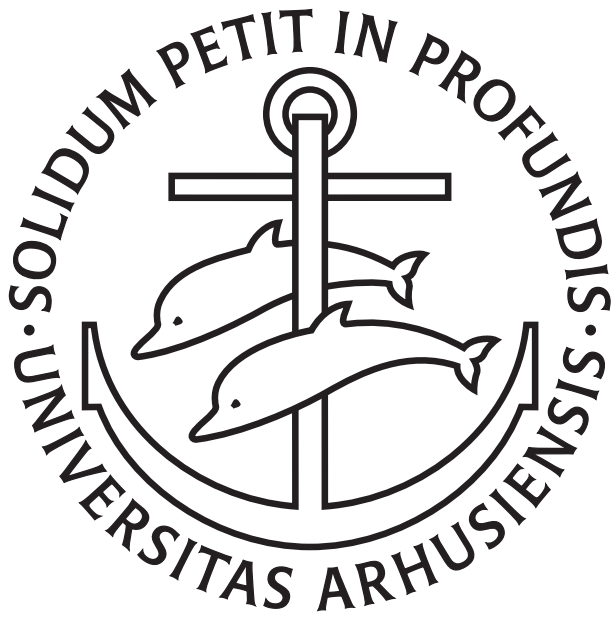


Aarhus University

PhD Dissertation

# Design and Analysis of Web Application Frameworks

Mathias Schwarz

*Supervisor:*
Anders Møller

Submitted: January 29, 2013

# Abstract

Numerous web application frameworks have been developed in recent years. These frameworks enable programmers to reuse common components and to avoid typical pitfalls in web application development. Although such frameworks help the programmer to avoid many common errors, we find that there are important, common errors that remain unhandled by web application frameworks.

Guided by a survey of common web application errors and of web application frameworks, we identify the need for techniques to help the programmer avoid HTML invalidity and security vulnerabilities, in particular client-state manipulation vulnerabilities. The hypothesis of this dissertation is that we can design frameworks and static analyses that aid the programmer to avoid such errors.

First, we present the JWIG web application framework for writing secure and maintainable web applications. We discuss how this framework solves some of the common errors through an API that is designed to be safe by default.

Second, we present a novel technique for checking HTML validity for output that is generated by web applications. Through string analysis, we approximate the output of web applications as context-free grammars. We model the HTML validation algorithm and the DTD language, and we generalize the validation algorithm to work for context-free grammars.

Third, we present a novel technique for identifying client-state manipulation vulnerabilities. The technique uses a combination of output analysis and information flow analysis to detect flow in the web application that might be exploited by malicious clients.

We implement and evaluate the techniques to study their usefulness in practice. We find that JWIG is useful for implementing large web applications. We furthermore evaluate the static analyses techniques and find that they are able to detect real bugs with few false positives in open-source applications.

# Resume

Igennem de senere år er et stort antal web applikations-frameworks blevet udviklet. Disse frameworks gør det muligt for programmører at genbruge løsninger og undgå typiske fejl i web applikationer. Selvom sådanne frameworks hjælper programmører med at undgå mange typer fejl, så viser det sig, at der alligevel er typer af fejl, som endnu ikke bliver håndteret tilstrækkeligt af frameworks.

Vi undersøger hvilke typer fejl, der ofte opstår i web applikationer, og hvilke løsninger forskellige frameworks har på problermerne. Denne undersøgelse viser, at der er brug for teknikker, som kan hjælpe programmøren med at undgå ugyldig HTML og sikkerhedsproblemer, især sårbarheder som skyldes manipulation af klientens tilstand. Hypotesen i denne afhandling er, at vi kan designe nye frameworks og statiske analyser som kan hjælpe programmøren med at undgå disse typer af fejl.

Først præsenterer vi JWIG frameworket der kan bruges til at skrive sikre web applikationer som er lette af vedligeholde. Vi diskuterer hvordan dette framework løser nogle af de typiske problemer med et API som er designet til at undgå fejl.

Dernæst præsenterer vi en ny teknik til at verificerere korrekthed af genereret HTML i web applikationer. Vi bruger streng-analyse til at tilnærme det mulige output med kontekst-frie grammatikker. Vi modellerer HTML valideringsalgoritmen og DTD specifikationssproget og viser, hvordan valideringsalgoritmen lader sig generalisere til at virke for kontekst-frie grammatikker.

Endelig præsenterer vi en ny teknik til at finde sårbarheder, som skyldes manipulation af klientens tilstand. Teknikken bruger en kombination af en output analyse og en information flow analyse til at finde flow af data som kan udnyttes af en ondsindet klient.

Vi implementerer og evaluerer teknikkerne for at undersøge, hvor godt de virker i praksis. Vi ser, at JWIG er brugbart til at implementere store web applikationer. Vi evaluerer også analyseteknikkerne og ser, at de kan bruges til at finde fejl i open-source programmer, som vi har fundet på nettet.

# Acknowledgments

Thanks to the members of the Programming Languages group at Aarhus University for insightful discussions, inspiring friendships, and countless foosball tournaments.

I am truly thankful to Anders Møller for his patient and insightful mentorship. He has been the best supervisor one can imagine.

I thank Simon Holm Jensen for his valuable feedback during the writing of this dissertation.

I would like to thank Henning Rohde and Anna Gringauze for an inspiring stay at Microsoft Research.

Thanks to my parents and brothers for supporting and encouraging me.

A very special thanks to my wife Rikke who has been a loving help and support throughout my studies. Our son Andreas has barely been with us for a year but I thank him for his happy smiles and for reminding me to never stop being curious.

*Mathias Schwarz,*
*Aarhus, January 29, 2013.*

# Contents

# Part I

# Overview

# Chapter 1

# Introduction

The hypothesis in this dissertation is that we can design frameworks and static analyses that aid the programmer to avoid invalid HTML and to avoid programs that are vulnerable against common security vulnerabilities, in particular *client-state manipulation* vulnerabilities. For such frameworks and analyses to be useful to a programmer it must be possible to apply them to large scale programs.

When the web was invented about 20 years ago, it was designed for publishing static content. During the following years it evolved into a rich platform for implementing software systems. With simple scripts that generated content, programmers became able to write applications. These applications allowed interactions between clients and servers without requiring the client to install additional software. This helped the web to quickly become a popular platform. Today, we call the combination of a client interface and a server that allows interactions and provides data for the client over the HTTP protocol a *web application*.

As sketched in Figure 1.1, web application interactions use the stateless HTTP protocol. Such HTML documents may include *forms* for server interaction and *JavaScript* code for implementing rich interfaces. In web applications, the server typically stores state in a database and generates HTML documents depending on this state. Similarly, the client may store some state in JavaScript, as cookies or as client-state parameters (see Section 4.3). As we will discuss in this dissertation, both client and server storage require careful security considerations.

With the increased popularity of the web as a platform, web application programmers began developing *web application frameworks* of reusable components to facilitate faster web application development and better application structure. Starting from simple CGI scripts [72], web application frameworks have today

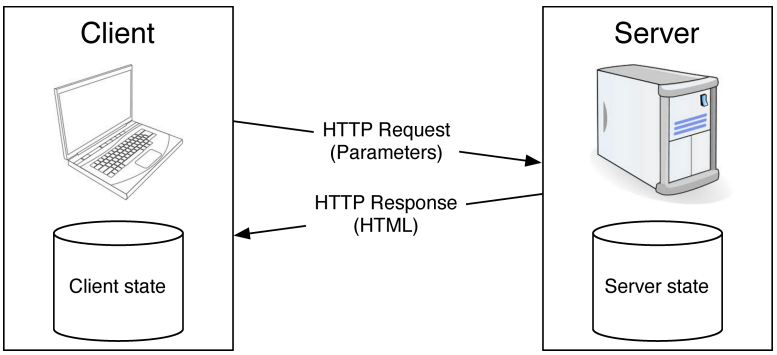


Figure 1.1: Web interactions follow a stateless request-response protocol but both client and server may store state.

evolved into complex libraries that allow rapid development of structured web applications. This development has resulted in numerous frameworks and techniques for web programming. This dissertation surveys the field of web application frameworks. From this survey, we identify the need to consider techniques to guarantee security and output correctness as central tasks of web application frameworks, and we argue that such techniques are useful for web application programmers.

Web application frameworks help the programmer avoid commons errors by reusing solutions to common problems. However, HTML invalidity and security vulnerabilities remain a problem in today's web applications. With framework design and static application analysis, this situation can be improved. This dissertation discusses such framework design and static analyses and argues that it is useful for programmers to employ these techniques while developing web applications. An important observation is that many such errors relate to HTML that is generated by the server and this dissertation investigates techniques that rely on analysis of this output.

The dissertation applies two different methods for avoiding errors in applications: 1) It presents a novel web application framework that is designed to be *safe by default* against common security problem and that allows for easy and precise analysis of HTML validity. 2) It presents analyses for existing web application frameworks. These analyses are evaluated on a set of third-party, open-source applications to study the usefulness of the techniques in practice.

## 1.1 Overview

The dissertation is composed of three parts. Chapter 2 contains an overview of the area of web application programming, including central concepts, frameworks, and security considerations. Furthermore, it presents a survey of currently popular web application frameworks. Through this survey, we argue that analyses are required to avoid the most common web application program errors, and that a new web application framework should be designed to avoid these errors.

Based on Chapter 2, Chapter 3 presents the design of the JWIG web application framework that is further discussed in the paper that is included as Chapter 6 of this dissertation. The JWIG framework is designed for writing web applications that are correct, secure, maintainable, and Chapter 3 will discuss how the JWIG framework solves the common problems identified for existing frameworks. The JWIG framework is a thoroughly redesigned successor to an older, namesake framework and we will also briefly study the conceptual differences between the two.

Chapter 4 presents static analyses for existing frameworks. These analyses allow reasoning about security and correctness for web applications written using existing frameworks. This work includes output validation for web applications as well as a security analysis to guard against client-state manipulation attacks.

The reader of this dissertation is assumed to be familiar with the areas of static program analysis, context-free grammars, and regular languages as well as with with the concepts of Java, HTML, XML, and HTTP.

Part II contains the publications I have co-authored as part of my PhD studies. Specifically, the following papers were co-authored as part of my PhD studies at Aarhus University. They are all submitted in extended form along with this dissertation:

**JWIG: Yet Another Framework for Maintainable and Secure Web Applications**
with Anders Møller. Appeared in Proc. *5th International Conference on Web Information Systems and Technologies*, March 2009

**HTML Validation of Context-Free Languages**
with Anders Møller. Appeared in Proc. *14th International Conference on Foundations of Software Science and Computation Structures*, 2011

**Automated Detection of Client-State Manipulation Vulnerabilities**
with Anders Møller. Appeared in Proc. *34th International Conference on Software Engineering*, 2012

## 1.2 Method

This section describes the methods that have been applied to investigate the hypothesis.

### Implementation

The techniques that are described in this dissertation have all been implemented as software packages (see Section 1.4). Implementation of web framework analysis requires specialization towards individual frameworks since analysis of frameworks themselves is likely to yield an imprecise result (Section 2.1.2).

In our implementation work, we have decided to focus on the Java programming language and the Servlet, JSP, and Struts web application frameworks. It is assumed that the result of analyzing framework based Java web applications is representative for the result that would arise from analyzing similar applications and frameworks in other languages. Section 2.4 will give a more detailed description of the structure of the web application frameworks that our tool can handle and compare them to widely used web application frameworks.

The JWIG web application framework is likewise implemented in Java and demonstrates an extension to the Java programming language that has useful properties for web application programmers. A similar extension could be created for other programming languages.

### Experimental evaluation

We have evaluated the analysis techniques by applying the software to open-source benchmarks. We have evaluated the techniques based on two criteria: 1) The precision of the analysis. In particular, how many false positives arise when running the tool on the benchmarks. We determine the number of false positives through manual inspection of the warnings given by the tool. 2) The usefulness to the programmer. In particular, how well does the technique guide the programmer towards correcting the issue in the code. We evaluate this property by discussing the process needed to asses the warnings given by the tool.

In the case of JWIG (see Chapter 3) we have implemented a software system CourseAdmin using the framework. CourseAdmin serves as a case study for evaluating JWIG in relation to the design goals of the JWIG web framework.

The experimental evaluation is also useful for identifying the need for future improvements of the evaluated techniques. The experimental evaluations will be described in further detail later in individual sections for each of the techniques.

## 1.3 Contributions

This dissertation presents the following main contributions:

- We survey the most influential web application frameworks and evaluate them in terms of output correctness, security properties and well-established software engineering principles.

- We identify the need and present the design of a new web application framework, *JWIG*, that avoids problems that are common to the existing frame-

works. The framework is evaluated through a comparison with existing frameworks and by implementing a large web application, CourseAdmin.

- We present and evaluate a novel algorithm for validating dynamically generated HTML pages. The algorithm is applicable to all SGML and XML based languages that are described by DTD schemas. The algorithm is furthermore able to include more SGML features in the validity check compared to previously available methods. These features include SGML content model exceptions and optional start tags. The usefulness of the approach is evaluated on a set of benchmark applications.

- We present and evaluate a technique for detecting security vulnerabilities that are related to client-state manipulation. Client-state manipulation vulnerabilities allow a malicious client to change data that the server stores as part of the document on the client side. The usefulness of the approach is evaluated on a set of benchmark applications.

## 1.4 Software packages

I have worked on a number of software packages as part of my PhD studies. First and foremost, these packages serve to evaluate the usefulness of the techniques described in my papers. In connection with my publications I have worked on the following software packages:

- **JWIG**[1][2] - A framework for writing Java web applications. This framework is further discussed in Chapter 3.

- **CourseAdmin** - A course administration tool that serves as a large-scale benchmark of JWIG. This tool currently serves as the course administration tool used for most courses at the Department of Computer Science.

- **WARLord**[3] - A tool for reasoning about Java web application response output and information flow in web applications. The tool implements the HTML validity analysis and the client-state manipulation analysis with a shared front end. These analyses are discussed in Chapter 4.

[1] `http://www.brics.dk/JWIG/`
[2] The JWIG analysis suite was implemented by Esben Andreasen.
[3] `http://www.brics.dk/WARLord`

# Chapter 2

# Web frameworks and web applications

Web application frameworks fall into two major categories: server-based frameworks concerned with programming the server side of web applications and client-based frameworks concerned with programming browsers. Client-based frameworks enable the programmer to write applications with rich and highly interactive user interfaces while server-based frameworks allow the application to run on a machine that is controlled by the application provider. This makes it possible for the programmer to implement his program in the language of his choice and allows him to draw on mature and well-known techniques and frameworks for implementing his application.

Some will use web applications in ways not anticipated by the programmer and some with malicious intent. Some will use browsers different from the one used by the developers of the applications. This results in challenges for web application programmers, and frameworks are useful to solve many such challenges.

Recently, there has been an increased focus on client-based web frameworks. The lack of type safety in the JavaScript programming language has motivated tools like TAJS [34] that assist the programmer in writing better client side programs. In spite of new client-based technologies, server-based frameworks remain widely used and it remains an important goal to improve stability and correctness of server-based web applications. Many of the problems that are common to server side web applications remain unsolved. The focus of this dissertation is on such server-based applications and in the following sections "web application framework" refers to server-based web application frameworks unless something else is explicitly stated.

Much work has been done to improve the quality of web applications. Testing and verification techniques that are applicable to programs in general, are also useful for web application programmers. Web applications, however, share common traits that make it possible to create specialized analysis techniques useful to analyze any web application. In this chapter, we will survey the area of web application frameworks. This will serve as a basis for understanding the applicability of the solutions presented later in this dissertation. Through this survey, we will identify the need for solutions to problems that are present across the current web application frameworks. In the following chapters, we will discuss solutions for these problems, both solutions that avoid the problems at the framework level and solutions through static analysis.

## 2.1 Purpose and structure of web frameworks

As discussed above, web application frameworks makes it possible for programmers to reuse commons implementations of some of the tasks and to abstract away lower-level details of the interaction with the client. In this section we will identify the core components of server-based web application frameworks and survey a portion of the most widely used web frameworks based on their design of these components.

### 2.1.1 Components of a web framework

Web application frameworks differ highly on their levels of abstraction and the amount of features they make available to the programmer. Most frameworks contain myriads of features, some of them general purpose, some of them specialized towards specific application architectures encouraged by the specific framework. Four basic components are, however, made available by all web application frameworks, a *dispatcher*, a *decoder*, a *generator*, and a *store*. This section presents an overview of these components and the design space for each of them.

In Section 2.4 we will survey web application frameworks to see examples of each of the points in the design space.

#### The dispatcher

A *dispatcher* defines the relationship between HTTP requests and web application code by locating and invoking code based on the contents of HTTP requests. A dispatcher can be *explicitly* configured through a configuration file or it may be *implicitly* configured through conventions. In the latter case, conventions typically determine how the dispatcher generates mappings from URLs to code based on the class or file structure of the program code.

The design of the dispatcher defines what the *basic unit* of the web application framework is. A basic unit corresponds to a single, possible *entry point* that can be invoked by the dispatcher. Basic units can be *source files* in which the dispatcher will typically start executing the code from the beginning of the file (see for example PHP and JSP in Section 2.4). It can be *classes* where the dispatcher can invoke a method based on a predefined interface (see for example Servlets or Seaside). It could also be individual *functions* as it is the case for Hop and JWIG. We will refer to an instance of such basic unit as a *page* in the application.

#### The decoder

A *decoder* decodes requests from clients and provides means for the web application to read parameters, headers, and request body data sent as part of the request. We can categorize decoders into two types:

1) *Pull decoders* that provide an interface to the decoder itself. The programmer retrieves the request parameters by invoking methods on this interface. A map from names to parameter values is the simplest form of such a pull decoder. Examples of such decoders exist in PHP, JSP, and the Servlet framework.

2) *Push decoders* that inject the request values into method parameters or Java Bean properties before the dispatcher invokes the entry point. The choice of parameters or properties typically depends on the choice of basic unit for the decoder: parameters are used if the basic unit is functions and Java Bean properties are used otherwise. Examples of frameworks that use push decoders include JSF, Struts, Hop, Seaside, and JWIG.

The push decoder approach makes it simple to identify the interface of a web application, while the pull approach provides the programmer with flexibility. Most of the push decoder frameworks also provide means for the programmer to read

parameters in pull style for situations where the added flexibility is necessary. Of the pull decoder frameworks discussed later in the chapter, only Hop and Seaside are purely push style and provide no pull features.

### The generator

The *generator* constructs output that is returned to clients as response to requests to the server. The generator has a large variety of design features which we will try to categorize here.

The web framework may provide a *domain specific language* (DSL) for writing *templates* intermixed with program code or it may rely on general purpose language syntax only. Template languages typically permit the programmer to write HTML code fragments in exactly the same syntax as is used for writing static HTML documents and they provide some way to combine these templates into a document. Frameworks *without templates* typically allow the programmer to generate output through function calls that have side effects in the generator.

Templates are used in the PHP, Servlets, JSP, JSF, Struts, and JWIG frameworks while the Servlets, Seaside and Hop frameworks rely on the syntax of Java, Smalltalk and a Scheme-like language respectively. Furthermore, JSP and Struts allow the programmer to intermix templates and calls that have side effect in the generator.

Orthogonally to the inclusion of template syntax, the output might be represented as a *first class* value in the programming language or be *implicitly represented* by the runtime system, for example as a result of appending data to an output stream read by the client. While first class values offer a high degree of freedom to the programmer, the stream approach allows data to be retrieved and rendered by the client while the server side is still executing.

A further design choice for the output representation is whether the values are *mutable* or *immutable*. On the client side, the document is represented as a first class, mutable structure, the DOM [30]. There are only few examples of such a representation in server side web application frameworks.

Orthogonally to these types, the generator may offer various degrees of protection against cross-site scripting attacks. We will get back to this issue in Section 2.3.2.

### The store

The *store* holds inter-request data. The store is often separated into various *scopes*, such as an *application* scope where the data is shared between all requests and clients, a *session* scope where the data is shared among requests from the same client, and a *request* scope where the data is shared only for the current request. Frameworks may provide fewer or more scopes than this or they may provide only the scope features of the hosting programming language.

The store may require *separation* from the rest of the code, so that the structure of the store must be represented as classes that are separate from the code that interacts with the generator. It might also allow *integration* so that the programmer can store data as part of the same code that interacts with the generator.

Finally, the store may be *typed* so that the type of a value is guaranteed by the type system of the programming language or it may be *untyped* and leave it up to the programmer to ensure type correctness. In Java the values may be represented as properties of Java Beans to have a typed store or represented as a string-to-object maps to have an untyped store. In the latter case, the programmer must cast the value to the expected type. Some frameworks, such as JSP, offer both a typed and an untyped storage.

### 2.1.2 Static analysis in the presence of web application frameworks

The large variety in web application frameworks and the differences in components pose a challenge when creating static analysis tool for web applications. Web application frameworks often rely on highly reflective code that is hard to analyze precisely. Therefore, tools are typically limited in scope to a single or a few web application frameworks for which specialized analysis support has been implemented in the tool. This is also the case for the WARLord tool discussed later in this dissertation.

So far, only little work has been done to overcome this challenge. Recently, Sridharan et al. presented the F4F (Framework for Frameworks) [78] system in which they are able to describe framework related data flow through a specification language WAFL. While they do not identify the components as such, they specify data flow from the decoder to the program code and through the framework store. Furthermore, the dispatcher is handled in enough detail to model entry points for an analysis. From the WAFL specification, the tool generates easily analyzable code for framework data flow and inserts the code into the Java byte code of the program. This allows the tool to analyze the program much more precisely compared to the result of analyzing the framework code. They demonstrate how they are able to implement a high precision taint analysis that is applicable to all frameworks supported by F4F and they demonstrate how Servlets, JSP, Spring MVC, and Struts can be modeled in the WAFL language. So far, the tool only supports Java but the approach seems applicable to other languages as well.

It might be possible to generalize the ideas of F4F to also include the generator and to generalize F4F to support additional frameworks and languages. The supported frameworks are very similar, and it is remains to be seen whether code generation is sufficient to handle frameworks that are substantially different from the JSP/Servlet family.

## 2.2 Software engineering principles in web frameworks

Web application developers must know a myriad of design patterns and architecture patterns to write and to understand modern web applications. In many cases such patterns are integral parts of web application frameworks. This section is not meant as a general introduction to software engineering best practices or to architecture principles in general; many good books already exist on this topic. Rather, in this section we will briefly discuss these principles at the level needed to understand the rest of this dissertation.

### 2.2.1 Cohesion, coupling, and the MVC pattern

The notions of *high cohesion* and *low coupling* describe applications that are structured into multiple loosely coupled components [79]. In lowly coupled applications, changes can be made locally without affecting other parts of the application. Low coupling improves maintainability of the code. Dually, in highly cohesive systems, code that is closely related in functionality is also closely related in the code. High cohesion improves readability of the code. High cohesion and low coupling can be properties of the overall architecture as well as the implementation and structure of the application code.

Several architectural patterns promote high cohesion and low coupling. In particular the model-view-controller (MVC) pattern [8, 43, 74] has gained popularity in web applications frameworks [46]. In this pattern, the use of the generator - the *view* - is separated from the *model* that represents the values in the store and the

*controller* that updates the model and view upon client interaction. Many good sources explain the MVC pattern and the reader is assumed to be familiar with the pattern.

In Chapter 3 we will discuss how a web framework can guide programmers towards high cohesion and low coupling in web applications without requiring the applications to adhere to a strict and specific implementation of the model-view-controller architecture pattern.

### 2.2.2 Safety by default

*Safety by default* is the principle that the framework rather than the application itself should guard against program errors and security vulnerabilities. In such a system, creating an application that has a certain type of error or vulnerability requires explicit action from the programmer. For example, in statically typed programming languages it is, assuming that the type system is sound, impossible to unawarely write a program that results in a type error at runtime. Most such languages do, however, provide specific features that allow the programmer to disable this safety. In the case of Java, one such feature is type casting which serves to override the type of an expression in the type checker.

Similarly, web application frameworks may guard against common types of errors and vulnerabilities by exposing an API that is designed to make it impossible to write an incorrect or vulnerable program. Applied at the web application framework level, the principle of safe by default can address common problems in web applications such as cross-site scripting and HTML invalidity. We will discuss common errors in web frameworks in Section 2.3 and see examples of safe by default design later.

## 2.3 Common web application errors

Two groups of errors relate directly to client interactions and are therefore of particular interest to web application developers: Output correctness and security vulnerabilities. Valid web application output guarantees to the programmer that all standards-compliant web browsers will render the documents consistently. Security vulnerabilities may allow malicious clients to attack the system and gain unintended privileges or knowledge. In this section, we will discuss these two groups of errors in more detail.

The solutions to web application errors fall into three categories: 1) *static* solutions where the program is analyzed by a tool prior running the program, 2) *dynamic* solutions where the error is detected and prevented at runtime, and 3) *framework* solutions where the language or the framework is constructed in such a way that it is impossible to unawarely write a vulnerable program. That is, the framework is safe by default.

### 2.3.1 Output correctness

When clients interact with web applications, clients expect the server to communicate using some the established web standards. The HTML family of languages [45] up to and including HTML 4 are specified using the DTD language. The XML version of XHTML is described in a similar specification language XML DTD [52]. In this dissertation the word HTML does not refer to XHTML unless explicitly stated.

A DTD description allows a *validator* to determine whether an HTML document is *valid*. We call an HTML document *valid* if it conforms to one of the HTML DTD specifications. Validity can be verified for a single HTML document by running an off-the-shelf validator that compares the document to a DTD.

```
1  import javax.servlet.http.*;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4
5  public class InvalidHTML extends HttpServlet {
6
7      public void doGet(HttpServletRequest request,
8                        HttpServletResponse response)
9              throws IOException {
10         PrintWriter writer = response.getWriter();
11         writer.write("<html><head><title>Incorect HTML" +
12                 "</title></head>");
13         String name = request.getParameter("name");
14         if (name.equals("World"))
15             writer.write("<body>Hello " + name + "!</body>");
16         else
17             writer.write("<body>Greetings " + name + "." +
18                     "May you live long and prosper.");
19         writer.write("</body></html>");
20     }
21 }
```

Figure 2.1: A Java servlet that outputs invalid HTML in one case.

Since the HTML specifications only prescribe the meaning of valid documents, invalid HTML documents are often rendered differently, depending on which browser is used. For this reason, careful HTML document authors validate their documents, for example using the validation tool provided by W3C[1]. Chen et al. surveyed a large set of web pages and showed that many existing HTML documents on the web are invalid [9] so validity is an important concern for programmers.

Web applications generate HTML dynamically, and while the W3C validator can validate individual documents, it cannot guarantee that all output that is generated by a web application is valid. Other methods are therefore needed to verify output correctness for web applications.

Figure 2.1 shows an example of a Java Servlet that will generate invalid HTML if the value of the input parameter `name` is the string `"World"`. The extra `</body>` tag is invalid according to the DTD descriptions of the HTML languages. While this error might be easy to spot for an experienced HTML author, the work presented in Section 4.2.2 reveals that examples of extra end tags do exist in the real world applications.

In essence, checking output correctness of a web application amounts to verifying that the application uses the generator of the framework in a way that always results in valid documents. As a runtime solution, it is possible to instrument the generator to validate each HTML page on the fly prior to sending the page to the client. However, such runtime validation will increase the load on the server significantly, and static guarantees are therefore desirable. In Sections 3.4 and 4.2.2 we will discuss static analysis solutions to validity problems.

A further refinement of output correctness analysis is to consider application specific rules about the structure of the output. For example, the programmer might want to ensure that a particular element with a specific `id` is always present in the output document. The CSS or JavaScript files may assume that this is the case, so the absence of this element could result in unexpected behavior. The programmer could employ a static output analyzer to verify that the program output always follows the structure that he expects. This dissertation will primarily present anal-

[1] `http://validator.w3.org`

yses for checking HTML validity but the algorithms could be extended to include application specific requirements.

### 2.3.2 Security

To protect web applications against malicious users, the programmer must be aware of numerous kinds of possible vulnerabilities and countermeasures. Among the most popular guidelines for programming safe web applications are those in the OWASP Top 10 report that covers "the 10 most critical web application security risks" [68].

In the OWASP Report, vulnerabilities are not necessarily the result of an incorrect program. Incorrect setup of SSL certificates, failure to use encrypted transport protocols, and insecure data storage are also included in the report. While such issues are in general important to consider, this dissertation focuses on program analysis and we will therefore only discuss the vulnerabilities that are directly related to program errors. The three categories *injection*, *cross-site scripting*, and *insecure direct object references* are of particular interest to this dissertation.

The two OWASP categories *injection* and *cross-site scripting* have been in particular been subject to research. A third OWASP category *insecure direct object references* is related to the two other categories in that all three categories of vulnerabilities are caused by trust in data sent from the client. In all three cases, the vulnerability could be avoided by inspecting the value before use. The class of *client-state manipulation vulnerabilities* are a superset of *insecure direct object references*. This category is not limited to values that act as database identifiers. It includes all values that are stored on the client side and returned to the server in a subsequent request. Changing such a value could lead to unexpected behavior on the server. We will discuss a technique to find such client-state manipulation vulnerabilities in Section 4.3.

In the rest of this section, we will discuss the three types of errors: cross-site scripting, command injection, and client-state manipulation attacks and discuss how they relate to each other.

#### Injection

According to OWASP Top 10 report, injection vulnerabilities can occur when untrusted data is sent to an interpreter as part of a command or query. The malicious client can exploit such a vulnerability to change the effect of the query arbitrarily to gain access to or to change data.

An example of an injection vulnerability is shown in Figure 2.2. The programmer intends to query the database to validate the username and password of a client. He inserts the parameter strings from the client into the SQL query and runs the query on the database to see whether a matching user exists. A malicious user can provide the strings `admin';--` for the `username` parameter and the empty string for the `password` parameter to generate the query string `SELECT * FROM user WHERE username='admin';--' AND password=''`. In SQL `--` starts a comment and in effect the malicious client will be logged in as `admin` without a password check.

The problem can be avoided by not inserting the untrusted string into the query directly. There are multiple solutions to this. One solution is to use *prepared statements* where the programmer writes the database SQL query with place-holders instead of values and lets the database framework be responsible for inserting the escaped values correctly into the queries. Another popular solution is object-relational mapping frameworks such as Hibernate [67] for Java and C♯ where the programmer uses object representations of the database contents and lets the mapping framework generate database queries to store and retrieve the data.

```
1  import javax.servlet.http.*;
2  import java.io.*;
3  import java.sql.*;
4
5  public class Injection extends HttpServlet {
6
7      public void doGet(HttpServletRequest request,
8                        HttpServletResponse response)
9              throws IOException {
10         PrintWriter writer = response.getWriter();
11         String username = request.getParameter("username");
12         String password = request.getParameter("password");
13         boolean loggedIn;
14
15         try {
16             Connection connection = DriverManager.getConnection
17                     ("","","");
18             String query = "SELECT * FROM user WHERE " +
19                             "username='" + username + "' AND " +
20                             "password='" + password+ "'";
21             Statement statement = connection.createStatement();
22             ResultSet resultSet = statement.executeQuery(query);
23             loggedIn = resultSet.next(); //true if any rows exist
24         } catch (SQLException e) {
25             e.printStackTrace(writer);
26             return;
27         }
28
29         writer.write("<html><head><title>Injection example" +
30                 "</title></head><body>");
31         if (loggedIn) {
32             request.setAttribute("user", username);
33             writer.write("Logged in as " + username);
34         } else {
35             writer.write("Incorrect user or password.");
36         }
37         writer.write("</body></html>");
38     }
39 }
```

Figure 2.2: Examples of a servlet that is vulnerable to injection attacks.

### Cross-site scripting

Cross-site scripting vulnerabilities occur when untrusted data is send to a web browser without proper validation or escaping. In a vulnerable application, the malicious client can gain control over the generator by constructing data that contains HTML tags. If such data is inserted into the document, it will be interpreted by the client in a way that was not intended by the application programmer. Effectively, such a vulnerability allows the malicious client to change the structure of the document and manipulate the way the page is presented to other clients. A malicious user can furthermore exploit a cross-site scripting vulnerability to execute JavaScript in the browser.

Figure 2.3 shows a simple example of a cross-site scripting vulnerability in a Java servlet. The servlet accepts the parameter `name` from the client and generates a page containing this value. To exploit the vulnerability, the client can send a request where the `name` parameter contains HTML. Examples of vulnerabilities also include programs where a parameter value is stored in a database by one page in the application and later inserted into a document on another page. In such examples, the connection between client input and generator use is easy for the

```
1  import javax.servlet.http.*;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4
5  public class XSS extends HttpServlet {
6
7      public void doGet(HttpServletRequest request,
8                        HttpServletResponse response)
9              throws IOException {
10         PrintWriter writer = response.getWriter();
11         writer.write("<html><head><title>XSS example" +
12                 "</title></head>");
13         String name = request.getParameter("name");
14         writer.write("<body>Hello " + name + "!");
15         writer.write("</body></html>");
16     }
17 }
```

Figure 2.3: Example of a servlet that is vulnerable to cross-site scripting.

programmer to miss.

Framework solutions for cross-site scripting range from framework-provided API methods for escaping HTML values to frameworks with template languages that prohibit direct insertion of unchecked data. We will discuss framework solutions in the following sections of the dissertation.

#### Client-state manipulation

Client-state manipulation vulnerabilities occur when the server stores trusted data as part of the document on the client side and expects this data to remain unchanged from one request to another. In Section 4.3 we furthermore argue that vulnerabilities are a result of updating or querying the store with untrusted data. Such vulnerabilities are similar to injection and cross-site scripting vulnerabilities in that untrusted data is used on the server without checking its contents. Client-state manipulation vulnerabilities are distinct in that the untrusted data is neither sent to a query as in injection vulnerabilities or to the output to a web browser as in cross-site scripting vulnerabilities. Rather, the vulnerability allows the malicious to change or read unauthorized data on the server. In client-state manipulation attacks, this is done by making the web application accept a modified value of a parameter rather than changing the structure of a command issued by the web server. In the OWASP Top 10 report, the vulnerability category *insecure direct object references* has significant overlap with client-state manipulation vulnerabilities but the problem is more general than that of object references. Our experiments in the work related to Section 4.3 indicate that object references are an important source of such vulnerabilities.

The example in Figures 2.4 and 2.5 shows a client-state manipulation vulnerability. This vulnerable code was found as part of the work described in Section 4.3. The JSP page in Figure 2.4 stores the user name of the current user in a hidden form field. In Figure 2.5, the programmer reads the value of the `nickname` parameter and expects it to remain unchanged. The `nickname` value is used for finding and updating the information about the user. If a malicious client changes the value of `nickname` he can change information about any user.

