AUTOMATIC UNIT TESTING TO DETECT SECURITY VULNERABILITIES IN WEB APPLICATIONS

by

Mahmoud Mohammadi

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2018

ABSTRACT

MAHMOUD MOHAMMADI. AUTOMATIC UNIT TESTING TO DETECT SECURITY VULNERABILITIES IN WEB APPLICATIONS. (Under the direction of DR. BEI-TSENG CHU)

Web applications consume data from different inputs. Some of these inputs originate from untrusted sources, such as user inputs which will be rendered in browsers or browser-based applications such as mobile apps. Many applications with these functions are subject to cross-site scripting attacks (XSS), as injected malicious inputs can cause undesired remote code execution in browsers. The prevalence of these script injection attacks is due to a mixture of data and code in web pages. To prevent such Cross Site Scripting (XSS) attacks, one of the most common security attacks today, web applications should sanitize untrusted data using output encoding functions before displaying them on web pages. To successfully prevent XSS attacks, the encoding must match the context in which untrusted data appears, such as HTML body, JavaScript, and style sheets. A common programming error is the use of a wrong type of encoder to sanitize untrusted data, leaving the application vulnerable.

I introduce a security unit testing approach to detect XSS vulnerabilities caused by improper encoding of untrusted data. Unit tests for the XSS vulnerability are constructed out of each web page and then evaluated by a unit test execution framework. A grammar-based attack generator is devised to automatically generate test inputs. I also introduce a vulnerability repair technique that can automatically fix detected vulnerabilities in many situations. Evaluation of this approach has been conducted on an open source medical record application written in JSP.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Bill Chu for his knowledge and support of my Ph.D study and research and his guidance in all the time writing of papers and this dissertation. And I also would like to express the deepest appreciation to Dr. Heather Richter Lipford for her continues support, enthusiasm, and immense knowledge.

Besides my advisers, I would like to thank the rest of my dissertation committee: Dr. Weichao Wang, Dr. Meera Sridhar, and Dr. Sheng-Guo Wang, for their encouragement, insightful comments, and questions.

DEDICATION

This dissertation is dedicated to my lovely wife Armaghan and my beloved kids Sheida and Neakan.

I also dedicate this work to my mother and father. You have successfully made me the person I am becoming.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AST	Abstract Syntax Trees
XSS	Cross-Site Scripting Attacks

CHAPTER 1: Introduction

Web applications consume data from different inputs. Some of these inputs originate from untrusted sources, such as user inputs which will be rendered in browsers or browser-based applications such as mobile apps. Many applications with these functions are subject to cross-site scripting attacks (XSS). Cross-Site Scripting (XSS) is one of the most common security vulnerabilities in web applications. These attacks occur when an attacker successfully injects a malicious JavaScript payload into a web page to be executed by users requesting that page. The Acunetix Web Application Vulnerability Report [4] showed that nearly 38% and 33% of web sites were vulnerable to XSS attacks in 2015 and 2016 respectively.

The prevalence of these script injection attacks is due to mixture of data and code in web pages. To prevent such Cross Site Scripting (XSS) attacks, one of the most common security attacks today, web applications should sanitize untrusted data using output encoding functions before displaying them on web pages. While one can prevent all XSS attacks by using the most strict encoder, that also takes away many useful web site functions. To balance security and functionality, developers must therefore choose the appropriate encoder depending on the context of the content. The following contexts have been identified for a typical web application: HTML, CSS, URL, and JavaScript. Well-tested encoders have been written for each of these contexts. A common programming error is that one chooses a wrong encoder for a given application context. Research shows that as many as 28% of encoders are used incorrectly [5].

```
1  <% String pid=(String)request.getParameter("pid");%>
2  <% String addr=(String) request.getParameter("addr");%>
3  <a onclick="fn('<%=escapeHtml(pid)%>')" href="#" >  mylink </a>
4  <p> <%=escapeHtml(addr) %>
```

Figure 1.1: Motivating Example

Consider the fragment of a JSP program shown in Figure 1.1. Native Java code is enclosed in <% and %>. This example has two user-provided inputs: *pid* and *addr*. Variable *pid* is used as part of rendering an HTML anchor element on line 3, and *addr* is displayed in the HTML body on line 4. A maliciously supplied input for *addr* might be

```
<script> atk(); </script>
```

If the encoding function, *escapeHtml()*, was not applied, the JavaScript function *atk()* on line 4 would be executed. The encoding function *escapeHtml()* replaces the < and > characters with &lt; and &gt; respectively and transforms the malicious input into the following string, preventing *atk()* from being interpreted as a JavaScript program by the browser:

```
&lt;script&gt; atk(); &lt;/script &gt;
```

However, the same encoding function does not work for the case on line 3. A malicious input for *pid* might be the following:

```
'+ atk() + '
```

It will pass *escapeHtml()* unchanged. The rendered anchor element would be as follows:

```
<a onclick= "fn(''+ atk()+'')" href= "#" > mylink </a>
```

JavaScript function *atk()* will be executed as part of evaluating the input parameter expression of function *fn()* when the link is clicked. The correct JavaScript encoder would, in this case, replace the single quote character with escaped single quote \' to prevent this attack.

There are also cases where more than one encoding function must be used (e.g. an untrusted input used in both JavaScript and HTML contexts). The order of applying encoders is sometimes important as well. For example, Figure 1.2 shows a case in which the order of encoders is incorrect. The order is incorrect because the JavaScript encoder on line 3 is intended to prevent successful attacks by encoding single and double quote characters as they can be used to shape successful attacks for the *onclick* attribute on line 4. However, in this case, the first encoder (*escapeHtmlDecimal()*) replaces single quote characters with &#39 characters and this character combination will not be changed by the second encoder (*escapeJavaScript()*). Thus, the encoded string by the first encoder (&#39); attack(); //) can pass through the second encoder and be sent to the browser intact. Unfortunately, browsers will decode (&#39); attack(); //) back to the original attack string ('); attack(); //) leading to a successful attack. This vulnerability can be fixed by reversing the order of the encoders used.

Applying the correct encoding is thus context-sensitive, meaning the encoder must match the web element context where an untrusted variable occurs. In practice, a variable can occur in one of the following four contexts: HTMLbody, JavaScript, CSS, and URL. Unfortunately, there is no systematic way to detect vulnerabilities due to mismatch of encoder and context [5]. Some researchers have looked at vulnerability prevention mechanisms using type inference to automatically detect the context of an

```
<% user= request.getParameter("user");
var1 = escapeHtmlDecimal(user);
var2 = escapeJavaScript(var1)+ "cnst"; %>
<a onclick="fn('<%= var2 %>');">Details</a>
```

Figure 1.2: Incorrect Order Of Encoders

untrusted variable so the correct encoding function can be automatically applied. To aid type inference, such efforts all rely on template languages with stronger type systems, such as Closure Templates [6] or HandleBars [7]. Such approaches have several limitations. First many web applications do not use such template languages. Second, type inference is not fully successful even with template languages. For example, a research team from Yahoo! found that they could identify the correct context in about 90.9% of applications written in HandleBars using type inference. Other researchers have also shown that type inference is not always accurate for some program constructs written in Closure Templates [6].

Static analysis techniques are commonly used to ensure that applications utilize encoding functions. However, static analysis only checks the existence of these functions and cannot verify their effectiveness. Detecting XSS vulnerabilities through dynamic analysis (black box testing) has also been researched, and there are several open source and commercial implementations [8, 9, 10, 11]. In these approaches, a vulnerability is detected by inspecting web application outputs. If an injected attack payload is found in the output, the application is deemed vulnerable. However, black box testing could have high false negative rates due to inadequate test path coverage [8].

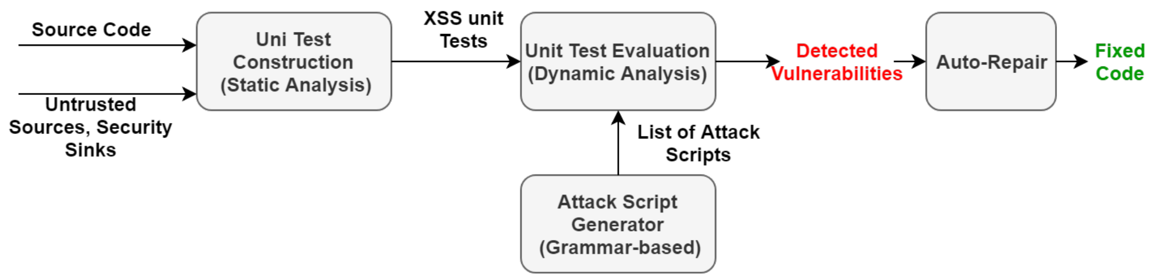

Source Code
Untrusted Sources, Security Sinks
Uni Test Construction (Static Analysis)
XSS unit Tests
Unit Test Evaluation (Dynamic Analysis)
List of Attack Scripts
Attack Script Generator (Grammar-based)
Detected Vulnerabilities
Auto-Repair
Fixed Code

Figure 1.3: Overall Architecture of Security Unit Testing

To overcome the limitations of static and dynamic analysis techniques, I present **Security Unit Testing** technique to detect and repair XSS vulnerabilities due to incorrect encoding function usage. Below is an overview of this technique and more details will be provided in chapters 3,4 and 5.

1.1 Security Unit Testing Architecture

The **Security Unit Testing** approach is a combined static-dynamic vulnerability detection technique based on unit testing which aims to reduce the false positive and false negative limitations of existing vulnerability detection techniques. I use static analysis to maximize the code coverage (and reduce false negatives) required to find all vulnerable points and dynamic analysis to ensure applicability of vulnerable points in a running application to minimize false positives. This approach is designed to be integrated into unit testing, so vulnerabilities can be mitigated early in the software development life-cycle.

The overall architecture of my approach is shown in Figure 1.3. This figure lists the approach inputs as:

- source code under the test
- configurations including sensitive operations and untrusted sources.

The output of this approach is a list of vulnerable points in the source code, many of which may be repaired by the auto-fixing component.

Key components of this XSS vulnerability detection technique are shown in 1.3 as following:

- First, the "**Unit Test Extraction**" component analyzes the source code in order to extract and generate the unit tests for XSS detection. This component represents the static analysis aspect of the architecture. Static analysis approaches used to find vulnerabilities have the advantage of complete source code coverage leading to low false negatives in detecting vulnerabilities. The Unit Test Extraction component analyzes the source code in order to automatically extract and generate unit tests for XSS detection in such a way to ensure code coverage.

- Second, the "**Unit Test Evaluation**" component confirms each vulnerability by rendering attacked pages using a headless browser in a unit testing framework. This component represents the dynamic analysis aspect of the technique. Dynamic analysis approaches can find the vulnerabilities with a low rate of false positives due to using the real results of source code execution leading to low false positive rates in detecting vulnerabilities. The Attack Evaluation component will use a repository of attack scripts, generated using the proposed Attack Generation component, to evaluate each unit test.

- Third, the "**Attack Generation**" component uses a grammar-based approach to generate attack vectors required to test the unit tests. The results of these three components will be a list of vulnerable unit tests with corresponding attack vectors used to exploit them.

- Forth, the "**Auto-fixing**" component is a mechanism based on the results of the "Attack Evaluation" component to repair many detected vulnerabilities using encoder placement. The result of the unit test evaluations is the list of vulnerable unit tests. Automatic fixing aims to fix these vulnerable unit tests. Fixing the vulnerable unit tests means refactoring the vulnerable code with different combinations of the encoders and, then re-evaluating the modified unit test with attack vectors to find an encoder that shows no vulnerability in evaluation results. Automatic repairing of the vulnerabilities is possible due to the limited number of encoders. This component uses the attack evaluation mechanism implemented in the "Attack Evaluation" component which leads avoiding false code repairs.

Throughout the paper I will use the following terminology to explain my approach.

Security Sinks refer to program statements performing operations that could be subject to XSS attacks. Specifically, these are server-side output generation commands sending values of variables to browser such as out.print() or <%= %> statements in JSP.

Untrusted Sources refer to statements retrieving information from sources that may contain malicious data. For example, *request.getParameter()* gets the data from the Internet. For my research, I assume untrusted sources are given as a set of API's returning untrusted values.

Tainted Variables are variables that obtain their values (directly or indirectly) from the untrusted sources.

Encoders are functions that are used to generate safe versions of their inputs using character encoding mechanisms such as *escapeHtml()*. Most developers use one of the widely used libraries of encoders such as ESAPI [12].

Tainted Data Flow is data flow of the tainted variable from an untrusted source to its destination in a security sink. I define each data flow as a tuple of untrusted source (U), security sink (S) and a list of encoders (E) between source and sink.

1.2 Dissertation Structure

The rest of the dissertation is organized as follow: Chapter 2 provides a literature review on different techniques used to handle the XSS vulnerabilities in web applications. This review covers two categories of techniques: 1) Vulnerability prevention mechanisms including auto-sanitization and secure coding, and 2) vulnerability detection techniques including static analysis and dynamic analysis.

Chapter 3 describes the "Unit Test Extraction" component of the approach which aims to construct the unit tests from the source code. This component uses static analyses to cover all execution paths of the web pages and generates unit tests out of these execution paths. Chapter 3.2 explains an "Attack Evaluation" technique which implements a testing mechanism to execute the generated unit tests in a unit testing framework using attack vectors. This component satisfies the need to run the unit test to prevent false positives in vulnerability detection.

Chapter 4 presents an "Attack Vector Generation" mechanism to generate the attack vectors required to evaluate the generated unit tests. This chapter introduces a grammar-based technique to cover the context-switching behavior of the browsers using grammars of HTML, CSS, JavaScript and URI.

Chapter 6 focuses on the evaluation results of the approach based on the ability of the approach to find the vulnerabilities, time efficiency of the approach and the attack coverage of the attack generation component.

Chapter 5 introduces an automatic fixing technique to fix the vulnerable unit tests reported by the **security unit testing** mechanism. This component re-factors the vulnerable unit tests with different combination of the encoders and evaluates the modified code to find the proper encoder.

Chapter 7 summarizes the dissertation contributions and outlines some future research directions.