In Section 4.3 we present a static solution for detecting client-state manipulation vulnerabilities [61]. Vulnerability detecting involves both an analysis of the output to determine which parameters contain client-state and an information flow analysis to track the values to where they are used in the application.

```
<% ChatRoomList roomList =
   (ChatRoomList)application.getAttribute("chatroomlist");
 ChatRoom chatRoom = roomList.getRoomOfChatter(nickname);
 Chatter chatter = chatRoom.getChatter(nickname); %>
<html><head>
<meta http-equiv="pragma" content="no-cache">
<title>
  Edit your (<%=chatter.getName()%>'s) Information
</title>
<link rel="stylesheet" type="text/css"
  href="<%=request.getContextPath()%>/chat.css">
</head>
<body bgcolor="#FFFFFF">
<form name="chatterinfo" method="post"
  action="<%=request.getContextPath()%>/servlet/saveInfo">
<table width="80%" border="0" cellspacing="0"
  cellpadding="2" align="center" bordercolor="#6633CC">
<tr><td valign="top"><h4>Nickname:</h4></td>
<td valign="top"><%=chatter.getName()%></td>
<input type="hidden" name="nickname"
  value="<%=chatter.getName()%>">
</tr>
<tr><td valign="top"><h4>Email:</h4></td>
<td valign="top"><input type="text" name="email"
  value="<%=chatter.getEmail()%>">
</td></tr>
<tr><td valign="top">
<input type="submit" name="Submit" value="Save">
</td></tr></table></form></body></html>
```

Figure 2.4: `editInfo.jsp`

```
public class SaveInfoServlet extends HttpServlet {
  String nickname = null;
  String email = null;
  HttpSession session = null;
  String contextPath = null;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
       throws IOException, ServletException {
    nickname = request.getParameter("nickname");
    contextPath = request.getContextPath();
    email = request.getParameter("email");
    session = request.getSession(true);
    ChatRoomList roomList = (ChatRoomList)
      getServletContext()
      .getAttribute("chatroomlist");
    ChatRoom chatRoom =
      roomList.getRoomOfChatter(nickname);
    if (chatRoom != null) {
      Chatter chatter = chatRoom.getChatter(nickname);
      chatter.setEmail(email);
      ...
    }
  }
}
```

Figure 2.5: `SaveInfoServlet.java`

No web frameworks seem to be safe by default against all such vulnerabilities but it has been proposed to create filters that automatically insert checks on client-state

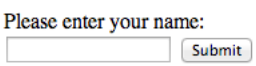


Figure 2.6: Page one shows an input form where the user can write his name.

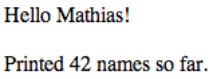


Figure 2.7: Page 2 shows a resulting page where the input is used.

parameters to ensure that they remain unchanged by the client [7, 75].

## 2.4 A survey of web application frameworks

In this section we will survey the current field of web application frameworks. Given the daunting number of such frameworks, we have to choose a set of frameworks to focus on; it is simply impossible to compare all of them in a single survey.

Based on the components presented in Section 2.1.1, we will discuss a set of existing web application frameworks that are representative in that each of the types of web application framework components are represented by one or more of the frameworks.

Through this survey we will identify the need for a new web application framework and show the need for analysis techniques to allow tools to give some guarantees about the behavior of programs written in existing frameworks. Furthermore, this section will serve as a reference for the remaining sections when discussing the implications and considerations of the presented analysis techniques.

We will show how the same simple example of a web application can be implemented in each of the frameworks. The application contains two pages. Page one (see Figure 2.6) presents a simple HTML form to the user to allow him to enter his name. Page two (see Figure 2.7) receives this parameter and prints it back to the user. Page two furthermore remembers how many names have been printed in the lifetime of the application. While this application is extremely simple, it does involve all of the surveyed components: the dispatcher, the decoder, the generator, and the store and it is therefore adequate to discuss the differences.

hello.html

```
1  <html>
2  <head>
3  <title>Enter your name</title>
4  </head>
5  <body>
6  Please enter your name:
7  <form action="hello.php" method="post">
8      <input type="text" name="name"/>
9      <input type="submit" value="Submit" />
10 </form>
11 </body>
12 </html>
```

Figure 2.8: Page one of the PHP application is a static HTML page.

sayhello.php

```
1  <html>
2  <head>
3  <title>Hello <?php echo $_POST["name"]; ?></title>
4  </head>
5  <body>
6  Hello <?php echo $_POST["name"]; ?>!<br/>
7
8  <?php
9  $file=file("data.txt");
10 $number = $file[0];
11 $file=fopen("data.txt");
12 fputs($file, number+1);
13 fclose($file);
14 ?><br/>
15 Printed <?php $number %> names so far.
16 </body>
17 </html>
```

Figure 2.9: A PHP application that receives input via an HTML form and prints it back to the user.

### 2.4.1 PHP

PHP is a framework as well an object oriented, imperative DSL for writing web applications [71]. It is very popular, in particular for writing small web applications. A multitude of frameworks build on top of PHP to extend it with new features. Here we will focus on PHP alone.

PHP imposes no specific structure on the web applications written in the framework. Each page in the application is a self-contained program and it is up to the programmer to ensure that the application has high cohesion and low coupling. It is possible to write MVC structured applications in PHP but the framework provides no features for easing the implementation such an architecture.

**Dispatcher:** The dispatcher of PHP is implicitly configured based on the set of PHP files on the disk. The basic unit is files and the PHP dispatcher (an Apache web server plugin) matches the URL of the request with the set of files available on the disk and invokes the PHP script in a file if it matches the request URL.

**Decoder:** The decoder is pull based and makes parameter values available through an implicitly defined variable. This variable holds a map from names to values. The name of this variable is `GET` or `POST` depending on the HTTP request method. Parameters are always strings and it is up to the programmer to ensure that these strings have the format that he expects them to.

**Generator:** PHP programs consist of snippets of HTML templates intermixed with PHP code. The PHP code allows the programmer to do computations and control the flow through the program. When the program runs, the snippets of HTML code are written to the stream-based generator as they are encountered. The generator offers no representation of the structure of the output and is not able to check any properties related to the output. The programmer must therefore carefully ensure that the structure of the output is well-formed and that inserted parameter values do not accidentally interfere with the intended structure of the output HTML document. For instance, the programmer must manually escape the $<$, $>$, $'$, and $"$ characters in content strings unless he wants it to be part of the HTML structure.

**Store:** PHP allows the user to store values in local variables. Such variables

are local to the current request. PHP also provides a simple mechanism for storing session values related to the current client, so that these values may be retrieved in later requests. PHP, however, offers no variables that are shared between all clients of the application. For such storage, PHP programmers usually resort to databases or other external storage.

Figure 2.9 shows a simple example of a PHP application consisting of two pages: the static `hello.html` containing an HTML form presented to the user. Page two is implemented by `sayhello.php`. This page receives the value in line 3 and 6 and prints it back to the user as part of a response page. Due to the lack of application scope storage in PHP, the example stores the value in a file and updates it in lines 9 through 13. As demonstrated in this example, PHP pages that generate a single constant document only contain the contents of that document.

#### Safety in PHP applications

PHP offers only very limited support for avoiding common web application errors. It is up to the programmer to ensure that cross-site scripting attacks are not possible. The PHP framework is therefore not safe by default against such vulnerabilities. Similarly, the SQL API of PHP is typically used in conjunction with SQL queries that are generated through string concatenation [80]. The programmer must remember to escape client provided values that become part of such query strings. PHP does provide an API for prepared statements. If the programmer uses this API exclusively then the application is free of SQL injection vulnerabilities. PHP has no support for avoiding client-state manipulation vulnerabilities. To our knowledge, no studies have yet been conducted to determine how wide-spread such vulnerabilities are but it is reasonable to expect that they are at least as wide-spread as in JSP applications.

### 2.4.2 Java Servlets

Java Servlets is a J2EE specification [64]. The Servlet framework provides a basic API for implementing web applications in the Java programming language. Java Servlets separate handling of the GET and POST request methods into two different Java methods (`doGet` and `doPost` respectively). According to the HTTP specification [21] GET requests are cachable and cannot have side effects. The Servlet framework does not enforce this but it makes it possible to write a Servlet that returns a view of the current server state from the `doGet` method and received requests that may update the state through the `doPost` method.

**Dispatcher:** The dispatcher is explicitly configured through a `web.xml` file or alternatively through Java annotations that map URLs to servlet classes. The basic unit is classes and to handle requests, each servlet class implements methods corresponding to each of the HTTP request methods (`GET`, `POST`, `PUT` etc.).

**Decoder:** The decoder is pull based and makes a request parameter map from names to values available to the web application programmer through an argument to the servlet method. Compared to PHP, Java Servlets abstract away the difference between the request methods and make all request parameters available through the same API methods on the `HttpServletRequest` class.

**Generator:** Similarly to PHP, the generator is stream based and offers no representation of the structure of the output. Since no template language is available in the Servlet framework, methods on the generator are explicitly called with string fragments of the desired output as arguments.

**Store:** The Servlet framework provides three scopes of untyped store: *application*, *session*, and *request*. The store is made available through maps from strings

SayHello.java

```
1  import javax.servlet.ServletException;
2  import javax.servlet.annotation.WebServlet;
3  import javax.servlet.http.*;
4  import java.io.*;
5
6  @WebServlet("/servlets/SayHello")
7  public class SayHello extends HttpServlet {
8      private int times;
9      protected void doPost(HttpServletRequest req,
10                           HttpServletResponse resp)
11             throws ServletException, IOException {
12         String name = req.getParameter("name");
13         PrintWriter out = resp.getWriter();
14         out.print("<html>\n" +
15                 "<head>\n" +
16                 "<title>Hello ");
17         out.print(name);
18         out.print("</title>\n" +
19                 "</head>\n" +
20                 "<body>\n" +
21                 "Hello ");
22         out.print(name);
23         out.print("!<br/>\n" +
24                 "<br/>\n");
25         out.print("Printed " + ++times + " names so far."+
26                 "</body>\n" +
27                 "</html>");
28     }
29 }
```

Figure 2.10: A Java Servlet application similar to the PHP application in Figure 2.9.

to objects and it is up to the programmer to ensure that the objects are of correct types. Furthermore, it is possible to create fields in servlet classes. Only one servlet class instance is created per declaration in the `web.xml` file, so such fields are essentially a way to store typed, application scoped data.

Similarly to PHP, page one of the example application is implemented as a static HTML file. The file shown in Figure 2.8 can also be used for the Java Servlet example with the sole exception that the value of the `action` attribute of the form is replaced by the value `servlets/SayHello`. The implementation of page two of the application is presented in Figure 2.10. The page is implemented as a servlet class and the parameter is received pull-style through to call to `getParameter`. The program sends output to the client by calling the `print` method on the `HttpServletResponse.getWriter`. As can be seen in Figure 2.10, Java Servlets have a significant syntactic overhead for writing typical HTML templates to the client. This results in programs that are significantly larger than equivalent PHP programs and Java Servlets are not often used exclusively for programming the server side of a web application. The Java Servlet framework is often used as a basis for implementing other frameworks that may replace all the components of the Servlet framework.

### Safety in servlet applications

Similarly to PHP, the Java Servlet framework leaves it to the programmer to make sure that the application is not vulnerable to the most common web application attacks. The Servlet framework does not contain a database API. It relies on

sayhello.jsp

```
1  <%@ taglib prefix="c"
2            uri="http://java.sun.com/jsp/jstl/core" %>
3  <%! int times; %>
4  <html>
5  <head>
6  <title>Hello <%= request.getParameter("name") %> </title>
7  </head>
8  <body>
9  Hello <c:out value="param.name"/>!<br/>
10 <br/>
11 Printed <%= ++times %> names so far.
12 </body>
13 </html>
```

Figure 2.11: The same example as in Figure 2.10, now using JSP. The `hello.html` remains the same in the JSP version of the program.

the Java Persistence API which makes prepared statements and escaping methods available.

The framework is not safe by default against the typical web application vulnerabilities. The generator requires the programmer to escape values before appending to the stream and the framework does not help the programmer to avoid client-state manipulation attacks.

The Java Servlets framework offers no features for ensuring output validity.

### 2.4.3 Java Server Pages

Java Server Pages (JSP) is an extension to the Servlet framework and is also described by a J2EE specification [14]. JSP is highly inspired by PHP and attempts to add some of the syntactic convenience of PHP to Java Servlets.

**Dispatcher:** Each JSP file is compiled into a Servlet but the JSP and Servlet frameworks differ on a few central points: The dispatcher is implicitly configured in that each JSP file is invoked by using the file name of the JSP file as the URL. As in PHP, the basic unit is the file. The generated class is never visible to the programmer.

**Decoder:** JSP allows the programmer to use the same decoder API as in the Java Servlet framework. Furthermore, parameters can also be fetched from the templates through the use of the JSTL Expression Language (EL) [51].

**Generator:** JSP replaces the generator of Java Servlets with a template DSL that allows the programmer to write snippets of HTML templates intermixed with Java program code. This results in a syntax for writing data to the output stream that will feel familiar to PHP programmers. As part of the template language, JSP provides a *tag* mechanism that allows the programmer to define tags. When encountered in the flow, these tags result in calls to Java code, and they allow the JSP files themselves to be free of embedded Java code.

**Store:** The store of JSP is the same as for the Servlet framework. An extra scope is added, the *page* scope in which values only exist throughout the execution of the JSP page. As in the Servlet framework, for JSP it is possible to store data in fields of the generated servlet class.

Figure 2.11 shows the hello example using a JSP page rather than a Java Servlet for retrieving the `name` parameter. The example demonstrates two ways of writing to the client. In line 6, Java syntax is used to write directly to the output stream

read by the client. In line 9, the JSTL library is responsible for writing the value to the client, and the value is fetched through the JSP Expression Language rather than by calling the Servlet API. The JSTL <*out*> tag escapes the value of `name` so the client is not able to exploit the request parameter for cross-site scripting.

#### Safety in JSP applications

The JSP standard tag library (JSTL) contains tags for reading values from the Servlet store and for writing these values to the client through the Expression Language (EL) [51]. These tags are safe by default against cross-site scripting since the programmer must explicitly set an `escapeXml` to false on the tag if he wants the data not be escaped. Unfortunately, the JSTL library is not the only way to print to write data to the client: The programmer may at any point embed Java code that is able to write output directly to the output stream as in the Servlet framework. JSP code often contains large snippets of Java code and it is convenient for the programmer to simply write values that computed in the Java code to the stream directly.

Similarly to Java Servlets, database persistence is not part of the framework and no security is built into the framework store.

Despite the tag-like structure of JSP page, the HTML tags are not parsed by the JSP engine and the framework offers no representation of the output. As a result, JSP gives no guarantees about the well-formedness or validity of the output.

### 2.4.4 JSF

Compared to JSP and Servlets, Java Server Faces (JSF) is a newer J2EE standard for implementing web applications. Unlike the two former frameworks, JSF imposes a MVC pattern on web applications that are built using the framework. Java beans are used for all data representation on the JSF framework. In this way JSF unifies the store with the way the client input is made available to the programmer and provides a simple an coherent way of defining applications that allows view and update of values in the beans.

**Dispatcher:** JSF offers a dispatching mechanism that is explicitly configured through a *faces-config.xml* file. The dispatcher invokes a Facelet template (see below) that is responsible for setting up the controller as an `action` on a form to update the state as a result of client interaction. Typically, a method on a Java bean is mapped to such a form action. The basic unit is the template files of the web application.

**Decoder:** The JSF decoder is push based. It pushes request parameters to the properties of a Java Bean through a mapping that is defined directly in the template language. The Java bean that receives the values is typically the same object as the dispatcher invokes a method on. This makes the input values available to the controller methods.

**Generator:** The JSF generator is based on an XML template language *Facelets* which on the surface is very similar to JSP. Contrary to JSP, the templates must be well-formed XML code and no Java code is allowed as part of the templates. JSF also provides a tag mechanism similar to that of JSP. As in JSP, tags can manipulate the control flow and insert values into the page shown to the client.

**Store:** Java beans also act as the store for JSF applications. Java beans can be marked with annotations to define the scope of the data. These scopes are similar to those of the JSP and Servlet frameworks. By using Java beans, the store becomes typed and external to the generator and different Facelets use different bean objects. This means that there is high degree of coherence between the structure of the store and the Facelets that use the data in the store.

hello.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<head>
    <title>Enter your name</title>
</head>
<body>
Please enter your name:
<h:form>
    <h:inputText value="#{hello.name}"/>
    <h:commandButton action="sayhello"/>
</h:form>
</body>
</html>
```

Figure 2.12: The JSF version of the hello program input page.

Hello.java

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;

@ManagedBean
@SessionScoped
public class Hello implements Serializable {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Figure 2.13: The bean where the input parameter `name` is stored.

Consider the example in Figure 2.12. This is the input page for the hello program. The page uses input elements from the JSF namespace to control how the request data is stored into beans upon receiving the request on the server. An EL expression defines the `name` request parameter should be stored in the `name` property of the `hello` bean on the server. The implementation of the `hello` bean is presented in Figure 2.13. The bean is annotated to be part of the session scope for JSF, so the value is specific to the client and remains until overridden by a subsequent request to a page that binds an input fields to this property. Furthermore, on the next request to the hello input page the value of the `name` property will be prefilled in the input field for the client. The `action` attribute of the form redirects the client to the `sayHello` page after setting the property. This page is shown in 2.14. This page writes the value of the `name` property of the `hello` bean back to the client and furthermore it calls a method `incTimes` on an application scoped bean `times` that increments and returns the value of a `times` property on that bean. The implementation of the `times` bean is straight-forward and is left out in this discussion.

sayHello.xhtml

```
1  <html xmlns="http://www.w3.org/1999/xhtml"
2        xmlns:h="http://java.sun.com/jsf/html">
3  <head>
4  <title>Hello <h:outputText value="#{hello.name}"/> </title>
5  </head>
6  <body>
7  Hello <h:outputText value="#{hello.name}"/>!<br/>
8  <br/>
9  Printed <h:outputText value="#{times.incTimes}"/> names so far.
10 </body>
11 </html>
```

Figure 2.14: The JSF version of the say hello page.

#### Safety in JSF applications

The JSF Facelets do not allow the programmer to write directly to the output and the framework can therefore escape any data and be safe by default against cross-site scripting. Similarly to Servlets and JSP, database interactions are left for other frameworks. JSF has no protecting against client-state manipulation.

While Facelets are well-formed XML fragments, the JSF frameworks does not guarantee output validity for the generated XHTML and it is up to the programmer to ensure validity. To some degree it is possible to validate the XHTML fragments in isolation using a validator but not all validity errors are easily found this way.

### 2.4.5 Struts 2

Struts 2 builds on top of JSP. There are significant differences between Struts 1 and Struts 2 applications. In this dissertation we will only discuss the more recent Struts 2. Struts applications are built as MVC applications where *Struts action* classes behave as the controller and the view is generated using JSP with a Struts specific tag library [53]. Contrary to JSP, the page are not invoked directly upon client request. Instead Struts invokes a Struts action which may update the state of the server. Afterwards, control is passed to the JSP page which then generates the view that is shown to the client.

**Dispatcher:** Struts replaces the dispatcher of JSP with an explicitly configured dispatcher which through a `struts.xml` file maps *Struts actions* to URLs. The flow from actions to JSP pages is controlled through the dispatcher configuration in `struts.xml` and is thus never explicitly in the action code.

**Decoder:** The decoder makes request parameters available through properties on the action class: the parameter named $p$ is simply stored in the action object by calling the setter for its `p` property. The decoder is therefore a push style decoder.

**Generator:** Struts uses JSP as its generator. Struts provides a large tag library and a more expressive query language for retrieving and inserting values from the Struts actions and other objects.

**Store:** The actions furthermore behave as store for values related to the current request. For session an application data, Struts provides untyped maps for storage in a way that is equivalent to JSP.

Figure 2.15 shows the Struts action for the Struts version of the Hello application. The action has a property `name` where the input parameter `name` is stored and a method `incTimes` that increments and stores the value of the `times` value. This method updates the value of `times` in the Java map that holds the store of appli-

StrutsHello.java

```
1  import com.opensymphony.xwork2.*;
2  import java.util.Map;
3
4  public class StrutsHello extends ActionSupport {
5      private String name;
6
7      public String getName() {
8          return name;
9      }
10
11     public void setName(String name) {
12         this.name = name;
13     }
14
15     public int incTimes() {
16         Map application = ActionContext.getContext().getApplication();
17         Integer times = (Integer) application.get("times");
18         if (times == null)
19             times = 0;
20         times++;
21         application.put("times",times);
22         return times;
23     }
24 }
```

Figure 2.15: The Struts action class for the Struts Hello application.

sayHelloStruts.jsp

```
1  %@ page contentType="text/html; charset=UTF-8" %>
2  <%@ taglib prefix="s" uri="/struts-tags" %>
3  <html>
4  <head>
5  <title>Hello <s:property value="name"/> </title>
6  </head>
7  <body>
8  Hello <s:property value="name"/>!<br/>
9  <br/>
10 Printed <s:property value="incTimes()"/> names so far.
11 </body>
12 </html>
```

Figure 2.16: Page two of the Struts Hello application.

cation scope values. Figure 2.16 shows page two of the application. The `property` tag reads the value of the `name` property on the action and inserts the value in the output. Similarly in line 11, the `incTimes` method is inserted and the value is printed to the output. We will omit page one as it is very similar the versions that we have seen already.

### Safety of Struts applications

Similarly to JSP, output can be done through the tag mechanism and Struts thus offers some protection against cross-site scripting vulnerabilities. As with JSP, Struts leaves data persistence to other frameworks and provides no protection against

client-state manipulation attacks.

Struts applications may also generate malformed and invalid HTML and the frameworks offers no features that helps the programmer avoid such problems.

### 2.4.6 Seaside

Seaside is a web application framework for Smalltalk [3, 23]. In Seaside, pages are represented as stateful *components*. These components form a tree structure and their state may be changed as as result of client interaction.

**Callback methods:** The `hello` method creates an anonymous *block* that is used as a callback. In Seaside, all interactions with the client happen through such callbacks, regardless whether the client submits a form, clicks a link, or interacts with the server through AJAX. As seen in the example, the callbacks are part of the object structure that represents the output that is sent to the client and the possible actions of the client therefore become part of the state. This stateful interaction style is highly different from the previously discussed frameworks in which all state was part of the application model. After an invocation of a callback method, the `renderContentOn` method is called to generate a new view of the application for the client.

**Dispatcher:** The Seaside dispatcher requires explicit configuration of the initial URL for each web application but is otherwise implicitly configured. For each client, the URL remains constant for all future views of the web application within the same session. This makes it impossible to create links to specific parts of the application.. The Seaside dispatcher sets up the tree of components when the client makes his initial request to the server. Each component may hold callback that can be invoked by the client. These callbacks are implicitly configured in the dispatcher and provide a way to update the component state. When a submit button is pressed, the decoder runs through each component and invokes the callback of each of them with the value provided in the user request.

**Decoder:** The decoder is a push decoder and parameter values are received as parameters to the callback methods (see below). To receive parameters on the server side, Seaside invokes a callback method for each input field or URL parameter.

**Generator:** The generator creates an XHTML view of the component tree and sends it to the client. Each web application class has a single `renderContentOn` method that is called to generate a new view of the application for the client. The programmer must manually dispatch between parts of his application from this single entry point.

**Store:** Each client has his own component tree that can be manipulated and the component state therefore roughly corresponds to a session state for each client. To share state between the clients, components may refer to shared objects in order to share state between clients. In this way, Seaside unifies part of the store with the generator and blurs the distinction between the view and the model of the application in that the components behave as both. However, the component structure also yields a strong cohesion between the stored data and the components that read and write this data.

Figure 2.17 presents and implementation of the hello web application for Seaside. In lieu of Smalltalk source code, the program is presented in a dump file of the class from the Pharo Smalltalk implementation. The class is instantiated each time a new client requests the URL that the class is mapped to. State related to the interaction each client is therefore stored as instance fields on the created objects and the `name` attribute of the class holds the value of the `name` input from the client. Similarly, the value of `Times` is held in a class variable to allow it to be shared for all clients. The class implements a `renderContentOn` method. This method is responsible for generating the output sent to the client and as noted above, this

hello.st

```
'From Pharo1.3 of 16 June 2011 [Latest update: #13327]'!
WAComponent subclass: #WAHello
  instanceVariableNames: 'name'
  classVariableNames: 'Times'
  poolDictionaries: ''
  category: 'WAHello'!

!WAHello methodsFor: 'ui' stamp: 'MathiasSchwarz'!
hello: html
      html text: 'Please enter your name:'.
  html text: name.
  html break.
  html textInput
    value: '';
    callback: [ :value |  name := value.  Times := Times + 1 ].
    html break.
  html submitButton: 'Submit'.! !

!WAHello methodsFor: 'ui' stamp: 'MathiasSchwarz'!
renderContentOn: html
Times isNil ifTrue: [Times := 0].
html form: [
  name isNil
    ifTrue: [
      self hello: html.
    ]
    ifFalse: [
      self sayHello: html.
    ]
  ]
! !

!WAHello methodsFor: 'ui' stamp: 'MathiasSchwarz'!
sayHello: html
  html text: 'Hello ', name, '!!'.
  html break.
  html text: 'Printed ',Times printString, ' names so far'.! !
```

Figure 2.17: A dump file for a Seaside class that implements the hello web application.

is done programmatically in Seaside. Depending on whether the `name` value has been set, the `renderContentOn` method will either call the `hello` method or the `sayHello` method to show the input form or the name output respectively. The `renderContentOn` method is re-run for each request to the Seaside application and serves only for rendering the current state of the application, not for updating the that state.

### Safety in Seaside

Seaside is responsible for generating HTML from the component tree and can therefore guarantee that the HTML is well-formed. Seaside does not claim any guarantees about the validity of the HTML output with respect to a DTD. To some degree, the components abstract away the HTML generation, and this abstraction allows the Seaside framework to prohibit certain types of invalid HTML. For example, the `break` message to `html` allows the programmer to generate a <*br*> tag and this `break` message does not allow the programmer to specify any children for the <*br*>

tag. However, in general is possible to combine components in ways that result in invalid HTML being generated, so validity is an issue for Seaside web applications too.

By generating output HTML through message passing to the `html` object, it becomes possible for Seaside to escape all strings before inserting them in the output document. While this means that the programmer has little direct control over the generated HTML, it also ensures that Seaside applications are not prone to XSS attacks. The use of callbacks also eliminates the need for hidden fields in forms, since variables containing such data may simply be closed over by the callback rather than passed through the form. This means that client-state manipulation vulnerabilities are less of a concern for Seaside applications. Seaside provides no built-in system for data persistence but many solutions, including SQL database drivers, are available. The issue of SQL injections is therefore not addressed by the Seaside framework.

### 2.4.7 Hop

Hop is a web application system by Serrano et al [77]. Hop includes a Scheme inspired, functional DSL for programming web applications as well as a web application framework. Hop code is compiled to a server side application as well as a client side (JavaScript) application which communicate through AJAX requests. The programmer explicitly marks the part of the code that he expect to run on the client side by marking s-expressions with `~` and explicitly defines the AJAX requests sent from the client to the server.

The work on Hop also includes a static type system and a formal semantics for the Hop language [5]. The latter can be used for ensuring that the compiler to the server side and the compiler to the client side code behave similarly. However, no work about correctness of the generated output with respect to a DTD has yet been published.

**Dispatcher:** The dispatcher is explicitly defined: Hop applications expose functions (called *service functions*) that may be invoked by the dispatcher to generate a response to a client. This is done through the Hop API operation `define-service`.

**Decoder:** The decoder makes the request parameters available as formal parameters to the service functions. Hop furthermore allows anonymous service functions that can be bound to e.g. form actions so that they will may be invoked as a consequence of posting a form to the server.

**Generator:** Functions named <`html`>, <`body`> etc. resemble the XHTML syntax and provide a mean of generating HTML elements. The programmer can define additional such functions to implement additional tags similar to the tag mechanism of JSP. When one of these tag functions is applied, a node value is generated. Each function can be applied to a variable number of arguments, each of which become a child node of the node that the function generates. The document is therefore a first class, immutable value in the language.

**Store:** Hop has no features for storing data other than what is provided by the Hop language itself and it has no data scope equivalent to the session scope of other frameworks. Hop does however provide ways of setting cookies on the client. As in Scheme, functions may hold mutable value and have side effects. Furthermore Hop applications can interface with a database for storing persistent data.

Figure 2.18 shows a Hop version of the Hello application. The application defines two services, `hello` and `sayHello` which show the input form and prints back to the user respectively. A `times` variable is declared for storing the `times` value. Since this variable is in declared in the scope outside the service definitions, it is shared between all requests to the `sayHello` service and behaves as application state. The

hello.hop

```
(define times 0)
(define-service (hello)
  (<HTML>
    (<BODY>
      (<FORM> :action "sayHello"
        (<INPUT> :type "text" :name "name")
        (<BR>)
        (<INPUT> :type "submit")
))))

(define-service (sayHello #!key name)
  (<HTML>
    (<BODY>
      "Hello " name "!"
      (<BR>)
      (set! times (+ times 1))
      "Printed " times " names so far"
)))
```

Figure 2.18: The Hop version of the hello application.

service implementations generate the two pages in a straight-forward manner, and `sayHello` receives the `name` input string as a function parameter.

#### Safety in Hop web applications

The Hop programmer explicitly creates the tree structure of the result document and Hop is safe by default against cross-site scripting since all strings pass through generator functions that escape data correctly. Injection vulnerabilities are not addressed by the Hop framework and there are no features to help the programmer avoid client-state manipulation attacks.

The Hop generator guarantees that the output document has a proper tree structure and that it has a single root node. Hop application output is therefore always well-formed but there are no guarantees about the validity of output from Hop web applications.

## 2.5 The need for a new web framework and analysis for the existing ones

In the survey presented in this chapter, we have seen a set of web application frameworks and discussed examples points in the design space for web application frameworks. We have seen that they are not safe by default against common web application errors. In particular, none of them provide ways to ensure that the output is valid HTML and all but one are not safe by default against client-state manipulation attacks.

The only safe by default framework for client-state manipulation attacks is the Seaside framework. Seaside binds a component state to each client and client-state becomes part of this component state rather than being stored at the client side. The stateful Seaside model is vastly different from the other surveyed frameworks, and it has some important drawbacks: When the server stores state for each client, the number of simultaneous clients becomes limited by server resources. The URL is specific to the session rather than to parts of the application and consequently

bookmarks and links do not work in Seaside programs. The Seaside solution is therefore not directly applicable to other frameworks.

To help the programmer, we can assist in two different ways: Either, tools can help the programmer dynamically or statically track errors in the program or else the frameworks can be designed to make it impossible to unawarely write incorrect programs. In the following chapters, we will study both directions: First, we will study the design of the JWIG framework that is designed to prevent commons security errors through framework solutions. Second, we will study output and security analysis of the frameworks discussed in this chapter. We will focus on the two problems that we have identified as the most prevalent ones: HTML invalidity and client-state manipulation vulnerabilities.

# Chapter 3

# Designing a new web application framework

Section 2.5 argued for the need for a new web application framework. In this chapter, we will discuss the JWIG web application framework that has been developed as part of the PhD project and its template transformation system Xact. The chapter consists of a general introduction to the framework and discusses how JWIG solves the common problems of the web application frameworks presented in the previous chapter. JWIG follows a strict safe by default design so that JWIG applications are not vulnerable to cross-site scripting attacks or insecure object references. Owing to the Hibernate framework, it is also to a large degree safe against command injection attacks.

Furthermore, JWIG application can be statically analyzed with high precision to ensure that the generated output conforms to the DTD descriptions of the family of XHTML languages. Additionally, the analysis suite can guarantee consistency in links between parts of the application and generate high precision page graphs for web applications.

## 3.1 Overview of JWIG

JWIG is a Java web application framework. JWIG includes a syntactic extension to Java for writing XML documents, and a simple, reflection-based API for implementing web applications. JWIG *web applications* are composed of *web methods* that are configured and invoked reflectively by the JWIG runtime system. Multiple web applications may be grouped into a single *web site* and multiple web applications may be bound to the same URL. This allows modularization of the code and allows the programmer to separate concerns in his program.

In overview, the components of JWIG are designed as follows:

**Dispatcher:** The JWIG dispatcher is an implicitly configured dispatcher which can be customized with explicit annotations if desired. The URLs of web applications and web methods can only overlap if they are explicitly configured to do so.

**Decoder:** The decoder provides request parameters through parameters to the web method and is also responsible for converting parameters to the desired data types.

**Generator:** JWIG uses the Xact XML system as generator. Xact documents are created by combining immutable XML templates that are treated as first class values in the program. The Xact system is discussed in details in Section 3.2.1.

**Store:** As store, JWIG allows the programmer to store data that is reachable through fields of the web application class and provides a database API for storing persistent data. JWIG offers no representation of the request that is available to the

SayHello.jwig

```
1  import dk.brics.jwig.*;
2  import dk.brics.xact.XML;
3
4  public class SayHelloJWIG extends WebApp {
5      private int times;
6
7      public XML hello(String name) {
8          XML form = [[
9          <html>
10           <head>
11             <title>Enter your name</title>
12           </head>
13           <body>
14             Please enter your name:
15             <form action=[ACTION] method="post">
16               <input type="text" name="name"/>
17               <input type="submit" value="Submit" />
18             </form>
19           </body>
20         </html>
21         ]];
22         return form.plug("ACTION", new SubmitHandler() {
23           public XML run(String name) {
24             XML greeting = [[
25                   <html>
26                     <head>
27                       <title>Hello <[NAME]> </title>
28                     </head>
29                     <body>
30                       Hello <[NAME]>!<br/>
31                       <br/>
32                       Printed <{++times}> names so far.
33                     </body>
34                   </html>]];
35             return greeting.plug("NAME", name);
36           }
37         });
38     }
39 }
```

Figure 3.1: JWIG version of the SayHello program from Section 2.4

programmer and contrary to Servlets, JSP, Struts, and JSF it has no values attached to the request scope. Instead, JWIG has a *response scope* where the programmer can store data related to generating the current response. This response scope is not shared between web applications, even if more than one web application generates a response for the HTTP request. JWIG has a session storage to allow data to be coupled with individual clients. Contrary to most other frameworks, session data is explicitly passed between web methods in JWIG. Both the request scope and the session scope ensure high cohesion between the code that represents the data and the web methods that use the data. As a consequence of explicit session representation, the same client may also participate in multiple simultaneous sessions. We will discuss JWIG sessions in more detail in Section 3.2.3.

Figure 3.1 shows an implementation of the Hello web application in JWIG. The web application contains a single web method `hello` that can be invoked by the client by sending a GET request to the relative URL `hello`. The web method generates an HTML page containing a form where the client can input his name.

The program furthermore contains a *submit handler* that is *plugged* into the `action` attribute of the form in line 22. The `run` method of the submit handler receives the input from the client and generates a response based on the name parameter. Similarly to the other Hello implementations, in line 32 the value of `times` is incremented and inserted into the response document. In JWIG, this value is stored in a field of the web application class.

## 3.2 Introduction to JWIG application programming

In this section, we will present the central components of the JWIG web application framework. The components will be studied in enough detail to serve as a basis for the discussions later in the chapter but it will leave out details that are relevant for programmers of JWIG web applications. This section builds on excerpts from the JWIG Users manual [62] and the reader is advised to read that user manual for a thorough presentation of all the components of JWIG.

### 3.2.1 Xact syntax

Web methods typically produce XHTML output, which is sent to the client to be shown in a browser window. A key constituent of JWIG is the Xact XML transformation language [10, 41]. This section provides a short introduction to Xact.

Xact is a general tool for transforming XML data. We here focus on its capabilities for constructing XHTML documents dynamically.

An *XML template* is a fragment of an XML document that may contain unspecified pieces called *gaps*. These gaps can be filled with strings or other templates to construct larger templates and complete XML (typically XHTML) documents. XML templates are represented in the JWIG program by immutable values of the `XML` type.

XML template constants are written in plain XML syntax enclosed in double square brackets. For example, the following line declares a variable to contain XML template values and initializes it to a simple constant value:

An XML variable and a simple template constant