CHAPTER 2: LITERATURE REVIEW

Finding cross-site scripting vulnerabilities due to encoding mistakes is the main motivation of this research. These mistakes are because of difficulties in inferring the correct context of the web application to apply the proper output encoder. In other words, there is not a general purpose data encoding function to be used in different contexts of HTML, JavaScript, CSS and URI. In order to deal with these encoding challenges, different approaches, including the vulnerability prevention and vulnerability detection mechanisms, have been studied. Section 2.1 highlights issues that make the effective output encoding a challenging problem for developers. In section 2.2, I review the vulnerability prevention mechanism including secure coding guidelines and auto-sanitization languages used to avoid the XSS vulnerabilities. Next, in section 2.3 I focus on vulnerability detection mechanisms based on static and dynamic analysis techniques and evaluate their context-sensitiveness in vulnerability detection. All of the prevention and detection mechanisms are compared based on Context Sensitiveness, Compatibility with Legacy Applications and Code Coverage.

2.1 Encoding Challenges

Applying an effective sanitization approach to prevent XSS attacks requires understanding how web browsers interpret web pages. Understanding the sanitization complexities and subtleties requires analyzing the sanitization process in terms of browsers' parsing behavior [6] . This analysis reveals that the interactions of the browser sub-interpreters can lead to a number of remarkable challenges in sanitization techniques. Web pages are composed of different contexts or grammars such as

HTML, CSS, JavaScript and URL and researchers have analyzed XSS defense and mitigation mechanisms based on data sanitization techniques that should consider differences of these contexts [13, 14, 15, 16, 17, 18, 19]. The encoding challenges are discussed in the following paragraphs.

Context-Sensitiveness: Encoding functions are widely used as the main defense mechanism to sanitize the data in web applications. The context-sensitiveness means that the programmer should be keenly aware of where (i.e. context) the tainted (user-controlled) variables are placed in a web page in order to apply the correct encoding. For example, when the tainted variable is used in the *href* attribute of tags such as an anchor tag (<a>) or *src* of the image tag (<img>), the HTML encoding is not sufficient. The reason is that an attack string starting with javascript: can switch the context of an *href* attribute from URI to JavaScript; leading to a successful attack. Figure 2.1 shows a piece of code with an UNTRUSTED (tainted) datum and an attack script which can be used to exploit the code. If the tainted datum is only sanitized by the HTML sanitizer, rather than the URI, it can be successfully exploited because the untrusted data is placed in the *href* attribute which actually is in the URL context and not HTML. This means an attack script starting with *javascript:* ,which can bypass the HTML encoder, can force the browser to execute the attack script appearing after the *javascript:* keyword.

```
Vulnerable Code : <a href="UNTRUSTED" > Link Text </a>
Attack Vector:   javascript: attack(); //
```

Figure 2.1: Context-Sensitive Sanitization

Nested Context : This challenge refers to cases when the data can simultaneously be interpreted in two contexts and so the encoders can easily fail if they do not

```
<% String val = escapeHTMLDecimal( UNTRUSTED); %>
<input type='button' onclick=" <%= escapeJavascriptl (  val ) %> "
/>
```

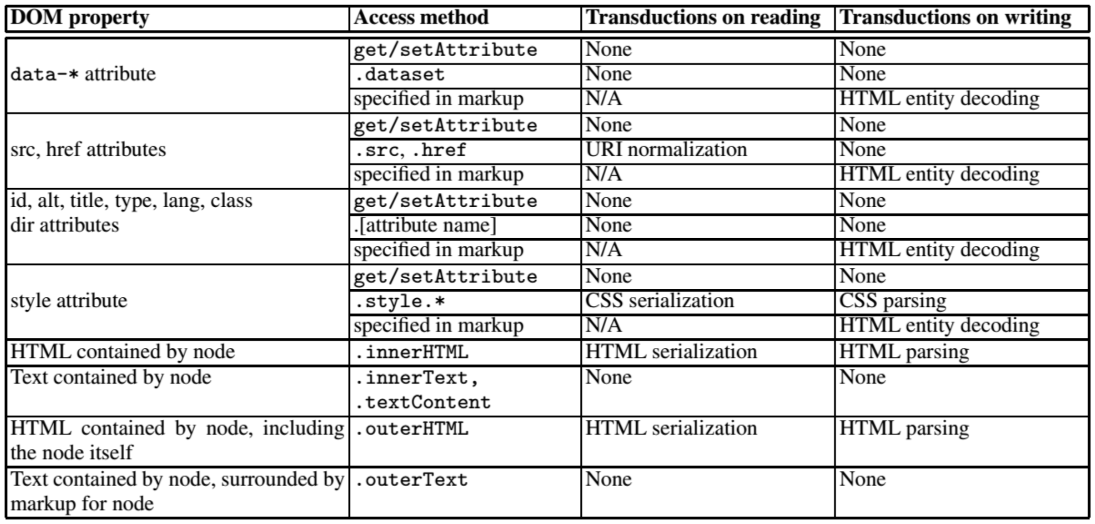
Figure 2.2: Vulnerable code having inconsistent encoders

detect these two contexts. For example, in an HTML block such as
`<a onclick="Show('UNTRUSTED');"  />`

The **UNTRUSTED** data is placed in two contexts. One context is the HTML context as the value of an HTML attribute. Simultaneously, it is interpreted as a JavaScript string, because the *onclick* is also an event attribute and all event attributes are in the JavaScript context. This means that the developer should consider both contexts to select a proper encoder, otherwise a vulnerability can occur.

Browser Decoding: This feature of the browsers introduces another issue that can undo the wrong-ordered sanitizations. In this case, the browser decodes the characters before shipping them to another sub-interpreter. This preprocessing causes the target interpreter to receive a decoded version of the previously encoded (sanitized) characters, which can undo the previously applied encoder. For example, the order of applied encoders in Figure 2.2 is inconsistent and the code is vulnerable. In this case, if the tainted (UNTRUSTED) value contains the single quote character {'} the *escapeHTMLDecimal()* escapes it to {&#39;}. Therefore, an attack string { ');attack(); // } will be changed to { &#39;);attack(); // }.

After this initial encoding, the *escapeJavaScript()* does nothing to this string because it does not contain a special token or character for this particular encoder (single quote for JavaScript encoder). Moreover, browsers have an internal decoding



DOM property	Access method	Transductions on reading	Transductions on writing
`data-*` attribute	`get/setAttribute`	None	None
	`.dataset`	None	None
	specified in markup	N/A	HTML entity decoding
src, href attributes	`get/setAttribute`	None	None
	`.src`, `.href`	URI normalization	None
	specified in markup	N/A	HTML entity decoding
id, alt, title, type, lang, class dir attributes	`get/setAttribute`	None	None
	.[attribute name]	None	None
	specified in markup	N/A	HTML entity decoding
style attribute	`get/setAttribute`	None	None
	`.style.*`	CSS serialization	CSS parsing
	specified in markup	N/A	HTML entity decoding
HTML contained by node	`.innerHTML`	HTML serialization	HTML parsing
Text contained by node	`.innerText`, `.textContent`	None	None
HTML contained by node, including the node itself	`.outerHTML`	HTML serialization	HTML parsing
Text contained by node, surrounded by markup for node	`.outerText`	None	None

Figure 2.3: Browser Transduction Details [1]

feature, known as implicit transduction [1] , that un-escapes(decodes) a string before sending it to another interpreter such as the JavaScript interpreter. In this case, the escaped attack string will be changed from {&#39;);attack; //} back to {'); attack ();// } which leaves it dangerous again. This browser decoding feature varies in different situations, and these variations can make the sanitization process even more complicated. Variations such as reading or writing HTML texts (e.g., innerText), getting or setting DOM properties using method calls (e.g. attributes set and get methods) or using property's names to access the properties, can lead to such implicit transductions. Figure 2.3 shows these differences in detail:

In summary, browsers are composed of multiple sub-interpreters for different contexts of HTML, JavaScript, CSS and URL. Interaction of these contexts introduces challenges such as context-sensitiveness, nested contexts and browser decoding which increases the complexity of the data sanitization process in web applications. Encoding functions are the main defense mechanism to sanitize data in web applications and there are correctly implemented encoders for each grammar. However, the complexities and challenges of the sanitization process can negate their effectiveness. In fact, inferring the correct context by the developers (and also by the vulnerability detection tools) can be sometimes ambiguous, shaping a vulnerability problem.

In order to deal with these sanitization challenges different approaches including the vulnerability prevention and vulnerability detection mechanisms have been studied. In section 2.2 (Vulnerability Prevention) and section 2.3 (Vulnerability Detection), I will explore these techniques in more details.

2.2 Vulnerability Prevention

Vulnerability prevention approaches aim to remove the root causes of software vulnerabilities. I explain two main categories of preventive techniques :

- 1) Secure Coding
- 2) Auto-Sanitization.

2.2.1 Secure Coding

The first approach to deal with the complexities of proper encoding and prevention of the security holes is **secure coding**. This approach is defined as guidelines to be followed by the developers in order to keep the source code free of security flaws. The goal of these programming practices, such as the XSS prevention cheat sheet by OWASP [20] or the secure coding guide lines introduced by Graff and Van Wyk [21], is to train the developers to write their code in a secure way.

While these guidelines are primarily developed to be applied manually, there are studies to extend and automate these programming guidelines. In one such study, Johns et al. [22], developed a technique to secure a host programming language by proposing an abstract data type that strictly enforces the data and code separation. The researchers achieved this by applying their technique to the Java language.

This approach introduced a data type called ELET (Embedded Language Encapsulation Type) to separate the data and code. The idea of this data type originated from the parsing algorithms used in compilers. Parsing algorithms usually contain a two-step process. The first step is lexical tokenizing, which reads and parses the source code into a stream of tokens. The second step maps identified tokens to the language's grammar elements resulting in parse trees with leaves expressing those

```
sql =  select -token , meta - char (*) , from -token , tablename -
token ( Users ), where -token , fieldname - token ( Passwd ),
metachar (=) , metachar ('), stringliteral ( mycatiscalled ),
metachar ( ')
```

Figure 2.4: A sample code in ELET data type [2]

tokens. The researchers stated that the tokens can be classified as follows: static (language's keywords), identifier (tokens with known values at compile time such as function names) and literals (tokens with dynamic values at run-time such as variables). In order to enforce the code and data separation, this classification is used to explicitly express these classes of tokens during development. For example, the SQL statement sql= SELECT * FROM Users WHERE Passwd = 'mycatiscalled' has an internal ELET representation as shown in Figure 2.4.

To integrate this data type into a programming language such as SQL or JavaScript two approaches are introduced: 1) API calls and 2) language preprocessors. API calls are based on an explicit call of an API for each token (e.g., sqlELET.addSelectToken () for 'SELECT' keyword). Using language preprocessors is a process that takes place before source code compilation in such way that every code fragment embedded between some predefined markers (e.g, $$) would be translated into API calls.

In the code shown in Figure 2.5, the *EletPrinter* is responsible for generating the final HTML output and is implemented as an external Servlet filter at the server side. This external component encodes all of its input data to mitigate any potential code injection vulnerability. This encoding happens just before generating the final output to be sent to the client-side. In fact, this server side filter parses the input HTML or JavaScript data (the hout variable in the example) and detects the proper context

```
String bad = req.getParameter (" bad ");
HTMLElet hout $=$ <h3 > Protection test </h3 >$$
EletPrinter.write (resp , hout);
```

Figure 2.5: ELET Data Type[2]

using syntactical code analysis. One notable result was that this technique could find all the known vulnerabilities of an application that was ported to this approach (JSPWiki). Although the server-side context-detection should have less overhead than client-side context detection; evaluations showed an average of 25% overhead (on a commodity computer), which means more optimizations are required.

In conclusion, if the output buffer maintained by the preprocessing mechanism can detect the correct parsing context at each point, this approach can be a context-sensitive prevention mechanism. Run-time overhead and the need to port a legacy application to the introduced data type are two main limitations of this technique.

2.2.2 Auto-Sanitization

Another similar category of vulnerability prevention techniques is **auto-sanitization mechanisms** [5, 23] which are based on type inference techniques to detect the correct context of an untrusted input [10]. However, prevailing web languages (e.g. HTML, CSS, JavaScript) are weakly typed and this context ambiguity prevents applying this approach effectively. Thus, researchers have focused on implementing auto-sanitization mechanisms using new context-aware template languages [6].
In these template languages, the developers specify the output generation logic in programing elements called templates that will be rendered or compiled to a host server- or client-side language. These templates allow developers to explicitly sepa-

rate the applications' logic from its display and then apply the context inference logic on the generated display data to detect the proper context [5, 23].

ScriptGrad is an auto-sanitization technique which aims to automatically sanitize the large scale legacy systems written in ASP.Net without applying changes in source code or the browser [23]. Its target is to detect and mitigate two classes of sanitizer placement errors: 1) context-mismatch and 2) inconsistent multiple sanitizers.

This technique is implemented as a two-step process. The first step is a training phase using a path profiling technique to learn all the sanitization paths and the corresponding contexts based on using benign user inputs which is implemented using the binary code instrumentation of the target application. This learning phase sends the result of each output generation statement to a component with browser-like logic to infer the sanitizer context at that statement. In the second step (run-time), the auto-correction logic embedded in the instrumented binary code will be activated. The goal is to detect the correct context of run time output generation statements and automatically sanitize the outputs. It is also worth noting that having a training phase to detect the proper sanitization context could strongly optimize the run-time context-detection step, reducing it with negligible overhead.

In summary, vulnerability prevention mechanisms can manage the sanitization complexities in two ways. The first approach, ***manual secure coding***, is an error prone task, due to complexities in detection of proper sanitization functions by the programmers. The second approach, ***auto-sanitization*** mechanisms, addresses the context detection and the corresponding ambiguities. In order to detect the correct context of HTML output generation statements, studies included have applied a browser-like parsing mechanism to output strings. In fact, output strings are sent to some auxiliary parsing components before sending them to the client-side to detect the context of that string. For example, the Scrip-Gard [23] uses instrumented server-side logics to capture the string outputs and sends them to a context-inference logic and the technique introduced in [5] uses a compile-time parsing engine for this purpose. Although sometimes flow control structures such as if/else statements can lead to ambiguities in context detection, these cases are not common and as Saxena et al. expressed in [23], only 1% of their evaluations had this condition which needs extra run-time detection overhead. In other words, the context detection with the help of the auxiliary parsing components, can almost be implemented as a static and high- precision prevention technique. Table 2.1 summarizes different aspects of this approach.

Table 2.1: Summary of Vulnerability Prevention

Approach Criteria	Context Sensitiveness	Legacy Applications Support	Code Coverage
Auto Sanitization	Yes (in a new language)	No	Yes
Secure Coding	Yes (Depends on the developer)	Yes	Yes

As mentioned before, context-aware auto-sanitization requires applying the type inference techniques in which runtime overheads are sometimes inevitable. For example, the technique Samuel et al. introduced in [5] states that their proposed technique, which is used to sanitize the Google Template Closure framework, has 3-9.6% runtime overhead. While the runtime overhead can be reduced using improved algorithms or better hardware, the need to partially (or completely) rewrite legacy applications in a new language or to instrument the compiled code is an important side-effect that make this technique a non-straight forward approach for such applications.

In the following section, vulnerability detection techniques will be discussed.

2.3 Vulnerability Detection

Another group of software protection approaches is vulnerability detection approaches. These approaches aim to detect security vulnerabilities using program analysis techniques. These techniques include static analysis, dynamic analysis or hybrid of both.

2.3.1 Static Analysis

Static analysis means reasoning about the run-time properties of software without executing it [13] . This technique is used to analyze the source code to detect security flaws. Many static analysis tools and techniques to detect security and non-security mistakes have been introduced by the researchers [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35]. In this regard, **taint analysis** or taint propagation analysis is the common technique that is frequently used to detect security flaws. This technique tracks a tainted variable (holding a value originated from an untrusted source such as user inputs) over all of its data flows in an application. Each tainted data flow starts from the first taint value assignment to a variable, passes through all statements affecting that variable until it reaches the final usage of that variable in a security sensitive operation (sink).

Jovanovic et al. in [28] have introduced an inter-procedural taint analysis technique to discover vulnerable points in PHP programs. They demonstrated that this approach can be applied to detect SQL injection, cross-site scripting and command injection vulnerabilities. They implemented their proposed analysis technique in a tool called **Pixy**. Pixy includes a literal analyzing technique (accompanied with variable alias analyzing) which is based on identifying the taint value of each single point (statement) in the source code. These points or nodes, annotated with taint values they hold, make it possible to inspect whether any sensitive sink in the program is receiving potentially malicious data, and hence, is vulnerable to injection attacks. This tool was applied to various PHP applications to evaluate its analysis effectiveness. Researchers used two factors for their evaluations. The first one is the ability to detect unknown vulnerabilities and the second one is the false positive rate. Empirical results showed that this tool can find new vulnerabilities with a false positive rate of 50%. The technique proposed in this study did not address the context-sensitiveness problem. While the introduced taint analysis can be applied to legacy PHP applications and even find new vulnerabilities, it only checks the existence of the sanitization functions and not their effectiveness. In fact, the subtleties and challenges regarding the context-sensitiveness of the sanitizers have not been addressed, and hence, cannot find these vulnerabilities.

Tripp et al. in [36] introduced a technique called **Taint Analysis for Java**(TAJ) as an approach to detect the these vulnerabilities: Cross-site scripting (XSS) attacks, Injection flaws, Malicious file executions and Information leakage. TAJ checks the Java source code with respect to a set of "security rules". Each security rule is a triple of *untrusted sources*, *sanitizer* and *sensitive operations or sinks*. At first, it performs a *pointer analysis* to build the application's function call graphs and then it runs an algorithm to track tainted data over this graph using single static assignments [25].

TAJ focuses on the challenges that industrial scale web applications face in applying taint analysis. Hence, they introduced a novel *thin source code slicing* technique to overcome performance and feasibility issues that usually happen during large scale taint analyses. In fact, this technique only tracks the tainted properties or fields of a parent object and the parent object itself is not the subject of data flow analysis. Technically speaking, this technique combines flow-insensitive data propagation through the heap with flow- and context-sensitive (different from the sanitization context-sensitiveness) data propagation through local variables. Moreover, it covers detection of taint values in the internal state of objects, Java Server Pages (JSP), Enterprise JavaBeans (EJB) and also the Struts and Spring frameworks. An important feature of this approach is its capability to perform constraint satisfying analysis. In other words, when it is needed to satisfy specific time and memory requirements, this technique can prioritize or limit the portions of the web application under analysis. TAJ has been implemented as an Eclipse plug-in on top of the IBM T. J. Watson Libraries for Analysis (WALA).

This approach, which is applied to Java applications, is similar to the technique used in Pixy for taint analysis of PHP codes [28]. While the proposed technique of thin slicing can improve the taint analysis mechanism, it has not addressed the context-sensitiveness of sanitization process. The reason is that the proposed technique, with good coverage of Java programming aspects such as EJBs and Spring framework, only checks the tainted data flows and the string constants affecting the final outputs (which can change the context in client-side) are not considered. However, this technique has the same advantages as other static analysis techniques which are the capability to be applied to legacy applications and zero runtime overhead.

In summary, static analysis techniques are used as a basic approach to detect security flaws. Due to difficulties in precise context detection (such as implicit browser transduction and nested contexts), the static analysis techniques do not have sufficient precision to detect sanitization errors. Static analysis tools only check the existence of the sanitization functions and not their effectiveness in source code. In other words, accurate context detection requires simulating the browser functionalities whether at run-time or compile-time, and hence, a static analysis technique cannot be solely used to address the context-sensitiveness of sanitization errors.

Taint analysis (or taint propagation analysis), which is a technique widely used to track the variables carrying the malicious inputs in an application, is based on static analysis. Taint analysis uses data flow analysis techniques [30] to detect a flow of(path of) data between the initial value assignment of variables by tainted values (e.g. user inputs) and the target point in which these variables are used. Since the tainted values are string values, the data flow analysis is only carried out for string variables (whether standalone variables or objects' properties) and other data types are ignored. Moreover, because of this string-based property, string manipulation operations such as concatenation, subtracting and replacing can affect the value of the tainted variables. Although string manipulation functions can modify the tainted values, they do not necessarily sanitize them. Thus, many taint propagation analysis techniques try to introduce string manipulation grammars [37] or notations to cover the string functions and describe them in a more formal way. This formal version of string functions can help reduce the ambiguities in string analysis.

One common technique of taint propagation analysis, introduced in different studies such as [38, 28], is based on tainted/untainted variables. In this technique, once tainted values are used as an input of a sanitization function, the result would be a safe or sanitized value. Therefore, if an expression is only composed of untainted variables, it would be safe, otherwise it is tainted. This way, taint analysis can investigate whether a sensitive function is receiving malicious data. In other words, if one variable with no sanitization functions in its data flows can be detected, any combination of that variable used in sensitive operations is also a tainted value, and thus, a vulnerability would be signaled. In this technique, if variables cannot be distinguished as tainted or untainted, they are marked as tainted. The input for these taint analysis techniques is the source code and a list of predefined sanitization functions. The result of these techniques is the list of vulnerabilities found in an analyzed source code.

One advantage of static analysis techniques is that regardless of flow control structures (e.g. if/else or loop statements) they can cover all data flows of tainted variables, providing a reliable base for further analysis. The capability to apply a technique to legacy applications and also minimum (or zero) runtime overhead are two important benefits considered to evaluate the program analysis techniques. In contrast, the false positive rate, measured as the ratio of truly detected vulnerabilities to total number of detected ones, is the drawback of these techniques. This disadvantage is due to uncertainty to precisely mark the data flows as tainted or untainted. In addition, other factors such as accuracy of sanitization context detection (very important), ability to analyze custom sanitizers and supporting different injection attacks are another measures that can be considered to evaluate these techniques. Table 2.2 summarizes different aspects of the static analysis approach.

Table 2.2: Static Analysis Summary

Inputs	Source Code Detection Rules (untrusted sources and sensitive operations)
Outputs	Vulnerabilities found in the source code
Pros	Can cover all the execution paths (Low False Negatives) Can be applied to legacy application without rewriting Zero runtime overhead Can be applied during development time
Cons	Context-insensitive (difficulties in writing detection rules) High False Positives

2.3.2 Dynamic Analysis

Dynamic analysis approaches are techniques for reasoning about the applications' behavior during runtime. One class of these approaches aims to enforce security policies in a runtime environment. In this class, the corresponding enforcement techniques have to observe the application or process during runtime and then apply the security policies using techniques such as reference (and inline reference) monitors. These security policy enforcements require modifications or rewriting the program (using native- or byte- code) in such a way that the applied policies cannot be circumvented. This class of dynamic analysis is out of scope of this study. Black box fuzzing is a widely used dynamic analysis approach to detect security flaws. This approach is focused on evaluating the runtime responses to different test inputs to infer the applications' behavior. Test input generation algorithms, execution path coverage and the testing time performance are the main specific research topics surveyed here [39, 40, 41, 42, 8, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 29, 58, 59, 60, 61, 62, 36, 63].

The WAVES technique [60] is a black-box fault injection approach to test web applications. In order to apply this technique to a web application, all of its entry points should be determined at first. Therefore, this approach uses a crawler to find all the HTML pages containing entry forms. The important challenge in this crawling is that many entry pages or points contain features that can prevent detecting or accessing it automatically. These features include dynamic contents (using JavaScript), session management or user input requirements prior to navigation to a particular page. To deal with these challenges, the researchers utilized a reverse engineering technique that can detect all possible entry points. Upon finding the entry points, the detected forms need to be filled properly to bypass the form validation mechanism used by the web application. Therefore, the researchers introduced the Injection Knowledge

Manager (IKM) as a mechanism to inject the test inputs in some selected fields and also filling the remaining fields with valid values required to bypass the form validation rules.
This technique has two important parts. The first one is a crawler component containing a JavaScript engine, DOM parser and event generation mechanism. Another part is a behavior monitoring component composed of a self-training anomaly detection and event stimulation mechanism. The combination of these two parts helps to detect XSS attacks by monitoring and comparing the normal and malicious behavior of the application at runtime. It can be seen that the advantage of combination of the introduced crawler with the self-training anomaly detection leads to low false positive rates in malicious behavior detection. The researchers did not mention how complete their approach is in generating attacks scripts for XSS attacks. This technique is mainly based on anomaly detection and does not require source code to generate the test inputs and so can be used for legacy applications.

According to context-sensitiveness of testing, the sufficiency of this technique depends on the attack repository used by them. In other words, while the security testing can reveal the XSS vulnerabilities as well as the context-sensitive sanitization errors, the attack generation mechanism used in this technique does not address this requirement.

Duchene [64] et al. introduced a XSS vulnerability detection approach based on control flow inference and evolutionary fuzzing. They believe that state transition of the application is an important factor in test input generation. Thus, they introduced a model inference mechanism to build a state transition model of the system under test. Then, this model guides the fuzzing process to start from an appropriate state. In addition, they used a genetic algorithm, as an evolutionary algorithm, to enhance the fuzzing process effectiveness. A repository of the known attack vectors is used for initial population of the genetic algorithm and both mutation and cross over functions of the genetic algorithm are influenced by a manually written attack grammar. The result of the genetic algorithm is fed to the application under test and the collected responses are used to improve the next generation of test inputs. This improvement is a result of the fitness function customized for reflected XSS vulnerabilities.

This approach can be used to detect the XSS vulnerabilities and because it doesn't propose any new mechanism to rewrite the old applications, it can be applied to legacy applications. However the attack generation coverage or completeness has not been addressed properly and the initial attack vectors used for the genetic algorithm affects the whole attack generation process. In addition, the advantage of this approach in detecting new vulnerabilities or false negative rate has not been clearly expressed. This approach uses lexical confinement to analyze the DOM structure of the response page using some predefined taint tree patterns. Taint patterns are trees with regular expressions in their nodes representing common patterns of injected HTML tags. There are two side effects for this approach: 1) The difficulty to cover all taint patterns can lead to false negatives. 2) The flow control inference algorithm cannot guarantee to find all execution paths of the application as the static analysis techniques do. The positive point is the attack grammar for guiding the test input mutations which can increase the chance of successful attacks and improves the vulnerability detection time.

The approach presented by Armando et al. in [39] is a security testing mechanism which is based on **model checking**. Model checking is a mechanism based on a formal representation of the systems under test. This approach builds an abstract representation (model) of a security mechanism (e.g. authentication) of the target web application and then this model would be checked through a model checking process. If any of the expected security properties of that abstract model are violated it will be used to generate the concrete test cases. Then the generated test cases are used to find vulnerabilities in the target web application using a testing engine. Applying this approach to an old application doesn't require any manual or automatic program rewriting and, hence it can be used for legacy applications. The researchers evaluated their technique in terms of attack generation speed, but the false positive or false negative rates or the ability to detect new vulnerabilities in comparison to other techniques have not denoted. While scope of this approach is general and not limited to any specific attack type, it seems to be a more appropriate mechanism for testing of the authentication protocols than cross-site vulnerabilities. In fact, using this technique to test and detect the cross-site scripting (XSS) vulnerabilities requires defining an abstract model of XSS protection mechanisms (especially context-sensitiveness of XSS sanitization process) with well-defined properties which is a very challenges task.

A **pattern-based combinatorial testing** technique has been proposed by Bozic et al. in [44] to detect stored and reflected XSS vulnerabilities. In this approach, an attack grammar represented in Backus-Naur Form (BNF) notation is used to design the test cases. This grammar, defined as a linear array of attack script components, is used as an input for a combinatorial test generation tool. The goal in combinatorial testing is selection of a few test inputs (or combinations) in a manner that a good coverage is still achievable. The coverage rate of combinatorial testing is determined by a parameter referred to as "strength factor" depicting the depth or number of the selections. The evaluations done by this study shows that a combinatorial test with strength of 2 has the same result as the higher numbers. This result highlights an important factor of achieving efficiency without losing the coverage. In addition, other evaluations show that this approach has low false negative rates and good detection time. Moreover, it can be applied to legacy applications and has no runtime overhead. While having a grammar to define the attack patterns is useful to organize and formulate the test case generation process, it might lead to search space explosion due to a huge number of strings that may belong to a grammar. The advantage of this approach is the combination of pattern based attack generation with combinatorial testing to overcome this challenge. This technique is based on one attack grammar and in order to cover context-sensitive features of XSS vulnerabilities either all context transfer situations in web pages or different grammars should be clearly defined.