```
1 XML hello = [[Hello <i>World</i>!]];
```

XML template constants must be well-formed, that is, the tags must be balanced and nest properly.

A *code gap* in an XML template is written using the syntax <{*code*}> where the content is a Java expression:

XML template with code gaps

```
1  String title = "JWIG";
2
3  XML getMessage() {
4    return [[Hello <b>World</b>]];
5  }
6
7  XML hello = [[
8    <html>
9      <head><title><{title}></title></head>
10     <body>
11       <{getMessage()}>
12     </body>
13   </html>
14 ]];
```

These gaps can be placed within text and tags, as if they were themselves tags. Gaps placed like this are called *template gaps*. When executed, the code in the code gaps is evaluated, and the resulting values are inserted in place of the gaps.

Code gaps can also be placed as the values of attributes. For example, `<a href={foo()}>` is an anchor start tag whose `href` attribute has a value obtained by invoking the method `foo`. Gaps placed like this are called *attribute gaps*.

A *named gap* is written using the syntax `<[name]>`. The gap name must be a legal Java identifier. Named gaps can also be placed inside tags, as the values of attributes.

For example, `<a href=[LINK]>` is an anchor start tag whose `href` attribute is a gap named `LINK`.

Named gaps are filled using the `plug` operation:

Building XML templates using `plug`

```
1 XML hello1 = [[<p align=[ALIGNMENT]>Hello <[WHAT]>!</p>]];
2 XML hello2 = hello1.plug("WHAT", [[<i><[THING]></i>]]);
3 XML hello3 = hello2.plug("THING", "World").plug("ALIGNMENT", "left");
```

After executing this code, the values of the variables will be:

```
hello1:  <p align=[ALIGNMENT]>Hello <[WHAT]>!</p>
hello2:  <p align=[ALIGNMENT]>Hello <i><[THING]></i>!</p>
hello3:  <p align="left">Hello <i>World</i>!</p>
```

As can be seen from the example, both strings and XML templates can be plugged into template gaps. However, only strings may be plugged into attribute gaps. When a string is plugged into a gap, characters that have a special meaning in XML (e.g. < and &) are automatically escaped.

The plug method allows any object to be plugged into a gap. XACT will convert the value into an XML fragment either by calling `ToXMLable` og `toString` on the object depending on the object type.

A common use of named gaps in XML templates is to avoid redundancy in dynamic XML construction, for example when many XHTML pages use the same overall structure:

XHTML wrapper

```
1  XML my_xhtml_wrapper = [[
2    <html>
3      <head><title><[TITLE]></title></head>
4      <body>
5        <h1><[TITLE]></h1>
6        <[BODY]>
7      </body>
8    </html>
9  ]];
10
11 XML page1 = my_xhtml_wrapper.plug("TITLE", "one")
12   .plug("BODY", [[This is <b>one page</b>]]);
13
14 XML page2 = my_xhtml_wrapper.plug("TITLE", "two")
15   .plug("BODY", [[This is <b>another page</b>]]);
```

Instead of inlining a template in a JWIG program, an XML template constant can be placed in an external file and loaded into the program as a resource.

### 3.2.2 Web methods

Each public method in the web app class is a *web method* that can be accessed via a HTTP GET request, as in the Hello World example above. Since web methods

Sessions

```
1  class UserState extends Session {
2    List<Item> shopping_cart; // contains user state
3  }
4
5  public URL entry() {
6    UserState s = new UserState();
7    ... // initialize the new session
8    return makeURL("shopping", s);
9  }
10
11 public XML shopping(UserState s) {
12   ... // s contains the session state
13 }
```

Figure 3.2: Use of session data in web methods

are associated with GET requests, they should be *safe* and *idempotent*, as required by the HTTP specification [21]. Web methods determine what is returned to the client. Depending on the return type of the web method, the behavior of the web method may be to send an XHTML page to the client, to redirect a client to another page or to send plain text.

The URLs of web methods depend only on the application code and remain unchanged unless the code changes. This makes it possible to bookmark links to specific parts of the web application.

### 3.2.3 Session data

The class `Session` makes it easy to encapsulate session state and transparently pass the data to web methods that are later invoked by the client.

A subclass of `Session` contains the session data. Session objects can be passed to other web methods through parameter passing. JWIG automatically provides a session ID to each session object. The example in Figure 3.2 illustrates a typical use of sessions data: The `entry` methods creates a piece of session data and possibly initializes the session with client specific state. This session object is then added explicitly to the URL generated for the `shopping` web method and the client is redirected to this web methods with the session.

Contrary to PHP, Servlets, and JSP that use string-to-object maps, the session data can then be stored in the fields of the object in a type safe manner. Such a type safe mechanism is also present in Struts and JSF but there the session data is not passed explicitly.

With explicit session passing, there is a strong cohesion between the session data and the code that reads the session data: The data cannot be read elsewhere in the web application and it is directly visible in the program code how session data flows from one web method to another.

### 3.2.4 Handlers

Forms constitute the main mechanism for obtaining user input in web applications. With most frameworks, the code that generates the form is separate from the code that reacts on the input being submitted, and these pieces of code are connected only indirectly via the URLs being generated and the configuration mapping from

URLs to code. This violates the principle of high cohesion and low coupling, and the flow of control and data becomes unclear.

Our solution is to extend the architecture with *event handlers* as a general mechanism for reacting to user input, so the handler mechanism also works for other kinds of DOM events. The technique is related to the use of action callbacks in Seaside (see Section 2.4.6).

Forms are created by building XML documents that contains a `<form>` tag with relevant input fields. A specialized event handler, called a *submit handler*, is plugged into the `action` attribute. The submit handler contains a method that can react when the form is submitted and read the form input data. (An example is shown in Figure 3.1.) Event handlers are typically anonymous classes located within the web methods.

The consequence of this structure is a high degree of code cohesion. Also, it becomes possible for static analysis tools to verify consistency between the input fields occurring in the form and the code that receives the input field data.

### 3.2.5 Client/server communication

Clients send messages through handlers (see section 3.2.4). In general, such messages may cause side effects in the web application.

The response of a web method is typically a view of some data from the underlying database. When the data changes, the view should ideally be updated automatically while only affecting the relevant parts of the pages to avoid interfering with form data being entered by the user. With many frameworks, this is a laborious task that requires many lines of code and insight into the technical details of JavaScript and AJAX, so many web application programmers settle with the primitive approach of requiring the user to manually reload the page to get updates.

We propose a simple solution that hides the technical details of the server-push techniques (aka. Comet) and integrates well with the Xact system. An *XML producer* is an object that can produce XML data when its `run` method is called. When the object is constructed, dependencies on the data model are registered according to the observer pattern. An XML producer can be inserted into an XHTML page such that whenever it is notified through its dependencies, `run` is automatically called and the resulting XML value is pushed to all clients viewing the page. All the technical details involving AJAX and Comet are hidden inside the framework, and it scales well to hundreds of concurrent connections.

Typically, the XML producer is created as an anonymous inner class within the web method that generates the XHTML page.

## 3.3 Relation to existing frameworks

We have already studied a variety of web application frameworks in Section 2.4. In this section we will study some of the similarities to these frameworks more closely.

### 3.3.1 Relation to the previous version of JWIG

The JWIG web application framework has a long history. The previous version of JWIG [11] originated as a Java version of the <bigwig> [6] web application framework. Similarly to the current version of JWIG, the previous JWIG framework had first-class, immutable templates and applications could be analyzed for output validity. An important difference was that the framework was *session oriented*. The control flow of an application was controlled by the control flow of a method, representing a session. Two new control flow statements were added to the Java

Language: the `view` statement that would pause execution of the method, send a document to the client and await client input before continuing execution and `exit` that would terminate the session and send a document to the client. An additional `receive` expression allows the programmer to read the value of a parameter from a request.

The session method made it relatively easy to encode a state machine into the flow of a web application and made it possible to store session state as local variables of the method. The approach, however, often results in tight coupling of the code that generates view and the code that receives user input. It furthermore makes it hard to write REST-like web applications and generate links to specific pages in the application.

The new JWIG framework tries to decouple views and actions (in JWIG called *web methods* and *handlers*) while maintaining the template system and some of the session features of the previous JWIG version.

### 3.3.2 The JWIG approach compared to MVC

With the MVC architecture, many frameworks try to guide the programmer towards applications that have low coupling between its parts. The design of JWIG aims for the same goal.

A classical MVC separates the model, view, and controller parts of the program into completely distinct files that may even be written in different programming languages. In for example JSF (see Section 2.4.4), the view is written in JSF's XML template language with no Java code involved. The template language has features for basic control flow but these features are used solely for controlling output generation, not for program logic.

In JWIG, the major distinction is between those methods that may have side effect and those that are not allowed to. With JWIG, XACT templates are combined using ordinary Java code as part of the flow of the program. The discipline of the programmer is guided by two properties of the JWIG framework:

1. As a result of the idempotency requirement imposed by the HTTP specification on GET requests [21], JWIG expects web methods to be idempotent and the result of the web method to be cachable.

2. JWIG will cache the return value (including handlers) of the web method. As a consequence, JWIG web methods must generate a self-contained, cachable value representing a view of the resource on the requested URL, and it is therefore very limited what program logic can be implemented in such a handler. As a result, web methods present a view of the data on the requested URL.

This yields an application structure that has some similarities with MVC application, and this section discusses how JWIG's web methods and handlers relate to the view and controller portions of MVC frameworks respectively.

In JWIG, the `Handler` objects have a role that is similar to the controller in an MVC framework. Handlers are invoked as a result of a POST request when submitting a form and they may therefore have side effects. Contrary to controllers in classical MVC frameworks, JWIG handlers may generate a new piece of XML and thus they may also behave as the view component. Such a handler generated view cannot contain further handlers though. The primary use of such a view is therefore to confirm the outcome of an operation that has a side effect.

The outcome of the JWIG application structure is a high cohesion between the code that generates the view of a resource and the code that changes that resource. As a result, both the web methods and the handlers are part of the same file,

though separated into two different methods. This typical structure is illustrated in Figure 3.1 where a web method generates a form and installs a handler on it. The handler will print back to the client in the same way as the programs shown in Section 2.4.

### 3.3.3 Relation between submit handlers and Seaside callbacks

In the Seaside programming model, input fields and links were paired with blocks that were called upon user interaction. JWIG SubmitHandlers are similar in that they are set as callback in the HTML documents and their `run` method is invoked upon form submission.

A key difference between the two is that the closure object in Seaside is specific to the client viewing the Seaside, whereas in JWIG the `SubmitHandler` object is shared among all clients that view. The JWIG programmer therefore cannot reliably use the submit handler to store any state related to the interaction with each client.

The purpose of JWIG submit handlers is to bind the code together, whereas the purpose of Seaside callbacks is to bind the client's interactions together with the component state.

## 3.4 Static analysis of JWIG applications

JWIG provides an analysis suite to analyze JWIG web applications for typical programming errors. We can divide web application errors into two categories:

1. Errors that are typical web applications regardless of the web application frameworks. Such errors include XHTML validity errors and dead links to other parts of the application.

2. Errors that are specific to but typical for applications of the concrete framework. While such errors may be a result of a design error in the framework, there might also be good reasons that such errors are typical for a specific framework. For example in JWIG, the analysis suite can check that parameters to a call to `makeURL` are type correct so that a later call to the target web method will not fail with a type error. While one could imagine that such guarantees could be given directly by a language, Java provides no useful features (such as first class functions) for this task and the type check is left to the JWIG analyzer instead.

We will discuss solutions to both categories of errors for JWIG applications in this section.

### 3.4.1 XHTML validation of output construction

The JWIG analysis suite guarantees that all output from web methods is well-formed and valid XHTML. The analyzer can exploit the fact that Xact templates are well-formed XML fragments to define a type system that is based on schema types from the schema description. Each `XML` variable in the program is assigned a schema type.

In Xact, a type is either a concrete schema type or a schema type provided gaps are plugged with `XML` values of specific Xact types. Web methods that return XHTML are expected to return XML of the type `HTML`, that is the DTD schema type for the <*html*> element of one of the XHTML DTD specifications.

The validity analysis builds directly on the Xact analysis suite [39], and we will not discuss it in further detail here.

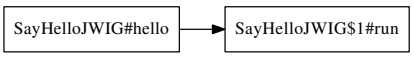


Figure 3.3: The page graph for the JWIG Hello application.

### 3.4.2 Web application page graphs

The JWIG analysis suite generates web application *page graphs* where nodes are web methods, handlers and edges are links and form submissions. The graph generation is done in three steps:

- First, the suite identifies the web methods and handlers for the application. This is done through the same introspection that is used when initializing the JWIG system at runtime.
- Second the targets of calls to `makeURL` are determined. The target of `makeURL` depends on the context in which `makeURL` is called. In order to match a call to `makeURL` with possible target web methods, the analysis must therefore create a call graph to find all possible web methods that may result in the call to the `makeURL` method.
- Third, the suite identifies all handlers that are instantiated by the web method and marks them as possible targets. Since web methods may call other methods that instantiate handlers, we use the same call graph that we did for `makeURL`.

Figure 3.3 shows the page graph for our simple Hello application in JWIG. There is a single web method `SayHelloJWIG#hello`, a submit handler `SayHello$1#run` and an edge between them corresponding to the possibility of a form submission.

The page graph can serve both as an illustration for documenting the system as well as provide a basis for further analysis of the web application.

### 3.4.3 Link consistency

The JWIG analysis suite checks that all calls to `makeURL` have a target and that the parameter types are correct. While `makeURL` guarantees to return a valid URL to a web method in the application, it may throw an exception if the method does not exist or if the parameters are of an incorrect type. The purpose of the type check is to ensure that `makeURL` will never fail at runtime. Using the page graph, the analysis suite determines whether each call to `makeURL` has a valid target. Afterwards, the analyzer determines the type of each of the remaining arguments to `makeURL`. These arguments are later passed to the web method as method parameters and they must therefore be subtypes of the parameter types of the web method. In the Java program they are provided to `makeURL` as a varargs list. The analysis suite implements an intra-procedural, flow-sensitive analysis that is based on control flow graphs generated by Soot [84]. This analysis determines a least common supertype for each of the entries in the varargs list of the `makeURL` call. These types are then compared to the formal parameters of the matched web methods through standard subtype check, point-wise per argument.

### 3.4.4 Filter coverage

Filters provide means for the programmer to collect data about the client or to ensure that the client is properly authenticated before viewing a page. To aid the

programmer to ensure that a specific filter always applies for a specific set of web methods, the JWIG analysis suite generates two sets of filters for each web method: filters that may apply before the web method is invoked and filters that are always applied before the web method is invoked. The two sets are determined by inspecting the `URLPattern` annotations on the web methods and filters (or using the default URL mapping if no annotation exists). Such a URL pattern is easily translated to a regular language. Using well-known techniques, the suite can determine whether the annotation language on the filter is included in the language of the web method (that is, the filter will always apply before invoking the web method) or whether the two languages have a non-empty overlap (that is, the filter may apply before invoking the web method).

### 3.4.5 Parameter names

Elbaum et al. found in an empirical study in 2005 that mismatching parameter names in web applications [19] are a common and serious problem. The `makeURL` feature of JWIG ensures that that generated URLs always match the expected parameter names for the target web method. The JWIG frameworks, however, does not directly ensure that parameter names match between form input fields and the recipient `SubmitHandler`. The analysis suite can determine whether parameter names to a `SubmitHandler` are included in the form that is submitted to the handler. The XML graph [58] generated by the XACT analysis allows the analysis suite to determine all possible parameter names for each form generated by a web method. Using the control flow graph that is generated by Soot, the analysis suite can then determine a set of possible types of `SubmitHandler` objects for each of the generated forms. The analysis suite can then compare the (constant) names of the parameters of the `run` method in each `SubmitHandler` to the names of the form input fields and determine whether the parameter name is a possible name for an input field in the form.

Halfond and Orso presented a similar analysis for Servlet and JSP web applications in 2008 [27]. In their work, the output of Servlet and JSP pages is approximated through string analysis of the output stream printed to by the application. From the string approximation they extract possible parameters to sent to other pages in the application. Another component of the analysis analyzes pages to determine which parameters they may read, that is identifies the interface of the page. In JWIG, the interface can be determined directly by inspecting the web method or the `run` method signature and reading the parameter names from the class file. As a consequence, the analysis for JWIG is much simpler than the corresponding parameter matching for Servlets and JSP.

## 3.5 Evaluation of the JWIG web application framework

We evaluate the JWIG web application frameworks in two ways: 1) We compare the features of JWIG to the safe by default short comings that we identified in Section 2.5. 2) We present af case study of a real web application developed using the JWIG framework.

We seek to answer the following research questions for JWIG:

Q1: To what degree is the JWIG framework safe by default against HTML invalidity and common security vulnerabilities?

Q2: Is JWIG useful for developing large web applications?

### 3.5.1 Safety of JWIG applications

To answer Q1, we review the design of the JWIG components. The template language of JWIG makes it impossible to insert unsanitized data into an XACT template. The `plug` operation escapes the inserted string to make the framework safe by default against cross-site scripting. In the rare occasion where strings that are based on client input need to be parsed as XML, JWIG allows the programmer to parse a string as an XML fragment.

JWIG integrates the Hibernate framework into `makeURL` and the decoder. This means that the programmer never converts between objects and direct object references in the code. JWIG will automatically serialize object references and store them in URL parameters and hidden fields and ensure that the references remain secure from one request to another. This makes JWIG applications safe by default against insecure direct object reference vulnerabilities. In Chapter 4, we will discuss analysis of client-state manipulation attacks and how JWIG applications can be analyzed to detect all such vulnerabilities. Furthermore, the use of Hibernate makes SQL injection attacks unlikely.

JWIG programs are guaranteed to always output well-formed XML. The compiler itself does not guarantee that the output is also valid. However, the JWIG analysis suite (discussed in Section 3.4) is able to verify output validity for JWIG applications, so if this suite is counted as part of the framework, JWIG applications are also safe by default against generation of invalid XHTML.

### 3.5.2 Case study: CourseAdmin

To answer Q2 and to test JWIG for programming applications beyond toy examples, we have used JWIG to develop the application CourseAdmin. The initial versions of the CourseAdmin application was developed by the authors of JWIG but the application has since been expanded by various other programmers. CourseAdmin is a large web application (more than 50,000 lines of code) for handling course administration at our department.

CourseAdmin is structured as five distinct web applications with a common model layer: the public web pages of classes and students, the course configuration application for course staff, an application for students to view their status and upload assignments, a forum application for course related discussions, and finally a configuration application for creating new courses.

High cohesion exists in this architecture because the web applications share common web methods and the division of the system into multiple web applications results in low coupling between unrelated program parts. This indicates that the patterns presented in the small example programs in this paper can be used to build much larger systems.

The main XML templates are located in separate files, and layout is controlled using CSS, which makes the page design consistent and easy to modify.

The experience with CourseAdmin has shown that extension of the application is easily done by adding new web methods and that the new web methods never break the functionality of the existing ones in practice. We conclude that JWIG is useful for developing large applications.

# Chapter 4

# Static analysis for existing frameworks

In this chapter, we will discuss output and security analysis of web applications. The techniques that are presented in this chapter will focus on frameworks with generators that represent the output implicitly and generate output by appending to a stream. As previously discussed, this includes widely used frameworks like PHP, Servlets, JSP, and Struts. We will argue that output analysis is useful to avoid HTML validity problems in web applications. Furthermore, the result of this analysis is also useful to find client-state manipulation vulnerabilities.

## 4.1 Approximating web application output

The analyses that are discussed in this chapter require *output analysis* of web applications. In the following sections, an output analysis is an analysis that for each page in a web application yields a static approximation of the possible output sent from the page as a response to an HTTP request. In essence the output stream analysis approximates the effect that a page has on the generator. In this work, the approximation will be presented as a context-free grammar [22]. The approximation is conservative, meaning that if a page can output a document then this document can also be generated by the context-free grammar describing the page.

Our approach for this approximation builds on earlier work by Christensen et al [12]. Their work demonstrated a technique for approximating the possible values of string expressions in Java programs as regular languages. This work involves an intermediate step where the approximation is on the form of a context-free grammar, and the approach is similar to what we do in this work. In this section we will formalize the output analysis, something that was not done in the previous work.

### 4.1.1 Output-stream flow graphs

An output-stream flow graph (OSFG) is a representation of a program that abstracts away everything that is not directly relevant for generating output to the output stream. The full description is not given in the papers, so we will study output-stream flow graphs in this section. OSFGs are formally defined as follows.

**Definition 4.1.** *An* output-stream flow graph $F$ *is a directed graph given as a tuple* $(I, E, C, L)$*:*

- $I$ *is a finite set of nodes, divided into four disjoint subsets:*
  - $I_{\mathsf{append}}$ *representing instructions that append strings to the output stream,*

  - $I_{invoke}$ *corresponding to method calls,*
  - $I_{return}$ *corresponding to method returns, and*
  - $I_{nop}$ *corresponding to operations that have no effect on the output.*
- $E \subseteq (I_{append} \cup I_{invoke}) \times I$ *is a set of* intra-procedural *edges,*
- $C \subseteq I_{invoke} \times I$ *is a set of* call *edges, and*
- $L : I_{append} \to \mathcal{R}$ *provides a regular string language (represented by a regular expression or a finite-state automaton) for every append node.*

Intuitively, the nodes correspond to primitive instructions in the program that is analyzed, and edges correspond to control flow between those instructions. As alphabet $\Sigma$ for the string languages, we use Unicode [81].

Output-stream flow graphs are abstract machines, and we define their behavior by an operational semantics:

**Definition 4.2.** *A* configuration *for $F$ is a triple $\rho = (\sigma, x, n)$ where $\sigma \in I^*$ is a stack of nodes (a call stack), $x \in \Sigma^*$ is a string (contents of the output stream), and $n \in I$ is a node (a program counter). The* execution step relation $\triangleright$ *on configurations is defined by the following rules:*

$$(\sigma, x, n) \triangleright (\sigma, xy, m) \text{ if } n \in I_{append},\ (n, m) \in E,\ \text{and } y \in L(n)$$

$$(\sigma, x, n) \triangleright (\sigma m, x, p) \text{ if } n \in I_{invoke},\ (n, m) \in E,\ \text{and } (n, p) \in C$$

$$(\sigma m, x, n) \triangleright (\sigma, x, m) \text{ if } n \in I_{return}$$

$$(\sigma, x, n) \triangleright (\sigma, x, m) \text{ if } n \in I_{nop},\ (n, m) \in E$$

*The reflexive transitive closure of $\triangleright$ is denoted $\triangleright^*$. The* language *of a node $n_0 \in I$, denoted $\mathcal{L}_F(n_0)$, is*

$$\mathcal{L}_F(n_0) = \{x \mid \exists n \in I_{return} : (\epsilon, \epsilon, n_0) \triangleright^* (\epsilon, x, n)\}.$$

An append node abstractly writes to the output stream and then continues execution nondeterministically at a successor node, an invoke node pushes a successor to the stack and then enters one of its target methods, and a return node exits the current method. Thus, the intention is that $\mathcal{L}_F(n_0)$ should contain a string $x$ if $x$ may appear as output when the program is executed starting from the instruction corresponding to $n_0$ with an empty stack and ending at a return node with an empty stack. In the next section, we describe a translation from Java Servlets and JSP code into output-stream flow graphs that meet this goal.

### 4.1.2 From Java Servlets and JSP to output-stream flow graphs

The translation from program files to an output-stream flow graph requires a representation of the control flow of the analyzed program. The control flow graph of the Servlet/JSP page can first be created using Soot [84].

The translation from CFG to OSFG requires a regular string language for each variable that references a string to be sent to the output. We use the Java String Analyzer [12] for obtaining such a regular language for each value.

The CFG is abstracted to an OSFG by replacing all nodes that output data with an append node and a regular string language corresponding to the values that can be printed. All nodes that call methods or other servlets are replaced with an invocation node and flow graph return nodes are replaced with a corresponding return node in the OSFG. All other nodes in the Soot flow graph are replaced with $I_{\mathsf{nop}}$ nodes that can afterwards be removed without changing the language the OSFG.

### 4.1.3 From output-stream flow graphs to context-free grammars

The next step is to translate the OSFG into a context-free grammar. We define context-free grammars as follows:

**Definition 4.3.** *A* context-free grammar *$G$ is a triple $(V, \Sigma, P)$ where*

- *$V$ is a set of nonterminals,*
- *$\Sigma$ is the terminal alphabet where $V \cap \Sigma = \emptyset$, and*
- *$P$ is a finite set of productions of the form $A \to \theta$ where $A \in V$ and $\theta \in U^*$ for $U = V \cup \Sigma$.*

We write $\alpha A \omega \Rightarrow \alpha\theta\omega$ when $A \to \theta \in P$ and $\alpha, \omega \in U^*$ and $\Rightarrow^+$ *and* $\Rightarrow^*$ are respectively the transitive closure and the reflexive transitive closure of $\Rightarrow$.

Our variant of context-free grammars has no fixed start nonterminal. Instead, we define the language of a nonterminal $A$ as $\mathcal{L}_G(A) = \{x \in \Sigma^* \mid A \Rightarrow^+ x\}$. The language of a terminal $a \in \Sigma$ is $\mathcal{L}_G(a) = \{a\}$, and the language of $\theta = u_1 \cdots u_k \in U^*$ is $\mathcal{L}_G(\theta) = \mathcal{L}_G(u_1) \cdots \mathcal{L}_G(u_k)$. We sometimes omit the subscript $G$ when it can be inferred from the context. We use the Unicode alphabet as $\Sigma$.

Although OSFGs have an operational flavor and context-free grammars are a declarative formalism, the step from the former to the latter is simple. Each node in the output-stream flow graph corresponds to a nonterminal in the grammar and the context-free grammar corresponds to the least solution to the following constraints:

- for each $n \in N_{\mathsf{append}}$ and $(n, m) \in E$, add a production $n \to r_n\ m$ to $P$ where the symbol $r_n$ denotes $L(n)$,
- for each $n \in N_{\mathsf{invoke}}$, $(n, m) \in E$ and $(n, p) \in C$, add a production $n \to p\ m$ to $P$, and
- for each $n \in N_{\mathsf{return}}$, add a production $n \to \epsilon$ to $P$.

This context free grammar approximates all possible output of the web application. We can describe the language of a single page in the web application as the language $\mathcal{L}(n)$ of the entry node $n$ of the page. We will use these page languages for output analysis and security analysis in the following sections.

## 4.2 HTML validation in WARLord

As discussed in Section 2.3.1 HTML validity plays an important role in the communication between servers and clients of web applications. It is therefore important to be able to verify validity of generated HTML in existing web application frameworks. The goal of the analysis presented in this section is to determine whether all strings that can be generated by a context-free grammar are valid HTML.

We want this analysis to be *sound*, in the sense that whenever it claims that the given program has this property that is in fact the case, *precise* meaning that it does not overwhelm the user with spurious warnings about potential invalidity problems, and *efficient* such that it can analyze non-trivial applications using modest time and space resources. Furthermore, all warning messages being produced must be *useful* toward guiding the programmer to the source of the potential errors.

To perform the validity analysis, we must first determine which tag sequences may be generated by the context-free grammar. Afterwards, we can determine whether the sequences respect the DTD description of the HTML language.

$$\delta(c,\rho) = \begin{cases} c & if\ \rho \notin S \text{ or } c = \bot \\ \text{tag} & if\ c = \text{content} \wedge \rho = \underline{<} \\ \text{tag} & if\ c = \text{attrval1} \wedge \rho = \underline{"} \\ \text{tag} & if\ c = \text{attrval2} \wedge \rho = \underline{'} \\ \text{ws} & if\ c \in \{\text{tag}, \text{ws}\} \wedge \rho \text{ is whitespace} \\ \text{attribute} & if\ c = \text{ws} \wedge \rho \text{ is not whitespace} \\ \text{attrval1} & if\ c = \text{attribute} \wedge \rho = \underline{="} \\ \text{attrval2} & if\ c = \text{attribute} \wedge \rho = \underline{='} \\ \text{content} & if\ c = \text{tag} \wedge \rho \in \{\ \underline{>}\ ,\ \underline{/>}\ \} \\ \text{content} & if\ c = \text{endtag} \wedge \rho = \underline{>} \\ \text{endtag} & if\ c = \text{content} \wedge \rho = \underline{</} \\ \text{error} & \text{otherwise} \end{cases}$$

Figure 4.1: Definition of the $\delta$ function used to assign contexts to all terminals in the context free grammar.

### 4.2.1 Annotating context-free grammars with contexts

We assign *contexts* to each terminal in the context-free grammar. In essence, contexts are annotations on the terminals describing what lexical construct (e.g. an attribute value or a tag name) that the terminal is part of in the generated output.

We extend the alphabet of the context-free grammar with three additional tokens $\underline{='}$, $\underline{</}$, and $\underline{="}$. We transform the context-free grammar so that whenever the original grammar would generate the sequence of tokens $\underline{=}\ \underline{'}$, it will instead generate $\underline{='}$. We transform the sequences $\underline{<}\ \underline{/}$ and $\underline{=}\ \underline{"}$ in a similar fashion.

For each terminal in the grammar, we want to assign a set of possible contexts. This set of contexts can contain the following possible values: *content*, *tag*, *endtag*, *attribute*, *ws*, *attrval1* and *attrval2* representing the syntactic categories of HTML and XML, and *error* denoting terminals in contexts that part of malformed output. We define a lattice, $\mathcal{C}$, as the power set of these elements.

We define a function $\delta : \mathcal{C} \times \Sigma \to \mathcal{C}$ shown in Figure 4.1. Møller and Kirkegaard showed [40] how a constraint system for a function like the one above can be defined such that a least solution always exists and all terminals in the grammar are assigned a set of possible contexts.

If no terminal contains the context *error* then we can transform the grammar into a context-free grammar where the alphabet consists of HTML start and end tags. We will use this grammar as a basis for the HTML validation algorithm presented in the following section.

### 4.2.2 HTML validation of an annotated grammar

This section gives an overview of our technique for checking validity of HTML that is generated by context free grammars. The work is presented in details in the paper "HTML Validation of Context-free Languages" [60] that is included as Chapter 7 of this dissertation. The goal of this section is therefore primarily to give an overview of the technique and relate to the current state of the art.

#### Validity of HTML documents

Parsing and validation constitute a single process for HTML documents. Contrary to documents written in XML languages, HTML documents cannot be parsed with-

out checking validity of nested document elements.

The HTML languages are described using the SGML DTD language [24]. In essence, for each element name the DTD describes a regular language of possible sequences of child elements. Such a set of sequences is called a *content model* of the element.

There are two major challenges when validating HTML documents compared to XHTML documents that use the less feature rich XML DTD language to describe validity:

- HTML allows certain tags to be omitted, for example the start tags <*body*> and <*tbody*> and the end tags </*html*> and </*p*>. This means that the document must be parsed before start and end tags can be matched. In XHTML there is always a match for each start and end tag in the document.

- HTML have tag inclusions and exclusions (collectively called exceptions) that allow the DTD to declare whether specific elements are allowed as descendents of the declared node, overruling the content models of the element declarations. While this makes it possible to for example forbid nesting of <*a*> elements at any level, it also means that the parser needs more contextual knowledge to verify document validity.

Both the SGML DTD language and the HTML specific features are discussed in detail in Chapter 7.

Previous solutions have not been able to address these two features to a satisfying degree. Our technique has two components: 1) A HTML parser that can validate a substring of an HTML document, given an initial parsing context for the string and 2) a technique for applying the substring parser to a context-free grammar to parse any terminal in the context-free grammar in any parsing context that it may appear in.

### Parsing substrings of HTML Documents

The first component of the analysis is a validity checking algorithm for substrings of HTML documents. The algorithm is based the Amsterdam parser [86] by Warmer and Egmond. The Amsterdam parser is a general parser that can parse any SGML language provided an SGML DTD description of the language. While the Amsterdam parser is not able to handle all SGML features, it is able to handle enough to support all existing versions of HTML and their solution is therefore sufficient for what we need.

HTML documents are parsed left-to-right with a context stack. For each point in the document, the stack contains 1) the open tags (that is the tags where a start tag has been seen but where no end tag has yet been encountered), 2) for each open element a pointer to the state of its content model and 3) information about content model exceptions. Given such a context stack, the parser can parse a sequence of start and end tags to yield a new context stack.

Section 7.3.1 of the paper gives pseudo code of the parsing algorithm and describes in detail how the algorithm works.

### Parsing context-free sets of documents

A main contribution of our work is to generalize the parsing algorithm for individual documents to work for sets of documents described by context-free grammars.

In essence, the parsing algorithm for document substrings is applied on each production in the context-free grammar. The algorithm parses the right-hand side of the production. If it reaches a reference to a nonterminal in the production, it

$A \rightarrow$ `<html><head><title></title></head>`$B$
$B \rightarrow$ `<body></body>`$C$
$B \rightarrow$ `<body>`$C$
$C \rightarrow$ `</body></html>`

Figure 4.2: The context-free grammar corresponding to the servlet in Figure 2.1.

| | $\mathcal{C}$ |
|---|---|
| $A$ | $(\mathsf{root}, q_{\mathsf{root}}) \mapsto \emptyset$ |
| $B$ | $(\mathsf{root}, q_{\mathsf{root}}) \cdot (\mathtt{html}, q) \mapsto \emptyset$ |
| $C$ | $(\mathsf{root}, q_{\mathsf{root}}) \cdot (\mathtt{html}, q_1) \cdot (\mathtt{body}, q_2) \mapsto \{(\mathsf{root}, q_3)\}$ |
| $C$ | $(\mathsf{root}, q_{\mathsf{root}}) \cdot (\mathtt{html}, q) \mapsto \lightning$ |

Figure 4.3: The incomplete set of context stacks that are generated for the context-free grammar in Figure 4.2. This figure leaves out the content model exceptions as they are not relevant to the error in question.

stores the parsing context. The algorithm parses each production in each stored context and continues until a fix-point is reached.

The generalization is based on the observation that for each element, DTD can only describe constraints for direct child elements. This bounds the number of parsing contexts in which each production may appear and allows the algorithm to abstract the parsing contexts to contexts of finite length. The analyzer is consequently able to reach a fix-point while still analyzing each production in every parsing context that it may appear in.

Consider the example in Figure 2.1. The corresponding context-free grammar of tags is shown in Figure 4.2. For readability this grammar has not been normalized as is done in Chapter 7 and the content model exceptions are omitted from the table since they are irrelevant to the error that is found. Figure 4.3 shows the context stacks that are generated by the analysis to determine HTML validity. The generated solution set ($\mathcal{C}$) is incomplete because the analysis finds an error ($\lightning$) in the grammar.

A thorough description of the parsing algorithm can be found in Section 7.4 where we also argue for the correctness of the approach.

### 4.2.3 On HTML5

Since we published our work, the HTML5 specification has largely been finished and browsers have begun implementing this standard. HTML5 has very recently been declared a *Candidate Recommendation* by the W3C and little is expected to be changed before HTML5 reaches status of *Recommendation* in 2016[1].

The HTML5 specification is a major departure from previous versions in that the parsing algorithm is not based on SGML and that there is no formal description (such as a DTD) for the language. It remains to be determined whether our work is applicable to HTML5. In practice, however, HTML5 is a relatively small addition to the previous HTML language: The change in the parser is backwards compatible with HTML 4 and furthermore, extra elements have been added to the language but only few of them are syntactically different from the elements that existed in HTML 4.

Compared to HTML 4, many of the content models have been simplified. For example, the <*table*> element may now be empty (it previously required at least

[1] `http://www.w3.org/TR/html5/`

one child element). W3C describe[2] HTML 5 as similar to HTML 4 transitional but with *presentational* elements and attributes removed and with structural elements such as <*section*> and <*article*> added. A major change is that the <*a*> element now inherits its content model from its parent in the document. Such a content model is not possible to describe in DTD.

It seems to be possible to describe all the HTML5 content models except the content model for <*a*> using the constructs of SGML DTD. This will require some effort, and the content model for <*a*> will require an addition to our technique.

### 4.2.4 Comparison to related work

Work has been done to make XML validation part of the type system of domain specific languages as well as general purpose languages. In a paper from 2005, Møller and Schwartzbach describe the design space of such solutions [57] but no such type systems exist for the SGML-based HTML languages.

However, surprisingly little work has been done on static analysis of HTML output correctness for web applications. Minamide and Tozawa present a technique for analyzing XML output of PHP programs. Their technique is able to validate this output against a schema description [56]. The schema description (DTD, and XML Schema, or Relax NG) is transformed to a regular hedge grammar. After checking whether the language generated by the PHP page is balanced, it is shown that it is decidable whether a balanced language is included in a regular hedge grammar.

With Nishiyama, Minamide later extended this technique to handle some of the features of the HTML family of languages [55, 66]. In this work the HTML DTD was translated into a regular hedge grammar and this made the validator able to handle omitted end tags. The authors tried to include content model exceptions and omitted start tags into the regular hedge grammar by inlining and expanding the grammar. It was, however, concluded that the runtime performance decreased too much as a result of the increased grammar size.

An alternative solution was proposed by Doh et al. in 2009. In their work on abstract parsing, LR(k) parsing was abstracted to work for context free grammars [16]. In the paper, they show how features the HTML DTD and the HTML parsing algorithm can be encoded as a an LR(0) grammar. In a later tech report they apply the parser to the same set of open-source benchmarks as done in our work [17]. Their findings are similar to ours, although the number of errors is not directly comparable.

In 2006, Kirkegaard and Møller presented a technique for validating context-free sets of XML documents with respect to a schema description [40]. The schema formalism for this work was XML graphs [58] rather than DTD. XML Graphs have an expressiveness equivalent to Relax NG while DTD is less expressive [65]. A translator allows DTD descriptions to be automatically rewritten to Relax NG so the technique could also verify that output was valid with respect to an XML DTD. The work builds on the work of Knuth on parenthesis languages [42]. The observation is that for XML languages neither start tags nor end tags are omissible. This allows us to consider $\underline{<}$ and $\underline{</}$ pairs in a parenthesis language and decide whether parentheses are always *balanced*. Knuth devices an algorithm for transforming a context-free grammar of balanced parenthesis strings into a context-free grammar on a form where the matching parentheses always appear in the same production in the grammar. Building on this algorithm, the technique brings the grammar on a form where start and end tags appear in the same productions and the papers shows how such balanced grammars can be translated to XML graphs. Finally, a language inclusion check between the XML graph that corresponds to the DTD and

[2] `http://www.w3.org/TR/html5-diff/`

the XML graph that corresponds to the application output can reveal whether the output is always valid according to the DTD.

WARLord implements the Kirkegaard and Møller technique. From small test examples it seems to perform well in practice. Only few existing programs, however, generate XML output through Servlets or JSP files, so a direct comparison to the work presented in this dissertation has not been possible on large scale open-source benchmarks.

Samimi et al. developed a technique for automatically correcting web applications that generate invalid HTML [73]. Their technique uses a string analysis to approximate the possible output of the web application and a constraint solver that is used for finding validity problems. The constraint solver can insert missing tags to make the program pass validation.

### 4.2.5 Evaluation

We have evaluated the approach on a series of open-source benchmarks and we found it to be precise in practice. The implementation is part of the WARLord analysis suite which also contains an implementation of the approach described in Section 4.2.4.

We evaluate the technique to find answers to the following research questions:

Q1: What is the typical analysis time for a Servlet/JSP page, and how is the analysis time affected by the absence or presence of validity errors?

Q2: What is the precision of the analysis in terms of false positives?

Q3: Are the warnings produced by the tool useful to locate the sources of the errors?

We found that the technique was fast in practice and yields few false positives. We were also able to correct the errors in the applications based on the output from the analysis tool. See Section 7.5 for the full evaluation, including discussion of the individual benchmarks.

## 4.3 Client-state manipulation vulnerability detection

In the paper "Automated Detection of Client-State Manipulation Vulnerabilities" [61] (see Chapter 8), we presented a static analysis for detecting client-state manipulation vulnerabilities in web applications. The analysis is presented in detail in the paper, so this section will only present an overview of the technique and discuss related work for detecting security problems.

The analysis uses parts of the output analysis presented in section 4.1. The analysis has been implemented as part of the WARLord tool to work for Servlets, JSP, and Struts applications.

### 4.3.1 Related security analysis techniques for web applications

This work is related to both output analysis and information flow analysis. Related work about output analysis was described in the previous section, so this section will focus on information flow analysis for web applications.

#### Information flow analysis to detect vulnerabilities

Information flow analysis is a main component in our technique for detecting client-state manipulation vulnerabilities. The goal of information flow analysis is to track