A **concolic (concrete + symbolic) testing** approach for web applications is introduced by Wasserman et al. in [63]. In concolic testing, symbolic constraints and concrete values are used to generate test inputs. In concolic testing, at first, a symbolic representation of the application is generated and then in cases where the symbolic constraints(composed out of symbolic conditions of if statements) cannot be solved, the concrete values will be used to solve those constraints. One of the important challenges in symbolic execution techniques is the difficulty of model is the string manipulation functions such as replace or substring functions. Therefore, the aim of this study is to cover string manipulations widely used in PHP source code, as well as extending the concolic testing beyond the single function testing (to extend it across multiple files).

In this technique, at first step, the target program is instrumented to collect both concrete and symbolic values from normal usage of the application; this step is referred to as the learning phase. This instrumentation takes place at memory level and neither source code modifications nor offline analysis is required. The second step, called the vulnerability detection step, has two inner phases: constraint generation and constraint resolution. In the first phase, the symbolic constraints are generated based on string manipulation functions using finite state transducers (FST). Next, to solve these constraints the FSTs are inverted using language equations, and in the case of intractable expressions the previously collected concrete values (from the learning phase) are used.

This approach is a general web application test input generation and is not targeted for security testing. But the introduced technique can be customized for security test input generation as well. While this study stressed that applying the concrete values in solving the symbolic constraints can increase the covered execution paths, it has not addressed to what extent the collected concrete values can solve these symbolic equations. The ambiguity is because of the fact that these values are collected during a normal application usage and may contain a limited number of execution paths and some corner cases (which are critical for security testing) may be missed.

In order to detect XSS attacks, context-sensitiveness of sanitizers should be expressed using symbolic execution, which as long as all string manipulation functions are covered can be done. On other hand, the browser transduction vulnerability is not a target application's source code feature to be symbolically represented and, hence may be missed. This approach can be used for legacy applications and has no runtime overhead.

Wang et al. [65] proposed a technique to generate mutated XSS attacks to test web applications. This technique crawls the public attack repositories to tokenize each attack script and then extract their elements. Next, a learning model builds a structural model of these extracted elements using a hidden Markov model. This structural model then will be applied to an algorithm to generate the mutated attack vectors out of initial attack scripts. In fact, the goal of this approach is extending a primary attack repository by generating new scripts instead of adding new ones. This way, even with limited repositories, a large number of attack scripts can be generated. One advantage of this approach is simulating the attacker's behavior in producing new attack vectors out of previous ones which can lead to more realistic attack scripts in comparison to random fuzzing algorithms. The evaluations show that it has a low false positive rate as well as zero runtime overhead. The ability to generate attack vectors to test the context-mismatch or browser transduction vulnerabilities is not clearly addressed in this technique. In other words, the mutation algorithm used to generate the new attack scripts should be aware of context switching tokens or keywords during the initial attack script tokenization. If all the conditions by which these tokens shape the attack scripts can be covered, this technique can be assumed as an effective approach to generate attack scripts with minimum false negatives.

Trip, Weisman, and Guy proposed a learning process [36], associated with XSS Analyzer, with the motivation to improve the efficiency of black-box testing by finding vulnerabilities faster in a large repository of attack strings. Different testing algorithms such as brute forcing (using all available attack vectors of the repository) or random testing (selecting repository entries randomly), cannot satisfy the performance requirements of a huge industry-level testing repository with the expected coverage and accuracy.

The proposed learning process uses the target application responses (resulted from black-box testing) to learn internal behavior of the system under test and prune the testing repository. In fact, it attempts to learn how some specific tokens or characters are blocked by internal sanitizers of the application and then filters all attack vectors containing them. In addition, it uses some strategies to generate the transformed version of blocked tokens, to check whether they can bypass the internal sanitizers or not. Applying these two learning and filtering processes can significantly reduce the number of repository entries used for testing and improve the detection time and efficiency of a huge attack repository. The learning and filtering process led to a low false positive rate and improved vulnerability detection times.

It is noteworthy to express that a huge database of attack vectors is an alternative to attack generation mechanisms. These repositories can cover many different situations or multiple attack generation technique and, hence have various advantages. These repositories are not dependent to the source code and so can be applied to legacy applications as well as under development applications. Moreover, having many different situations can significantly lower the false negative rate of attack generation. In order to cover the attack vectors to check context-inconsistency of the application sanitizers, these attack vectors should previously prepared which depends on the organization preparing them (in this case IBM). Therefore, one cannot prove or deny these attack script are covered or not. Moreover, increasing the number of the attack vectors can downgrade the efficiency and performance of these approaches.

One of the main concerns in security testing is generating attack vectors. ARDILLA [61] is a novel tool developed by Kiezena et al. to automatically generate XSS and SQL injection attack scripts for PHP web applications. It is a vulnerability detection technique based on input generation and dynamic taint analysis. The proposed technique consists of these components: input generator, taint propagator, concrete-symbolic database and attack generator/checker.

The input generator part creates concrete inputs (not attack vectors) for the taint propagator. Next, the taint propagator component uses these inputs to detect whether each input flows to sensitive sinks (including passing through a database).

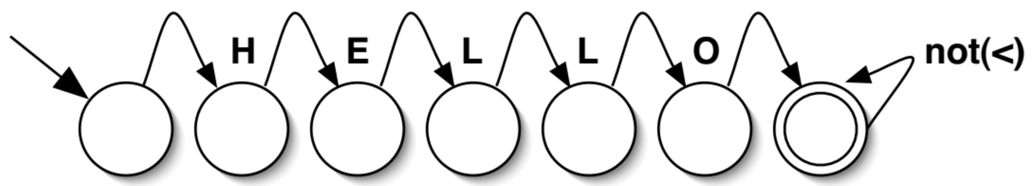
This component uses tainted inputs to determine taint sets associated with runtime values. The taint sets are a set of tainted input parameters used altogether in a sensitive sinks. For example a variable might be derived from two taint sources inp1, inp2 using string concatenation and then this variable used in a sensitive operation. Taint propagator keeps the taint sets unchanged during assignments expressions and function calls but changes them to empty sets as a return value of a 'taint filter'(e.g. sanitizer function) function call. Therefore, if there is a non-empty taint set at a sensitive sink, it means that there is a potential vulnerable dataflow from untrusted sources to that sink and, so can be used for the next step.

Next, the attack generator takes these taint sets and generates concrete attack vectors by applying these inputs using a repository of the attack patterns. The attack checker component, next, runs the application using these concrete attack scripts to find successful ones. In fact, this technique runs the application under test with two inputs (innocuous and malicious values) and compares the result of the two executions. This technique signals a XSS attack if the HTML output produced from the application using the attack script contain additional script structure than the benign inputs.

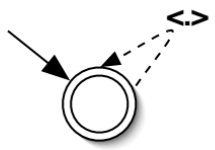
One of the novelties of this approach is its relational database that can execute both concrete and symbolic commands. This feature allows the accurate dataflow analysis of the taint values through the database, which is crucial to detect stored XSS attacks. The attack checker component uses a repository of prebuilt attack scripts developed by security professionals. The researchers also developed a prototype [62] using a constraint solver instead of attack libraries. They compare the result of the repository-based approach with this new constraint-solving approach and concluded that the concrete attack generation can be reduced to a string constraint solving problem. However, the constraint solving requires formally defining attack strings using grammars and string constraints. In comparison with traditional black-box fuzzing, they configured a fuzzer (Burp Intruder) with the same attack library and target application as the repository-based approach and discovered that the fuzzing technique found the vulnerabilities slower than the proposed approach.

This approach has two main limitations. The first one is that it only can generate attack vectors for inputs generated by the input generator component. The input generator is a component that crawls the application and attempts to generate inputs (not attack vectors) for PHP forms and if these inputs can reach the sensitive operations then the next steps (attack generation and checking) can be accomplished. In fact, input generation for PHP can be complicated because of dynamic features of the language and also inputs that require user interactions. The source code coverage rate is the most important factor for this component which does not support sessions, resulting in low code coverage.

Although this approach can use different attack libraries, its completeness or false negative rate depends on the completeness of its underlying attack repositories. Though the evaluations revealed that the proposed approach has fast attack generation time as well as low false positive rate, they did not address the attack coverage challenge. With regard to context-sensitiveness of XSS data sanitizations, the generated attack vectors should satisfy all conditions in which this requirement can be checked and any context inconsistency in the source code can be revealed. While using attack repositories implies that attack vectors requiring checking the context-sensitiveness should be inserted by security professionals, in the constraint solving alternative this requirement should be defined in the constraint grammars. In other words, the grammars used by the constraint solver should cover all situations in which



(a)



(b)

```
$custom = str_replace('<', '', $input);
$custom = 'Hello ' . $custom
```

Figure 2.6: PHP Code and its Automaton

a context switch from any context to a JavaScript context can take place. This coverage doesn't require including full context definitions (grammars) but a subset of the target context (HTML, JavaScript, CSS, and URI) should be sufficient.

Combining the static and dynamic analysis to check the effectiveness of input sanitization is a novel approach implemented in Saner [38]. It uses static analysis techniques to detect the paths between the untrusted (tainted) sources and the places these tainted inputs are used and then evaluates the sanitization used over these paths. They use Pixy's approach (a static analysis tool for PHP) as the core of their static analysis part but Pixy only checks the built-in PHP sanitizers and cannot be used for custom input sanitization functions. For example the PHP code and its corresponding automaton are as follows:

In Figure 2.6 (a) the constant string 'Hello' is represented and a general tainted (untrusted) string value is also in (b) using a dashed line, indicating any value is accepted.

Pixy uses finite state transducers (string automata) to model the string functions but it is not a taint-aware technique and cannot model taint propagations through the custom functions. Therefore, they extended it by an over-approximation or safe approximation of string values. In fact, the resulting values of the custom sanitization functions used in a proposed mechanism are referred to as implicit taint propagation. In this proposed mechanism, in order to distinguish between tainted and untainted values, only tainted values are considered in modeling the functions. In other words, any embedded or hardcoded string values assigned to variables by the programmers (that are assumed to be safe) are replaced with empty strings to reduce false positive rates. However, in cases where these functions have input tainted parameters, the function output would be represented using an automaton that describes all possible string values.

To detect vulnerabilities they use an automaton intersection approach in which the automaton representing the string values of the sink point intersects with a target XSS automaton (pre-built automaton) shown in Figure 2.7. That is, if the intersection result is not empty, it means that the string automaton of that sink point, which is a result of a sanitization path from first assignment to this point, is malicious. For example, the intersection of the automaton on the left side of the Figure 2.6(a) with this target automaton is empty, which means that this input cannot have an attack script represented by the target XSS automaton. In contrast, the automaton on the right side of the Figure 2.6 (b) yields a non-empty set, which means an attack script can be accepted or generated using that automaton.

The static analysis mechanism used in evaluating the custom sanitization function is still lacking a reliable proof for reaching an adequate level of precision. The goal of

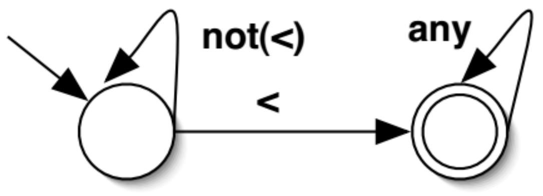


Figure 2.7: Target XSS Automaton

the dynamic analysis part is to evaluate the effectiveness of such custom sanitization functions. Therefore, once a sanitization path is marked as malicious (its automaton has non-empty intersection with target XSS automaton), the dynamic analysis part is used to investigate any false positive vulnerability.

Table 2.3: Dynamic Analysis Summary

Inputs	Compiled Code Test Inputs(Attack Vectors)
Outputs	Vulnerable/Safe Test Points Successful Attack Scripts
Pros	Context-Sensitive Low False Positive Zero runtime overhead Applicable for new and legacy applications No runtime overhead
Cons	Code Coverage Problem (False Negatives)

In order to evaluate this mechanism against the context-sensitiveness factor, the static and dynamic analysis should be investigated individually. In this regard, the dynamic analysis part uses a predefined set of the attack vectors and, hence its effectiveness or ability to detect the context-inconsistency issues is dependent on this repository. Moreover, the dynamic analysis part only runs on custom sanitization functions and the effectiveness of the sanitization paths based on standard encoder or sanitizers are only determined by static analysis. On the other hand, the efficiency of the static analysis to detect such problemss merely depends on how standard sanitizers are represented using automaton. Although the automaton introduced as the target XSS automaton is not complete, the intersection of this automaton with standard sanitizers should be empty. Moreover, the proposed mechanism states that the constant strings (which can be used to generate the HTML outputs using command echo) are treated as empty strings in the proposed implicit taint propagation. In terms of HTML specifications, these constants can contain keywords that affect the proper sanitizer (context-sensitiveness) and this way they are ignored by the static analysis. In other words, the static analysis can only check their existence and not their context consistency.

2.4 Conclusion

Vulnerability prevention approaches aim to remove the root causes of software vulnerabilities using manual and automatic approaches. While the manual approaches are mainly based on secure coding guidelines, the automatic approaches introduce type inference techniques to detect the proper contexts at compile-time or run-time [5, 23]. In fact, applying the type inference techniques to current imperative web languages (HTML, JavaScript, CSS and URI) requires introducing new languages (over the host languages such as PHP, Java or JavaScript) to distinguish data and code. Having separated data and code helps keep track of the final outputs generated by the special commands and statements of these new languages (e.g. using data markup instead of echo command in PHP). These new output generation markups can be used by the compile-time or run-time parsers, which use browser logic to detect the proper context at each point of the application. The generated outputs, qualified with correct contexts at each output generation point, are used to automatically apply the proper sanitizer.

Based on the reviewed studies in the vulnerability prevention approach, auto-sanitization techniques can detect the proper contexts statically at compile-time or in a few situations at runtime (with corresponding overhead). Relevant academic studies and emerging trends of use of these techniques by big Internet companies such as Google [6] and Yahoo [7], positions them as an important approach for context-aware sanitization.

On other hand, these approaches have some drawbacks. First, the need to rewrite an application's source code (partially or completely) makes it important to ask whether this approach can be applied to legacy applications. Table 2.4 summarizes different features of this approach.

Table 2.4: Vulnerability Prevention using Automatic Encoding Summary

Technique	**Type Inference**
Inputs	Source Code/Binary Code
Outputs	source code/binary code with injected encoder
Pros	Shifts the burden of context detection from developer to tools Can be applied during development time
Cons	Run-time overhead Needs application rewriting for legacy applications

Vulnerability Detection approaches have focused on detecting and finding the sanitization mistakes using static analysis, dynamic analysis (testing) and human based techniques. The human based techniques such as code reviewing or manual penetration testing are not covered in this review.

Static analysis : Static analysis is a method of reasoning about the application's behavior without executing it. While static analysis techniques can be applied for different purposes (e.g. compilers use static analysis in their code optimization components), it is used as taint analysis for vulnerabilities detection [38, 28, 30, 32, 33]

Applying the data flow analysis in inter-procedural cases and ambiguous language statements such as aliasing in PHP would increase the complexities of the data flow analysis. However, using optimization techniques such as implicit taint analysis and thin slicing can improve the time and memory efficiency of taint-aware static analysis as well as it accuracy. The main advantage of this approach is that it can cover all execution paths for both new and legacy applications and also can be applied during development time. Alternatively, its offline reasoning feature leads to this problem that it only checks the existence of sanitizers to signal the vulnerabilities and not their effectiveness leading to context-insensitiveness. Table 2.5 summarizes different aspects of this approach.

Dynamic Analysis: This approach, as a vulnerability detection technique, aims

Table 2.5: Static Analysis Summary

Inputs	Source Code Detection Rules (untrusted sources and sensitive operations)
Outputs	Vulnerabilities found in the source code
Pros	Can cover all the execution paths (Low False Negatives) Can be applied to legacy application without rewriting Zero runtime overhead Can be applied during development time
Cons	Context-insensitive (difficulties in writing detection rules) High False Positives

to evaluate the applications' responses to detect any security mistakes. Although dynamic analysis is a broad technical term to refer to different techniques, in this review I refer to dynamic analysis as a technique utilizing the testing approach to evaluate the applications' encoding behavior, which is known as black-box fuzzing. In fact, in this approach an application is vulnerable when it is tested against the attack vectors and the testing output reveals that at least one of the applied attack scripts is successful. This highlights that an effective dynamic analysis depends on the evaluation technique used to detect the successful attacks as well as the capability to find and cover all the testing points (code coverage).

Many scholars hold the view that code coverage is the main challenge in this topic. Different algorithms such as genetic algorithms [66], pattern-based attack generation [44, 45, 67, 32] and attack repositories [10] have been reviewed. Regarding to context-sensitive encoding errors, there would be a definite need to evaluate the application's behavior from the browser's view, and dynamic analysis is the right choice for this purpose. In other words, having correctly crafted attack vectors (test inputs) can reveal the corner cases in which these subtle mistakes lie. Although this technique can detect the context-sensitive encoding errors for legacy and new applications, applying it during the development time can be limited by runtime dependencies affected by application logic and scenarios. In fact, these limitations lead to constraints such as waiting for user inputs, database systems dependency, and authentication requirements that actually are not required to test the security aspect of the applications. Table 2.6 summarizes different aspects of this approach.

Table 2.6: Dynamic Analysis Summary

Inputs	Compiled Code Test Inputs(Attack Vectors)
Outputs	Vulnerable/Safe Test Points Successful Attack Scripts
Pros	Context-Sensitive Low False Positive Zero runtime overhead Applicable for new and legacy applications No runtime overhead
Cons	Code Coverage Problem (False Negatives)

To sum up, although investigating the encoding effectiveness of static and dynamic analysis, in terms of **context sensitiveness** or consistency, was the main criterion for this review, I also considered additional important requirements that affect introducing a novel. These factors are:

- 1) Legacy Applications Support(not requiring to rewrite the code)
- 2) Development Time Support (being able to to be applied during development time)

Based on the reviewed studies, none of the static and dynamic analysis alone can satisfy all of the above requirements and thus, a combined approach covering the advantages of both static and dynamic analysis with an effective integration capability into the software development phase is needed.

Table 2.7: Summary of Approaches

	Context Sensitiveness	Legacy Applications Support	Code Coverage
Auto Sanitization	Yes (in a new language)	No	Yes
Static Analysis	No	Yes	Yes
Dynamic Analysis	Yes (depends on attack vectors)	Yes	No

In the following sections, I will explain the different components of the proposed unit testing-based vulnerability detection architecture in detail.

CHAPTER 3: Unit Test Extraction and Evaluation

3.1 Unit Test Extraction

The need to analyze each web page to find its vulnerabilities with low false positive and false negative rates requires a mechanism to both find and then verify vulnerable points in the web applications. Flow control statements (e.g., if statements) cause different execution paths in a web page. Static analysis can find the suspicious points in all of the execution paths. To avoid false positives, all of the detected suspicious points should be verified by executing their corresponding execution path. This led me to define each execution path as a unit test to be executed and verified against the test inputs. If the unit tests are vulnerable the original source code should be vulnerable and if the original source is vulnerable at least one unit test should be vulnerable. This means that to detect the true vulnerabilities the unit tests should preserve the HTML context and sanitization functionality of the original source code.

The goal of the unit test generation is to extract unit tests proper for XSS vulnerability detection from the source code. To ensure source code coverage, I construct a set of unit tests based on execution paths in each web page (here each JSP files) with the goal that if the original JSP file has an XSS vulnerability due to incorrect encoder usage, at least one of the corresponding unit tests will be similarly vulnerable as well. The following are inputs for XSS unit test construction:

- (1) Web page source code
- (2) Untrusted sources and
- (3) Security sensitive operations or security sinks.

```
<% String ordID =       request.getParameter("order");
ordID = escapeHtml(orderID);
 if(editMode){ %>
   <a onclick="edit('<%= ordID %>')"  href="#" > Edit Order </a>
 <% } else { %>
  <span> Order:<%= ordID %> </span>
 <% } %>
```

Figure 3.1: Original Source Code

```
String ordID = request.getParameter("ord");
ordID= escapeHtml(ordID);
boolean e1= (editMode);
//then-branch of if statemenet
out.write("<a onclick=\"edit('");
out.write(ordID);
out.write(" ')\" href=\"#\" > Edit Order</a>");
```

Figure 3.2: Generated Unit Test

Untrusted sources are Java methods from which malicious data can be brought into the web application, such as *request.getParameter()*. Security sinks are statements (or function calls) used to generate the HTML outputs to be rendered by browsers. There are a number of sinks in the context of JSP applications: *out.write()* , *out.print()*, *out.println()*, out.append(), or <%= %>. I illustrate the unit test generation using Figure 3.1 as the original code and the Figure 3.2 as one of the constructed XSS unit tests.

To focus the discussion, I assume the application encodes all untrusted variables using known encoding functions. Taint analysis can readily discover cases where an untrusted variable appears in a sink without encoding. The vulnerability model is a situation where an encoder does not match the application's HTML document context.

A typical JSP file contains both HTML and JavaScript specifications as well as Java variables and statements, referred to as host variables and statements. The term HTML document context refers to HTML and JavaScript specifications in the JSP file. To avoid false negatives, I capture all sinks in all possible HTML document contexts. For convenience of performing program analysis, I replace all HTML elements with equivalent Java statements. This task is accomplished by using a JSP code analyzer that uses Java output generation commands such as *out.write()* to enclose HTML and JavaScript parts of the JSP files. For example, the HTML elements in line 4 of Figure 3.1 are replaced by lines 5-7 in Figure 3.2.

Java branch statements could impact a sink's HTML document context as illustrated by Figure 3.2. Untrusted variable *ordID* is in a JavaScript context in the "then" branch of a Java if statement (line 4). In the "else" branch of the same Java if statement, variable *ordID* is in HTML body context (line 6). A control flow analysis is performed to generate the control flow graphs for each JSP file to infer the execution paths. Multiple execution paths or unit tests will be created when the JSP file contains *if* or *switch/case* Java statements. For example, the source code of Figure 3.1 contains the following two possible execution paths:

- Line numbers 1,2,3,4 (then branch)
- Line numbers 1,2,3,6 (else branch)

Two XSS unit tests are generated for this example, each corresponds to one execution path containing no branching logic and each has a sink containing one untrusted variable. Execution paths without sinks and untrusted variables are discarded as they are not vulnerable to XSS attacks.

Figure 3.2 is a XSS unit test extracted from the "then-branch" of the Java if-statement in Figure 3.1. The untrusted variable in this unit test is *ordID*, which appears in a sink statement (<%= %>). The sanitizing function is on line 4. To avoid any runtime exceptions or miss any statements affecting the HTML context I also keep the conditional expression used in the if statement in both branches, which is shown as line number 3 of Figure 3.2 by assigning the value of the conditional expression to a Boolean variable e1.

While it is possible for branch statements written in JavaScript to change the HTML document context of a sink, I expect such cases to be rare. This is because sinks are written in Java. It is therefore natural for developers to use Java to express changes in HTML document context. I thus assume that JavaScript code does not change the HTML document context of sinks. I will assess this and other assumptions in the evaluation section.

I assume that each JSP web page is set up for unit testing. This means there is a runtime environment with web server, application server, and database server. Running XSS unit tests does not have additional requirements. The original source code can run with no error if all the environment requirements are met. The original JSP page is launched on a web server to set up the session. A proxy captures the session information. The captured session is sent as part of subsequent requests for XSS unit testing. Then all the XSS unit tests on the same web server will be executed (using the attack evaluation component). The captured session information is sent along with attack strings to run the XSS unit tests. Each XSS unit test shares the same session as the original unit test, as session is global across all JSP pages on the server.The process described above is standard practice for unit testing web pages.

Single Variable: In the ideal case, the original JSP file contains one untrusted variable as is the case in Figure 3.1. For such a case, there are no false negatives because all possible HTML document contexts are captured by at least one XSS unit test. If the original code was vulnerable due to using the wrong encoding function, then at least one of the XSS unit tests would be vulnerable. I define a **false positive** as a situation where the application's context and the applied encoding function are matched (safe) in the original source code, but the encoding function is detected as vulnerable (mismatched) by a XSS unit test. This is not possible for the ideal case because the XSS unit test construction process preserves the HTML document contexts of the original JSP file.

Injection points for XSS unit tests. I assume that untrusted sources are specified as a set of Java API's, such as user forms and database queries. Taint flow analysis is used to identify injection points in the program. Injection points are places where variables containing an attack string (as an input parameter for the unit test) are injected into a unit test. These variables are used as an argument of the first encoder function in its data-flow from untrusted source to security sinks. Since an XSS unit test contains no branching logic, detection of such injection points is straightforward. Figure 3.3 shows part of an original source code. Untrusted variable *fName* is used in a sink on line 4 after being sanitized using the encoder on line 3. Variable *fName* gets value from variable *prf* as a result of a database call, *searchProfile()*, a tainted source on line 2. In the corresponding unit test in Figure 3.4, the variable containing the

```
1  <% List<Profile> prf;
2  prf= searchProfile(customerID);
3  fName = escapeHtml(prf.Name); %>
4  <a onclick="profile('<%= fName %> ')" href="#" >
```

Figure 3.3: Code with Untrusted Source from a Database call

```
1  //param is an input parameter containing an attack string at test
   time
2  param = request.getParameter("param");
3  prf= searchProfile(customerID);
4  // Injection point is in place of prf.Name in original code
5  fName = escapeHtml(param);
6  out.write("<a onclick=\"profile(\'");
7  // sink line in original code = 4
8  out.write( addLine(fName , "4") );
9  out.write(" ')\" href='#' >");
```

Figure 3.4: Generated Unit Test with Injection Point

attack string *param* will be injected into the XSS unit test as the input parameter of the *escapeHtml()* encoder, as its first application in a statement, on line 3 of Figure 3.3.

I also instrument each XSS unit test so that it reports the line number in the source code if a vulnerability is found as shown on line 8 of Figure 3.4. I identify the line number of each sink statement in the original JSP file. Suppose the line number of a sink in the original JSP file is 4:

```
4: <%= tainted + "constant" %>
```

I add a function to each unit test to add the line number of the sink statement to the attack string:

```
out.write(addLine(tainted + "constant",4)))
```

Function *addLine()* is a server-side function which adds the line number of the sink statement as a parameter to the attack payload. This line number is calculated during the static analysis process generating XSS unit tests. In my evaluation described below, this line number will be used to identify the vulnerable statement line number to the developer and also used to guide the auto-fixing component to replace the incorrect encoders.

Multiple Variables: An XSS unit test may contain multiple untrusted variables. Figure 3.5 shows two examples. Best secure programing practices [50] suggest that if both variables are properly sanitized with respect to the expected HTML document context, their combination should be safe as well. I refer to this as the *independent encoding assumption*. This assumption allows us to test one variable at a time by holding the rest of the untrusted variables constant.

```
<%= "User : " + escapeHtml(user) + "(" + escapeHtml(email) + ")" %>
<%= escapeHtml( "Patient:" + firstName + " " + lastName) %>
```

Figure 3.5: Multiple Tainted Variables in one Sink

3.2 Unit Test Evaluation

The goal of the attack evaluation component is to assess whether a unit test is vulnerable to any of the XSS attack strings. This requires that the unit tests get invoked and the response page is evaluated using a real browser. Major approaches to verifying the success of the attack include string matching assessment, DOM structure assessment and Attack payload hook.

String Matching Assessment: One widely used approach to evaluate the response page of applications in security testing, exemplified by the popular black box testing tools ZAP [11] and XSS filters such as Firefox NoScript [68] and Google XSSAuditor [69], is to look up the attack payload in the response page. The rationale for this approach is that if an attack payload can bypass the encoder function intact, an attack could occur. Unfortunately this approach can lead to false positives. A successful attack payload must be compatible with the context it is injected into. For instance, Figure 3.6 shows a situation in which an HTML body encoder is used to sanitize a user-entered parameter on line 1. Line 2 is an attack string in ZAP's attack repository. Line 3 shows the output of the web page when this attack string is applied. Since the encoder does not alter the attack string, ZAP's test evaluation mechanism would report this page as vulnerable. This is a false positive because this attack cannot be executed in the HTML context.

DOM Structure Assessment: This approach is based on the observation that a successful attack can change the DOM structure of the response page. Assessment of this effect, known as lexical confinement, can be done by comparing the DOM structure of the response page using taint-aware policies written as taint tree patterns [49, 69]. Taint tree patterns are trees with regular expressions in their nodes describing different cases in which a successful attack can change the parse tree of an injected HTML node. This approach needs to define all possible taint tree patterns, which can be very difficult especially for JavaScript code leading to false negatives.