```
1 class A {
2     private String a;
3
4     public String foo() {
5         String s = source();
6         this.a = s + "foo";
7         return this.a;
8     }
9 }
```

Figure 4.4: Example of information flow from `source` to `return`.

how information propagates through a program [15]. There are three constituents in information flow analysis: *sources*, *sinks*, and *sanitizers*. Sources are expressions where values enter the program, sinks are expressions where values leave the program and sanitizers are operations that take values and make them harmless according to the threat model of the domain. An information flow analysis tries to find flow of information from sources to sinks that does not pass through a sanitizer.

Consider the example in Figure 4.4. In the method `foo`, information enters the program from the return value of the call to `source` in line 5. The program computes a new string in line 6 and stores it in the field `a`. Since the computed string contains all the information from `s`, there is also information flow from `source` to the return value `a` of the method `foo`. If we consider the return statement a sink in the program, the value from `source` can pass to the sink without going through a sanitizer.

Information flow analysis or taint analysis has previously been applied to detect vulnerabilities that relate to unsanitized input parameters.

The WebSSARI tool by Huang et al. [31] pioneered the use of static information flow analysis to enforce web application security, and numerous researchers have since followed that path.

Wassermann and Su developed a static analysis technique for finding injection vulnerabilities [80, 87]. The technique uses a combination of taint analysis and string analysis to detect untrusted substrings that could change the structure of the queries. The observation behind using such a string analysis is that applications are only vulnerable if the client-supplied string is able to change the structure of the query. In later work, they apply a similar approach to find cross-site scripting vulnerabilities [88].

Earlier work by Jovanovic et al. [36, 38] and by Xie and Aiken [89] used taint analysis exclusively for detecting the same type of vulnerabilities. In both lines of work, it is stated that the technique is applicable for detecting cross-site scripting as well.

Livshits and Lam presented a taint analysis that is based on a points-to-analysis [49]. In their work, vulnerabilities are described through their *Program Query Language* (PQL) and the analysis algorithm detects flow from sources to sinks based on the vulnerability descriptions. They apply their tool to a variety of vulnerability types including SQL injections and cross-site scripting.

Tripp et al. presented Taint Analysis for Java (TAJ) [83]. TAJ allows fast and effective taint analysis of large applications. Their tools can be applied to detect the information flow related vulnerabilities that we have discussed.

#### Dynamic taint checking

Dynamic solutions include tools for dynamically tracking tainted, unchecked data. The Perl programming language has the *taint checking*[3] feature to allow the programmer to mark strings as tainted. To avoid cross-site scripting, code that generates HTML can check for the taint flag on the string and stop the execution if a tainted string is inserted without any checks or sanitation. The same technique can be applied to avoid injection attacks.

Ruby has a similar tainting feature. Ruby will automatically mark input values as tainted. The programmer can set a *safe level* for his code. This level is used by Ruby to determine which values to mark as tainted (keyboard input, network data etc.).

Taint checking, however, gives no guarantee of the absence of vulnerabilities and is merely a way to stop execution when the problem is detected.

### 4.3.2 Overview of the analysis technique in WARLord

As defined in Section 2.3.2, client-state manipulation vulnerabilities occur when the server stores trusted data as part of the document on the client side and expects this data to remain trust-worthy from one request to another. In essence, client-state manipulation vulnerabilities may appear if there is flow of untrusted data from the client-state parameters provided by the decoder to the store. If the server code does not verify that the data has not been tampered with by the client, then it might be possible for the client to create an exploit and gain access to changing or reading data on the server.

Contrary to injections or cross-site scripting, detecting client-state manipulation vulnerabilities requires knowledge about the origin of each parameter to determine whether the parameter carries client-state. The analysis therefore has three components: 1) It finds client-state parameters through output analysis of the pages of the web application. 2) It determines sink locations in the code that must not be written to unless the value has been sanitized. 3) It runs an information flow analysis to track client-state parameter values to the sink locations.

The analyses are described in detail in Chapter 8, and we will just discuss an overview of the three components in this section.

#### Finding client-state parameters

To find client-state parameters, we analyze all output of the application based on the approximation described in Section 4.1. We characterize hidden form fields and URL parameters as client-state values and we analyze the output to find names of such fields and parameters.

By combining parameter names with targets of links and form submissions, we construct a *page graph* that describes the flow of parameters between the pages of the web application. From this graph, we can determine which client-state parameters flow into each page in the web application.

Figure 4.5 shows an example of such a page graph for the application *JSPChat* that was analyzed as part of the evaluation in the paper. The page graphs shows an excerpt of the pages in *JSPChat* with edges representing flow of client-state parameters from one page to another. For each page, the set of in-going edges show the set of client-state parameters for the page. For example, `nickname` is a client-state parameter to `SaveInfo`. This corresponds to the client-state parameter we discussed in Section 2.3.2. JSPChat is explained in further details in Section 8.8.

[3] `http://perldoc.perl.org/perlsec.html`

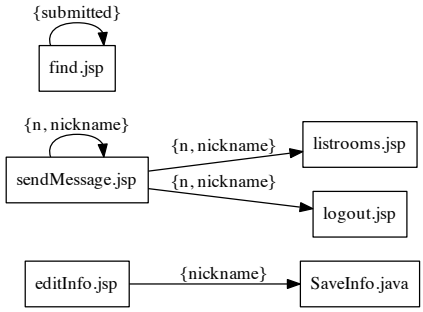


Figure 4.5: The automatically constructed page graph for the parts of *JSPChat* that involve client-state.

The result of these steps is for each page a set of parameter names that may hold client-state values.

### Finding application state

In the second analysis component, we determine which locations in the program that may hold values related to the part of the store that is shared between requests and clients. That is, we determine which variables may hold *application state*. The analysis distinguishes between *internal* application state, that is heap values that are reachable in multiple requests (for example values that are stored in the `HTTPServletContext` object) and *external* application state, that is values that are stored in the file system, database etc. We find all locations that may hold such values through a simple fix point procedure that is described in more detail in the paper.

### Finding flow from client-state parameters to application state

The remaining analysis determines the information flow through the web application. This is done as an information flow analysis.

- Sources in the analysis are method calls that may read the client-state parameters that we detected through the output analysis.
- Sinks are operations in the code where the application writes to fields of internal application state objects or calls methods that involve external application state.
- Sanitizers are application dependent. They correspond to operations where the programmer checks whether a value has been manipulated. The user of the analysis provides a description of these sanitizers.

The information flow analysis is a whole-program data flow analysis and it is described in more detail in the paper.

The analysis excludes library code and considers method parameters to library method calls as potential sinks. This behavior can be changed through a *customization* feature that allows the user of the analysis to specify derivation rules that describe the behavior of the method more precisely. The customization mechanism furthermore allows the user to specify application specific sanitizers to eliminate spurious warnings.

### 4.3.3 Evaluation

The tool has been implemented as an analysis on top of the Soot analysis framework [84] as part of the WARLord analysis suite. Currently, the tool supports analysis of Servlet, JSP and Struts applications.

In the paper, we investigated the following research questions:

Q1: Is the analysis precise enough to detect client-state vulnerabilities with a low number of false positives? Specifically, can it identify the common uses of client-state, and is it capable of distinguishing between safe and unsafe uses of client-state in the sense described in Section 8.2?

Q2: Are the warning messages produced by the tool useful to the programmer for deciding whether they are false positives or indicate exploitable vulnerabilities?

Q3: In situations where the programmer decides that a vulnerability warning is a false positive, is it practically feasible to exploit the customization mechanism to eliminate the false positive?

Q4: Is the analysis fast enough to be practically useful during web application development?

We evaluated the tool on a series of open-source benchmark applications and found that we could answer all the questions affirmatively. The full evaluation can be found in Section 8.8.

# Chapter 5

# Conclusion

We have argued that security vulnerabilities, client-state manipulation vulnerabilities in particular, and output invalidity are important problems in web applications. We have surveyed a set of web application frameworks and seen how these problems remain unsolved in current web application frameworks.

We set out to investigate the hypothesis that web application framework design and static analysis could help the programmer avoid these errors.

We have studied a new web application framework JWIG which handles these problems through a combination of framework design and an analysis suite. We evaluated JWIG based on the safe by default principle and found that JWIG was able to support the programmer in avoiding common errors. We evaluated the usefulness of JWIG by implementing a large application CourseAdmin and concluded that the design of JWIG was also useful for implementing large applications.

We have discussed static analysis for current web application frameworks and seen how we can approximate the output of a page of an application.

To handle validation of the SGML based HTML language, we studied the parsing and validation model for HTML and generalized the parsing algorithm to work for context-free languages. Through evaluation on open-source benchmarks, we found the analysis useful for verifying validity of the generated HTML and we concluded that the analyses had high precision when analyzing a set of open-source web applications.

We applied a combination of output analysis and information flow analysis to detect and analyze the use of client-state parameters in order to find client-state manipulation vulnerabilities. We evaluated the technique on a series of open-source benchmarks to study usefulness of this approach to detecting client-state manipulation vulnerabilities in practice. The evaluation showed that the technique had high precision and that we were able to find real vulnerabilities in the applications.

We conclude that framework design and static analysis can help the programmer to avoid HTML invalidity errors and client-state manipulation vulnerabilities.

# Part II

# Publications

# Chapter 6

# JWIG: Yet Another Framework for Maintainable and Secure Web Applications

This chapter contains the paper "JWIG: Yet Another Framework for Maintainable and Secure Web Applications" [59]. The paper originally appeared at the 5th International Conference on Web Information Systems and Technologies (WEBIST 2009).

The version in this chapter is an extended version of the paper which can also be downloaded as a tech report from the authors' home pages. Compared to the published version, this version has an extra section describing techniques for caching an authorization. It also contains additional examples. The paper is supplemented by the JWIG User's Manual [62] which is not included in this dissertation.

# JWIG: Yet Another Framework for Maintainable and Secure Web Applications

Anders Møller Mathias Schwarz

**Abstract**

Although numerous frameworks for web application programming have been developed in recent years, writing web applications remains a challenging task. Guided by a collection of classical design principles, we propose yet another framework. It is based on a simple but flexible server-oriented architecture that coherently supports general aspects of modern web applications, including dynamic XML construction, session management, data persistence, caching, and authentication, but it also simplifies programming of server-push communication and integration of XHTML-based applications and XML-based web services. The resulting framework provides a novel foundation for developing maintainable and secure web applications.

## 6.1 Introduction

Web applications build on a platform of fundamental web technologies, in particular, XHTML, HTTP, and JavaScript. Although these are relatively manageable technologies, it is generally regarded as difficult to write and maintain nontrivial, secure applications. The programmer must master not only the fundamental technologies but also techniques for session management, caching, data persistence, form input validation, and rich user interfaces. In addition, most web applications are exposed to all hackers in the world, so the programmer must also consider potential security problems, such as, injection attacks, cross site scripting, and insufficient authentication or encryption.

In recent years, a multiplicity of *web application frameworks* have emerged, all claiming to make web application development easier. Among the most widely known are Struts [53], Spring MVC [35], Google Web Toolkit [25], Ruby on Rails (RoR) [28], PHP [47], and ASP.NET [54]. These frameworks represent different points in the design space, and choosing one for a given task is often based on subjective arguments. However, a general limitation is the lack of a simple unified communication model that supports both the traditional server-oriented structure and the more modern style using AJAX[1], server-push[2], and JSON or XML-based web services [13, 85].

In this paper, we try to take a fresh view on the problem. Based on essential design principles, we exhibit some of the limitations of existing frameworks, propose yet another web application framework, and compare it with the existing ones. The main contributions of this paper can be summarized as follows:

[1] `http://en.wikipedia.org/wiki/Ajax_(programming)`
[2] `http://en.wikipedia.org/wiki/Comet_(programming)`

- First, we identify and argue for a small collection of requirements and design principles for creating a web application framework.

- In Section 6.2 we propose a simple architecture that supports the basic features of receiving HTTP client input and producing XHTML output, in accordance with the design goals.

- In Sections 6.3–6.9 we demonstrate the flexibility of the architecture by showing how it can smoothly handle other important aspects that can be challenging to work with in other frameworks, including server-push communication, session state, persistence, and authentication.

- The resulting framework provides a clear structure of both control and data, which makes the application program code amenable to specialized static analysis. For example, one analysis checks—at compile-time—that only valid XHTML 1.0 data is sent to the clients during execution.

- The new framework is evaluated through a series of short but nontrivial example programs and a case study that demonstrate the resulting system in relation to the requirements and design principles, in comparison with alternative frameworks.

### Premises and Requirements

To specify the aims of our task and narrow the design space, we begin by formulating the basic premises and requirements:

- We focus on Java, for the simple reasons that this language is well-known to most programmers, especially those developing web applications, and it comes with a massive collection of useful, open source libraries that we and the application programmers can build on. An immediate consequence of this choice is that we cannot benefit from, for example, general higher-order functions, closures and continuations, as known from functional languages and frameworks such as Seaside [3] and the PLT Scheme Web Server [44]. Such features can be emulated in Java, but not elegantly. On the other hand, we can exploit Java's reflection capabilities. We permit simple syntactic extensions of Java, in particular to support XML processing.

- We assume that the browsers are capable of rendering XHTML and executing JavaScript code, as in Internet Explorer 8 and Firefox 3, but the framework cannot use browser plugins. This means that the applications will be readily deployable to all users with modern browsers.

- We postulate that a majority of web applications do not require massive scalability and have less than a thousand concurrent uses. Our framework design should focus on this category.

- The framework must uniformly support both XHTML applications (where the clients are browsers) and XML-based web services (where the clients are other types of software). It should be possible to statically check that only valid XML output is produced, relative to a given XML schema. This requires a clear flow of control and data in the code. For the back-end, the framework must integrate with existing object-relational mapping (ORM) systems, e.g. Hibernate [67], such that data persistence can be obtained with minimal effort to the programmer, but without being tied to one particular system.

- Representational state transfer (REST) is a collection of network architecture principles that the Web is based on [70]. We focus on its use of URLs for addressing resources with generic interfaces as HTTP methods (GET, POST, etc.) and caching. In contrast, the remote procedure call (RPC) approach encourages the use of URLs for addressing operations, not resources, which sometimes fits better with the abstractions of the programming language in use. Both approaches should be uniformly supported by the framework.

### Design Principles

To guide the design of the framework, we adhere to the following key principles:

***High cohesion and low coupling*** Cohesion is a measure of how strongly related and focused the responsibilities of a software unit is. The related concept of coupling is a measure of how strongly dependent one software unit is on other software units. It is common knowledge in software engineering that high cohesion and weak coupling are important to maintainability and reliability of the software [79]. Nevertheless, many web programming frameworks fail in these measures when considering, for example, handling of form data and asynchronous client–server communication. Examples of this are shown in Sections 6.4 and 6.5.

***Secure by design*** Buffer overruns are an example of a security concern that has largely been eliminated by the use of languages that are secure by design, for example Java or C# instead of C or C++. However, injection attacks, cross-site scripting, insecure direct object references, broken session management, and failure to restrict URL access remain among the most serious classes of web application vulnerabilities [69]. We believe that the principle of 'secure by design' should be applied at the level of web application frameworks to address those classes of vulnerabilities.

***Convention over configuration*** All configuration should have sensible defaults, in order to minimize the number of detailed decisions that developers need to make for the common cases. This principle was popularized by Ruby on Rails.

In the following sections, we give examples of how existing frameworks violate these principles and explain our proposal for a solution. We omit discussions of how our framework handles input validation and XHTML extensions for making catchy user interfaces, and we only briefly touch upon the relations to other frameworks.

## 6.2 Architecture

Web clients in general cannot be trusted, and essentially all web applications contain sensitive data, so according to the 'secure by design' principle and for simplicity our starting point is a classical *server-oriented* approach where the application code is executed on the server rather than on the client. This means that we avoid many of the security considerations that programmers have if using a client-oriented architecture, as e.g. GWT. (A concrete example of this advantage is shown in section 6.10.2.) One of the typical arguments in favor of a client-oriented approach is the opportunity to create rich user interfaces—however, such effects are possible also in server-oriented frameworks using tag libraries containing JavaScript code. Also, pushing computation to the client-side may imply a decreased load on the server, but, on the other hand, it often conflicts with the use of ORM systems: It is

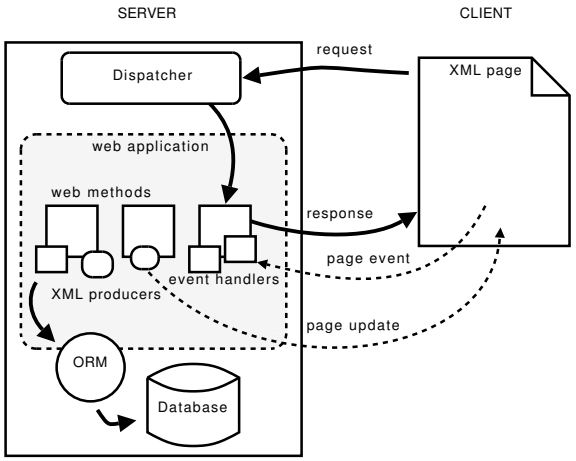


Figure 6.1: The framework architecture. The boxes and circles represent the main static components, and the arrows show the dynamic interactions between the components.

```
public class Main extends WebApp {
  public XML hello(String what) {
    return [[
      <html>
        <head><title>Example</title></head>
        <body>
         <p>Hello <{ what }></p>
        </body>
      </html>
    ]];
  }
}
```

Figure 6.2: A "hello world" web application.

difficult to ensure high performance if the application code is physically separated from the ORM system.

Figure 6.1 illustrates the basic architecture. As in many other newer server-oriented frameworks, a *dispatcher* on the server receives the client HTTP requests and invokes the appropriate server code, here called *web methods*. Many frameworks rely on separate configuration files for mapping from request URLs to application code. To decrease coupling, our web methods are instead discovered using introspection of the web application code, as in e.g. CherryPy. The web methods have formal parameters for receiving arguments from the clients and return values to be sent as response.

As an example, the tiny web application shown in Figure 6.2 accepts HTTP GET requests of the form `http://example.org/Main/hello?what=World` and returns a small XHTML page as response. The class `Main` is a web application that contains a single web method `hello`. We explain the XML construction and the `[[...]]` notation in Section 6.3. Following the 'convention over configuration' principle, no application specific configuration is required to make this run—unlike the situation in, for example, Struts or RIFE [4].

The architecture is a variant of the Model-View-Controller pattern that many other frameworks also apply. Most importantly, the model (the database) is sepa-

rated from the main web application code. However, we propose a less rigid division between the view and the controller than used in other frameworks, as we explain in Section 6.3.

In the following sections, we go into more details and extend this basic architecture to fulfill the requirements we defined, which includes introducing the notions of *XML producers* and *event handlers* and the *page event* and *page update* actions appearing in Figure 6.1.

## 6.3 Generating XML Output

A common approach to generating XHTML output dynamically in web application frameworks is to use a template system, either where templates contain code for dynamic construction of content (as in e.g. ASP.NET, PHP, or RoR) or where templates contain placeholders where the code can insert content (as in e.g. RIFE). Another approach is to use GUI components that are assembled programatically (as e.g. GWT or Wicket). Web services, on the other hand, need the ability to also receive and decompose XML data, which is often done completely differently in a DOM-style fashion.

All those systems lack the ability to ensure, at compile time, that output is always well-formed and valid XML data. This is known to affect reliability and may lead to, for example, cross site scripting vulnerabilities.

We propose a solution that (1) unifies the template approach and the DOM-style approach, (2) permits static validation analysis, and (3) avoids many security issues by design.

We build on a new version of the XACT system [41]. XML data is represented as well-formed XML fragments that are first-class values, meaning that they can be stored in variables and passed between methods (unlike most other template systems). Figure 6.2 shows an XML value constant (enclosed by `[[...]]`, using a simple syntax extension to Java) that contains a snippet of code (enclosed by `<{...}>` in XML content or `{...}` in attribute values), which at runtime evaluates to a value that gets inserted. The syntax extension reduces the burden of working with XML data inside Java programs and is desugared to ordinary Java code.

XML values may also contain named *gaps* where other XML values or strings can be inserted, which makes it easy to reuse them. As an example, the following code defines a wrapper for XHTML pages and stores it in a variable `w`:

```
XML w = [[
 <html>
  <head><title>My Pink Web App</title></head>
  <body bgcolor="pink"><[BODY]></body>
 </html>
]];
```

This wrapper can then be used whenever a complete page needs to be generated, for example in this web method:

```
XML time() {
  return w.plug("BODY", [[
   <p>The time is: <b><{ new Date() }></b></p>
  ]] );
}
```

To obtain separation of concerns between programmers and web page designers, XML values can also be stored in separate files. In fact, contracts can be established to formalize and verify this separation, as explained in the article by Böttget et al. (2006).

XML values can also be decomposed and transformed with operations inspired by JDOM and XPath. By design, XML values are always well-formed (i.e. tags are balanced, etc.). When inserting text strings into fragments, special characters are automatically escaped, which eliminates a significant class of security vulnerabilities. In addition, variables can be annotated with XML Schema type information, and a program analysis can perform type checking to ensure that output is always valid according to a given schema [39].

As an example (from the CourseAdmin system described in Section 6.11) in the RPC-style web service category, the following web method uses the features of XACT to produce an XML document describing the status of hand-in exercises for a given student:

```
XML<c:handins> getHandins(XML<c:student> s) {
 String sid =
  db.getStudents().get(s.getAttribute("id"));
 List<Handin> hs = db.getHandins().get(sid);
 if (hs == null)
  throw new NotFoundException();
 XML x = [[ <c:handins> <[H]> </c:handins> ]];
 for (Handin h : hs)
  x = x.plug("H", [[
   <c:handin number={ h.getNumber() }
             status={ h.getStatus() } />
   <[H]>
  ]] );
 return x;
}
```

We here assume that the underlying model is represented by a data structure `db`. By default, web methods react only on HTTP GET requests. This can be changed using an annotation, for example `@POST` for web methods that are unsafe in the HTTP sense.

## 6.4 XML Producers and Page Updates

The response of a web method is generally a view of some data from the underlying database. When the data changes, the view should ideally be updated automatically while only affecting the relevant parts of the pages to avoid interfering with form data being entered by the user. With many frameworks, this is a laborious task that requires many lines of code and insight into the technical details of JavaScript and AJAX, so many web application programmers settle with the primitive approach of requiring the user to manually reload the page to get updates.

We propose a simple solution that hides the technical details of the server-push techniques (aka. Comet) and integrates well with the XACT system. An *XML producer* is an object that can produce XML data when its `run` method is called. When the object is constructed, dependencies on the data model are registered according to the observer pattern. An XML producer can be inserted into an XHTML page such that whenever it is notified through its dependencies, `run` is automatically called and the resulting XML value is pushed to all clients viewing the page. All the technical details involving AJAX and Comet are hidden inside the framework, and it scales well to hundreds of concurrent uses.

Typically, the XML producer is created as an anonymous inner class within the web method that generates the XHTML page. This ensures high cohesion for that software component and low coupling by only depending on the model of the data that is shown to the client. An example is shown in Section 6.6.

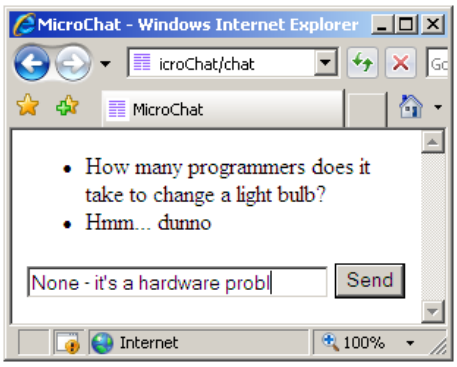


Figure 6.3: The running MicroChat application.

## 6.5 Forms and Event Handlers

Forms constitute the main mechanism for obtaining user input in web applications. With most frameworks, the code that generates the form is separate from the code that reacts on the input being submitted, and these pieces of code are connected only indirectly via the URLs being generated and the configuration mapping from URLs to code. This violates the principle of high cohesion and low coupling, and the flow of control and data becomes unclear.

Our solution is to extend the architecture with *event handlers* as a general mechanism for reacting to user input. We here focus on forms although the mechanism also works for other kinds of DOM events. The technique is related to the use of action callbacks in Seaside.

Forms are created by building XML documents that contains a `form` tag with relevant input fields. A specialized event handler, called a submit handler, is plugged into the `action` attribute. The submit handler contains a method that can react when the form is submitted and read the form input data. (An example is shown in Section 6.6.) As with XML producers, event handlers are typically anonymous classes located within the web methods.

The consequence of this structure is a high degree of code cohesion. Also, it becomes possible by static analysis to verify consistency between the input fields occurring in the form and the code that receives the input field data.

## 6.6 Example: MicroChat

The following example shows a tiny chat application that uses the features explained in the previous sections. It shows a list of messages and a text field where users can write new messages (see Figure 6.3).

```
public class MicroChat extends WebApp {
 List<String> messages =
  new ArrayList<String>();

 public XML chat() {
  return [[
   <html>
    <head><title>MicroChat</title></head>
    <body>
     <{ new XMLProducer(messages) {
       XML run() {
       if (!messages.isEmpty())
```

```
        return [[
         <ul>
          <{ [[<li><[MSG]></li>]]
              .plugWrap("MSG", messages) }>
         </ul>
        ]];
       else
        return [[]];
    }
    } }>
    <form method="post" action=[SEND]>
     <p>
      <input type="text" name="msg"/>
      <input type="submit" value="Send"/>
     </p>
    </form>
   </body>
  </html>
 ]]
 .plug("SEND", new SubmitHandler() {
  void run(String msg) {
   messages.add(msg);
   update(messages);
  }
 });
}}
```

For simplicity, the application state is here represented as a field `messages` in the application class; we discuss persistence in Section 6.8. The XHTML document being created when the user invokes the `run` web method contains an instance of an `XMLProducer` that declares itself to be an observer of `messages` and writes the messages in the document. The `plugWrap` method here builds a list of `<li>` items containing the messages. Secondly a submit handler is plugged into the `action` attribute of the form. The handler method reads the form field `msg`, adds it to the list of messages, and then invoke `update` to notify all observers of the list, in this case the XML producer. The effect is that whenever a user posts a new message, the list of messages is automatically updated in all browsers viewing the page.

Notice the high degree of cohesion within the `chat` web method. In most other frameworks (with Seaside and as a notable exception) the equivalent code would be more fragmented and hence less maintainable.

## 6.7 Parameters and References to Web Methods

Web methods and event handlers can be parameterized, as shown in the previous examples. Any Java class that contains a static method `valueOf` can be used for such parameters for deserializing values, as known from the basic Java library. Conversely, serialization for output is performed using `toString` (for strings) or `toXML` (for XML values). The dispatcher then transparently handles the serialization and deserialization. For example, JSON data is trivial to transfer with this mechanism.

The URL format used in requests to web methods can be controlled by an annotation, to support REST-style addressing of resources. As an example, the following annotation could be placed on the `hello` web method from Figure 6.2 to override the default format, such that it can be invoked by URLs like `http://example.org/foo/World`:

```
@URLPattern("foo/$what")
```

Links between pages can be created using the method `makeURL`:

```
XML my_link = [[
 <a href={ makeURL("hello", "John Doe") } >
  click here
 </a>
]];
```

The web method name is passed as a constant string (since methods are not first-class values in Java), but it is straightforward to statically type check that it matches one of the web methods within the application. Parameters are serialized as explained above, and the resulting URL is generated according to the URL pattern of the web method.

This approach ensures that the coupling between web pages becomes explicit in the application source code, in contrast to frameworks that involve URL mappings in separate configuration files. As we explain in the next section, it additionally provides a novel foundation for managing session state and shared state.

## 6.8 Session State and Persistence

A classical challenge in web application programming is how to represent session state on top of the inherently stateless HTTP protocol. Although REST prescribes the use of stateless communication, without ever storing session state on the server, we believe that the benefits of allowing session state on the server outweigh the disadvantages—a concrete example is shown in section 6.10.2.

Many frameworks track clients using, for example, cookies or URL rewriting and provide a string-to-object map for each client for storing session state. Using cookies in this way conflicts with the REST principle that resources should be addressable by URIs: With cookie-based session management, one client cannot participate simultaneously in several sessions of the same web application, and it is difficult to transfer a session from one client to another.

Our approach is to store session state in objects derived from an abstract class named `Session`. Using a class instead of a map means that the ordinary Java type checker will check that expected properties are present when a web method accesses the session state. The class defines the `valueOf` and `toString` methods so the objects can be given as parameters to web methods using the mechanism described in Section 6.7. Serialization assigns a long random key to each session object, and deserialization finds the session object using this key.

The following example extends the "Hello World" example from Section 6.2 so it now saves the `what` parameter in a newly created session object and then redirects to the `sayHi` method, which reads the session data and embeds it in its XML output:

```
URL hello(String what) {
 return makeURL("sayHi", new HelloSession(what));
}

class HelloSession extends Session {
 String name;
 public HelloSession(String s) { name = s; }
}
public XML sayHi(HelloSession s) {
 return [[
  <html>
   <head><title>Example</title></head>
   <body><p>Hello <{ s.name }></p></body>
  </html> ]];
}
```

Session state is here encapsulated in the `HelloSession` objects, and it is clear from the signature of the `sayHi` web method that it requires the client to provide a session

key. This way of tracking clients is effectively a variant of URL rewriting that is integrated with the dispatching mechanism of the framework.

A session garbage collector thread takes care of removing session objects that are not accessed by a client within a certain period of time. As a convenient extra feature, the generated XHTML pages automatically (using AJAX) inform the server that the relevant session objects are alive as long as the pages are being viewed in a browser.

All web applications and web services, beyond the level of toy examples, encompass a database for data persistence. As mentioned in the introduction, we focus on ORM systems. To this end, we utilize the same pattern as for session state: Persistable data must implement the `Persistable` interface, which requires it to define an ID string for each object. This ID can be used to query the object from the database during deserialization. It can be passed around in the URLs in a call-by-reference style via the `makeURL` mechanism or hidden inside session objects.

Among the currently most serious web application vulnerabilities is *insecure direct object references*, i.e. situations where malicious users can modify data references passed in URLs with insufficient authentication on the server [69]. Employing the 'safe by default' principle, our framework requires that permission must explicitly be given to use an ID of a persistable object. This is done coherently by defining a method named `access` that returns true when the use of the given ID is allowed in the context of the HTTP request concerned. Other issues related to authentication are discussed in the next section.

## 6.9 Caching and Authentication

We propose a notion of *filters* that uniformly handles caching, authentication, and logging following the design principles from Section 6.1. The dispatcher permits a given request URL to match multiple web methods, which are then processed in turn until one produces a response. The order can be controlled by a `@Priority` annotation. A filter is a web method with return type `void` and by default has higher priority than ordinary web methods.

The class `WebApp`, from which all web applications are derived, has a filter named `cache` with a URL pattern that matches all requests. This filter transparently performs server-side caching by storing a number of responses generated by the ordinary web methods. It also handles conditional GET requests to support client-side caching.

The cache can be connected by the observer pattern to the underlying data for removal of stale pages. This currently requires the programmer to specify the dependencies: For example, invoking `addPageInvalidator(x)` ensures that the current page gets invalidated when `update(x)` is invoked. Obviously, this violates the 'secure by default' principle since the programmer may forget to specify all dependencies, in which case the users may get stale data. To our knowledge, no existing web programming framework solves this problem; we currently investigate the use of static dependency analysis to address this.

Filters can also be used for decoupling authentication from the application logic:

```
@URLPattern("restricted/**")
public void authenticate() {
  if (!isSecure())
    throw new AccessDeniedException();
  User u = getUser();
  if (u == null ||
      !u.getUsername().equals("jdoe") ||
      !u.getPassword().equals("42"))
    throw new
    AuthorizationRequiredException("MyRealm");
```

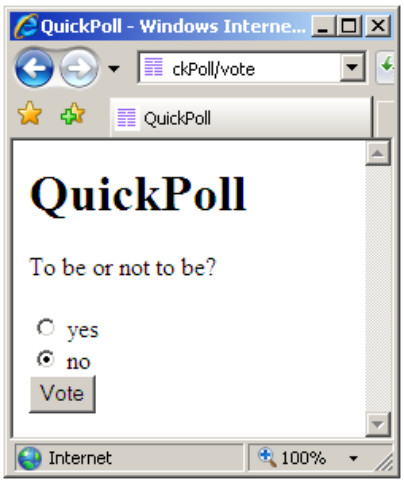


Figure 6.4: Voting in QuickPoll.