Attack Payload Hook: This approach captures the execution of the attack payload to signal the successful attack. Client-side JavaScript code is used to hook the attack payload functions such as widely used alert(), confirm() and prompt() functions, or user defined ones as introduced in XSSValidator [70].

```
<p> <%= escapeHtml(request.getParameter("atk"))%>  </p>
Attack String :  + alert(1)
<p>  + alert(1)  </p>
```

Figure 3.6: False Positive in Attack Detection Listing 8.

I propose to use a modified version of the "attack payload hook" approach to detect the success of attacks. A limitation of the original "attack payload hook" approach is that the attack payload should be executed by the browser in order to detect the attack. It means that if the vulnerability enclosed in situations such as tag "events" or JavaScript flow control blocks (e.g., if statements) which need user interactions or particular conditions to be activated, the attack cannot be detected. I need to find a way to overcome this limitation so that events can be evaluated successfully under all circumstances.

3.2.1 Attack payload

Our approach is to execute each attack string using JWebUnit, a widely used open source unit testing tool for web pages. Vulnerabilities are only reported if successful execution of an attack payload by JWebUnit is detected. For the attack payload, I use a JavaScript function attack(n), which takes as parameter the line number of the code being tested. It changes the web page title by appending that line number.

3.2.2 Test driver

Figure 3.7 shows the XSS unit test driver. Lines 1 and 2 are for test preparation. Function sessionPreparation() sets up the execution environment by applying captured session information (as shown in the discussion of Figure 8). The rest of the test driver invokes the XSS unit test by applying attack strings. After initializing an instance of WebTester (a subclass of JWebUnit) on line 1, each iteration of the loop on line 3 takes one attack vector (atk) and invokes the XSS unit test page (unitTest.jsp) with the attack string as a parameter (line 4). Line 5 pauses to let the unit test page be rendered completely. Line 6 asserts whether the attack is successful by checking the title of the response page. If the attack is successful, the page title contains the line number(s) of the vulnerable sinks, helping developers to fix vulnerabilities.

3.2.3 Handling events

In order to find vulnerabilities associated with tag events, I must trigger each event with a JavaScript body. There are 88 possible events in HTML5, some of them can only be triggered based on particular user interaction such as onmouseover or a run time condition such as *onerror()*.

However, since all events share the same syntax, I can substitute events that cannot be easily simulated in a test environment with an event that can be easily triggered. We verified that in major browsers (Chrome, Safari, Firefox) event onclick can be associated with every HTML tag and it can be triggered using a JavaScript API. Figure 9 shows a JavaScript program I use to go through all tags in the DOM. For each tag, the program checks if the tag has an event with an event body (line 4). If a tag has a body, the program assigns the event body to an onclick event and triggers

```
Public void prepare()
{
wt = new  WebTester();
sessionPreparation(); //other preparations such as proxy
}
public void run() {
  for( String atk :atkVectors){
// Invoking the Unit Test
  wt.gotoPage("unit_1.jsp?atk="+atk);
  sleep();
  verifyResponse(wt);
   }
}
```

Figure 3.7: Test Driver and Test Preparation

it programmatically.

In summary, the unit test extraction component aims to extract unit tests from each web page for XSS vulnerability detection. The server side code of each web page (e.g., JSP pages) contains multiple flow control statements like if/else leading to multiple execution paths. Each execution path covers the whole life cycle of a web page from the time it is requested (using URL of that page or redirecting from other pages) to the last statement of that web page. Execution paths are mutually exclusive at run-time and only one of them is executed depending on the flow control statements. Each unit test corresponds to one execution path and can be requested independent of other unit tests like its parent web page. This unit test extraction

```
var tags = document.all;
for (var i=0; i <tags.length;i++){  e= tags[i];
 if (typeof e.onfocus == "function")   {
    event = e.onfocus;
    e.onclick=event;
    e.click();
  }
  // checking for other events
```

Figure 3.8: Assigning body of events

uses static analysis techniques (taint analysis techniques) to find the execution paths and their corresponding unit tests.

I construct a set of unit tests based on execution paths in each web page with the goal that if the original JSP file has an XSS vulnerability due to incorrect encoder usage, at least one of the corresponding unit tests will be similarly vulnerable as well. The following are inputs for XSS unit test construction: (1) web page source code, (2) untrusted sources and (3) security sensitive operations or security sinks. This process represents the static analysis aspect of the proposed architecture. Static analysis approaches that are used to find the vulnerabilities have the advantage of complete source code coverage. Unit test extraction only generates the unit tests and does not report whether the unit test contains vulnerability. To confirm having vulnerability in each unit test I need the unit test evaluation component.

The "unit test evaluation" component aims to run the extracted unit tests in a real environment to confirm having vulnerabilities. This confirmation is done through a unit test evaluation process which runs unit tests in an execution framework like JUnit. In fact, to avoid false positives, all of the detected suspicious unit tests should be verified by executing in a real environment. If the unit tests are vulnerable the original source code should be vulnerable, and if the original source is vulnerable at least one unit test should be vulnerable. This means that to detect the true vulnerabilities the unit tests should preserve the HTML context and sanitization functionality of the original source code.

To execute unit tests in unit testing frameworks (e.g., JUnit) test drivers are required. I use a test driver that invokes and renders unit test pages using a headless browser like JWebUnit. The test driver invokes each unit test against all the attack scripts I generated using the attack generation component. Vulnerable unit tests can successfully execute the JavaScript payload of the attack scripts. This leads to modifications in the returned response of the invoked unit test (e.g., changes in the title of the web page) that can be detected by the test driver. The test driver then reports the detected modifications in the invoked unit tests as the vulnerable unit test. This process represents the dynamic analysis aspect of the proposed technique. Dynamic analysis approaches can find the vulnerabilities with a low rate of false positives in detecting vulnerabilities due to using the real results of code execution. The Attack Evaluation component will use a repository of attack scripts, generated using the proposed Attack Generation component, to evaluate each unit test.

In the following section I will explain the details of the attack generation technique which is used to synthesize attack scripts used by the unit test evaluation component.

CHAPTER 4: Attack Vector Generation

Because my test evaluation is based on execution of attack strings, I must make sure attack strings are syntactically correct. Furthermore, I want to include all possible types of attack scenarios. Related work in generating XSS attacks rely on either expert input [71], or on reported attacks[10, 72]. It is difficult to show that all possible attack scenarios are included using these approaches.

My approach consists of two components. First I use context free grammar rules to model how JavaScript payloads are interpreted by a typical browser. Assuming they are accurate, then a successful attack must follow these grammar rules. Second, I devise an algorithm to derive attack strings systematically based on these grammar rules. Assuming the grammar rules accurately model the way the browser interprets JavaScript programs, and assuming that the attack derivation algorithm can generate at least one attack string for every type of attack, then my approach would cover all possible attack scenarios. It is possible that either I may have missed some grammar rules by which a browser interprets JavaScript programs, or the attack enumeration algorithm failed to consider a possible derivation path. Through peer review, I can improve both components in a way similar to how any security algorithms are revised. The advantage of this approach is I rely on expert know-how on the more manageable task of modeling browser behavior as opposed to the more open-ended task of enumerating possible attack scenarios.

4.1 Browser Modeling

A typical Ib browser contains multiple interpreters: HTML, CSS, URI and JavaScript. The browser behavior can be modeled as one interpreter passing control to another upon parsing specific input tokens while rendering HTML documents. I refer to the event of interpreter switching as **context switching**. For example, the URI parser transfers the control to the JavaScript parser if it detects input *javascript:* as in the case:

```
<img src="javascript:attack();" >
```

A successful XSS attack is to induce the JavaScript interpreter to execute an attack payload. I use a set of context free grammar (CFG) rules to specify possible input strings that cause the browser to activate the JavaScript interpreter to execute an attack payload. Portners et. al. [73] observed that a successful XSS attack must either call a JavaScript function (e.g. an API), or make an assignment (e.g. change the DOM). According to JavaScript language syntax, wherever an assignment operation can be executed, a function call can also be made. Therefore, without loss of generality, I assume the attack payload (referred to as PAYLOAD in the following grammars) is a function *attack()* that changes the title of the web page.

Like Halfond et. al [74], I divide the CFG into these sections: URI, CSS, HTML, Event and JavaScript. In each section I specify possible transitions to cause a JavaScript interpreter to execute an attack payload. I will then integrate these sections of grammar rules to generate attack strings. For clarity, I will use the following convention in grammar definitions: upper case words for non-terminals, lower case words for terminals, symbols sq, dq, eq for single quote, double quote and equal sign characters respectively.

```
URIATRI ::= URIHOST eq URIVAL
URIHOST ::= src | href | codebase | cite|action | background | data | classid |
longdesc|profile |usemap | formaction|icon | manifest | poster | srcset | archive
URIVAL ::= sq URI sq | dq URI dq | URI
URI ::= javascript: PAYLOAD
```

Figure 4.1: URI Grammar

URI Context: URI (Uniform Resource Identifier) strings identify locations of resources such as images or script files. Based on RFC 3986, they have the following generic syntax:

scheme: [//[user:password@] host [:port]][/] path [?query] [#fragment]

Here, the scheme represents the protocol type (such as ftp or http) used to access a resource, and the rest of the string expresses the authority and path information required to identify the resource. To cause the URI interpreter to switch to the JavaScript interpreter, the scheme must be equal to the keyword *javascript*, followed by JavaScript statements. Other possible schemes include http, ftp, and https. Since no JavaScript can be injected into schemes other than scheme JavaScript, I concentrate on describing URIs that contain the JavaScript scheme [75]. An URI can be properly interpreted by a browser only as a value of an expected attribute of a host context. I continue with the example of

```
<img src="javascript:attack();">
```

where *src* is the source attribute of the HTML *img* tag and referred to as URIHOST. Figure 4.1 represents the grammar for URI. Rule URIATRIB specifies a URI attribute consisting of a URIHOST name and the URLVAL. Rule URIHOST lists all possible URI host contexts in an HTML document. Again, for the purpose of generating attack strings, I only consider a URI of the JavaScript scheme. PAYLOAD is a special

```
STYLEATRIB ::= style eq STYLEVAL
STYLEVAL ::= (sq STYLE sq) | (dq STYLE dq) | (STYLE)
STYLE ::= CSSPROP*
CSSPROP ::= PROPNAME : PROPVAL;
PROPNAME ::= background-image | list- style-image| content | cursor | cue-after
| cue-before
PROPVAL ::= url(URI)
```

Figure 4.2: CSS Grammar

nonterminal representing a JavaScript attack payload. It signals to the attack generator that a context switch to JavaScript is possible at this point.

CSS Context : Cascading Style Sheets (CSS) specifications can be either contained in a CSS file or placed directly in HTML elements, e.g. tag definitions (using the *style* attribute or style blocks). A context switch from the CSS interpreter to the JavaScript interpreter is possible only when a URI is a property of a CSS-style element, specified by function *url()*. The argument to the *url()* function must follow the definition of URI in Figure 4.1. Figure 4.2 lists rules for URI to be included as part of a CSS-style element.

Attribute Event Context: HTML events, such as onfocus and onload, can cause context switches to JavaScript. Grammar rules in Figure 4.3 define an HTML event attribute composed of an event name EVENTNAME and value EVENTVAL. Although types of possible events vary with HTML tags, I found that the *onclick* event can be triggered in all HTML tags. As mentioned in the attack evaluation section, I change all events in the source code to the *onclick* event for attack evaluation. Rule EVENTVAL defines the value of the event which is a JavaScript statement to be executed upon the specified event.

```
EVENTATRIB ::= EVENTNAME eq EVENTVAL
EVENTNAME ::= onclick
EVENTVAL ::= sq PAYLOAD sq | dq PAYLOAD dq | PAYLOAD
```

Figure 4.3: Event Attributes Grammar

```
HTML ::= ELEM*
ELEM ::= IMG | STYLE | SCRIPT | SPECIAL
IMG ::= <img ATRIBLIST >
ATRIBLIST ::= ATTRIBUTE*
ATTRIBUTE ::= URIATRIB | STYLEATRIB | EVENTATRIB
STYLE ::= <style> CSSPROP* </style>
SCRIPT ::= <script> PAYLOAD </script>
SPECIAL ::= ( </textarea> | </title> )
```

Figure 4.4: Integration Grammar

HTML : Having modeled context switches in URI, CSS, and Event, I integrate them in a single grammar to model JavaScript execution in HTML as shown in Figure 4.4. A XSS attack script can be injected either in a tag's attribute or tag's body. Rule HTML in Figure 4.4 defines tags as a set of elements represented by the ELEM rule to cover these cases.

Since all HTML tags attributes share identical syntax, I use rule IMG to define tag *img* as a representative to model all possible context switching patterns via tag attributes. The browser can switch to the JavaScript interpreter only in the following tag attribute types: URI, CSS, and EVENT. Grammar rules for these elements have been discussed above.

In the case of injection into tag bodies, JavaScript must be enclosed by the <script> </script> tags, as specified by the SCRIP rule. However, there are a few exceptions. First, inside the <style> tag body, JavaScript can only be included as part of some

```
ADDITIVEEXP ::= PRIMARYEXP ADDITIVEPART
ADDITIVEPART::= (+ PRIMARYEXP)*
PRIMARYEXP ::= PAYLOAD | LITERAL
LITERAL = dq 1 dq | sq 1 sq | 1
```

Figure 4.5: JavaScript Additive Expressions Grammar

```
var x = " const <%= hostVar %> " ;

var x = 19<%= hostVar %>;

var x = 20 * <%= hostVar %>;

func("const" + <%= hostVar %> , param2);

if ( <%= hostVar %> == 2017) {...}
```

Figure 4.6: Injection Points in JavaScript Code

CSS properties, as specified by the STYLE rule. Second, no JavaScript is allowed in bodies of <textarea> and <title> tags. To inject JavaScript into bodies of these tags, these tags must first be closed as specified by rule SPECIAL.

JavaScript : JavaScript code can be placed either directly in HTML elements (e.g. through tag events such as *onclick*) or in <*script*> blocks. Attackers can inject a malicious payload into a block of vulnerable JavaScript code. A successful attack must manipulate the JavaScript interpreter into executing the payload, *attack()*.