```
  else
    next();
}
```

This filter restricts access to all resources with URLs matching the pattern `restricted/**` relative to the base URL of the web application. If the connection is insecure (i.e. not using SSL/TLS), a 403 Forbidden response is made. The `getUser` method returns the credentials provided via HTTP Basic authentication. If the user is not authorized, a 401 Unauthorized response is made to request the client for acceptable credentials. Otherwise, control is passed to the next web method in line using the `next` method.

## 6.10 Additional Examples

This section provides a two examples showing how the techniques are used to build an small web application. The QuickPoll example demonstrates the use of the server push techniques and the GuessingGame example demonstrates how sessions can be used to curb the control flow in a web application. The source code of these two examples can be found in the appendix.

### 6.10.1 QuickPoll

The first example, QuickPoll, is a poll system. The web method `main` produces a menu page; `init` is used by the poll administrator to set the question to be asked; `vote` produces a page containing the question and a yes/no form and processes the user's choice; and finally, `results` shows an overview of the votes. It uses all the framework features covered by Sections 6.2–6.7—in particular, two submit handlers and one XML producer. The application state could easily be made persistent, as explained in Section 6.8, and authentication could be added as in Section 6.9.

### 6.10.2 GuessingGame

The second example, GuessingGame, is the classical toy where each user must guess a number between 1 and 100 in as few attempts as possible. The `start` web method creates a new session object, corresponding to a user starting a new game,

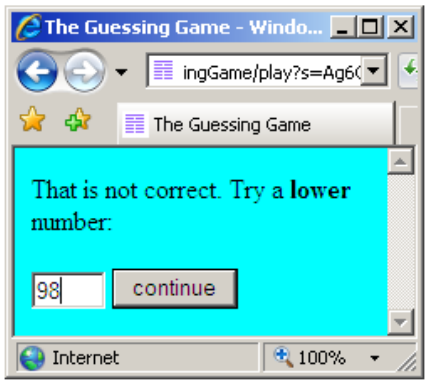


Figure 6.5: Playing the GuessingGame.

and immediately redirects to `play`. The session object contains the secret number to be guessed, the number of attempts, and also a snapshot of the current XHTML page. The `play` web method simply shows the page of the given session. The main functionality is located in the constructor of the session object: it first creates the initial XHTML page and then uses submit handlers together with the features of XACT (that we shall not explain in detail here) to modify the page contents when the users provides input. Finally, `record` shows the total number of plays and the current record holder. The global game state is stored in a persistable `GameState` object. The source code for the `GameState` class it not included here, since it belongs to the data model and does not contain any web-specific functionality.

Note that all the web methods react on GET requests whereas the submit handlers use POST, reflecting the fact that the former are safe (in the HTTP sense) unlike the latter which have side-effects. In particular, a URL for the `play` method, which contains a session key as parameter, can always be used to get the current page of the session.

We here omit a thorough comparison of how web applications with similar functionality could be programmed with other frameworks. However, we claim that the benefits discussed in the previous sections regarding code structure and maintainability also apply to these examples. In particular, the functionality of GuessingGame is cumbersome to express without storing session state on the server since it is crucial that the users are not able to manipulate it or backtrack during a game.

## 6.11 Case Study: CourseAdmin

To evaluate the framework in a more realistic setting and to test it for programming applications beyond toy examples we have used it to develop the application CourseAdmin. This is a medium sized web application (more than 20,000 lines of code) for handling course administration at our institute. The application includes management of assignments, a webboard, and attendance counting. The application has close to 1200 users with about 200 daily unique users. A part of the system is written using the XML syntactic sugar while the rest is written in pure Java, and even without the extra convenience of the syntactic extension, the XML model remains usable.

CourseAdmin is structured as four distinct web applications with a common model layer: the public web pages of classes and students, the course configuration application for course staff, an application for students to view their status and upload assignments, and finally a configuration application for creating new courses.

High cohesion exists in this architecture because the web applications share common web methods, including authentication filters, by inheriting from common abstract super classes, and the division of the system into multiple web applications results in low coupling between unrelated program parts. This indicates that the patterns presented in the small example programs in this paper can be used to build much larger systems.

The main XML templates are located in separate files, and layout is controlled using CSS, which makes the page design consistent and easy to modify.

The 'convention over configuration' approach means that CourseAdmin only needs a handful of configuration lines for database credentials and email server names.

The 'secure by default' principle is important in CourseAdmin, since user provided content in the webboard is shared between the clients, and the system contains confidential data about student grades. By the use of the `access` mechanism for protecting data in the model layer as explained in Section 6.8 and authentication filters as in Section 6.9, all necessary security checks are collected in one cohesive component, and new functionality can be added to the system without risking violations of the existing policies.

## 6.12 Conclusion

We have presented a novel minimalistic Java-based framework that provides uniform support for common tasks in web programming. By the use of a reflection-based *dispatcher*, XACT for XML processing, *XML producers* for server-push communication, *event handlers* for user input processing, and *filters* for caching and authentication, we obtain a framework for making maintainable and secure web applications with high cohesion, low coupling, and security-by-design.

As ongoing and future work, we aim to provide a more detailed analysis of the capabilities and limitations of other frameworks and their relation to the one we have presented here. Also, we remain to show how the framework can handle user input validation through a combination of XML producers and event handlers and rich user interfaces by integrating existing JavaScript libraries using tag-like XHTML extensions.

# APPENDIX

This appendix contains the source code for the GuessingGame and QuickPoll example web applications programmed with our framework. A couple of screenshots are shown in Fig. 6.4 and 6.5. Our primary interest here is code maintainability, so we settle for a primitive design of the XHTML pages. (As explained in Section 6.3, separation of concerns between XHTML page design and Java code can be accomplished by moving the XML templates into separate files.)

## Example: QuickPoll

```
public class QuickPoll extends WebApp {

 private XML wrapper = [[
  <html>
   <head><title>QuickPoll</title></head>
    <body>
     <h1>QuickPoll</h1>
     <[BODY]>
    </body>
   </html>
 ]];

 class State {
  String question;
  int yes;
  int no;
 }

 State state = new State();

 public XML main() {
  return wrapper.plug("BODY", [[
   <ul>
    <li>
     <a href={makeURL("init")}>Initialize</a>
    </li>
    <li>
     <a href={makeURL("vote")}>Vote</a>
    </li>
    <li>
     <a href={makeURL("results")}>View results</a>
    </li>
   </ul>
  ]]);
 }

 public XML init() {
  return wrapper.plug("BODY", [[
   <form method="post" action=[INIT]>
    <p>What is your question?</p>
    <p>
     <input name="question" type="text" size="40"/>?<br/>
     <input type="submit" value="Register my question"/>
    </p>
   </form>
  ]]).plug("INIT", new SubmitHandler() {
   XML run(String question) {
    state.question = question;
    state.yes = state.no = 0;
    update(state);
    return wrapper.plug("BODY", [[
     <p>Your question has been registered.</p>
     <p>Let the vote begin!</p>
```

```
   ]]);
  }
 });
}

public XML vote() {
 if (state.question == null)
  throw new AccessDeniedException
      ("QuickPoll not yet initialized");
 addResponseInvalidator(state);
 return wrapper.plug("BODY", [[
  <{state.question}>?<p/>
  <form method="post" action=[VOTE]>
   <p>
    <input name="vote" type="radio" value="yes"/>
    yes<br/>
    <input name="vote" type="radio" value="no"/>
    no<p/>
    <input type="submit" value="Vote"/>
   </p>
  </form>
 ]]).plug("VOTE", new SubmitHandler() {
  XML run(String vote) {
   if ("yes".equals(vote))
    state.yes++;
   else if ("no".equals(vote))
    state.no++;
   update(state);
   return wrapper.plug("BODY", [[
    <p>Thank you for your vote!</p>
   ]]);
  }
 });
}

public XML results() {
 return wrapper.plug("BODY",
    new XMLProducer(state) {
  XML run() {
   synchronized (state) {
    int total = state.yes + state.no;
    if (total == 0)
     return [[ <p>No votes yet...</p> ]];
    else
     return [[
      <p><{state.question}>?</p>
      <table border="0">
       <tr>
        <td>Yes:</td>
        <td><{drawBar(300*state.yes/total)}></td>
        <td><{state.yes}></td>
       </tr>
       <tr>
        <td>No:</td>
        <td><{drawBar(300*state.no/total)}></td>
        <td><{state.no}></td>
       </tr>
      </table>
     ]];
    }
   }
 });
}

private XML drawBar(int length) {
 return [[
  <table>
```

```
    <tr>
     <td bgcolor="black" height="20"
         width={length}/>
    </tr>
   </table>
  ]];
 }
}
```

## Example: GuessingGame

```
public class GuessingGame extends WebApp {

 private XML wrapper = [[
  <html>
   <head><title>The Guessing Game</title></head>
   <body style="background-color: aqua">
    <[BODY]>
   </body>
  </html>
 ]];

 Random rnd = new Random();

 class UserState extends Session {

  int number;
  int guesses;
  XML page;

  UserState() {
   number = rnd.nextInt(100)+1;
   guesses = 0;
   page = wrapper.plug("BODY", [[
    <p id="MSG">
      Please guess a number between 1 and 100:
    </p>
    <form method="post" action=[GUESS]>
     <p>
      <input name="guess" type="text" size="3"/>
      <input type="submit" value="continue"/>
     </p>
    </form>
   ]]).plug("GUESS", new SubmitHandler() {
    void run(int guess) {
     guesses++;
     if (guess != number)
      page = page.setContentOfID("MSG", [[
       That is not correct. Try a
       <b><{guess>number?"lower":"higher"}></b>
       number:
      ]]);
     else {
      boolean record =
         game.getHolder() != null &&
         guesses >= game.getRecord();
      final XML thanks = [[
       <p>
        Thank you for playing
        this exciting game!
       </p>
      ]];
      page = wrapper.plug("BODY", [[
       <p>
        You got it, using
        <b><{guesses}></b> guesses.
       </p>
       <{ !record ? thanks : [[
         <p>
          That makes you the new
          record holder!
         </p>
         <p>
          Please enter your name for
          the hi-score list:
         </p>
```

```
        <form method="post" action=[RECORD]>
         <p>
          <input name="name" type="text" size="20"/>
          <input type="submit" value="continue"/>
         </p>
        </form>
       ]].plug("RECORD", new SubmitHandler() {
        void run(String name) {
         synchronized (GuessingGame.class) {
          GameState game = GameState.load();
          if (guesses < game.getRecord())
           game.setRecord(guesses, name);
         }
         page = wrapper.plug("BODY", thanks);
        }
       })
      }>
     ]]);
    }
  };
 });
}

public URL start() {
 GameState.load().incrementPlays();
 return makeURL("play", new UserState());
}

public XML play(UserState s) {
 return s.page;
}

public XML record() {
 GameState game = GameState.load();
 return wrapper.plug("BODY", new XMLProducer(game) {
  XML run() {
   synchronized (GuessingGame.class) {
    if (game.getHolder() != null)
     return [[
      <p>
       In <{game.getPlays()}> plays of this game,
       the record holder is <b><{game.getHolder()}></b>
       with <b><{game.getRecord()}></b> guesses.
      </p>
     ]];
    else
     return [[
      <p>
       <{game.getPlays()}> plays started.
       No players finished yet.
      </p>
     ]];
    }
   }
  });
 }
}}
```

# Chapter 7

# HTML Validation of Context-Free Languages

This chapter contains the paper "HTML Validation of Context-Free Languages" [60]. The paper appeared on the 14th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2011).

Compared to the published version, the version that is included in this chapter includes an additional section containing proofs.

# HTML Validation of Context-Free Languages

Anders Møller Mathias Schwarz

**Abstract**

We present an algorithm that generalizes HTML validation of individual documents to work on context-free sets of documents. Together with a program analysis that soundly approximates the output of Java Servlets and JSP web applications as context-free languages, we obtain a method for statically checking that such web applications never produce invalid HTML at runtime. Experiments with our prototype implementation demonstrate that the approach is useful: On 6 open source web applications consisting of a total of 104 pages, our tool finds 64 errors in less than a second per page, with 0 false positives. It produces detailed error messages that help the programmer locate the sources of the errors. After manually correcting the errors reported by the tool, the soundness of the analysis ensures that no more validity errors exist in the applications.

## 7.1 Introduction

An HTML document is *valid* if it syntactically conforms to a DTD for one of the versions of HTML. Since the HTML specifications only prescribe the meaning of valid documents, invalid HTML documents are often rendered differently, depending on which browser is used [9]. For this reason, careful HTML document authors validate their documents, for example using the validation tool provided by W3C[1]. An increasing number of HTML documents are, however, produced dynamically by programs running on web servers. It is well known that errors caught early in development are cheaper to fix. Our goal is to develop a program analysis that can check statically, that is, at the time programs are written, that they will never produce invalid HTML when running. We want this analysis to be *sound*, in the sense that whenever it claims that the given program has this property that is in fact the case, *precise* meaning that it does not overwhelm the user with spurious warnings about potential invalidity problems, and *efficient* such that it can analyze non-trivial applications with modest time and space resources. Furthermore, all warning messages being produced must be *useful* toward guiding the programmer to the source of the potential errors.

The task can be divided into two challenges: 1) Web applications typically generate HTML either by printing page fragments as strings to an output stream (as in e.g. Java Servlets) or with template systems (as e.g. JSP, PHP, or ASP). In any case, the analysis front-end must extract a formal description of the set of possible outputs of the application, for example in the form of a context-free grammar. 2) The analysis back-end must analyze this formal description of the output to check that all strings that it represents are valid HTML. Several existing techniques follow this pattern, although considering XHTML instead of HTML [40, 56]. In

[1] `http://validator.w3.org`

practice, however, many web applications output HTML data, not XHTML data, and the existing techniques – with the exception of the work by Nishiyama and Minamide [66], which we discuss in Section 7.2 – do not work for HTML.

The key differences between HTML and XHTML are that the former allows certain tags to be omitted, for example the start tags `<html>` and `<tbody>` and the end tags `</html>` and `</p>`, and that it uses tag inclusions and exclusions, for example to forbid deep nesting of `a` elements. This extra flexibility of HTML is precisely what makes it popular, compared to its XML variant XHTML. On the other hand, this flexibility means that the process of checking *well-formedness*, i.e. that a document defines a proper tree structure, cannot be separated from the process of checking *validity*, i.e. that the tree structure satisfies the requirements of the DTD.

In this paper, we present an algorithm that, given as input a context-free grammar $G$ and an SGML DTD $D$ (one of the DTDs that exist for the different versions of HTML[2]), checks whether every string in the language of $G$ is valid according to $D$, written $\mathcal{L}(G) \subseteq \mathcal{L}(D)$. The key idea in our approach is a generalization of a core algorithm for SGML parsing [24, 86] to work on context-free sets of documents rather than individual documents.

### 7.1.1 Outline of the Paper

The paper is organized as follows. We first give an overview of related approaches in Section 7.2. In Section 7.3 we then present a formal model of SGML/HTML validation that captures the essence of the features that distinguish it from XML/XHTML validation. Based on this model, in Section 7.4 we present our generalization for validating context-free sets of documents. We have implemented the algorithm together with an analysis front-end for Java Servlets and JSP, which constitute a widely used platform for server-based web application development. (Due to the limited space we focus on the back-end in this paper.) In Section 7.5, we report on experiments on a range of open source web applications. Our results show that the algorithm is fast and able to pinpoint programming errors. After manually correcting the errors based on the messages generated by the tool, the analysis is able to prove that the output will always be valid HTML when the applications are executed.

### 7.1.2 Example

Figure 7.1 shows an example of a JSP program that outputs a dynamically generated table from a list of data using a combination of many of the JSP and SGML features that appear in typical applications. The `meta` element is not part of the content model of `head`, but it is allowed by an SGML inclusion rule. The `body` element contains a table where both the start and the end tag of the `tbody` element are omitted, and a parser needs to insert those to validate a generated document. Similarly, all `td` and `th` end tags are omitted. The contents of the table are generated by a combination of tags from JSP Standard Tag Library, embedded Java code that prints to the output stream, and ordinary JSP template code.

The static analysis that we present is able to soundly check that the output from such code is always valid according to e.g. the HTML 4.01 Transitional specification.

[2]The HTML 5 language currently under development will likely evoke renewed interest in HTML. Although it technically does not use SGML, its syntax closely resembles that of the earlier versions.

```
1  <%@ page import="java.util.*, org.example" %>
2  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
3  <html><head><meta name="description" content="Joke Collection">
4    <title>Jokes</title>
5    <%! List<Joke> js = Jokes.get();%>
6    <body><table>
7      <tr><th>Question<th>Punch line</tr>
8        <% if (js.size() > 0) {
9          request.setParameter("Jokes", js); %>
10         <c:forEach items="${Jokes}" var="joke">
11           <tr><td><c:out value="${joke.question}"/>
12             <td><c:out value="${joke.punchline}"/></tr>
13         </c:forEach>
14       <% } else {
15         out.print("<td>No more jokes</tr>");
16       } %>
17   </table></body>
18 </html>
```

Figure 7.1: A JSP page that uses the JSTL tag library and embedded Java code. The example takes advantage of SGML features such as tag omission and inclusions.

## 7.2 Related Work

Previous work on reasoning about programs that dynamically generate semi-structured data has focused on XML [57], not SGML, despite the fact that the SGML language HTML remains widely used. (Since XML languages are essentially the subclass of SGML languages that do not use the tag omission and exception features, our algorithm also works for XML.) Most closely related to our approach is the work by Minamide et al. [55, 56, 66] and Kirkegaard and Møller [40].

In [55] context-free grammars are derived from PHP programs. From such a grammar, sample documents are derived and processed by an ordinary HTML or XHTML validator. Unless the nesting depth of the elements in the generated documents is bounded, this approach is unsound as it may miss errors. Later, an alternative grammar analysis was suggested for soundly validating dynamically generated XML data [56]. That algorithm relies on the theory of balanced grammars over an alphabet of tag names, which does not easily generalize to handle the tag omission and inclusion/exclusion features that exist in HTML. The approach in [40] is comparable to [56], however considering the more fine-grained alphabet of individual Unicode characters instead of entire tag names and using XML graphs for representing sets of XML documents.

Yet another grammar analysis algorithm is presented by Nishiyama and Minamide [66]. They define a subclass of SGML DTDs that includes HTML and shows a translation into regular hedge grammars, such that the validation problem reduces to checking inclusion of a context-free language in a regular language. That approach has some limitations, however: 1) it does not support start tag omission, although that feature of SGML is used in HTML (e.g. `tbody` and `head`); 2) the exclusion feature is handled by a transformation of the DTD that may lead to an exponential blow-up prohibiting practical use; and 3) the inclusion feature is not supported. The alternative approach we suggest overcomes all these limitations.

The abstract parsing algorithm by Doh et al. [16] and the grammar-based analysis by Thiemann [82] are also based on the idea of generalizing existing parsing algorithms. The approach in [16] relies on abstract interpretation with a domain of LR($k$) parse stacks constructed from an LR($k$) grammar for XHTML, and [82] is based on Earley's parsing algorithm. By instead using SGML parsing as a starting point, we avoid the abstraction and we handle the special features of HTML:

Given a context-free grammar describing the output of a program, our algorithm for checking that all derivable strings are valid HTML is both sound and complete.

## 7.3 Parsing HTML Documents

Although HTML is based on the SGML standard [24] it uses only a small subset of the features of the full standard. SGML languages are formally described using the DTD language (not to confuse with the DTD language for XML). Such a description provides a formal description for the parser on how a document is parsed from its textual form into a tree structure. Specifically, in SGML both start and end tags may be omitted if 1) allowed by the DTD, and 2) the omission does not result in ambiguities in the parsing of the document. The DTD description provides the content models, that is, the allowed children of each element, as deterministic regular expressions over sequences of elements. Furthermore special exceptions, called inclusions and exclusions, are possible for allowing additional element children or disallowing nesting of certain elements. An inclusion rule permits elements anywhere in the descendant tree even if not allowed by the content model expressions. Conversely, an exclusion rule prohibits elements, overriding the content model expressions and inclusions.

Consider a small example DTD:

```
<!ELEMENT inventory - - (item*) +(note)>
<!ELEMENT item - O (#PCDATA)>
<!ELEMENT note - O (#PCDATA)>
```

In each element declaration, `O` means "optional" and `-` means "required", for the start tag and the end tag, respectively. This DTD declares an element `inventory` where the start and end tags are both required. (Following the usual SGML terminology, an *element* generally consists of a start tag and its matching end tag, although certain tags may be omitted in the textual representation of the documents.) The content model of `inventory` allows a sequence of `item` elements as children in the document tree. In addition, `note` is included such that `note` elements may be descendants of `inventory` elements even though they are not allowed directly in the content models of the descendants. The second line declares an element `item` that requires a start tag but allows omission of the end tag. The content model of `item` allows only text (PCDATA) and no child elements in the document tree. Finally, the element `note` is also declared with end tag omission and PCDATA content. An example of a valid document for this DTD is the following:

```
<inventory><item>gadget<item>widget</inventory>
```

The parser inserts the omitted end tags for `item` to obtain the following document, which is valid according to the DTD content models for `inventory` and `item`:

```
<inventory><item>gadget</item><item>widget</item></inventory>
```

Because of the inclusion of `note` elements in the declaration of `inventory`, the following document is also parsed as a valid instance:

```
<inventory><item>gadget<note>new</note><item>widget</inventory>
```

SGML is similar to XML but it has looser requirements on the syntax of the input documents. For the features used by HTML, the only relevant differences are that XML does not support tag omissions nor content model exceptions.

We consider only DTDs that are acyclic:

**Definition 7.1.** *An SGML DTD is* acyclic *if it satisfies the following requirement: For elements that allow end tag omissions there must be a bound on the possible depth of the direct nesting of those elements. That is, if we create a directed graph where the nodes correspond to the declared elements whose end tags may be omitted and there is an edge from a node A to a node B if the content model of A contains B, then there must be no cycles in this graph.*

This requirement also exists in Nishiyama and Minamide's approach [66], and it is fulfilled by all versions of the HTML DTD. Contrary to their approach we do not impose any further restrictions and our algorithm thus works for all the HTML DTDs without any limitations or rewritings.

### 7.3.1 A Model of HTML Parsing

As our algorithm is a generalization of the traditional SGML parsing algorithm we first present a formal description of the essence of that algorithm. We base our description on the work by Warmer and van Egmond [86]. The algorithm provides the basis for explaining our main contribution in the next section.

We abstract away from SGML features such as text (i.e. PCDATA), comments, and attributes. These features are straightforward to add subsequently. Furthermore, a lexing phase allows us to consider strings over the alphabet of start and end tags, written <*a*> and </*a*>, respectively, for every element $a$ declared in the DTD. (This lexing phase is far from trivial; our implementation is based on the technique used in [40], and we omit the details here due to the limited space.) More formally, we consider strings over the alphabet $\Sigma = \{\texttt{<}a\texttt{>} \mid a \in \mathcal{E}\} \cup \{\texttt{</}a\texttt{>} \mid a \in \mathcal{E}\}$ where $\mathcal{E}$ is the set of declared element names in the DTD. We assume that $\texttt{root} \in \mathcal{E}$ is a pseudo-element representing the root node of the document, with a content model that accepts a single element of any kind (or, one specific, such as `html` for HTML). The sets of included and excluded elements of an element $a \in \mathcal{E}$ are denoted $I_a$ and $E_a$, respectively.

For simplicity, we represent all content models together as one finite-state automaton [29] defined as follows:

**Definition 7.2.** *A* content model automaton *for a DTD D is a tuple* $(Q, \mathcal{E}, [q_a]_{a\in\mathcal{E}}, F, \delta)$ *where Q is a set of states, its alphabet is* $\mathcal{E}$ *as defined above,* $[q_a]_{a\in\mathcal{E}}$ *is a family of initial states (one for each declared element),* $F \subseteq Q$ *is a set of accept states and* $\delta : Q \times \Sigma \hookrightarrow Q$ *is a partial transition function (with* $\perp$ *representing undefined).*

Following the requirement from the SGML standard that content models must be unambiguous, this content model automaton can be assumed to be deterministic by construction. Also, we assume that all states in the automaton can reach some accept state. Each state in the automaton uniquely corresponds to a position in a content model expression in $D$.

SGML documents are parsed in a single left-to-right scan with a look-ahead of 1. The state of the parser is represented by a *context stack*. The set of possible contexts is $\mathcal{H} = \mathcal{E} \times Q \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{E})$. ($\mathcal{P}(\mathcal{E})$ denotes the powerset of $\mathcal{E}$.) We refer to the context $c_n = (a, q, \iota, \eta)$ at the top of a stack $c_1 \cdots c_n \in \mathcal{H}^*$ as the *current context*, and $a$, $q$, $\iota$, and $\eta$ are then the *current element*, the *current state*, the *current inclusions*, and the *current exclusions*, respectively. An element $b$ is *permitted* in the current context $(a, q, \iota, \eta)$ if $\delta(q, b) \neq \perp$. We refer to a tag $a$ just below another tag $b$ in the context stack as $b$'s *parent*. We say that $\textsc{OmitStart}(a, q)$ holds if the start tag of $a$ elements may be omitted according to $D$ when the current state is $q$, and, similarly, $\textsc{OmitEnd}(a, q)$ holds if the end tag of $a$ elements may be omitted in state $q$. (The precise rules defining OmitStart and OmitEnd from $D$ are

1. **function** $Parse_D(p \in \mathcal{H}^*,\ x \in \Sigma^*)$ :
2. **if** $|x| = 0$ **then**
3. *// reached end of input*
4. **return** $p$
5. **else if** $|p| = 0$ **then**
6. *// empty stack error*
7. **return** $\bigcirc$
8. **let** $p_1 \cdots p_{n-1} \cdot (a_n, s_n, \iota_n, \eta_n) = p$
9. **let** $x_1 \cdots x_m = x$
10. **if** $x_1 =$ <a> $\wedge\ a \notin \eta_n$ for some $a \in \mathcal{E}$ **then**
11. *// reading a non-excluded start tag*
12. **if** $\delta(s_n, a) \neq \bot$ **then**
13. *// the start tag is permitted by the content model, push onto stack and proceed*
14. **return** $Parse_D\big(p_1 \cdots p_{n-1} \cdot (a_n, \delta(s_n, a), \iota_n, \eta_n) \cdot (a, q_a, \iota_n \cup I_a, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
15. **else if** $a \in \iota_n$ **then**
16. *// the start tag is permitted by inclusion, push onto stack and proceed*
17. **return** $Parse_D\big(p_1 \cdots p_n \cdot (a, q_a, \emptyset, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
18. **else if** $x_1 =$ </a> $\wedge\ a = a_n \wedge s_n \in F$ for some $a \in \mathcal{E}$ **then**
19. *// reading an end tag that is permitted, pop from stack and proceed*
20. **return** $Parse_D\big(p_1 \cdots p_{n-1},\ x_2 \cdots x_m\big)$
21. **else if** OmitEnd$(a_n, s_n)$ **then**
22. *// insert omitted end tag, then retry*
23. **return** $Parse_D(p,$ </$a_n$> $\cdot\ x)$
24. **else if** $\exists a' \in \mathcal{E}$ : OmitStart$(a', s_n)$ **then**
25. *// insert omitted start tag, then retry*
26. **return** $Parse_D(p,$ <$a'$> $\cdot\ x)$
27. **else**
28. *// parse error*
29. **return** $\lightning$

Figure 7.2: The $Parse_D$ function for checking validity of a given document.

quite complicated; we refer to [24, 86] for the details.) The current inclusions and exclusions reflect the sets of included and excluded elements, respectively. These two sets can in principle be determined from the element names appearing in the context stack, but we maintain them in each context for reasons that will become clear in Section 7.4.

Informally, when encountering a start tag <$a$> that is permitted in the current context, its content automaton state is modified accordingly, and a new context is pushed onto the stack. When an end tag </$a$> is encountered, the current context is popped off the stack if it matches the element name $a$.

An end tag may be omitted only if it is followed by either the end tag of another open element or a start tag that is not allowed at this place. A start tag may be omitted only if omission does not cause an ambiguity during parsing. These conditions, which define OmitEnd and OmitStart, can be determined from the current state and either the next tag in the input or the current element on the stack, respectively, without considering the rest of the parse stack and input. Moreover, OmitStart has the property that no more than $|\mathcal{E}|$ omitted start tags can be inserted before the next tag from the input is consumed.

Our formalization of SGML parsing is expressed as the function $Parse_D : \mathcal{H}^* \times \Sigma^* \to \big(\mathcal{H}^* \cup \{\lightning, \bigcirc\}\big)$ shown in Figure 7.2. The result $\bigcirc$ arises if an end tag is encountered while the stack is empty, and $\lightning$ represents other kinds of parse errors. In this algorithm, OmitEnd and OmitStart allow us to abstract away from the precise rules for tag omission, to keep the presentation simple. The algorithm captures an essential property of SGML parsing: a substring $x \in \Sigma^*$ of a document is parsed relative to a parse stack $p \in \mathcal{H}^*$ as defined above, and it outputs a new parse stack or one of the error indicators $\bigcirc$ and $\lightning$. We distinguish between the two kinds of errors for reasons that become clear in Section 7.4.

With this, we can define validity of a document relative to the DTD $D$:

**Definition 7.3.** *A string* $x \in \Sigma^*$ *is a* valid document *if*

$$Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), x\big) = (\mathsf{root}, q, \emptyset, \emptyset)$$

*for some* $q \in F$.

The $Parse_D$ function has some interesting properties that we shall need in Section 7.4:

**Observation 7.1.** *Notice that the* $Parse_D$ *function either returns directly or via a tail call to itself. Let* $(p^1, x^1), (p^2, x^2), \ldots$ *be the sequence of parameters to* $Parse_D$ *that appear if executing* $Parse_D(p^1, x^1)$ *for some* $p^1 \in \mathcal{H}^*, x^1 \in \Sigma^*$. *Now, because the DTD is acyclic, for all* $i = 1, 2, \ldots$ *we have* $|x^{i+|\mathcal{E}|}| < |x^i|$, *that is, after at most* $|\mathcal{E}|$ *recursive calls, one more input symbol is consumed. Moreover, in each step in the recursion sequence, the decisions made depend only on the current context and the next input symbol.*

## 7.4 Parsing Context-Free Sets of Documents

We now show that the parsing algorithm described in the previous section can be generalized to work for *sets* of documents, or more precisely, context-free languages over the alphabet $\Sigma$. The resulting algorithm determines whether or not all strings in a given language are valid according to a given DTD. The languages are represented as context-free grammars that are constructed by the analysis front-end from the programs being analyzed.

The definitions of context-free grammars and their languages are standard:

**Definition 7.4.** *A* context-free grammar *(CFG) is a tuple* $G = (N, \Sigma, P, S)$ *where* $N$ *is the set of nonterminal symbols,* $\Sigma$ *is the alphabet (of start and end tag symbols, as in Section 7.3.1),* $P$ *is the set of productions of the form* $A \to r$ *where* $A \in N$, $r \in (\Sigma \cup N)^*$, *and* $S$ *is the start nonterminal. The* language *of* $G$ *is* $\mathcal{L}(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$ *where* $\Rightarrow^*$ *is the reflexive transitive closure of the derivation relation* $\Rightarrow$ *defined by* $u_1 A u_2 \Rightarrow u_1 r u_2$ *whenever* $u_1, u_2 \in (\Sigma \cup N)^*$ *and* $A \to r \in P$.

**Definition 7.5.** *A CFG* $G$ *is* valid *if* $x$ *is valid for every* $x \in \mathcal{L}(G)$.

To simplify the presentation we will assume that $G$ is in Chomsky normal form, so that all productions are of the form $A \to s$ or $A \to A'A''$ where $s \in \Sigma$ and $A, A', A'' \in N$, and that there are no useless nonterminals. It is well-known how to transform an arbitrary CFG to this form [29]. We can disregard the empty string since that is never valid for any DTD, and the empty language is trivially valid.

The idea behind the generalization of the parse algorithm is to find out for every occurrence of an alphabet symbol $s$ in the given CFG which context stacks may appear when encountering $s$ during parsing of a string. The context stacks may of course be unbounded in general. However, because of Observation 7.1 *we only need to keep track of a bounded size top (i.e. a postfix) of each context stack*, and hence a bounded number of context stacks, at every point in the grammar.

### 7.4.1 Generating Constraints

To make the idea more concrete, we define a family of *context functions*, one for each nonterminal $A \in N$. Each is a partial function that takes as input a context stack and returns a set of context stacks:

$$\mathcal{C}_A : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*)$$

Informally, the domain of $\mathcal{C}_A$ consists of the context stacks that appear during parsing when entering a substring derived from $A$, and the co-domain similarly consists of the context stacks that appear immediately after the substring has been parsed. Formally, assume $x \in \mathcal{L}(G)$ such that $S \Rightarrow^* u_1 A u_2 \Rightarrow^* u_1 y u_2 = x$, that is, the nonterminal $A$ is used in the derivation of $x$, and $y$ is the substring derived from $A$. The domain $dom(\mathcal{C}_A)$ then contains the context stack $p$ that arises after parsing of $u_1$, that is, $p = Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1\big) \in dom(\mathcal{C}_A)$. Similarly, $\mathcal{C}_A(p)$ contains the context stack that arises after parsing of $u_1 y$, that is, $Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1 y\big) = Parse_D(p, y) \in \mathcal{C}_A(p)$ if $p \notin \{\lightning, \bigcirc\}$. As explained in detail below, we truncate the context stacks and only store the top of the stacks in these sets. To obtain an efficient algorithm, we truncate as much as possible and exploit the fact that $Parse_D$ returns $\bigcirc$ if a too short context stack is given.

The context functions are defined from the DTD as a solution to the set of constraints defined by the following three rules:

§1 Following Definition 7.3, parsing starts with the initial context stack at the start nonterminal $S$ and must end in a valid final stack:

$$\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$$

§2 For every production of the form $A \to s$ in $P$ where $s \in \Sigma$, the context function for $A$ respects the $Parse_D$ function, which must not return $\lightning$ or $\bigcirc$:

$$\forall p \in dom(\mathcal{C}_A) : p' \notin \{\lightning, \bigcirc\} \ \wedge \ p' \in \mathcal{C}_A(p) \text{ where } p' = Parse_D(p, s)$$

§3 For every production of the form $A \to A'A''$ in $P$, the entry context stacks of $A$ are also entry context stacks for $A'$, the exit context stacks for $A'$ are also entry context stacks for $A''$, and the exit context stacks for $A''$ are also exit context stacks for $A$. However, we allow the context stacks to be truncated when propagated from one nonterminal to the next:

$$\begin{aligned}\forall p \in dom(\mathcal{C}_A) : \exists p_1, p_2 : p = p_1 \cdot p_2 \ \wedge \ p_2 \in dom(\mathcal{C}_{A'}) \wedge \\ \forall p_2' \in \mathcal{C}_{A'}(p_2) : \exists t_1, t_2 : p_1 \cdot p_2' = t_1 \cdot t_2 \ \wedge \ t_2 \in dom(\mathcal{C}_{A''}) \wedge \\ \forall t_2' \in \mathcal{C}_{A''}(t_2) \Rightarrow t_1 \cdot t_2' \in \mathcal{C}_A(p)\end{aligned}$$

Note that rule §3 permits the context stacks to be truncated; on the other hand, rule §2 ensures that the stacks are not truncated too much since that would lead to the error value $\bigcirc$.

**Theorem 7.1.** *There exists a solution to the constraints defined by the rules above for a grammar $G$ if and only if $G$ is valid.*

*Proof.* See the appendix. □

### 7.4.2 Solving Constraints

It is relatively simple to construct an algorithm that searches for a solution to the collection of constraints generated from a CFG by the rules defined in Section 7.4.1. Figure 7.3 shows the pseudo-code for such an algorithm, $ParseCFG_D$. We write $w$ DEFS $A$ for $w \in P$, $A \in N$ if $A$ appears on the left-hand side of $w$, and $w$ USES $A$ if $A$ appears on the right-hand side of $w$. The solution being constructed is represented by the family of context functions, denoted $[\mathcal{C}_A]_{A \in N}$.

The idea in the algorithm is to search for a solution by truncating the context stacks as much as possible, iteratively trying longer context stacks, until the special error value $\bigcirc$ no longer appears. The algorithm initializes $[\mathcal{C}_A]_{A \in N}$ on line 6 and

1. **function** $ParseCFG_D(N, \Sigma, P, S)$ :
2. **declare** $W \subseteq P,\ [\mathcal{C}_A]_{A \in N} : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*),\ \Delta : N \to \mathcal{P}(\mathcal{H}^*)$
3. *// initialize worklist and context functions*
4. $W := [\ w \in P \mid w \text{ DEFS } S\ ]$
5. **for all** $A \in N$, $p \in \mathcal{H}^*$ **do**
6. $\quad \mathcal{C}_A(p) := \begin{cases} \emptyset & \text{if } A = S \ \wedge\ p = (\mathsf{root}, q_{\mathsf{root},\emptyset,\emptyset}) \\ \bot & \text{otherwise} \end{cases}$
7. $\quad \Delta(A) := \emptyset$
8. *// iterate until fixpoint*
9. **while** $W \neq \emptyset$ **do**
10. $\quad$ **remove** the next production $A \to r$ from $W$
11. $\quad$ **for all** $p \in dom(\mathcal{C}_A)$ **do**
12. $\quad\quad$ **if** $A \to r$ is of the form $A \to s$ where $s \in \Sigma$ **then**
13. $\quad\quad\quad$ *// rule §2*
14. $\quad\quad\quad$ **let** $p' = Parse_D(p, s)$
15. $\quad\quad\quad$ **if** $p' = \bigcirc$ **then**
16. $\quad\quad\quad\quad$ *// record that entry context stack p is too short for A*
17. $\quad\quad\quad\quad \Delta(A) := \Delta(A) \cup \{p\}$
18. $\quad\quad\quad\quad \mathcal{C}_A(p) := \bot$
19. $\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ USES $A$ **add** $w$ to $W$
20. $\quad\quad\quad$ **else if** $p' = \lightning$ **then**
21. $\quad\quad\quad\quad$ *// fail right away*
22. $\quad\quad\quad\quad$ **fail**
23. $\quad\quad\quad$ **else if** $p' \notin \mathcal{C}_A(p)$ **then**
24. $\quad\quad\quad\quad$ *// add new final context stack p′ for A*
25. $\quad\quad\quad\quad \mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{p'\}$
26. $\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ USES $A$ **add** $w$ to $W$
27. $\quad\quad$ **else if** $A \to r$ is of the form $A \to A'A''$ where $A', A'' \in N$ **then**
28. $\quad\quad\quad$ *// rule §3*
29. $\quad\quad\quad$ **let** $p_2$ be the smallest string such that $p = p_1 \cdot p_2$ and $p_2 \notin \Delta(A')$
30. $\quad\quad\quad$ **if** no such $p_2$ exists **then**
31. $\quad\quad\quad\quad$ *// record that entry context stack p is too short for A*
32. $\quad\quad\quad\quad \Delta(A) := \Delta(A) \cup \{p\}$
33. $\quad\quad\quad\quad \mathcal{C}_A(p) := \bot$
34. $\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ USES $A$ **add** $w$ to $W$
35. $\quad\quad\quad$ **else if** $p_2 \in dom(\mathcal{C}_{A'})$ **then**
36. $\quad\quad\quad\quad$ **for all** $p_2' \in \mathcal{C}_{A'}(p_2)$ **do**
37. $\quad\quad\quad\quad\quad$ **let** $t_2$ be the smallest string such that $p_1 \cdot p_2' = t_1 \cdot t_2$ and $t_2 \notin \Delta(A'')$
38. $\quad\quad\quad\quad\quad$ **if** no such $t_2$ exists **then**
39. $\quad\quad\quad\quad\quad\quad$ *// record that entry context stack p is too short for A*
40. $\quad\quad\quad\quad\quad\quad \Delta(A) := \Delta(A) \cup \{p\}$
41. $\quad\quad\quad\quad\quad\quad \mathcal{C}_A(p) := \bot$
42. $\quad\quad\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ USES $A$ **add** $w$ to $W$
43. $\quad\quad\quad\quad\quad$ **else if** $t_2 \in dom(\mathcal{C}_{A''})$ **then**
44. $\quad\quad\quad\quad\quad\quad$ **if** $\{t_1 \cdot t_2' \mid t_2' \in \mathcal{C}_{A''}(t_2)\} \not\subseteq \mathcal{C}_A(p)$ **then**
45. $\quad\quad\quad\quad\quad\quad\quad$ *// add new final context stacks for A*
46. $\quad\quad\quad\quad\quad\quad\quad \mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{t_1 \cdot t_2' \mid t_2' \in \mathcal{C}_{A''}(t_2)\}$
47. $\quad\quad\quad\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ USES $A$ **add** $w$ to $W$
48. $\quad\quad\quad\quad\quad\quad$ **else**
49. $\quad\quad\quad\quad\quad\quad\quad$ *// add new entry context stack $t_2$ for $A''$*
50. $\quad\quad\quad\quad\quad\quad\quad \mathcal{C}_{A''}(t_2) := \emptyset$
51. $\quad\quad\quad\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ DEFS $A''$ **add** $w$ to $W$
52. $\quad\quad\quad\quad$ **else**
53. $\quad\quad\quad\quad\quad$ *// add new entry context stack $p_2$ for $A'$*
54. $\quad\quad\quad\quad\quad \mathcal{C}_{A'}(p_2) := \emptyset$
55. $\quad\quad\quad\quad\quad$ **for all** $w \in P$ where $w$ DEFS $A'$ **add** $w$ to $W$
56. $\quad$ *// rule §1*
57. $\quad$ **if** $\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \not\subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$ **then**
58. $\quad\quad$ **fail**
59. **return** $[\mathcal{C}_A]_{A \in N}$

Figure 7.3: The $ParseCFG_D$ algorithm for solving the parse constraints for a given CFG.

iteratively on lines 9–58 extends these functions to build a solution. The worklist $W$ (a queue, without duplicates) consists of productions that need to be processed because the domains of the context functions of their left-hand-side nonterminals have changed. The function $\Delta$ maintains for each nonterminal a set of context

stacks that are known to lead to $\bigcirc$.

Each production in the worklist of the form $A \to s$ is parsed according to rule §2 on lines 14– 26, relative to each context stack $p$ in $dom(\mathcal{C}_A)$. If this results in $\bigcirc$, the corresponding context stack is added to $\Delta(A)$, and all productions that use $A$ are added to the worklist to make sure that the information that the context stack was too short is propagated back to those productions. If a parse error $\lightning$ occurs (line 20), the algorithm terminates with a failure. If the parsing is successful (line 23), the resulting context stack $p'$ is added to $\mathcal{C}_A$.

For a production of two nonterminals, $A \to A'A''$, we proceed according to rule §3. For each context stack $p$ in $dom(\mathcal{C}_A)$ on line 29 we pick the smallest possible postfix $p_2$ of $p$ that is not in $\Delta(A')$ and propagate this to $\mathcal{C}_{A'}$. If no such postfix exists, we know that $p$ is too short, so we update $\Delta(A)$ and $W$ as before. Otherwise, we repeat the process (line 37) to propagate the resulting context stack through $A''$ and further to $\mathcal{C}_A$ (line 46).

Finally, on line 57 we check that rule §1 is satisfied.

**Theorem 7.2.** *The ParseCFG$_D$ algorithm always terminates, and it terminates successfully if and only if a solution exists to the constraints from Section 7.4.1 for the given CFG.*
*(We leave a proof of this theorem as future work.)*

**Corollary 7.1.** *Combining Theorem 7.1 and Theorem 7.2, we see that ParseCFG$_D$ always terminates, and it terminates successfully if and only if the given CFG is valid.*

### 7.4.3 Example

As an example of a normalized grammar, consider $G_{ul} = (N, \Sigma, P, S)$ where $N = \{A_1, A_2, A_3, A_4, A_5, A_6\}$, $\Sigma = \{\texttt{<ul>}, \texttt{</ul>}, \texttt{<li>}, \texttt{</li>}\}$, $S = A_1$, and $P$ consists of the following productions:

$$\begin{array}{ll} A_1 \to A_5\ A_2 & A_2 \to A_6\ A_3 \\ A_3 \to \texttt{</ul>} & A_4 \to \texttt{<li>} \\ A_5 \to \texttt{<ul>} & A_6 \to \texttt{<li>} \\ A_6 \to A_4\ A_1 & \end{array}$$

The language generated by $G_{ul}$ consists of documents that have a `ul` root element containing a single `li` element that in turn contains zero or one `ul` element. The grammar can thus generate deeply nested `ul` and `li` elements, and truncation of context stacks is therefore crucial for the *ParseCFG$_D$* algorithm to terminate. Notice that all `</li>` end tags are omitted in the documents.

We wish to ensure that the strings generated from $G_{ul}$ are valid relative to the following DTD, which mimics a very small fraction of the HTML DTD for unordered lists:

```
<!ELEMENT ul - - (li*)>
<!ELEMENT li - O (ul*)>
```

For this combination of a CFG and a DTD, the *ParseCFG$_D$* algorithm produces

the following solution to the constraints:

| | $\mathcal{C}$ |
|---|---|
| $A_1$ | $(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\mathsf{root}, q, \emptyset, \emptyset)\}$<br>$(\mathtt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\mathtt{li}, q_{li}, \emptyset, \emptyset)\}$ |
| $A_2$ | $(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{\epsilon\}$ |
| $A_3$ | $(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\mathtt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{\epsilon\}$ |
| $A_4$ | $(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\mathtt{li}, q_{li}, \emptyset, \emptyset)\}$ |
| $A_5$ | $(\mathtt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\mathtt{li}, q_{li}, \emptyset, \emptyset) \cdot (\mathtt{ul}, q_{ul}, \emptyset, \emptyset)\}$<br>$(\mathtt{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\mathtt{root}, q, \emptyset, \emptyset) \cdot (\mathtt{ul}, q_{ul}, \emptyset, \emptyset)\}$ |
| $A_6$ | $(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\mathtt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\mathtt{li}, q_{li}, \emptyset, \emptyset)\}$ |

Although the context stacks may grow arbitrarily when parsing individual documents with $Parse_D$, the truncation trick ensures that $ParseCFG_D$ terminates and succeeds in capturing the relevant top-most parts of the context stacks.

## 7.5 Experimental Results

We have implemented the algorithm from Section 7.4.2 in Java, together with an analysis front-end for constructing CFGs that soundly approximate the output of web applications written with Java Servlets and JSP. The front-end follows the structure described in [40], extended with specialized support for JSP, and builds on Soot [84] and the Java String Analyzer [12]. (We omit a more detailed explanation of this front-end, due to the limited space.)

The purpose of the prototype implementation is to obtain preliminary answers to the following research questions:

- What is the typical analysis time for a Servlet/JSP page, and how is the analysis time affected by the absence or presence of validity errors?
- What is the precision of the analysis in terms of false positives?
- Are the warnings produced by the tool useful to locate the sources of the errors?

We have run the analysis on six open source programs found on the web. The programs range from simple one man projects, such as the JSP Chat application (JSP Chat[3]), the official J2EE tutorial Servlet and JSP examples (J2EE Bookstore 1 and 2[4]) to the widely used blogging framework Pebble[5], which included dozens of pages and features. We have also included the largest example from a book on JSTL (JSTL Book ex.[6]) and an application named JPivot[7]. The tests have been performed on a 2.4 GHz Core i5 laptop with 4GB RAM running OS X. As DTD, we use HTML 4.01 Transitional.

Figure 7.4 summarizes the results. For each program, it shows the number of JSP pages, the time it takes to run the whole analysis on all pages (excluding the time used by Soot), the time spent in the CFG parser algorithm, the number of warnings from the analyzer, and the number of false positives determined by manual inspection of the analyzed source code.

The tool currently has two limitations, which we expect to remedy with a modest additional implementation effort. First, validation of attributes is currently not

[3] http://www.web-tech-india.com/software/jsp_chat.php
[4] http://download.oracle.com/javaee/5/tutorial/doc/bnaey.html
[5] http://pebble.sourceforge.net/
[6] http://www.manning.com/bayern/
[7] http://jpivot.sourceforge.net/

supported. Second, the implementation can track a validity error to the place in the generated Java code where the invalid element is generated, but not all the way back to the JSP source in the case of JSP pages.

Furthermore, the tool considers all uses of the SGML SHORTTAG feature as an error. The HTML specification specifically states that the use of this feature should be avoided because it is unlikely to work with existing HTML implementation.

In some cases when an unknown value is inserted into the output without escaping special XML characters (for example, by using the `out` tag from JSTL), the front-end is unable to reason about the language of that value. This may for instance happen when the value is read from disk or input at runtime. The analysis will in such cases issue an additional warning, which is not included in the count in Figure 7.4, and treat the unknown value as a special alphabet symbol and continue analyzing the grammar. In practice, there are typically a few such symbols per page. While they may be indications of cross site scripting vulnerabilities, there may also be invariants in the program ensuring that there is no problem at runtime.

The typical analysis time for a single JSP page is around 200-600 ms. As can be seen from the table, only a small fraction of the time is spent on parsing the CFG. The worklist algorithm typically requires between 1 and 100 iterations for each JSP page, which means that each nonterminal is visited between 1 and 10 times.

Validity errors were found in all the applications. The following is an example of a warning generated by the tool on the JSP Chat application:

```
ERROR: Invalid string printed in
  dk.brics.servletvalidator.jsp.generated.editInfo_jsp on line 94:
Start tag INPUT not allowed in TBODY
Parse context is [root HTML BODY DIV CENTER FORM TABLE TBODY]
```

This warning indicates that the programmer forgot both a `tr` start tag and a `td` start tag in which the `input` element would be allowed, causing the `input` tag to appear directly inside the `tbody` element. This may very well lead to browsers rendering the page differently.

The reason that all JSP pages of the J2EE Bookstore applications are invalid it that there is an unmatched `</center>` tag and a nonstandard `<comment>` tag in a header used by all pages. After removing these two tags, only one page of this application is (correctly) rejected by the analysis. While Pebble seems to be programmed with the goal of only outputting valid HTML, the general problem in this web application is that the `table`, `ul`, and `tr` elements require non-empty contents, which is not always respected by Pebble. Furthermore, several more serious errors, such as forgotten `td` tags, exist in the application. The JSP Chat application is written in JSP but makes heavy use of embedded Java code. The tool is able to analyze it precisely enough to find several errors that are mostly due to unobvious (but feasible) flow in the program.

Based on the warnings generated by the tool, we managed to manually correct all the errors within a few hours without any prior knowledge of the applications. After running the analysis again, no more warnings were produced. This second round of analysis took essentially the same time as before the errors were corrected. Since the analysis is sound, we can trust that the applications after the corrections cannot output invalid HTML.

## 7.6 Conclusion

We have presented an algorithm for validating context-free sets of documents relative to an HTML DTD. The key idea – to generalize a parsing algorithm for SGML to work on grammars instead of concrete documents – has lead to an approach that

| Program | Pages | Time | CFG Parser time | Warnings | False positives |
|---|---|---|---|---|---|
| Pebble[3] | 61 | 24.0 s | 369 ms | 32 | 0 |
| J2EE Bookstore 1[4] | 5 | 6.7 s | 93 ms | 5 | 0 |
| J2EE Bookstore 2[4] | 7 | 9.0 s | <1 ms | 7 | 0 |
| JPivot[5] | 3 | 2.8 s | 8 ms | 1 | 0 |
| JSP Chat[6] | 15 | 6.8 s | 100 ms | 10 | 0 |
| JSTL Book ex.[7] | 14 | 4.9 s | 24 ms | 6 | 0 |

Figure 7.4: Analysis times and results for various open source web applications written in Java Servlets and JSP.

smoothly handles the intricate features of HTML, in particular tag omissions and exceptions. Preliminary experiments with our prototype implementation indicate that the approach is sufficiently efficient and precise to function as a practically useful tool during development of web applications. In future work, we plan to improve the tool to accommodate for attributes and to trace error messages all the way back to the JSP source (which is tricky because of the JSP tag file mechanism) and to perform a more extensive evaluation.

## 7.7 Proof of Theorem 7.1

We begin with some lemmas and a proposition that help us give a simple proof of the theorem. The first lemma shows a compositionality property of the $Parse_D$ function:

**Lemma 7.1.** *Given $p \in \mathcal{H}^*$ and $x_1, x_2 \in \Sigma^*$, let $p' = Parse_D(p, x_1)$. If $p' \in \{\lightning, \bigcirc\}$ then $Parse_D(p, x_1x_2) = p'$; otherwise $Parse_D(p, x_1x_2) = Parse_D(p', x_2)$.*

Another property of $Parse_D$ is that providing a larger context stack cannot lead to more parse errors:

**Lemma 7.2.** *Given $p_1, p_2 \in \mathcal{H}^*$ and $x \in \Sigma^*$, let $p' = Parse_D(p_2, x)$. If $p' \neq \bigcirc$ then $Parse_D(p_1 \cdot p_2, x) = p'$.*

(We omit the proofs of these lemmas.)

We henceforth abbreviate the initial context stack by $\diamond$:

$$\diamond = (\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset)$$

The following proposition captures the essential properties that were described intuitively in Section 7.4.1 of solutions to the context function constraints:

**Proposition 7.1.** *Assume $x \in \mathcal{L}(G)$ and $[\mathcal{C}_A]_{A \in N}$ satisfies the constraints from Section 7.4.1 for a given CFG $G$. Let $A$ be a nonterminal used in a derivation of $x$ such that $S \Rightarrow^* u_1 A u_2 \Rightarrow^* u_1 y u_2 = x$ for some $u_1, y, u_2 \in \Sigma^*$. Now, $[\mathcal{C}_A]_{A \in N}$ has the following properties:*

*(a) Let $p = Parse_D(\diamond, u_1)$. If $p \notin \{\lightning, \bigcirc\}$ then there exist $p_1, p_2 \in \mathcal{H}^*$ such that $p = p_1 \cdot p_2$ and $p_2 \in dom(\mathcal{C}_A)$. That is, $dom(\mathcal{C}_A)$ contains a postfix $p_2$ of the context stack that arises after parsing $u_1$, unless a parse error has occurred.*

*(b) Let $p' = Parse_D(p_2, y)$ for some $p_2 \in dom(\mathcal{C}_A)$. If $p' \notin \{\lightning, \bigcirc\}$ then $p' \in \mathcal{C}_A(p_2)$. That is, $\mathcal{C}_A(p_2)$ contains the context stack that arises after parsing $y$ if starting in the context stack $p_2$ and no parse error occurs.*

*(In fact, as we show later, parse errors cannot occur when there exists a solution to the constraints.)*

*Proof.* Consider a left-to-right depth-first traversal of a derivation tree of $x$ where we visit each node (corresponding to a terminal or a nonterminal) both on the way down and the way up. We now show by induction in $k = 0, 1, 2, \ldots$ that (1) all the nonterminal nodes that have been visited on the way down (and maybe also on the way up) after the first $k$ steps of this traversal have property (a), and (2) all the nonterminal nodes that have been visited both on the way down and on the way up after the first $k$ steps of this traversal have property (b).

For the base case, $k = 0$, we only need to show that $\diamond \in dom(\mathcal{C}_A)$, however this follows directly from rule §1 (see Section 7.4.1).

For the induction step, $k > 0$, if the node being visited is a terminal, our goal follows immediately from the induction hypothesis. If the node is instead a nonterminal, we split into two cases: either the $k$'th step is downward or it is upward. If it is downward, we are visiting a nonterminal node $A'$ with a right sibling $A''$ and a parent $A$, or a nonterminal node $A''$ with a left sibling $A'$ and a parent $A$, corresponding to a production on the form $A \to A'A''$. We need to show that the new node being visited satisfies property (a). Now, $u_1$ is the string formed by the sequence of terminals visited so far. By the induction hypothesis, $dom(\mathcal{C}_A)$ contains

a postfix of $Parse_D(\diamond, u_1)$, and by the first line of rule §3, $dom(\mathcal{C}_{A'})$ thereby also contains a postfix of $Parse_D(\diamond, u_1)$, hence property (a) is satisfied. The case for $A''$ is similar, using the second line of rule §3. If the $k$'th step is instead upward, we need to show that the new node being visited satisfies property (b). (Property (a) follows immediately from the induction hypothesis.) Let $A$ be the nonterminal of the node. Either the node has a single child, corresponding to a production of the form $A \to s$, or two children, corresponding to a production of the form $A \to A'A''$. In the former case, property (b) follows from rule §2; in the latter case, we use rule §3. □

The proof of Theorem 7.1 has two parts:

*1.* We first show that the CFG $G$ is valid (according to Definition 7.5) if there exists some $[\mathcal{C}_A]_{A \in N}$ that satisfies the constraints from Section 7.4.1.

Let $x \in \mathcal{L}(G)$ and $p' = Parse_D(\diamond, x)$. From rule §1 we know that $\diamond \in dom(\mathcal{C}_S)$. By part (b) of Proposition 7.1, either $p' \in \{\lightning, \bigcirc\}$ or $p' \in \mathcal{C}_S(\diamond)$. However, $p' \in \{\lightning, \bigcirc\}$ is not possible. To see this, assume $p' \in \{\lightning, \bigcirc\}$ and let $x = s_1 s_2 \cdots s_n$ where $s_1, s_2, \ldots, s_n \in \Sigma$. Then there exists a position $i \in \{1, \ldots, n\}$ such that $p'' = Parse_D(\diamond, s_1 s_2 \cdots s_{i-1}) \notin \{\lightning, \bigcirc\}$ and $Parse_D(\diamond, s_1 s_2 \cdots s_i) \in \{\lightning, \bigcirc\}$. Let $A$ be the nonterminal that derives $s_i$ in $x$. Part (a) of Proposition 7.1 now tells us that $dom(\mathcal{C}_A)$ contains a postfix $p_2''$ of $p''$, and by rule §2, $Parse_D(p_2'', s_i) \notin \{\lightning, \bigcirc\}$. Lemma 7.2 then gives us that $Parse_D(p'', s_i) \notin \{\lightning, \bigcirc\}$. By Lemma 7.1, $Parse_D(\diamond, s_1 s_2 \cdots s_i) = Parse_D(p'', s_i) \notin \{\lightning, \bigcirc\}$, which contradicts $Parse_D(\diamond, s_1 s_2 \cdots s_i) \in \{\lightning, \bigcirc\}$. Thus, $p' \notin \{\lightning, \bigcirc\}$, so $p' \in \mathcal{C}_S(\diamond)$. By rule §1, $\mathcal{C}_S(\diamond) \subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$, so $p' = (\mathsf{root}, q, \emptyset, \emptyset)$ for some $q \in F$, which means that $x$ is valid according to Definition 7.3.

*2.* Next, we show conversely that validity of $G$ implies that a (not necessarily finite) solution exists to the constraints.

Assume $G$ is valid. Construct $[\mathcal{C}_A]_{A \in N}$ as follows, for each $A \in N$:

$$dom(\mathcal{C}_A) = \bigcup_{S \Rightarrow^* u_1 A u_2 \text{ where } u_1, u_2 \in \Sigma^*} Parse_D(\diamond, u_1)$$

$$\mathcal{C}_A(p) = \bigcup_{A \Rightarrow^* y \text{ where } y \in \Sigma^*} Parse_D(p, y) \quad \text{for any } p \in dom(\mathcal{C}_A)$$

That is, we construct the context functions such that $dom(\mathcal{C}_A)$ contains all context stacks that may appear when entering $A$, without performing any truncation, and similarly for their output. (Note that these applications of $Parse_D$ never return $\lightning$ or $\bigcirc$ due to Lemma 7.1 since $G$ is assumed to be valid, so $dom(\mathcal{C}_A)$ and $\mathcal{C}_A(p)$ are well-defined.) With this construction, we argue that $[\mathcal{C}_A]_{A \in N}$ is a solution to the constraints:

- Rule §1 is satisfied because, by construction, $\mathcal{C}_S(\diamond) = \bigcup_{S \Rightarrow^* y} Parse_D(\diamond, y)$, and $y \in \Sigma^*$ is valid when $S \Rightarrow^* y$.

- To see that rule §2 is satisfied, consider a production of the form $A \to s$ in $P$ where $s \in \Sigma$, and assume $p \in dom(\mathcal{C}_A)$ and $p' = Parse_D(p, s)$. By construction of $dom(\mathcal{C}_A)$, we have $p = Parse_D(\diamond, u_1)$ where $S \Rightarrow^* u_1 A u_2$ for some $u_1, u_2 \in \Sigma^*$. Now, $p' \in \{\lightning, \bigcirc\}$ would contradict the assumption that $G$ is valid using Lemma 7.1 as above, so $p' \notin \{\lightning, \bigcirc\}$. By construction of $\mathcal{C}_A(p)$, we also get $p' \in \mathcal{C}_A(p)$.

- Rule §3 is satisfied for any production $A \to A'A''$ because no truncation appears in our present construction of $dom(\mathcal{C}_A)$, so the following property is satisfied, which clearly implies the condition in rule §3:

$$\begin{aligned}\forall p \in dom(\mathcal{C}_A) : p \in dom(\mathcal{C}_{A'}) \wedge \\ \forall p' \in \mathcal{C}_{A'}(p) : p' \in dom(\mathcal{C}_{A''}) \wedge \\ \forall p'' \in \mathcal{C}_{A''}(p') \Rightarrow p'' \in \mathcal{C}_A(p)\end{aligned}$$

  To see that this property is satisfied, consider derivations of the form $S \Rightarrow^* u_1 A u_2 \Rightarrow u_1 A' A'' u_2 \Rightarrow^* u_1 y_1 A'' u_2 \Rightarrow^* u_1 y_1 y_2 u_2$ where $u_1, u_2, y_1, y_2 \in \Sigma^*$. The first line then directly follows from the construction of $dom(\mathcal{C}_A)$. For the second line, we have $p \in Parse_D(\diamond, u_1)$ and $p' \in Parse_D(p, y_1)$. By Lemma 7.1, $p' \in Parse_D(\diamond, u_1 y_1)$ so $p' \in dom(\mathcal{C}_{A''})$. For the third line, we have $p'' \in Parse_D(p', y_2)$, and $p'' \in Parse_D(p, y_1 y_2)$ then follows from Lemma 7.1.

# Chapter 8

# Automated Detection of Client-State Manipulation Vulnerabilities

This chapter contains a revised version of the paper "Automated Detection of Client-State Manipulation Vulnerabilities" [61]. The paper originally appeared at the 34th International Conference on Software Engineering (ICSE 2012) where it was awarded with a distinguished paper award.

Compared the orginial paper, this version explains the analysis in much more detail and explains the output analysis of web pages. The evaluation section furthermore contains three additional benchmarks. This version of the paper has been submitted for journal publication.

# Automated Detection of Client-State Manipulation Vulnerabilities

Anders Møller Mathias Schwarz

**Abstract**

Web application programmers must be aware of a wide range of potential security risks. Although the most common pitfalls are well described and categorized in the literature, it remains a challenging task to ensure that all guidelines are followed. For this reason, it is desirable to construct automated tools that can assist the programmers in the application development process by detecting weaknesses. Many vulnerabilities are related to web application code that stores references to application state in the generated HTML documents to work around the statelessness of the HTTP protocol. In this article, we show that such client-state manipulation vulnerabilities are amenable to tool supported detection.

We present a static analysis for the widely used frameworks Java Servlets, JSP, and Struts. Given a web application archive as input, the analysis identifies occurrences of client state and infers the information flow between the client state and the shared application state on the server. This makes it possible to check how client-state manipulation performed by malicious users may affect the shared application state and cause leakage or modifications of sensitive information. The warnings produced by the tool help the application programmer identify vulnerabilities before deployment. Alternatively, the inferred information can be applied to configure a security filter that automatically guards against attacks at runtime. Experiments on a collection of open source web applications indicate that the static analysis is able to effectively help the programmer prevent client-state manipulation vulnerabilities.

## 8.1 Introduction

Errors in web applications are often critical. To protect web applications against malicious users, the programmers must be aware of numerous kinds of possible vulnerabilities and countermeasures. Among the most popular guidelines for programming safe web applications are those in the OWASP Top 10 report that covers "the 10 most critical web application security risks" [68]. Many security properties depend on the flow of untrusted data in the programs. This flow is often not explicit in the program code, so it can be difficult to ensure that sensitive data is properly protected. Although tool support exists for detecting and preventing some risks, manual code review and testing remain crucial to ensure safety. However, code review and testing are tedious and error-prone means, so it is desirable to identify classes of vulnerabilities that are amenable to tool support.

As an example, consider the category *A4 - Insecure Direct Object References* from the 2010 OWASP Top 10 list. A direct object reference is a reference to an internal implementation object, such as a database record, that is exposed to the user as a form field or a URL parameter in an HTML document. Such references

are examples of *client state*, which is used extensively in web applications to work around the statelessness of the HTTP protocol, for example to store session state in the HTML documents at the clients.

Figure 8.1 shows two excerpts of source code from a web application named *JSPChat*[1]. Part (a) shows a JSP page containing an HTML form for saving personal information in a chat service, and part (b) shows the servlet code that is executed when the form data is submitted by the user. The first thing to notice is that the `nickname` form field on line 20 in the JSP page functions as a direct object reference that refers to a `ChatRoom` object and a `Chatter` object that are stored on the server. As the application programmer cannot trust that the user does not modify such references in an attempt to access resources that belong to other users, it is important to ensure that object references are protected. This can be done for example using a layer of indirection (i.e. using a map stored on the server from client-state values to the actual object references), via cryptographic signatures or encryption of the client state, or by checking that the user is authorized to access the resources being referenced in the requests. This is, however, easy to forget when programming the web application. In the example, the servlet reads the `nickname` parameter, stores it in a field in the servlet object, and then uses it – without any security measures – to look up the corresponding `ChatRoom` and `Chatter` objects in the shared application state on lines 47 and 49. Obviously, a malicious user can easily forge the parameter value and thereby access another person's data. (The careful reader may have noticed another vulnerability in the program code; we return to that in Section 8.8.)

As documented in security alerts and reports by, e.g., ISS [33], MSC [7], Advosys [1] and Sanctum [91], vulnerabilities of this kind have been known—and exploited—for more than a decade. However, they remain widespread on the web, as evident from the 2010 OWASP report. A recent study shows that application developers are still unaware of common classes of related vulnerabilities, despite awareness programs provided by, for example, OWASP, MITRE, and SANS Institute [75]. A notable recent example of client-state manipulation is the attack on the Citigroup website that allowed hackers to disclose account numbers and transaction history for 200,000 credit cards [63].

According to OWASP, "automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe". Nevertheless, in this article we show that it is possible to develop automated tools that can detect many of these flaws. Our approach is based on a simple observation: *Vulnerability involving client-state manipulation is strongly correlated to information flow from hidden fields or other kinds of client state to operations involving the shared application state on the server.* This approach is along the lines of previous work on static taint analysis [49, 83], however with crucial differences in how we characterize the sources and sinks of the information flow. We discuss related work in Section 8.9.

[1] `http://www.web-tech-india.com/software/jsp_chat.php`

```
<% ChatRoomList roomList =
   (ChatRoomList)application.getAttribute("chatroomlist
       ");
 ChatRoom chatRoom = roomList.getRoomOfChatter(nickname)
     ;
 Chatter chatter = chatRoom.getChatter(nickname); %>
<html><head>
<meta http-equiv="pragma" content="no-cache">
<title>
  Edit your (<%=chatter.getName()%>'s) Information
</title>
<link rel="stylesheet" type="text/css"
  href="<%=request.getContextPath()%>/chat.css">
</head>
<body bgcolor="#FFFFFF">
<form name="chatterinfo" method="post"
  action="<%=request.getContextPath()%>/servlet/saveInfo
      ">
<table width="80%" border="0" cellspacing="0"
  cellpadding="2" align="center" bordercolor="#6633CC">
<tr><td valign="top"><h4>Nickname:</h4></td>
<td valign="top"><%=chatter.getName()%></td>
<input type="hidden" name="nickname"
  value="<%=chatter.getName()%>">
</tr>
<tr><td valign="top"><h4>Email:</h4></td>
<td valign="top"><input type="text" name="email"
  value="<%=chatter.getEmail()%>">
</td></tr>
<tr><td valign="top">
<input type="submit" name="Submit" value="Save">
</td></tr></table></form></body></html>
```

(a) `editInfo.jsp`