Injection points in JavaScript are (Java) host variables. While host variables could be used in any JavaScript construct, such as part of a variable or function name (e.g., `var vname<%= hostVar %> = 'value';`) such cases make little sense. Host variables are primarily used to pass server-side values to JavaScript code. Thus I only consider scenarios where attack scripts are injected as part of a string or a numeric literal in expressions as illustrated in Figure 4.6.

The goal of the each attack script is to turn the host variable into an expression so a function call can be made. A successful attack can be any syntactically correct JavaScript expression. Without loss of generality, I generate attack expressions using only the plus(+) operator as it can be used on both string and numeric data. The resulting expression is referred to as an additive expression. Its grammar is shown in Figure 4.5. The first two lines in Figure 4.5 define JavaScript additive expressions as expressions composed of multiple string/numeric literals or expressions concatenated to each other using the plus(+) operator in JavaScript. The PAYLOAD non-terminal is a placeholder for attack payloads.

4.2 Attack String Generation

The goal of the attack string generation is to generate all possible types of attacks using the grammar rules described in the previous section. I describe the generation process in this section.

Sentence Derivation : I generate XSS attack strings based on any of the grammars described above by constructing a leftmost derivation tree [76] from the start symbol of each grammar. The following are derivation steps for a sentence based on the HTML *img* tag grammar.

ELEM ::= IMG

::= <img ATTRIBUTE*>

::= <img EVENTATRIB >

::= <img EVENTNAME eq EVENTVAL >

::= <img onclick = PAYLOAD >

Generating Attack Strings : Attacks can be injected in any part of an HTML element, a CSS block, or JavaScript expression. Consider the following example where a host variable, *hostVar*, is a function parameter on the right hand side of the assignment statement for variable *fName*.

```
var fName =func("Dr. <%= hostVar %> ");
```

The attack script must take into consideration existing characters both to the left and to the right of the injection point (point in which the *hostVar* is placed), referred to as left context and right context respectively as described in my previous work[77].

To fit the attack into the left context, one may close the string parameter with character " followed by a context switch using a new additive expression. The resulting attack string would be: " + **attack()** + " and the successful injection is shown as follows:

```
var fName =func("Dr." + attack() + "");
```

I first derive a sentence based on the start symbol of the grammar. Each sentence will lead to successful execution of a JavaScript attack. To systematically generate attacks for all possible existing left and right contexts, I must produce all possible partial sentences. The following is a possible derivation for an additive expression in JavaScript leading to a complete sentence:

ADDITIVEEXP ::= PRIMARYTEXP ADDITIVEPART

. . . ::= LITERAL ADDITIVEPART

. . . ::= "1" ADDITIVEPART

. . . ::= "1" (+ PRIMARYTEXP)*

. . . ::= "1" + PRIMARYTEXP + PRIMARYTEXP

. . . ::= "1" + PAYLOAD + "1"

For each complete sentence derived from the grammar, I generate multiple versions of partial sentence as potential attack strings. Each version will be shaped by removing one token from the either the beginning or from the end of the previous version starting from the initial sentence. These versions represent different possible ways an attack can be successfully interpreted by the browser taking advantage of the injection point's left and right contexts.

This removing process will continue until the first PAYLOAD symbol is reached. For example, given the additive expression derived earlier, the following versions of attack strings can be generated.

1. "1" + PAYLOAD + "1"
2. 1" + PAYLOAD + "1"
3. " + PAYLOAD + "1"
4. + PAYLOAD + "1"
5. PAYLOAD + "1"

To consider existing contexts to the right of the injection point, I systematically generate multiple versions of any partial attack string by removing one token from the end of the previous one until the PAYLOAD symbol is reached. The following four versions of attack strings are derived based on attack string 3 from the previous list:

6. "+PAYLOAD + "1
7. **"+PAYLOAD + "**
8. "+PAYLOAD +
9. "+PAYLOAD

Attack string in item 7 can be successfully injected into host variable *hostVar*.

Closures Operators: Closure operators (*, +) in my grammar rules may result in an infinite number of derivations. The following example shows a derivation by applying the closure operator up to two times on the ELEM rule. A total of six derivations are possible for the ELEM non-terminal:

HTML ::= ELEM*
ELEM ::= (IMG | SCRIPT)*
::= IMG
::= IMG IMG
::= IMG SCRIPT
::= SCRIPT
::= SCRIPT SCRIPT
::= SCRIPT IMG

I observed that attack strings containing more than one attack payload are redundant. This is because a successful attack only needs to execute one payload. If an attack pattern is not successful, repeating the same pattern multiple times will not help it succeed. I empirically determine the number of times closure operators need to be applied. I compute leftmost derivations by applying different upper bounds on closure operators until no more attack strings with one payload can be added. For example, for the grammar rules presented here, applying each closure operator 3 times does not generate new attack strings with one payload over applying each closure operator 2 times. Note that current grammars do not contain recursive rules.

In summary, the attack generation component focuses on synthesizing the attack scripts using a grammar-based approach. The grammar describing the pattern of attack scripts is based on internal behavior of the web browsers. Browsers are composed of different interpreters like HTML, CSS, JavaScript and URI. Upon detecting and parsing special tokens in web pages the browsers switch control from one interpreter to another one. For example, the HTML interpreter transfers control to JavaScript interprer by detecting an event attribute (e.g., onclick) in a HTML tag. I call this interpreter switching as context-switching, which defines the basic concept behind the proposed grammar-based attack generation. Because the main goal of each XSS attack script is to run a JavaScript payload, the proposed technique focuses on combining specific aspects of each grammar(context) that can transfer control to another interpreter to finally reach the JavaScript interpreter. This way an attack script can be a list of context-switching tokens to finally force the browser to run the payload in a JavaScript context, leading to a successful attack.

I selected context-switching parts of different grammars (HTML, URI and CSS) to build a central grammar to cover many attack scripts patterns. In addition, to cover cases that the attack scripts should be able to be activated inside a JavaScript context(e.g., inside a <script> tag) which does not require context-switching, I selected some parts of the JavaScript grammar. This part of the JavaScript grammar aims to cover patterns in which a JavaScript statement (a function call or assignment expression) can be interpreted successfully, regardless of the injection point. Having these grammars, the next step is using a sentence derivation algorithm to generate all possible sentences (potential attack scrips) from these grammars. These generated sentences from the grammars will be stored in a repository to be used by the attack evaluation component.

The grammar-based approach to generate the attack scrips has two advantages. First, it provides a way to formally study the attack patterns and verify which patterns are not covered by them, a goal which is very hard to achieve by using experts-provided attack repositories. Second, it provides a flexible mechanism to add new features that lead to successful attacks introduced both by browsers or new versions of the interpreters (e.g., covering HTML 5 features). This way the uncovered attack patterns can also be considered leading to less false negatives initiated by the lack of proper attack scripts.

CHAPTER 5: Automatic Repairing

After detecting vulnerabilities, the repair phase aims to automatically fix discovered vulnerabilities by replacing the incorrect encoders with proper ones. Consider the sample code snippet of Figure 5.3, security unit testing will reveal an XSS vulnerability on line 4 as the HTML encoder used on line 2 will not prevent XSS attacks in the JavaScript context.

I observed that there are four choices of contexts of base encoders: HTML, JavaScript, URL and CSS. One must also consider combinations of multiple encoders. As I discussed in the introduction, a wrong order of encoders can also lead to vulnerabilities [1]. The OWASP secure programming guideline [78], a highly regarded source for secure programming, suggests the following six possible encoders and their combinations: HTMLEncoder, JavaScriptEncoder , CSSEncoder, URLEncoder ,JavaScript(HTML()) and JavaScript(URL()) as adequate for preventing the vast majority of XSS vulnerabilities. I refer to this list of six choices as *candidate encoders*.

I explore the possibility of automatically fixing XSS vulnerabilities by trying each of the possible candidate encoders to replace vulnerable encoders and use the attack evaluation mechanism described in section 3.2 to verify if the replacement produces a program not susceptible to XSS attacks. This repair strategy is computationally feasible for most program structures due to the limited number of candidate encoders (6 encoders) and the short time required to verify each encoder replacement. However, there are a few caveats to the auto fixing approach.

First, I may be able to fix a vulnerability but unintentionally lead to unexpected

```
<% cmp= request.getParameter("cmp"); %>
<input name="cmp" >
<script>
var x=document.getElementByName("cmp");
x.setAttribute("value", '<%=escapeJavaScript(escapeHtml(cmp)%>
</script>
```

Figure 5.1: Over-encoding: Safe but broken output[3]

behavior. For example, a *JavaScript(Html())* encoding sequence may be applied to encode a variable (e.g., *companyName*) inside a JavaScript block. If the value of *companyName* happens to be a safe value of Johnson & Johnson, what would be displayed to the end user could be "Johnson &amp; Johnson". I refer to this problem as *over-encoding.* Although this encoder combination shows no vulnerability, but the repaired code (newly added encoder) will transform the normal web page contents to a safe but broken one. For example, Figure 5.13 shows a verified proof of the concept code snippet that the combination of the JavaScript and HTML encoders leads to safe code but it can destroy the displayed value of the *input* tag.

It is very difficult to avoid over encoding as there is no precise definition. I can minimize the likelihood of over-encoding by considering encoders in the candidate encoder list. Furthermore, I choose repairs with a single encoder over repairs that involve double-encoder.

Second, popular encoding libraries such as Apache, Spring framework, and ESAPI, differ in implementation details that can cause vulnerabilities, as illustrated on lines 1 and 2 of Figure 5.2. On line 1, a JavaScript encoder should be used to sanitize vari-

```
1  1) <%  String param =
   JavaScriptEncoder(request.getParameter("param"));%>
2  <script> Func( 9<%= param %>); </script>
3  // Attack Script : + attack();
4  2) <script> window.setInterval('<%= param %>');</script>
5  // Successful Attack Script: attack();
```

Figure 5.2: Proper Encoders but Vulnerable

```
<% String user = request.getParameter("username");
user = escapeHtml(user); %>
User Name: <div ><%= user %> </div>
<img src="plus.gif" onclick="details('<%= user %>')" >
```

Figure 5.3: Original Source Code

able *param*. Both Apache and Spring libraries did not prevent the attack string listed on line 3, but the JavaScript encoder from the ESAPI library is safe. The reason is that ESAPI encodes character plus(+) while the the other two leave it unchanged.

Third, it is possible that no fix can be found by using one of the candidate encoders. In such cases, I will defer the fix to developers. For example, I cannot fix unsafe programming practices outlined by OWASP, as illustrated on lines 2 and 4 of Figure 5.2. JavaScript API *setInterval()* is inherently unsafe because it may take *attack()* directly as an argument. No encoders can fix this vulnerability. I do not consider repairs that require structural changes to the program, like adding a new variable or deleting statements.

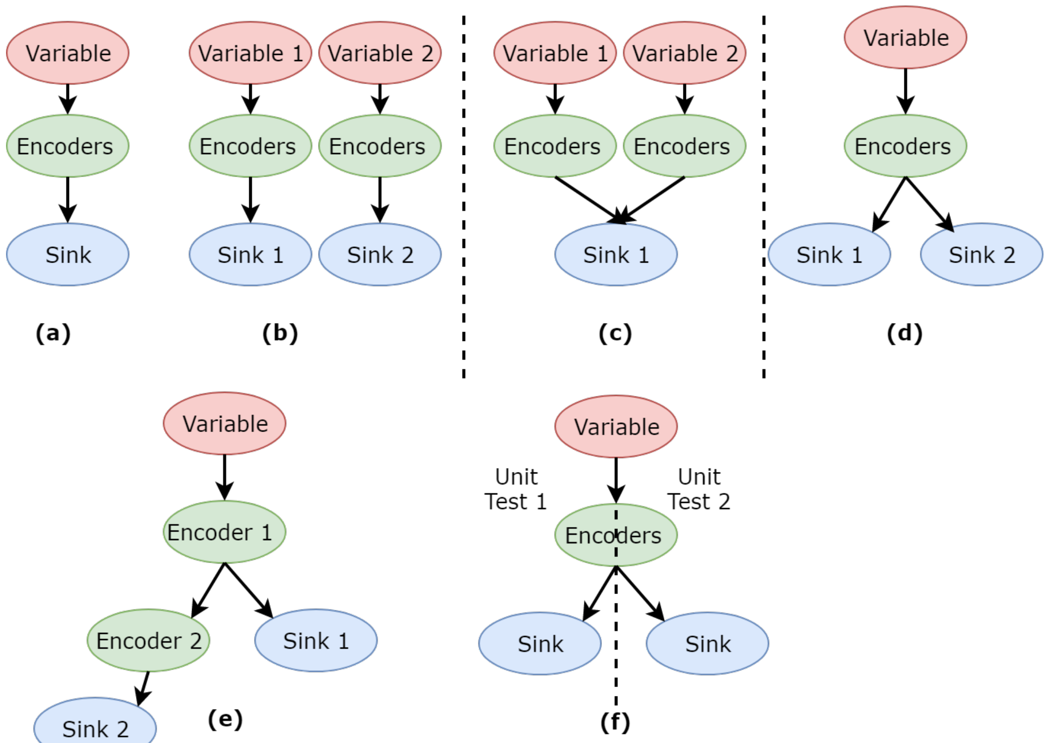


Figure 5.4: Different Cases of the Automatic Repair: (a) Single Variable, Single/Multiple Encoders. (b) Multiple Variables (c) Multiple Variables -Single Sink (d,e) Multiple Sinks - Shared Encoder. (f) Multiple Unit Tests

Finally, I only consider vulnerable code where up to two encoders are used in a sequence, which should cover the vast majority of cases [79]. Vulnerabilities with more than two encoders in a sequence are referred to the developer as auto-fixing is likely to lead to over encoding.

The core task for repair is to replace the vulnerable encoder and perform XSS unit testing to either accept or reject the fix. Figure 5.4 shows all possible scenarios for encoder replacement. I examine each scenario in detail.