```
public class SaveInfoServlet extends HttpServlet {
  String nickname = null;
  String email = null;
  HttpSession session = null;
  String contextPath = null;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
       throws IOException, ServletException {
    nickname = request.getParameter("nickname");
    contextPath = request.getContextPath();
    email = request.getParameter("email");
    session = request.getSession(true);
    ChatRoomList roomList = (ChatRoomList)
      getServletContext()
      .getAttribute("chatroomlist");
    ChatRoom chatRoom =
      roomList.getRoomOfChatter(nickname);
    if (chatRoom != null) {
      Chatter chatter = chatRoom.getChatter(nickname);
      chatter.setEmail(email);
      ...
    }
  }
}
```

(b) `SaveInfo.java`

Figure 8.1: A simplified version of the *JSPChat* web application.

In summary, the main contributions of this article are as follows:

- Our starting point is a characterization of *client-state manipulation vulnerabilities* (Section 8.2) that has considerable overlap with category A4 from the OWASP 2010 list of the most critical risks. In particular, we describe safety conditions under which the use of client state is likely not to cause vulnerabilities.
- Based on this characterization, we present an automated approach to detect occurrences of client state in a given web application and check whether the safety conditions are satisfied (Sections 8.3–8.6).
- In addition to reporting the detected vulnerabilities to the application programmer, we describe how the information obtained by the analysis can also be used for automatically configuring a security filter (Section 8.7) that guards against client-state manipulation attacks at runtime.
- Through experiments performed on 7 open source web applications with a prototype implementation of our analysis, we show that the approach is effective for helping the programmer detect client-state manipulation vulnerabilities. On a total of 1575 servlets, JSP pages, and Struts actions, our tool identifies 4802 possible occurrences of hidden fields and other client-state parameters, and it reveals 230 exploitable vulnerabilities involving 58 different field names.

The static analysis that underlies our automated approach to detect client-state manipulation vulnerabilities consists of three components. The first component (Section 8.4) infers the dataflow between the individual servlets and pages that constitute the application, in order to identify the *client-state parameters*. This requires a static approximation of the dynamically constructed output of the servlets and pages and extraction of relevant URLs and parameter fields in forms and hyperlinks. The second component (Section 8.5) analyzes the program code to find out which objects represent *shared application state*, i.e., state that is persistent or shared between multiple clients, as opposed to session state or transient state. The third component (Section 8.6) performs an information flow analysis to identify the possible flow of user controllable input from client-state parameters to shared application state objects. The vulnerability warnings being produced by the tool can be used either to guide the programmer to apply appropriate countermeasures by modifying the application source code or to automatically configure a security filter.

Our goal is not to develop a technique that can fully guarantee absence of client-state manipulation vulnerabilities. Rather, we aim for a pragmatic approach that can detect many real vulnerabilities while producing as few spurious warnings as possible. Since authorization checks and other countermeasures come in many different forms, the information flow analysis component may require some customization, but the analysis is otherwise fully automatic.

## 8.2 Client-State Manipulation Vulnerabilities

In a web application, *client state* comprises information that is stored within a dynamically generated HTML document in order to be transmitted back to the server at a subsequent interaction, for example when a form is submitted. Since the HTTP protocol is stateless, client state is widely used for keeping track of users and session state that involves multiple interactions between the each client and the server. Storing session state at the client instead of at the server can have several benefits. Most importantly, it decreases the load on the server and avoids the need for a

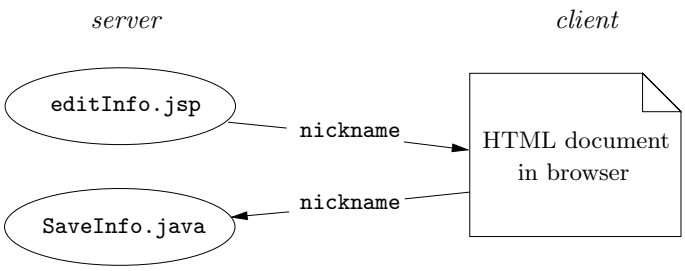


Figure 8.2: Data flow of client state from a JSP page to a servlet via an HTML document. The `nickname` parameter is sent from `editInfo.jsp` to `SaveInfo.java` via the HTML document.

session state expiration mechanism. Client-state appears as hidden fields in HTML forms – as in the example in Section 8.1 – and as URL query parameters in hyperlinks. Such state is not intended to be modified by the user, but nothing prevents malicious users from doing so, and this is easy to forget when programming web applications. A related situation occurs with HTML select menus, radio buttons, and checkboxes, which also contain fixed sets of values that the user is intended to choose from. We commonly refer to HTTP GET/POST request parameters that contain such state as *client-state parameters*. Cookies provide a related mechanism; in this article we focus on ordinary HTTP request parameters, but our approach in principle also works for cookies.

For the discussion we consider Java-based web applications, specifically ones based on Java Servlets, JSP or Struts. We use the general term *page* to refer to a servlet instance, a JSP page, or a Struts action instance; each produces an HTML *document* when executed.

Figure 8.2 illustrates the data flow of client state for the *JSPChat* example. The value of `nickname` is passed as client state from a JSP page, `editInfo.jsp`, to a servlet, `SaveInfo.java`, using a hidden field in the HTML document.

The following characterization is our key to automate detection of the vulnerabilities we consider: A web application is vulnerable to *client-state manipulation* if is is possible, by users modifying client state, to access or manipulate shared application state that is not otherwise possible.

This class of vulnerabilities is closely related to MITRE's weakness categories CWE-472 (External Control of Assumed-Immutable Web Parameter)[2] and CWE-639 (Authorization Bypass Through User-Controlled Key)[3] – and, as discussed in the previous section, to OWASP's risk category A4 (Insecure Direct Object References). Moreover, the page for CWE-472 mentions that it "is a primary weakness for many other weaknesses and functional consequences, including XSS, SQL injection, path disclosure, and file inclusion". Descriptions of the categories are shown in Figure 8.3.

All client-state parameters – most importantly, those that originate from hidden fields in HTML forms – are potential sources of client-state manipulation vulnerability.
On the other hand, we observe that uses of client state are *safe*, that is, not vulnerable to client-state manipulation, if at least one of the following conditions is satisfied:

[2] `http://cwe.mitre.org/data/definitions/472.html`
[3] `http://cwe.mitre.org/data/definitions/639.html`

CWE-472 (External Control of Assumed-Immutable Web Parameter)
*"If a web product does not properly protect assumed-immutable values from modification in hidden form fields, parameters, cookies, or URLs, this can lead to modification of critical data. Web applications often mistakenly make the assumption that data passed to the client in hidden fields or cookies is not susceptible to tampering. Improper validation of data that are user-controllable can lead to the application processing incorrect, and often malicious, input."*

CWE-639 (Authorization Bypass Through User-Controlled Key)
*"Retrieval of a user record occurs in the system based on some key value that is under user control. The key would typically identify a user related record stored in the system and would be used to lookup that record for presentation to the user."*

OWASP A4 (Insecure Direct Object References)
*"A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data."*

Figure 8.3: Weakness categories from MITRE and OWASP that are related to client-state manipulation.

1) The client-state parameter value stored in the HTML document is encrypted using a key private to the server. The server then decrypts the value when it is returned. A variant is to leave the value unencrypted but add an extra hidden field or URL parameter containing a digital signature (or MAC, message authentication code) computed from the client-state value and the server's private key. The server then verifies that the client-state value is unaltered by checking the signature when the form data is returned. To prevent against replay attacks and impersonation attacks, a time stamp and a client ID can be included in the encryption or signature generation. A drawback of this approach is that extra work is needed when producing and receiving the client state.

2) The client state entirely consists of large random values that are practically impossible to predict by attackers. A typical example is the use of *session IDs*: in many web applications, all session state is stored on the server and the only client state being used consists of session IDs, i.e. references to the session state on the server. A drawback of this approach is that it requires extra space on the server to store the session state.

3) An indirection is used: the client state consists of, for example, only numbers between 1 and some small constant, and these numbers are then mapped to the actual application state on the server. This approach is particularly useful for select menus, radio buttons, and checkboxes. In this way, client-state manipulation cannot provide access to data beyond what is accessible from this map. A drawback of this approach is the burden involved in maintaining the indirection map.

4) The client-state parameter is treated as untrusted input, no different from other kinds of parameters, and any access to application state involving the given client-state value is guarded by an authorization check.

5) Finally, a sufficient condition for safety according to the definition above is that all the shared application state that can be accessed through client-state

manipulation is already available by other means – that is, the information should not be considered sensitive.

We see uses of several of these techniques in the web applications *Hipergate*, *Pebble*, and *JWMA* that we study in Section 8.8. OWASP's ESAPI[4] library contains support for implementing the first four of these safe usage conditions. As an example, to apply the ESAPI encryption approach to the *JSPChat* web application from Figure 8.1, the programmer would wrap the expression `chatter.getName()` on line 21 into a call to `ESAPI.httpUtilities().encryptHiddenField(...)` and insert a matching call to `decryptHiddenField` on line 39. The `decryptHiddenField` method throws an `IntrusionException` if tampering is detected. In .NET, the `LosFormatter`[5] class provides support for MAC protection of client state (or view state, as it is called in .NET). The use of encryption and signatures to prevent manipulation of hidden form fields was originally suggested by MSC [7] and Advosys [1]. Thus, the countermeasures are well-known; the goal of our analysis is to detect when they are applied inadequately.

## 8.3 Outline of the Analysis

We adapt the well-known approach to static information flow analysis for identifying the possible dataflow from *sources* to *sinks* that does not pass through *sanitizers* [31, 37, 49, 83, 88, 90]:

- The sources in our setting are the locations in the code where client-state parameters are read. With common web application frameworks, such as Java Servlets, JSP, and Struts, it is not explicit in the application source code which parameters contain client state, so we need a static analysis to infer this information.

- The sinks are the operations in the source code that affect shared application state. We conservatively assume that this application state is not accessed by other means. This assumption may lead to false positives, which we consider experimentally in Section 8.8. As is it not explicit in the source code which objects and methods involve shared application state (in contrast to session state or transient state), we need another static analysis component to extract this information.

- The sanitizers correspond to the various kinds of protection described in Section 8.2. For example, decrypting an encrypted client-state value is one kind of sanitization.

Our analysis tool has built in mechanisms for identifying sinks and sanitizers, independently of the applications. The user can customize the analysis by providing additional application specific patterns.

We propose the following procedure for analyzing a given web application: (1) Run the analysis on the application, with only the default sink and sanitizer patterns. (2) Study the warnings being produced and add customization rules to those that are considered false positives, to enable the analysis to reason more precisely about the relevant parts of the application. (3) Then run the analysis again, using the new customization. Provided that the analysis is sufficiently precise, most warnings now indicate actual exploitable vulnerabilities.

[4] `https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API`
[5] `http://msdn.microsoft.com/en-us/library/system.web.ui.losformatter.aspx`

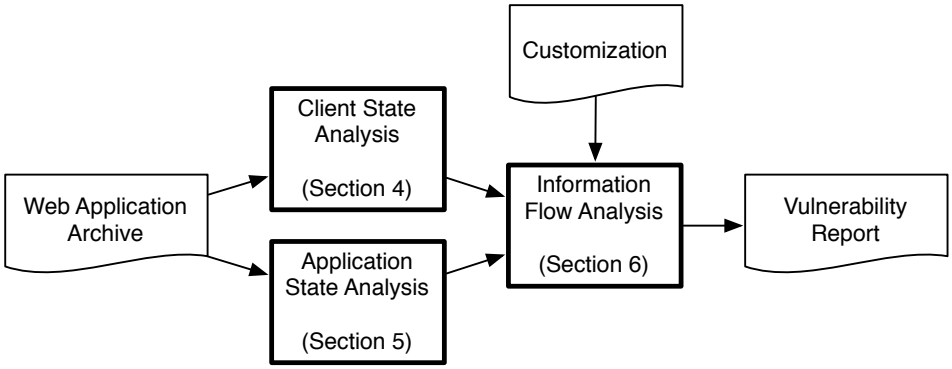


Figure 8.4: Structure of the analysis.

One approach to remedy the vulnerabilities detected by the analysis is that the programmer manually incorporates appropriate countermeasures into the web application source code as discussed in Section 8.2. Another option is to feed the vulnerability report to a security filter, which we describe in Section 8.7, for automatic protection. Our experiments (see Section 8.8) indicate that the burden of the customization step is manageable. However, we note that a fully automatic approach to protect against client-state manipulation can be obtained by omitting customization entirely and applying the security filter without having eliminated false positives. Compared to the more manual approach involving customization, the price is a modest runtime overhead incurred by the security filter since it may protect some client state unnecessarily.

The following sections explain how we identify client-state parameters and application state and perform the information flow analysis. The structure of the combined analysis is illustrated in Figure 8.4.

## 8.4 Identifying Client State

When a page $p$ reads an HTTP parameter, for example on lines 39 and 41 in Figure 8.1, the only way we can find out whether that is a client-state parameter is to analyze all the pages of the application that dynamically construct HTML documents with links or forms referring to $p$. Specifically, we need to recognize the construction of the hidden field named `nickname` on line 20 in `editInfo.jsp`, and via the `action` attribute of the `form` element in `editInfo.jsp`, establish the connection from `editInfo.jsp` to line 39 in the servlet `SaveInfo.java`.

For each page $q$ in the web application we first generate a context-free grammar $G_q$ that conservatively approximates the set of HTML documents that may be generated by $q$. This can be done as in our previous work on analysis of dynamically generated HTML documents [60]. To find the client-state parameters in the HTML documents generated by $q$, we identify all elements that define hidden fields, select boxes, radio buttons, and links in $G_q$ and collect the corresponding parameter names. Since these names may be generated dynamically in the program we approximate them conservatively by a regular language $C_{out}(q)$. In the *JSPChat* example from Figure 8.1 this step identifies `nickname` as the only client-state parameter originating from `editInfo.jsp`, thus $C_{out}(\texttt{editInfo.jsp}) = \{\texttt{nickname}\}$. We also infer the references between the pages by identifying `href` attributes in `a` elements and `action` attributes in `form` elements in $G_q$. This results in a map $S$ that holds the set of possible successor pages for each page. For example,

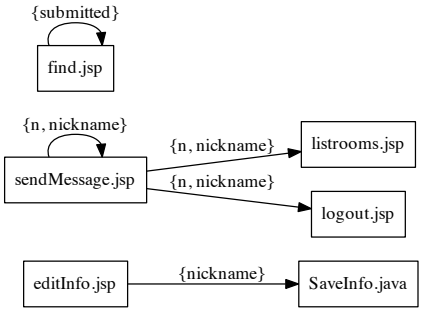


Figure 8.5: The automatically constructed page graph for the parts of *JSPChat* that involve client state.

`SaveInfo.java` $\in S($`editInfo.jsp`$)$. Section 8.4.1 explains in more detail how to infer this information from the web application code.

Combined with information extracted from the deployment descriptors (`web.xml` in Servlets and `struts.xml` in Struts) this results in a *page graph* in which nodes correspond to pages and edges correspond to $S$ describing the possible links and form actions. In the example in Figure 8.1, this step identifies the edge from `editInfo.jsp` to `SaveInfo.java`. Figure 8.5 shows the page graph for the 6 pages in *JSPChat* that involve flow of client-state.

The names of the incoming client-state parameters to a page $p$ can now be expressed as $C_{in}(p) = \cup_{q_i} C_{out}(q_i)$ for each page $q_i$ where $q_i \in S(p)$. (In principle, $p$ may have multiple incoming edges with different $C_{out}$ sets, in which case the analysis issues a warning, though that never happens in our experiments described in Section 8.8.) For the example, this gives $C_{in}($`SaveInfo.java`$) = \{$`nickname`$\}$. Section 8.4.2 explains how to use $C_{in}$ for locating operations in the application code that read client-state parameters.

The result of these steps is a set of method calls in the application code that will serve as sources of client-state values in the information flow analysis in Section 8.6. All of the steps can be done soundly in the sense that every call to `getParameter` and related operations that may return client state is always included in the statically inferred set of client-state value sources. In our experiments, we never observe any imprecision of this phase.

### 8.4.1 Analyzing HTML Output

As outlined above, we need to analyze the source code of the web application to find the hidden fields, URL parameters, links, and form actions that may appear in the generated HTML pages. With Java Servlets, output is generated by printing string fragments to an output stream. JSP and Struts compile to Servlets, so we can handle each of these frameworks by focusing on Servlets. We assume that the HTML documents being generated do not use JavaScript code in ways that interfere with forms and links. Starting from the Java bytecode of the web application, the first step is to construct an *output stream flow graph*, which is a representation of the program that abstracts away everything not directly relevant for generating output to the output stream. More formally, an output stream flow graph $F$ is a directed graph given as a tuple $(N, E, C, L)$:

- $N$ is a finite set of nodes, divided into three disjoint subsets:
  - $N_{\mathsf{append}}$ are *append nodes* representing instructions that print strings to the output stream,

  - $N_{\mathsf{invoke}}$ are *invoke nodes* corresponding to method calls, and
  - $N_{\mathsf{return}}$ are *return nodes* corresponding to method returns.

- $E \subseteq (N_{\mathsf{append}} \cup N_{\mathsf{invoke}}) \times N$ is a set of *intra-procedural* edges,
- $C \subseteq N_{\mathsf{invoke}} \times N$ is a set of *call* edges, and
- $L : N_{\mathsf{append}} \to \mathcal{R}$ gives a regular string language (represented by a regular expression or a finite-state automaton over the Unicode alphabet) for every append node.

Output stream flow graphs are abstract machines. Intuitively, the nodes correspond to primitive instructions in the program being analyzed, and edges correspond to control flow between those instructions. An append node abstractly writes to the output stream and then continues execution nondeterministically at a successor node. An invoke node pushes a successor node to the call stack and then enters one of its target methods. A return node exits the current method, pops a node from the call stack, and continues execution from that node. We define the language $\mathcal{L}_F(n_0)$ of $F$ relative to an entry node $n_0 \in N$ as the set of strings that may appear as output when $F$ is executed starting from $n_0$ with an empty stack and ending at a return node with an empty stack.

Our implementation constructs output stream flow graphs from Java bytecode using the Soot program analysis framework [84] and the JSA string analysis tool [12, 20]. From Soot, we use the Jimple intermediate representation, the built-in class-hierarchy analysis for obtaining call graphs, and the Spark points-to analysis to find the operations that affect the HTTP output stream. JSA gives us the regular string languages for the invoke nodes. The resulting output stream flow graph has one entry node in $F$ for each page in the web application.

A *context-free grammar* $G$ is a tuple $(V, \Sigma, s, P)$ where

- $V$ is a set of nonterminals,
- $\Sigma$ is the terminal alphabet where $V \cap \Sigma = \emptyset$,
- $s \in V$ is a start nonterminal, and
- $P$ is a finite set of productions of the form $v \to \theta$ where $v \in V$ and $\theta \in (V \cup \Sigma)^*$.

For convenience, we also allow right-hand-sides of productions in $P$ to be symbols that denote regular string languages over $\Sigma$. (In principle, these can always be reduced to regular grammars.) We define the language $\mathcal{L}_G(v_0)$ of $G$ relative to a nonterminal $v_0$ as the set of strings over $\Sigma$ that can be derived starting from $v_0$ using the productions in $P$. In particular we are interested in the language relative to the start nonterminal $s$, written $\mathcal{L}(G) = \mathcal{L}_G(s)$.

We now construct a family of context-free grammars $\{G_q\}_q$, where the index $q$ is a page in the web application, from the output stream flow graph $F$. All the grammars have the same nonterminals, terminals, and productions; only the start nonterminal differs. The nonterminals are the nodes from $F$, that is, $V = N$, and $\Sigma$ is the Unicode alphabet. The start nonterminal for $G_q$ is the entry node of the page $q$. The productions are constructed such that $\mathcal{L}_{G_q}(n) = \mathcal{L}_F(n)$ for all $n \in N$. Although output stream flow graphs have an operational flavor and context-free grammars are a declarative formalism, this construction of the productions is straightforward:

- for each $n \in N_{\mathsf{append}}$ and $(n, m) \in E$, add a production $n \to r_n\ m$ to $P$ where the symbol $r_n$ denotes $L(n)$,

- for each $n \in N_{\mathsf{invoke}}$, $(n, m) \in E$ and $(n, p) \in C$, add a production $n \to p\ m$ to $P$, and
- for each $n \in N_{\mathsf{return}}$, add a production $n \to \epsilon$ to $P$.

The correctness of this translation from output stream flow graphs to context-free grammars follows from the observation that the semantics of both formalisms can be expressed as the smallest solution to the following constraints where $\mathcal{L}$ assigns a language over the Unicode alphabet to each node or nonterminal:

$$\forall n \in I_{\mathsf{append}}, (n, m) \in E : \ L(n)\mathcal{L}(m) \subseteq \mathcal{L}(n)$$
$$\forall n \in I_{\mathsf{invoke}}, (n, m) \in E, (n, p) \in C : \ \mathcal{L}(p)\mathcal{L}(m) \subseteq \mathcal{L}(n)$$
$$\forall n \in I_{\mathsf{return}} : \ \epsilon \in \mathcal{L}(n)$$

We now have a family of context-free grammars where $\mathcal{L}(G_q)$ is an over-approximation of the strings that the web application page $q$ may possibly produce as output. The next step is to analyze each $G_q$ to produce an annotated grammar $G'_q$ as described in the following.

Consider the way an HTML parser performs a left-to-right scan through the characters of an HTML document. The parser is initially in a state *contents*. When it encounters a "<" character, it switches to another state *tagname* meaning that it now expects to see a tag name. It stays in this state until it encounters, for example, a whitespace character, which causes a switch to the state *attname* meaning that it is now prepared to see an attribute name. Similarly, it recognizes the different kinds of attribute value syntax, entity references, comments, etc., as different parse states. Let $H$ denote the set of all parse states that are necessary for parsing HTML documents, $H = \{\mathit{contents}, \mathit{tagname}, \dots\}$, and let $\delta : H \times \Sigma \to H$ denote the transition function that determines the next state after each character is read. An actual HTML parser also maintains a stack to keep track of the nesting of elements; for our purposes, only the $\delta$ function is relevant. We now generalize this process to operate on a context-free grammar $G_q$, which defines a set of HTML documents, rather than on individual HTML documents. The result is a function $\rho : P \times \mathbb{N} \to \mathcal{P}(H)$ that assigns a set of HTML parse states to each position in the productions of $G_q$. As an example, for a production $p_3 = v_7 \to v_8$ `<u>` $v_9$ `</u>` we may have $\rho(p_3, 0) = \rho(p_3, 1) = \{\mathit{contents}\}$ and $\rho(p_3, 2) = \{\mathit{tagname}\}$ where the numbers 0, 1 and 2 correspond to the position at the beginning of the right-hand-side, the position immediately after $v_8$, and the position after the first "<", respectively. We construct the function $\rho$ as the least solution that satisfies the following constraints:

- for each production $p = v \to \theta$ where $v$ is the start nonterminal: $\mathit{contents} \in \rho(p, 0)$
- for each production $p = v \to \theta$ and each $i = 1, \dots, |\theta|$, let $a_i$ be the $i$'th terminal or nonterminal in $\theta$,
  - if $a_i$ is a terminal: $h \in \rho(p, i-1) \Rightarrow \delta(h, a_i) \in \rho(p, i)$
  - if $a_i$ is a nonterminal and $p' = a_i \to \theta'$ is a production: $\rho(p', |\theta'|) \in \rho(p, i)$

The least solution can be computed using a simple fixpoint algorithm. We now define the annotated grammar $G'_q = (G_q, \rho)$. Each set $\rho(p, i)$ is usually a singleton, meaning that the parse context has been determined uniquely, however, in situations where a piece of program code generates output that may result, for example, either in contents between HTML tags or in attribute values, the sets may contain multiple parse states.

The last step of the analysis of the dynamically generated HTML output consists of a simple traversal through the annotated grammar $G'_q$, looking for specific

elements and attributes. To construct $C_{out}$, that is, the required information about names of hidden fields and URL parameters, we look for the `name` attributes in `input` elements that have a `type` attribute with value `hidden` and for `href` attributes in `a` elements. To construct the page graph edges $S$ we look for the `action` attributes in `form` elements and for `href` attributes in `a` elements.

### 8.4.2 Analyzing Input Parameters

Parameter values in the Java Servlet framework are read using the `getParameter` method of the `HttpServletRequest` object. We conservatively assumes that all objects of type `HttpServletRequest` are relevant. The Servlet framework instantiates all request objects and provides no implementation of the `HttpServletRequest` interface to the programmer, so this assumption is unlikely to result in false positives in practice. As the request parameter name that is given as argument to this method may not be a constant in the source code, we approximate for each call to `getParameter` the possible values as a regular language. We obtain this information using the JSA string analysis tool as in Section 8.4.1. If the language overlaps with $C_{in}(p)$, we mark the method call as a *client-state value source*. This step will mark the method call on line 39 in Figure 8.1 as such a source. The call on line 41 will not be marked since `email` is not in $C_{in}$(`SaveInfo.java`).

Most JSP pages that read request parameters use the underlying mechanism from Servlets, although the expression language (EL) and the JSTL tag library may also be involved. In the Struts framework, parameters are read in a different manner. Rather than retrieving the values from a request object, Struts populates a Java bean object with the parameter values. In each case, we can identify parameter read operations in the code using simple pattern matching on Soot's Jimple code.

## 8.5 Identifying Shared Application State

To find the operations in the code that affect shared application state, i.e. state that is shared between all requests, we first identify the application state that is stored in memory, which we call the *internal* application state. This includes:

1. all `HttpServlet` objects (and hence the value of `this` inside servlet classes) and `ServletContext` objects, and all values of static fields,
2. all values of fields of objects that have been classified as internal application state, and conversely, all objects that have non-static fields containing internal application state, and
3. all values returned from static methods or from methods on internal application state objects.

Notice that in situations where session state or transient state points to shared application state or vice versa, the second rule may conservatively classify such state as application state.

Finding all expressions in the code that may yield internal application state according to these rules can be done with a simple iterative fixpoint algorithm combined with an alias analysis, such as the points-to analysis provided by the Soot tool that we also used in Section 8.4.1. We first define a set of abstract locations $K = \mathit{Field} \cup \mathit{Local}$ where *Field* and *Local* denote the fields in classes and the local variables and method parameters, respectively, in the application code. For a field $f \in \mathit{Field}$, the abstract location $f$ corresponds to the set of fields named $f$ in objects at runtime. Similarly, a local variable or method parameter $x \in \mathit{Local}$ corresponds to all occurrences of $x$ at runtime. Every name in *Field* and *Local* is implicitly

qualified by the signature of the surrounding class and method, respectively, to distinguish between variables of the same name in different contexts. We assume that nested expressions have been linearized by Soot using extra local variables, and the keyword `this` is treated as a local variable. The points-to analysis gives us a may-alias equivalence relation $\sim \subseteq K \times K$ such that $k_1 \sim k_2$ if $k_1$ and $k_2$ may point to the same object at runtime. We now find the internal application state by computing a subset of the abstract locations $A \subseteq K$ as the least solution to the following constraints where $x, y \in \mathit{Local}$, $f \in \mathit{Field}$, $c$ is a class, and $m$ is a method:

- for every abstract location $k$ that has type `HttpServlet` or `ServletContext` or is a static field: $k \in A$,
- for every field read operation $x$ `=` $y.f$: $y \in A \Rightarrow x \in A$,
- for every field write operation $x.f$ `=` $y$: $y \in A \Rightarrow x \in A$,
- for every static method call operation $x$ `=` $c.m(\ldots)$: $x \in A$,
- for every non-static method call operation $x$ `=` $y.m(\ldots)$: $y \in A \Rightarrow x \in A$,
- for every pair of abstract locations, $k_1$ and $k_2$, where $k_1 \sim k_2$: $k_1 \in A \Leftrightarrow k_2 \in A$.

The first condition corresponds to rule (1) above, the next two correspond to rule (2), and the two after those correspond to rule (3). The last rule takes aliasing into account. This computation of $A$ captures all internal application state, although obviously as an approximation. As mentioned above, we may conservatively classify some session state or transient state as application state. The coarse heap abstraction and alias analysis, as well as the lack of, for example, flow- and context-sensitivity may also contribute to imprecision. Nevertheless, the experiments described in Section 8.8 indicate that this simple analysis is sufficient.

Continuing the *JSPChat* example from Figure 8.1, the variables `nickname`, `email`, `session`, and `contextPath` in `SaveInfo.java` are fields in the servlet class, so their values are correctly classified as internal application state. (That is, however, presumably not intended by the programmer, which we return to in Section 8.8.) The `roomList` variable gets its value from an attribute in the servlet context object using the method calls `getServletContext().getAttribute(...)`, so its value is also classified as internal application state. In contrast, the variables `request` and `response` are not included as internal application state.

We also find the *external* application state stored in files and databases. Such state is read and written using special API functions. The analysis treats all parameters to all methods from the standard Java libraries as sinks, except from a built-in collection of method parameters that have a special meaning for the information flow. We describe these exceptions and a customization mechanism in Section 8.6.

Web applications often rely on libraries, such as Hibernate or Apache Commons, which are typically provided in separate jar files. We allow libraries to be omitted from the analysis for analysis performance reasons. This will simply cause the analysis to treat all method calls to those libraries conservatively as operations on external application state.

The result of this analysis component is an over-approximation of the set of expressions in the code that yield internal application state and of the set of method calls that involve external application state. We use this information in the following section.

## 8.6 Information Flow from Client State to Shared Application State

As outlined in Section 8.3, we use an information flow analysis to identify flow of the client-state values in the program to the shared application state. In general, information flow analysis considers of two kinds of flow: *explicit* and *implicit* flow [15]. Explicit flow is caused by assignments and parameter passing. Other forms of explicit flow may be described using customized derivation rules, as described below. Implicit flow arises when the value of a variable depends on a branch condition involving another variable. Other work involving information flow in web applications typically disregards implicit flow [50,83]. According to Tripp et al. [83], "experience shows that attacks based on control dependence are rare and complex, and thus less important than direct vulnerabilities." To simplify our analysis, we also choose to consider only the explicit flow.

Information flow analysis requires a characterization of sources, sinks, and sanitizers. The sources in our analysis are the client-state value sources that were identified in Section 8.4. The sinks are operations in the code where the application writes to fields of internal application state objects or calls methods that involve external application state, which we found in Section 8.5.

Sanitizers can be methods that determine whether a given client-state value is safe, for example by performing access control or MAC checking, and methods that convert unsafe values to safe ones, for example by decrypting the values. An example is the ESAPI method `decryptHiddenField` mentioned in Section 8.2. As sanitizers are highly application specific, they are provided through customization, and none are built into the analysis.

The information flow analysis we use is a simple whole-program dataflow analysis. It is flow sensitive, meaning that different information is obtained at different program points. It is context sensitive using one level of call-site sensitivity. The state abstraction uses the same definition of abstract locations, $K$, as the analysis in Section 8.5. Each abstract state provides a set of client-state parameter names for each abstract location. For example, at the program point after the assignment on line 39 in Figure 8.1, the abstract state maps the `email` field of the servlet class to the singleton set {`nickname`} and all other locations are mapped to the empty set. Our implementation uses Soot, as in the other analysis components, with class-hierarchy analysis for call graph construction. The analysis scales well since it only tracks client-state parameters, and relatively few fields and variables involve client state in typical web applications. Another important factor is that the analysis omits library code.

The information flow analysis can be customized to improve precision for sanitizers and sinks. As mentioned above, calls to library methods are treated as sinks by default. This behavior can be changed by specifying derivation rules, each consisting of a method signature and a description of the relevant information flow between arguments, the base object, and the return value. Such derivation rules can also be provided for methods in application code to override the ordinary analysis of information flow between calls to those methods and their bodies, typically for describing sanitizers that convert unsafe values to safe ones. Another variant of customization rules allow description of sanitizers that return a boolean indicating whether the given value is safe or not. When this boolean is used as a branch condition, the analysis will consider the sanitized value as safe in the true branch.

The customization rules can be given either as annotations in the code or in a separate file. Application-specific rules can be added by the user of the analysis. Examples of such customizations are presented in Section 8.8. Additionally, we provide a collection of predefined rules for the Java standard library. Figure 8.6 shows

```
java.lang.String.replace(java.lang.CharSequence,java.lang.CharSequence):
    Flow from parameters 1 and 2 to return value
java.lang.StringBuffer.append(java.lang.String):
    Flow from parameter 1 to base and return value
java.lang.Integer.parseInt(java.lang.String):
    Flow from parameter 1 to return value
java.io.Writer.write(java.lang.String):
    Flow from parameter 1 to base value
java.util.List<E>.add(E):
    Flow from parameter 1 to base value
java.util.List<E>.iterator():
    Flow from base value to return value
java.util.Iterator<E>.next():
    Flow from base value to return value
java.util.HashMap<K,V>.put(K,V):
    Flow from parameters 1 and 2 to base value
java.util.HashMap<K,V>.get(java.lang.Object):
    Flow from base value to return value
java.io.File(java.lang.String):
    Flow from parameter 1 to return value
```

Figure 8.6: Examples of predefined information flow rules.

some examples. The first four rules involve operations on strings that propagate client-state information from parameters to return values or to the base value. For the `add` method on a `List` object, the `List` object is marked as client-state if the object being added to the list has that status. All `Iterator` objects being produced from such `List` objects also become marked as client state, and similarly for objects that are returned from the `next` method on these `Iterator` objects. This accounts for the common pattern of information flow to and from `List` containers. Other containers, such as `HashMap` objects, are treated similarly. The last rule shown in the list tells the analysis that creating a `File` object is harmless – in fact, such objects are often used in authentication checks, c.f. condition 4 in Section 8.2 – so the `File` constructor should not be treated as a sink. However, we specify information flow from the parameter to the constructed object since that object may later be used for constructing, for example, `FileWriter` objects, which are treated as sinks.

## 8.7 Automatic Configuration of a Security Filter

The approach of using MACs to protect against client-state manipulation attacks that we discussed in Section 8.2 can be implemented with a generic servlet filter that intercepts all HTML documents generated by the application at runtime and all HTTP request that are sent by the clients, without modifying the web application code [76]. For every use of client state in the HTML documents, an additional hidden field or query parameter containing the MAC is automatically inserted. Whenever an HTTP request is received from a client, the MAC check is performed on the appropriate request parameters. For this to work, the filter needs to be configured with information about which fields and parameters contain client state that should not be manipulated, and this information is precisely what our static analysis can provide. It is of course important that the client-state analysis is precise enough to correctly distinguish between parameters that carry client state and ones that do not. It is less critical that the information flow analysis is able to correctly distinguish between safe and unsafe uses of client state. However, to avoid the

overhead of generating and checking MACs for parameters that are already safe by other means, it is nevertheless useful that also this analysis component is as precise as possible.

Note that using this security filter is optional; as discussed in Section 8.3 it can be viewed as an alternative or supplement to manually eliminating the vulnerabilities by appropriately patching the application source code.

## 8.8 Evaluation

Our prototype implementation, WARLORD[6], reads in a Java web archive (.war) file containing a web application built with Java Servlets, JSP or Struts, together with an analysis customization file, and performs the analysis described in Sections 8.3–8.6. As mentioned in previous sections, the implementation is based on the Soot analysis infrastructure [84], the JSP compiler from Tomcat[7], and our tools for HTML grammar analysis [60] and string analysis [12, 20]. With this implementation, we aim to answer the following research questions:

Q1: Is the analysis precise enough to detect client-state vulnerabilities with a low number of false positives? Specifically, can it identify the common uses of client state, and is it capable of distinguishing between safe and unsafe uses of client state in the sense described in Section 8.2?

Q2: Are the warning messages produced by the tool useful to the programmer for deciding whether they are false positives or indicate exploitable vulnerabilities?

Q3: In situations where the programmer decides that a vulnerability warning is a false positive, is it practically feasible to exploit the customization mechanism to eliminate the false positive?

Q4: Is the analysis fast enough to be practically useful during web application development?

To answer these questions, we experiment with a collection of web applications. For each application, we go through the process suggested in Section 8.3: We first run the WARLORD tool on the application with no customization. After a manual study of the warnings being produced, appropriate customization is added, if possible, to address the false positives. If any exploitable vulnerabilities are found after running the analysis again, this time with the new customization, we fix them manually using one of the techniques mentioned in Section 8.2.

Our experiments are based on 10 open source web applications found on the web: *JSPChat*[1] (the small chat application mentioned in Section 8.1), *Hipergate*[8] (a customer resource management application written entirely in JSP), *Takatu*[9] (a large tax administration system), *Pebble*[10] (a widely used blogging application), *Roller*[11] (another blogging application), *JWMA*[12] (a web mail application), *JsForum*[13] (a forum application), *JavaLibrary*[14] (a book library management application), *BodgeIt*[15] (a web shop written to demonstrate common security problems in

[6] http://www.brics.dk/WARlord/
[7] http://tomcat.apache.org/tomcat-7.0-doc/jasper-howto.html
[8] http://hipergate.sourceforge.net/
[9] http://takatu.sourceforge.net/
[10] http://pebble.sourceforge.net/
[11] http://roller.apache.org/
[12] http://jwma.sourceforge.net/
[13] http://sourceforge.net/projects/jsforum/
[14] http://sourceforge.net/projects/javalibrary/
[15] http://code.google.com/p/bodgeit/

| | Frameworks | Pages | Client-state parameters | Unique names |
|---|---|---|---|---|
| *JSPChat* | Servlets, JSP | 16 | 19 | 3 |
| *Hipergate* | JSP | 760 | 2680 | 264 |
| *Takatu* | JSP, Struts | 558 | 1840 | 31 |
| *Pebble* | Servlets, JSP | 122 | 22 | 11 |
| *Roller* | JSP, Struts | 53 | 86 | 27 |
| *JWMA* | Servlets, JSP | 26 | 40 | 10 |
| *JsForum* | Servlets, JSP | 10 | 14 | 8 |
| *JavaLibrary* | JSP | 20 | 92 | 35 |
| *BodgeIt* | Servlets, JSP | 9 | 6 | 6 |
| *WebGoat* | Servlets | 1 | 1 | 1 |

Figure 8.7: List of benchmarks. The 'Frameworks' column shows which web frameworks that are used in each benchmark; 'Pages' is the total number of JSP pages, servlet classes, and Structs action classes; 'Client-state parameters' is the number of client-state parameters inferred by the analysis, and 'Unique names' is the number of distinct names of such parameters.

web applications), and *WebGoat*[16] (another web application that has been made to demonstrate typical security problems, written by OWASP). Our prototype supports Struts 2 but not version 1, so we do not include the full list of benchmarks from Stanford SecuriBench [48]. The benchmarks on our list cover a variety of application kinds of different size, they are written by different programmers, and they use different web frameworks (a mix of Java Servlets, JSP, and Struts). The *Takatu* and *JsForum* projects do not appear to be active but represent interesting snapshots of incomplete web applications. Some characteristics of the benchmarks are listed in Figure 8.7. The column 'Client-state parameters' shows the total number of client-state parameters computed as $\sum_p |C_{in}(p)|$ for all pages $p$. Although $C_{in}(p)$ may in principle be infinite, each of the sets is a singleton in most cases. Note that client-state values appear in all the benchmarks. The number of distinct names of the parameters, i.e. $|\bigcup_p C_{in}(p)|$, shown in the last column gives an indication of how many different kinds of client state that occur.

### 8.8.1 Experiments

#### JSPChat

The analysis identifies uses of 19 client-state parameters, and only 1 warning is produced about potential client-state manipulation vulnerability. The single warning is shown in Figure 8.8: as hinted in Section 8.1, the application is prone to a timing attack since the values of the request variables are stored in fields on the servlet object, which the analysis reveals. Since this is indeed shared application state, such a vulnerability falls within our characterization of client-state manipulation vulnerabilities. Notice that the analysis output includes a trace from the source to the sink, which can make it easier to confirm or dismiss the error by manual inspection. If we manually correct this error by changing the field into a local variable, the analysis

[16] `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`

```
Write of client-state value 'nickname' to application state
on line 23 of sukhwinder.chat.servlet.SaveInfoServlet
Trace:
 sukhwinder.chat.servlet.SaveInfoServlet:
  void doGet(HttpServletRequest,HttpServletResponse)
```

Figure 8.8: Output from the WARLORD tool for the *JSPChat* benchmark.

finds another error: the application is also prone to a classical client-state manipulation attack, since a malicious user may change the `nickname` request parameter and consequently change the information for another user. This error can be corrected by fetching the nickname from the session instead of a client-state parameter. After also correcting this error, WARLORD gives no more warnings. A manual inspection confirms that the remaining occurrences of client-state parameters are indeed safe. No customization is necessary for this application.

### Hipergate

An extreme number of client-state parameters to pass data between pages in this web application. All client-specific values are passed around using hidden fields. Running the analysis yields 197 warnings. With 14 customizations this number is brought down to 132 warnings, almost all of which are caused by client-state parameter values that flow into parameterized database queries without any checks. We have inspected all of the warnings, and many of them correspond to code that is vulnerable to attacks. The main source of false positives originates from a use of randomly generated ID strings for database rows. Such strings are hard to guess and we do not consider this as vulnerable. If we exclude warnings given on uses of these random strings, 71 warnings remain.

All in all, 40 of the warnings reveal exploitable client-state manipulation vulnerabilities. One of the warnings reveals that a file can be read from the disk using a file name originating from a client-state parameter in `wb_style_persist.jsp`. This parameter can be exploited to change files on the disk. Although the programmer has carefully inserted authorization checks to ensure that the user should be granted access to the page in question, no checks are made for any of the client-state parameters, and they can therefore be manipulated by the client. The tool also gives a warning on the page `docrename_store.jsp`, which can be exploited to rename files. The programmer has inserted a check to ensure that the user has rights to rename the files, but this is performed on another parameter than the one holding the file name, and an attacker can therefore create an exploit that changes only the file name. Furthermore, the tool emits 4 warnings for the page `reference.jsp` where parameters can be injected into an SQL string. 1 warning on the page `catusrs_store.jsp` reveals that a client-state parameter can give access to update permissions for any user, and 2 warnings reveal a similar problem for `catgrps_store.jsp`. Similarly, 31 warnings in 18 other pages reveal places where client-state values give direct access to the database. In all cases, data is queried or changed in the database using a client-state parameter.

For the remaining 31 warnings, we found that they could not be exploited. In three cases, the parameters control settings for querying the database without affecting the result, for example the number of rows queried at a time. In additional three cases, the values are references to objects that are owned by the user and changing these values does not give access to new information. In the remaining

cases, values flow to the database API but the queries are only used for logging client actions or for retrieving data that is used for access control. The current customization mechanism is not able to express the precise behavior of SQL expressions that are executed through calls to the JDBC API and therefore the analysis considers all such calls as sinks. The tool is able to classify 2548 out of 2680 uses of client-state parameters as safe.

### Takatu

The analysis identifies 1840 client-state parameters. 184 warnings are issued, all but 14 caused by reading from the database using an ID that comes from a hidden field. These IDs are used for querying objects from the database. After manually inspecting the warnings we can see that 162 of them can be exploited to change or read data on the server. Other 6 warnings indicate places where values are read from the database in ways that are not vulnerable, for example for searching for values in the database. The remaining 14 warnings indicate places where a client-state parameter holds the value of a log flag that is used to query the database but none of them can be exploited. No customization is required for this application.

Interestingly, this web application at multiple places asks the user to confirm the deletion of an object. The ID of the object is stored in a hidden field that is not protected, so the client can delete any object of the same type by modifying the ID used as object reference. The errors are easily corrected, for example by signing the vulnerable parameters and checking the signature when the parameter is sent back to the server.

### Pebble

WARlord identifies 22 uses of client-state parameters and initially produces 4 warnings. This web application uses a dispatcher, so all requests except those to JSP pages go through a single servlet. The number of client-state parameters seems small because of this structure, but the classes being dispatched to make heavy use of the client-state parameters.

The web application stores files on the disk such that each blog has its own directory, and it uses the value of a parameter from a hidden field to determine the name of the file to save to, which is the cause of 2 warnings. However, each value used this way is verified to be a child of the blog folder, so the folder structure ensures that users cannot overwrite each other's files. The two first customization rules shown in Figure 8.9 handle this check of the parent folder.

```
net.sourceforge.pebble.util.FileUtils.underneathRoot(File,File):
    Sanitizer for arg 2
net.sourceforge.pebble.domain.FileManager.isUnderneathRootDirectory(File):
    Sanitizer for arg 1
net.sf.ehcache.Element.get(Serializable):
    Not a sink
net.sourceforge.pebble.index.StaticPageIndex.getStaticPage(String):
    Not a sink
net.sourceforge.pebble.util.FileUtils.getContentType(String):
    Not a sink
```

Figure 8.9: Customization rules for the *Pebble* benchmark.

Only 1 warning is produced after the customization. It is caused by the page where a new blog is added. This page uses an `id` parameter originating from a hidden field to set the database ID of the newly created blog and to create a directory for the files belonging to the blog. The `id` parameter is verified to only contain letters, and another check ensures that the ID is not already in use. Together, these two checks mean that there are no exploitable vulnerabilities related to the 4 warnings. The safety depends on a subtle invariant about the directory structure where files are stored on the disk. While this invariant is beyond what we can express with the customization mechanism, extracting the relevant code into a separate method would make the code easier to read, less prone to become vulnerable as a result of future changes, and it would become expressible as a sanitizer using the customization mechanism.

### Roller

The developers of this web application have systematically reviewed the code for the class of vulnerabilities we are trying to detect. All client-state parameters are protected with authorization checks that are well-documented in the code. Running WARLORD initially results in 53 warnings on the 53 pages. We added 14 customization rules, which mainly describe information flow for a few string manipulation functions and information about queries of public information such as blog comments. Those functions are part of the Apache Commons API, so these rules are generally useful in all applications that use this API.

Only 1 warning remains after adding these rules. That warning refers to a page that allows blog comments to be deleted using a client-state parameter to identify the blog comments. All comments belong to a blog, and user rights are defined for each blog. The page checks whether each comment belongs to the blog and refuses any attempt to delete comments on other blogs in a way that cannot be modeled with our customization mechanism. However, if the code was rewritten slightly to use a separate method to check the ownership directly, this method could be marked as a sanitizer. That would also make it possible to check that future changes to this code do not create vulnerabilities, and it would make the code more readable.

### JWMA

This web application acts as a front-end for an email server using the Java Mail API, and it stores almost all data in the session state belonging to the user. It has little shared application state, but it uses some client-state parameters as part of it flow.

The HTML view is generated through JSP pages and form data is handled using servlets. The behavior of the receiving servlet is determined by one of two hidden fields, `acton` and `todo`. The behavior depends only on implicit information flow from these two parameters and no warnings are issued in relation to them. Inspecting the use of the parameter values manually does not reveal any vulnerabilities either.

With no customizations, WARLORD produces 3 warnings. Two of them are spurious warnings related to reading and using the values of the client-state parameters `paths` and `contact.id` in the servlet `JwmaController`. Request parameters are read using a method on the class `JwmaSession` and WARLORD is unable to analyze this precisely enough to determine that these two parameters are not read by `JwmaController`.

The third warning relates to the client-state parameter `numbers`, which is used for moving and deleting messages in `JwmaController`. Through manual inspection we have found that this parameter is not vulnerable since it only allows manipulation of data in the client's own folder.

### JsForum

This web application uses a combination of JSP pages for generating the HTML view and Servlets for updating data in the database. The database connection uses the standard JDBC API for accessing a MySQL database.

Client-state parameters are primarily used for storing database identifiers. WARLORD reveals that the programmers have not protected the application against client-state manipulation attacks. Without customization, WARLORD produces 12 warnings, all of which relate to the use of database identifiers. In the servlet `AddThread`, the analysis warns that the client-state parameters `lastThread_id` and `forum_id` are stored in an application state object. This happens because the servlet generates an SQL query based on these parameters and stores the query string in a field reachable from the servlet class. The methods are not synchronized and another request might therefore override the value before it is sent to the database. Other warnings reveal that clients can change the values of `forum_id` and `lastThread_id` to post to a different forum and to manipulate the identifier of a newly created thread. Furthermore, the servlet `AddThread` allows the client to post as a different user by changing the value of the hidden field named `user`.

Further inspection of the other pages reveals similar vulnerabilities in the servlets `ChangeMessage`, `AddReply`, and `AddForum`. In `DeleteForum`, however, the application code checks that the client is an administrator before deleting a forum. We therefore do not consider that servlet to be vulnerable. The customization mechanism is not able to express such a property. Of the 12 warnings, 11 corresponded to actual client-state manipulation vulnerabilities. No customization was used.

### JavaLibrary

This small JSP application for managing a book library with operations for reserving and lending books and managing a list of users with varying levels of privileges. WARLORD detects 92 client-state parameters in the application and deems 65 of them safe. Of the remaining 27 parameters, 22 are read by the servlet `FormProcess`.

JavaLibrary uses a bean for representing all values related to users. This bean is updated from client-state values when a user is added or edited. The JSP page `user_form.jsp` is used for creating and editing users. Depending on the rights of the user, the page prefills the HTML form with hidden fields. The `FormProcess` does not check for client-state manipulation and it is therefore possible to modify many of these parameters to gain privileges similar to that of an administrator when creating or editing users. This accounts for 7 of the 22 warnings. Furthermore, client-state manipulation through other forms can be exploited to change reservation dates and due dates for borrowed books and to borrow books for other users. All of the 22 parameters can be exploited for attacks.

The remaining 5 warnings that are not related to `FormProcess` result from client-state in JSP pages. According to comments in the code, state is saved in these fields to allow the client to return to the page later and complete the data entry. Similarly to *JSPChat*, this creates a possibility of a timing attack, and in this application it also allows clients to read values entered by other clients.

No customization was necessary for this web application.

### BodgeIt

This web application was written as a benchmark for penetration testing tools. It contains what the authors call “hidden (but unprotected) content” and “insecure object references”, which are within our definition of client-state vulnerability. It therefore serves well as a test for our static analysis.

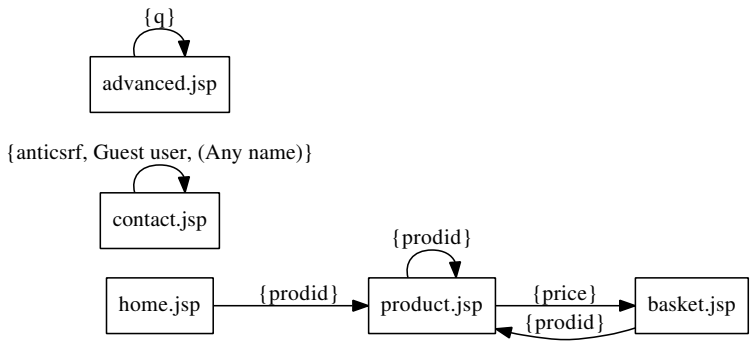


Figure 8.10: An excerpt of the page graph for *BodgeIt* showing the nodes and edges that involve client state.

Figure 8.10 shows the part of the page graph for *BodgeIt* that involves client state. Before customization, WARLORD reports 5 warnings. We added the two customization rules shown in Figure 8.11. They relate to the use of an encrypted token in `contact.jsp` for protecting against cross-site request forgery. One warning refers to a parameter named `prodid` in the JSP page `product.jsp`. The value originates from a URL parameter and is used to query the database for a product with the corresponding database ID. While this page demonstrates the possibility for client-state manipulation, changing the parameter does not give the client access to additional information. Consequently, there is no vulnerability in this case. Another warning, which originates from the URL parameter `typeid`, is also used for a database query. Manipulating this parameter does not give access to new information either.

In the JSP page `contact.jsp`, WARLORD detects a hidden field. Rather unusually, the name of this hidden field can be arbitrary, because it is set to the name of the current user. This causes WARLORD to consider all parameters in the successor page, which is `contact.jsp` itself, as client-state parameters, which results in a false positive when storing the value of the `comments` parameter. Although this warning does not indicate a possible client-state manipulation vulnerability, it reveals the possibility of a name clash if a user is registered with the name "`comments`".

The JSP page `basket.jsp` page places an item in a shopping basket along with the price of the item. WARLORD gives a warning for the `productid` parameter. The client is able to arbitrarily change this parameter to add any item to the basket, however, we classify this as another false positive because the client is already able to add any item to the basket without client-state manipulation. The item price is stored in a hidden field called `price` but this field is never read, so the user cannot gain any extra privileges by changing its value, and WARLORD correctly yields no warning in this case.

In conclusion, we find that there are, surprisingly, no exploitable client-state manipulation vulnerabilities in this web application.

### WebGoat

We have analyzed a single servlet in this web application. The purpose of the servlet, which uses a single hidden field, is to demonstrate vulnerabilities of exactly the kind we want to detect. Unlike the other benchmarks, this application generates output using a custom DOM-like framework and we decided to manually create the set of parameters that may hold client-state values.

Perhaps surprisingly, our tool reports 0 warnings for this application. The reason is that the servlet does not use the input variable for anything else than selecting a message to send back to the client. This usage does not violate any of the safe usages presented in Section 8.2 and we therefore conclude that while the illustrative servlet of course mimics the behavior of a vulnerable piece of server code, it is actually not vulnerable to any attack. A manual inspection of the code confirms that the client is indeed not able to change the shared application state in any way by changing the value of the hidden field.

### 8.8.2 Summary of Results

Figure 8.12 summarizes the benchmark results from the previous section. The first column, 'Client-state parameters', is the same as in Figure 8.7. The following columns show the number of warnings before customization, the number of customization rules, and the number of warnings after customization. The tool produces at most one warning for each of the client-state parameters from the first column (however each warning may contain multiple traces from sources to sinks). The next column, 'Exploitable', shows how many of the warnings we could manually verify to be exploitable by malicious clients performing client-state manipulation attacks. The column 'Safe client-state parameters' shows the number of client-state parameters that the analysis after customization determines *not* to be vulnerable. The final column shows the time spent for the full analysis. Figure 8.13 shows

`com.thebodgeitstore.util.AES.hexStringToByteArray(java.lang.String):`
Flow from parameter 1 to return value
`com.thebodgeitstore.util.AES.decryptCrt(java.lang.String):`
Not a sink (the method returns a safe value)

Figure 8.11: Customization rules for the *BodgeIt* benchmark.

| | Client-state parameters | Warnings before customization | Customization rules | Warnings after customization | Exploitable | Safe client-state parameters | Time |
|---|---|---|---|---|---|---|---|
| *JSPChat* | 19 | 1 | 0 | 1 | 1 | 18 | 30 s |
| *Hipergate* | 2680 | 197 | 14 | 132 | 40 | 2548 | 116 m |
| *Takatu* | 1840 | 184 | 0 | 184 | 162 | 1656 | 20 m |
| *Pebble* | 22 | 4 | 5 | 1 | 0 | 21 | 10 m |
| *Roller* | 86 | 53 | 14 | 1 | 0 | 85 | 4 m |
| *JWMA* | 40 | 3 | 0 | 3 | 0 | 37 | 3 m |
| *JsForum* | 14 | 12 | 0 | 12 | 11 | 2 | 1 m |
| *JavaLibrary* | 92 | 27 | 0 | 27 | 27 | 65 | 2 m |
| *BodgeIt* | 8 | 5 | 2 | 4 | 0 | 4 | 2 m |
| *WebGoat* | 1 | 0 | 0 | 0 | 0 | 1 | 30 s |

Figure 8.12: Summary of experimental results.

the results after grouping together data that involve parameters of the same name, which, gives an indication of the variety as in Figure 8.7.

The tests have been performed on a 2.4 GHz Core i5 laptop running OS X. The JVM was given 1 GB of heap space for each benchmark. The time and memory was primarily used by the Soot framework for loading classes and performing the pointer analysis.

With this, we are able to answer the research questions:

Q1: A manual inspection of the application code confirms that the client-state analysis succeeds in finding all client-state value sources. This amounts to a total of 4802 client-state parameters. The analysis determines that 92% of those parameters are safe, that is, they are not involved in any warnings. Moreover, 241 of the 353 warnings that are produced in total reveal exploitable vulnerabilities. The false positives are not evenly distributed among the benchmarks, and they are concentrated on a small number of different parameter names.

Q2: Based on the warnings given by the tool, especially the trace information, it was in each case possible for us to quickly determine whether it indicated a vulnerability or not. The entire process of classifying the warnings and adding customization rules for all 10 benchmarks took one person less than a day, despite having no prior knowledge of the benchmark code.

Q3: Adding customization rules in many cases reduced the number of spurious warnings considerably. As discussed for the individual benchmarks, the remaining cases typically involve subtle, undocumented invariants. Moreover, if allowing simple refactorings, such as extracting a safety check to a separate method, most of these cases could be captured within the existing customization framework. In the case of *Hipergate*, however, some uses of client state are safe for reasons that go beyond the current capabilities of customization. The decision mentioned in Section 8.3 that the analysis ignores condition 5

| | Client-state parameters | Warnings before customization | Customization rules | Warnings after customization | Exploitable | Safe client-state parameters | Time |
|---|---|---|---|---|---|---|---|
| *JSPChat* | 3 | 1 | 0 | 1 | 1 | 2 | 30 s |
| *Hipergate* | 264 | 99 | 14 | 75 | 30 | 189 | 116 m |
| *Takatu* | 31 | 10 | 0 | 9 | 9 | 22 | 20 m |
| *Pebble* | 11 | 4 | 5 | 1 | 0 | 10 | 10 m |
| *Roller* | 27 | 1 | 14 | 1 | 0 | 26 | 4 m |
| *JWMA* | 10 | 3 | 0 | 3 | 0 | 7 | 3 m |
| *JsForum* | 8 | 7 | 0 | 7 | 7 | 1 | 1 m |
| *JavaLibrary* | 35 | 22 | 0 | 22 | 22 | 13 | 2 m |
| *BodgeIt* | 8 | 5 | 2 | 4 | 0 | 4 | 2 m |
| *WebGoat* | 1 | 0 | 0 | 0 | 0 | 1 | 30 s |

Figure 8.13: Summary of experimental results, as Figure 8.12 but here grouping together the warnings according to the client-state parameter names to illustrate the variety of the potential vulnerabilities.

from Section 8.2 results in a few false positives in *BodgeIt*, *Hipergate*, and *Takatu*.

Q4: The tool analyzes between 10 and 200 pages per minute. Pages can be analyzed individually, so when a programmer is modifying the application, he can decide to run the tool only on pages that have changed.

## 8.9 Related Work

Client-state manipulation vulnerabilities, in particular the kind involving hidden fields, have been known for many years, as described in Sections 8.1 and 8.2. Likewise, automated techniques for protecting against security vulnerabilities in web applications have a long history. We here explain the connections between our approach and the most closely related alternatives that have been proposed.

One direction of work is using runtime enforcement of security policies, as exemplified by the security gateway proposed by Scott and Sharp [76]. Given a security policy, their gateway can, for example, automatically attach MACs to hidden fields. The approach requires that the programmer specifies which input fields need this kind of protection, which, as discussed in Section 8.1, is too easy to forget. In contrast, the idea in our approach is to inform the programmer – using static analysis of the application source code – that protection may be inadequate. We adopt Scott and Sharp's security gateway as presented in Section 8.7, however the configuration of our security filter is provided by static analysis, not by the programmer.

An essential constituent of our approach is the observation that client-state manipulation vulnerabilities are correlated to information flow from client state to application state. Together with automatic inference of client state (Section 8.4) and shared application state (Section 8.5), this allows us to detect likely errors largely without requiring the programmers to provide any specifications. Some application specific customization is required though, as seen in Section 8.8. For future work, it may be interesting to apply probabilistic specification inference [50] to automate this phase.

The WebSSARI tool by Huang et al. [31] pioneered the use of static information flow analysis to enforce web application security, and numerous researchers have since followed that path (see for example [37, 49, 83, 88, 90]). Our proof-of-concept implementation uses a simple information flow analysis, as described in Section 8.6. More advanced alternatives include the algorithms by Livshits and Lam [49] and Tripp et al. [83].

The first phase of our analysis that identifies the client-state parameters (Section 8.4) applies techniques from our earlier work on static analysis of HTML output of Java-based web applications [40,60]. The WAM-SE and WAIVE analysis tools by Halfond et al. [26,27] also infer interface specifications for web applications, however without identifying which parameters contain client state, for example originating from hidden fields.

Providing comprehensive support for diverse web application frameworks, such as Java Servlets, JSP, and Struts, is a challenging endeavor. A general framework, F4F, has recently been proposed by Sridharan et al. [78], however we have found that it is not sufficiently flexible for our setting, in particular for the client state identification phase. Still, the ideas in F4F may be adapted in future work to enable support for additional web application frameworks.

Finally, we note that several commercial tools are capable of detecting security vulnerabilities in web applications. According to a 2007 IBM white paper [32], the AppScan tool is capable of detecting vulnerabilities involving hidden field manipulation and parameter tampering. The latest version uses techniques from TAJ [83], however we have been unable to perform a proper comparison and obtain further

information about the techniques applied by AppScan. Microsoft's CAT.NET[17] also uses static information flow analysis, but it cannot detect client-state manipulation vulnerabilities without detailed specifications provided by the user. Other commercial tools include NTOSpider[18] from NT OBJECTives, WebInspect[19] from Fortify/HP, and CodeSecure[20] and HackAlert[21] from Armorize. To our knowledge, most of these tools (with the exception of CodeSecure, which is developed from WebSSARI) employ crawling [2, 18], not static analysis. We believe static analysis can be a promising supplement to dynamic approaches as it may provide better coverage of the web application source code.

## 8.10 Conclusion

We have demonstrated that it is possible to provide tool support that can effectively help programmers prevent client-state manipulation vulnerabilities in web application code. The static analysis we have presented is capable of precisely identifying client state, in particular state stored in hidden fields, and help distinguishing between safe and unsafe use of such state. With WARLORD, our prototype implementation of the analysis, we quickly discovered 230 exploitable weaknesses in 10 web applications. The analysis has high precision: for a total of 4802 non-exploitable client-state parameters, 92% were classified as safe.

Moreover, we have argued that the information inferred by the analysis can also be used for automatic configuration of a security filter that at runtime protects against client-state manipulation attacks.

Our experiments also indicate potential for improvements. Specifically, although analyzing the *Hipergate* benchmark revealed 40 weaknesses, it also resulted in a number of false positives originating from a small group of client-state parameters. It appears that many of these false positives can be avoided if the analysis is extended to also infer the provenance of the client-state values, which can be a subject for future work. It may also be worthwhile to extend the technique to reason about client state stored in cookies.

[17] `http://blogs.msdn.com/b/securitytools/archive/2010/02/04/cat-net-2-0-beta.aspx`
[18] `http://www.ntobjectives.com/ntospider`
[19] `https://www.fortify.com/products/web_inspect.html`
[20] `http://armorize.com/index.php?link_id=codesecure`
[21] `http://armorize.com/index.php?link_id=hackalert`

# Bibliography


[1] Advosys Consulting. Preventing HTML form tampering, 2000. `http://advosys.ca/tips/form-tampering.html`.

[2] Jason Bau, Elie Bursztein, Divij Gupta, and John C. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proc. 31st IEEE Symposium on Security and Privacy*, 2010.

[3] Alexandre Bergel, Stéphane Ducasse, and Lukas Renggli. Seaside - advanced composition and control flow for dynamic web applications. *ERCIM News*, 2008(72), 2008.

[4] Geert Bevin. RIFE, 2007. `http://rifers.org/`.

[5] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Reasoning about web applications: An operational semantics for hop. *ACM Trans. Program. Lang. Syst.*, 34(2):10, 2012.

[6] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[7] David I. Brussin. A white paper analyzing the MSC hidden form field web site vulnerability, 1998. Miora Systems Consulting.

[8] Steve Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987.

[9] Shan Chen, Dan Hong, and Vincent Y. Shen. An experimental study on validation problems with existing HTML webpages. In *Proc. International Conference on Internet Computing, ICOMP '05*, June 2005.

[10] Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. A runtime system for XML transformations in Java. In *Proc. Database and XML Technologies, 2nd International XML Database Symposium (XSym)*, volume 3186 of *LNCS*. Springer-Verlag, August 2004.

[11] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[12] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[13] Douglas Crockford. JavaScript Object Notation (JSON), 2006. RFC4627. `http://tools.ietf.org/html/rfc4627`.

[14] Pierre Delisle, Jan Luehe, Mark Roth, and Kin man Chung. *JavaServer Pages Specification*. Sun Microsystems Inc., 2.2 edition, November 2009.

[15] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20, July 1977.

[16] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[17] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Static command-injection-attack detection based on abstract parsing. Technical report, Hanyang University, Kansas State University, 2012.

[18] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2010.

[19] Sebastian G. Elbaum, Kalyan-Ram Chilakamarri, Bhuvana Gopal, and Gregg Rothermel. Helping end-users "engineer" dependable web applications. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 31–40, 2005.

[20] Asger Feldthaus and Anders Møller. *The Big Manual for the Java String Analyzer*. Department of Computer Science, Aarhus University, November 2009.

[21] Roy Fielding. *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group, 2009.

[22] Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., New York, NY, USA, 1966.

[23] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[24] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1991.

[25] Google. Google Web toolkit - build AJAX apps in the Java language. `http://code.google.com/webtoolkit/`.

[26] William G. J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proc. International Symposium on Software Testing and Analysis*. ACM, July 2009.

[27] William G. J. Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2008.

[28] David Heinemeier Hansson et al. Ruby on Rails, 2008. `http://www.rubyonrails.org/`.

[29] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[30] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrve. Document object model (dom) level 3 core specification. W3C Recommendation, April 2004.

[31] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proc. 13th International World Wide Web Conference*, May 2004.

[32] IBM. The dirty dozen: preventing common application-level hack attacks, 2007. `ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_dirtydozen.pdf`.

[33] Internet Security Systems. Form tampering vulnerabilities in several web-based shopping cart applications, 2000. ISS E-Security Alert, `http://www.iss.net/threats/advise42.html`.

[34] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[35] Rod Johnson et al. Spring MVC, 2008. `http://www.springframework.org/`.

[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[37] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[38] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 258–263, 2006.

[39] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, September 2005. Presented at Programming Language Technologies for XML (PLAN-X).

[40] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006.

[41] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[42] Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11:269–289, 1967.

[43] G. Krasner and S. Pope. A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[44] Shriram Krishnamurthi, Peter Walton Hopkins, Jay A. McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[45] Arnaud Le Hors, Dave Raggett, and Ian Jacobs. *HTML 4.01 Specification*, December 1999.

[46] Avraham Leff and James T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, EDOC '01, pages 118–, Washington, DC, USA, 2001. IEEE Computer Society.

[47] Rasmus Lerdorf et al. PHP, 2008. `http://www.php.net/`.

[48] Benjamin Livshits. Defining a set of common benchmarks for web application security. In *Workshop on Defining the State of the Art in Software Security Tools*, August 2005.

[49] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. 14th USENIX Security Symposium*, August 2005.

[50] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[51] Kin man Chung, Pierre Delisle, Jan Luehe, and Mark Roth. *Expression Language Specification*. Sun Microsystems Inc., 2.1 edition, May 2006.

[52] Shane McCarron, Masayasu Ishikawa, and Murray Altheim. *HTML 1.1 - Module-based XHTML - Second Edition*, November 2010.

[53] Craig R. McClanahan et al. Struts, 2002. `http://jakarta.apache.org/struts/`.

[54] Microsoft. ASP.NET, 2008. `http://www.asp.net/`.

[55] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proc. 14th International Conference on World Wide Web, WWW '05*, pages 432–441. ACM, May 2005.

[56] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In *Proc. 4th Asian Symposium on Programming Languages and Systems, APLAS '06*, November 2006.

[57] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. 10th International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[58] Anders Møller and Michael I. Schwartzbach. XML graphs in program analysis. *Science of Computer Programming*, 76(6):492–515, June 2011. Earlier version in Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM) 2007.

[59] Anders Møller and Mathias Schwarz. JWIG: Yet another framework for maintainable and secure web applications. In *Proc. 5th International Conference on Web Information Systems and Technologies*, March 2009.

[60] Anders Møller and Mathias Schwarz. HTML validation of context-free languages. In *Proc. 14th International Conference on Foundations of Software Science and Computation Structures*, 2011.

[61] Anders Møller and Mathias Schwarz. Automated detection of client-state manipulation vulnerabilities. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 749–759, 2012.

[62] Anders Møller and Mathias Schwarz. *JWIG user manual*. Department of Computer Science, Aarhus University, 2012.

[63] Lee Moran. How Citigroup hackers broke in 'through the front door' using bank's website, 2011. http://www.dailymail.co.uk/news/article-2003393/How-Citigroup-hackers-broke-door-using-banks-website.html.

[64] Rajiv Mordani. *Java Servlet Specification*. Sun Microsystems Inc., version 3.0 rev a edition, December 2009.

[65] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[66] Takuya Nishiyama and Yasuhiko Minamide. A translation from the HTML DTD into a regular hedge grammar. In *Proc. 13th International Conference on Implementation and Application of Automata, CIAA '08*, volume 5148 of *LNCS*, July 2008.

[67] Elizabeth J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1351–1356, 2008.

[68] Open Web Application Security Project. OWASP top 10, 2010. `https://www.owasp.org/`.

[69] OWASP. The ten most critical web application security vulnerabilities, 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[70] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[71] Php: Hypertext preprocessor.

[72] David Robinson. *The WWW Common Gateway Interface Version 1.1*, May 1995.

[73] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[74] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[75] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proc. 15th International Conference on Financial Crypto*, February 2011.

[76] David Scott and Richard Sharp. Abstracting application-level web security. In *Proc. 11th International World Wide Web Conference*, May 2002.

[77] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 975–985, 2006.

[78] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint analysis of framework-based web applications. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2011.

[79] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[80] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, January 2006.

[81] The Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley, 1996. `http://www.unicode.org/`.

[82] Peter Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI '05*, 2005.

[83] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[84] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON '99*. IBM, November 1999.

[85] W3C. Web services activity, 2008. `http://www.w3.org/2002/ws/`.

[86] Jos Warmer and Sylvia van Egmond. The implementation of the Amsterdam SGML parser. *Electronic Publishing*, 2(2):65–90, 1988.

[87] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 32–41, 2007.

[88] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proc. 30th International Conference on Software Engineering*. ACM, May 2008.

[89] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[90] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symposium*, July/August 2006.

[91] ZDNet. New e-rip-off maneuver: Swapping price tags, 2001. http://www.zdnetasia.com/new-e-rip-off-maneuver-swapping-price-tags-21187583.htm.