5.1 Single Variable

In this case a tainted data-flow only contains one tainted variable and the tainted variable contains up to two encoders in its path from the untrusted origin to the sink as shown in Figure 5.4(a). Figure 5.5 shows a sample code snippet for this case in which the tainted variable *user* has been used in a sink on line 3 after being sanitized using encoder *escapeHtml()* on line 2.

```
1  <% user= request.getParameter("user");
2  user= escapeHtml(user); %>
3  <a onclick="fn('<%= user %>');">Details</a>
```

Figure 5.5: Single Variable, Single Encoder

```
1  <% user= request.getParameter("user");
2  user = escapeHtmlDecimal(user);
3  user2 = escapeJavaScript(user)+ "constant"; %>
4  <a onclick="fn('<%= user2 %>');"> More Details</a>
```

Figure 5.6: Single Variable, Multiple Encoder

Repairing this vulnerability entails replacing the vulnerable encoder (*escapeHtml*) on line 2 with another one from the list of *candidate encoders*. After modifying the code with each of the candidate encoders the modified code should be tested again. In this example, encoder (*escapeJavaScript(escapeHtml*)) would fix the vulnerability. The computational complexity of this case is the time required to test all the candidate encoders.

Figure 5.6 shows an example where an untrusted variable is sanitized by two encoders on lines 2 and 3 before sending its value to the browser via the sink statement of line 4. This code is vulnerable because of the order of encoders. Encoder *escapeHtmlDecimal()* on line 2 replaces the single quote character with its decimal equivalent which can bypass the JavaScript encoder and be sent to the browser unchanged. Once

```
1 <% user = request.getParameter("user");
2 user = escapeHtml(user); %>
3 <a onclick=" details('<%= user %> ');" > Details </a>
4 <% email=request.getParameter("email");
5 email = escapeHtml(email); %>
6 <a onclick=" fn(' <%= email %> '); " > Send </a>
```

Figure 5.7: Multiple independent tainted variables

decimal encoded characters are parsed by the browser they will be decoded back to the original single code leading to a successful attack as mentioned in introduction section.

To find a solution for this two-encoder case, I test the following combinations from the candidate list where e1 and e2 refer to encoders on lines 2 and 3 respectively.

- { e1: escapeJavaScript , e2: escapeHtml }
- { e1: escapeJavaScript , e2: escapeURL }

This single-variable case also covers situations in which two encoders are nested in one statement such as the code below:

```
user = escapeJavaScript( escapeHtml(user));
```

The single-variable case can be generalized to situations where a unit test contains multiple independent data-flows as shown in Figure 5.4(b). Figure 5.7 shows a unit test that contains two tainted variables (*user* and *email*), they are used in independent different sinks (line 3 , 6) with separate encoders on lines 2 and 5. In this example, both encoders are incorrect. My approach can automatically fix these

```
1 <% user = request.getParameter("user");
2 user = escapeHtml(user);
3 email = request.getParameter("email");
4 email = escapeHtml(email);
5 fullusr = user+"(" + email +")"; %>
6 <a onclick=" fn('<%= fullusr %>'); " > Details </a>
```

Figure 5.8: Multiple Variable - Single Sink

vulnerabilities by replacing both encoders as *escapeJavaScript()*. Because the vulnerable sinks have independent data-flows they can be evaluated at the same time. The computational complexity for cases in Figure 5.4(a) and (b) are the same.

5.2 Multiple Variables - Single Sink

In this scenario one security sink is the end point of multiple untrusted variables with separate encoders in their data-flows as shown in Figure 5.4(c). A vulnerability is reported if at least one of the encoders is incorrect. Figure 5.8 shows such a case in which two tainted variables *user* and *email* are concatenated to shape the third variable *fulluser* to be used in the sink on line 6 after *user* is sanitized on line 2 (refereed as e1) and *email* on line 4 (refereed to as e2).

I observe that in most cases, variables in a given sink appear in the same web application context. This implies all variables should use same encoders. In the example of Figure 5.8, because variables *user* and *email* appear in the same context (i.e. JavaScript argument in an event), considering the following replacements are sufficient.

1. { e1: escapeJavaScript , e2: escapeJavaScript }

```
1  <% user = request.getParameter("user");
2  user =escapeHtml(user); %>
3  <a onclick= " Add( ' <%= user %> ' ) " > Add </a>
4  <a onclick= " Edit( ' <%= user %> ' ) " > Edit </a>
```

Figure 5.9: Multiple Sinks-Shared Encoder

2. { e1: escapeHtml, e2: escapeHtml }
3. { e1: escapeCSS, e2: escapeCSS }
4. { e1: escapeURI, e2: escapeURI }
5. { e1: escapeJavaScript (escapeHtml()) , e2: escapeJavaScript (escapeHtml()) }
6. { e1: escapeJavaScript (escapeURI()) , e2: escapeJavaScript (escapeURI()) }

However, one could imagine rare cases where multiple variables in one sink may be rendered in two or more contexts. For such cases, I must consider testing replacements where e1 and e2 are different, or 6*6=36 encoder combinations. This would be computationally expensive if many variables are involved. I believe such cases are rare. So my proposal is to only test the same encoder sequence for all variables at the same time. If a repair cannot be found, this may indicate multiple contexts are involved for the same sink. I defer repair for such vulnerabilities to developers.

Multiple Sinks - Shared Encoder: These are cases where different sinks share the same set of encoders as shown in Figure 5.4(d) and (e). A vulnerability appears when a sink's context does not match the shared encoder. Figure 5.9 shows such a case in which the sinks on lines 3 and 4 use the same encoder of line 2. To fix this vulnerability, the encoder on line 2 need to be replaced by

```
1 <% user = request.getParameter("user");
2 user =escapeJavaScript(user); %>
3 <p> <%= user %> </p>
4 user =escapeHtml(user);
5 <a onclick= " Add( ' <%= user %> ' ) " > Edit </a>
```

Figure 5.10: Shared Encoder: Different Contexts with solution

```
1 <% user = request.getParameter("user");
2 user =escapeJavaScript(user); %>
3 <a onclick= " Add( ' <%= user %> ' ) " > Add </a>
4 user =escapeHtml(user);
5 <p> <%= user %>  </p>
```

Figure 5.11: Shared Encoder: Different Contexts and no solution

```
user=escapeJavaScript(escapeHtml(user))
```

Moreover, a developer may add an extra encoder before one of the sinks as on line 4 of Figure 5.10. The more general pattern for this case of multiple sinks sharing common encoders is shown in Figure 5.4(e). This code is vulnerable because the encoder on line 2 does not prevent attacks to line 3. Using the list of candidate encoders, the repair found for the encoders on lines 2 and 4 would be:

Line 2: `user = escapeHtml(user)`

Line 4: `user = escapeJavaScript(user)`

However, there are situations where no repair can be made for this pattern of code.

```
1  <% user = request.getParameter("user");
2  user =escapeHtml(user)
3  if(editMode){ %>
4  <a onclick="fn('<%= user %>')" > Edit User </a>
5  <% } else { %>
6  <div> User Name : <%= user %> </div> <% } %>
```

Figure 5.12: Share encoder in multiple unit tests

Consider the example in Figure 5.11 where encoded variable *user* is used in two different contexts: JavaScript on line 3 and HTML on line 5. The code is vulnerable and a repair cannot be found for encoders on lines 2 and 4. The reason is that none of the OWASP two-encoder combinations ({ Line 2: HTML , line 4: JavaScript} or { Line 2: URL , Line 4: JavaScript} will lead to safe code.

To repair this vulnerability, a new variable will have to be created, changing the structure of the program. My current approach does not consider such moves. Future research is needed to thoroughly explore this strategy.

5.3 Multiple Unit Tests

So far I have considered possible scenarios to repair a vulnerability within a single XSS unit test through encoder replacement. I consider next situations where vulnerabilities are discovered in two different XSS unit tests derived from the same JSP page as illustrated in Figure 5.4(f). As long as fixes for each XSS unit recommend same replacements, the final fix for the JSP page can be easily constructed. An example of such a case is illustrated in Figure 5.12. In this case, each XSS unit test is based on a different branch of *if/else* statements. Lines 1,2,3,4 are in one XSS unit test

```
1  <% cmp= request.getParameter("cmp"); %>
2  <input name="cmp" >
3  <script>
4  var x=document.getElementByName("cmp");
5  x.setAttribute("value", '<%=escapeJavaScript(escapeHtml(cmp)%>
6  </script>
```

Figure 5.13: Over-encoding: Safe but broken output[3]

and lines 1,2,3,6 are in another one. By comparing the AST (Abstract Syntax Tree) of the unit tests and the original source code, the shared statements (including the encoders) can be determined. Line 2 contains the shared encoder between the two unit tests. Similar to the shared encoder in Figure 5.10, the correct encoder on line 2 should satisfy two contexts, as in:

```
cmp = escapeJavaScript( escapeHtml(cmp))
```

However, it is possible that there is a conflict in repairs for each XSS unit test. In such a situation structural changes to the code are required by a developer to fix this vulnerability.

In summary, I explored the possibility of automatically fixing the XSS vulnerabilities by trying each of the candidate encoders to replace vulnerable encoders. The list of candidate encoders is limited (6 choices) leading to a computationally feasible solution. This code repair mechanism covers different combinations of tainted variables and their corresponding encoders and security sinks as below:

- Single Variable, Multiple Encoders: One tainted variables with one/multiple enders in its dataflow ending in one security sink.(Figure 5.4(a)

- Multiple Variables : Multiple tainted variables (with their encoder and sinks) in one unit test (Figure 5.4(b).

- Multiple Variables -Single Sink : Multiple tainted variables with separated encoders ending at one security sink (Figure 5.4(c).

- Multiple Sinks - Shared Encoder: One tainted variable with one/multiple encoders used in multiple security sinks (Figure 5.4(d,e).

- (f) Multiple Unit Tests: One tainted variable and with its encoder and security sinks shared in two unit tests (Figure 5.4(f).

I used code re-factoring to replace the vulnerable encoders with candidate ones without restructuring the code and statements. The effectiveness of each of the replaced encoders will be verified using the attack evaluation mechanism used to discover the vulnerabilities (Section 3.2). The encoder showing no vulnerability in this evaluation step will be used as the solution to replace the vulnerable one in the code. There are situations where the repairing of the vulnerable encoders needs code restructuring (e.g., adding new variables) which is not covered in my study. The introduced auto-fixing technique can sometimes lead to over-encoding, in which the displayed output is safe but broken. Also, encoding libraries have different implementation details leading to different vulnerability evaluation results.

CHAPTER 6: Evaluation

Our evaluations use iTrust, an open source medical records application with 112,000 lines of Java/JSP code [80]. Project iTrust has 235 JSP files and I use all of them for this evaluation. I seek to evaluate the following research questions.

(1) Are the assumptions made in my approach valid in iTrust?
(2) How effective is the described approach at detecting XSS vulnerabilities?
(3) How does XSS Unit-Testing compare with existing tools in detecting vulnerabilities?
(4) What is the computational performance of the described approach at detecting XSS vulnerabilities?
(5) How effective is the describe approach at auto-fixing detected vulnerabilities?

6.1 Assumption Verification

We assume that all web pages can be executed in a unit test environment without runtime errors. This implies that all resources required to run these web pages, such as application servers, database servers and external libraries are available for both vulnerability detection and repair phases. These requirements are met with the iTrust project. Each of the 235 JSP pages can be executed successfully as unit tests. I use the Apache TomCat as the application server and mySQL server as the database. iTrust uses Apache StringEscapeUtils libraries to encode the outputs and traditional JSP tags to generate outputs.

We assume that untrusted variables are independent of each other. This means that if a unit test contains more than one variable that may contain malicious input, I can find all XSS vulnerabilities by testing each variable independently. Out of 2268 sinks in iTrust, 27 contain multi-variables. In all these cases my encoding independence assumption is true.

We also assume that the JavaScript code does not change the web context of sinks and server-side variables are only used as values in JavaScript programs. I found these assumptions are true in all cases in where server-side values are passed to JavaScript blocks. We also could not find any JavaScript code inducing vulnerable points that cannot be triggered at runtime, which means I have not missed any injection points in client-side code that can lead to a false negative.

We manually observed that all the requests from the Internet are handled via standard Java HttpServerRequest library in JSP pages and there are no web services or REST API calls to any other part of the application. All the generated responses are rendered as full web pages (HTML + CSS + JavaScript) and there are no partial requests using Asynchronous JavaScript calls (AJAX) or any data-only (e.g., JSON) data communications between server and client. It shows that all the security sinks are placed in serve-side JSP code and all the output data should be encoded at server-side which means covering the server-side code in generating unit tests can avoid false negatives due to missing any vulnerable sinks.

6.2 Vulnerability Detection

We compared my XSS unit testing approach with security black box testing using a popular open source security testing tool ZAP [11]. Table 6.1 summarizes my evaluation results.

We found 24 zero-day vulnerabilities due to misuse of encoders. The following code snippet provides an example from iTrust where HTML encoding is used in a

Table 6.1: Summary of vulnerability findings

	Detected Vuln.	True Positives	False Positives
ZAP	119	10	109
XSS Unit Tesing	24	24	0

JavaScript context.

```
<a onclick="func('<%= escapeHtml(input) %>')" > Link</a>
```

ZAP has a very high false positive rate: 91%. No false positives were reported by my approach. The reason for ZAP's high false positive rate is because it does not confirm findings through execution. Instead it uses string matches to find attack scripts in output pages, as illustrated in section 3.2.

This means that once an attack vector appears in the output web page, ZAP will report it as a successful attack even though the attack string cannot be successfully executed. For example, on line 1 below, an HTML encoder is correctly used to sanitize input but it is incorrectly reported as a vulnerable code by ZAP. The reason is that attack vector **'+attack() + '** which is used to attack a JavaScript block can bypass the HTML encoder and appears intact in the body of <p> tag (on line 2) and thus, will be reported by ZAP as a successful attack.

```
1)<p> <%= escapeHtml(input)%> <p>


2)<p> '+attack() + ' <p>
```

Our approach found 14 vulnerabilities ZAP did not find. All these cases are due to the lack of test coverage by ZAP. ZAP does not test all execution paths. In my approach, a separate XSS unit test is created for each possible execution branch in a JSP file. In addition, some vulnerabilities are triggered by events, such as failure to load an image. My test evaluation approach handles such situations.

6.3 Attack Generation

My attack evaluation component reads attack strings required to evaluate the unit tests from a repository of attack scripts. This repository can be prepared using the results of my attack generator or from other sources. I compared my grammar based attack generation results with two open source XSS attack repositories: ZAP repository and the HTML5Sec web site [71]. The HTML5Sec attack repository found fewer vulnerabilities than the ZAP repository. However, I found vulnerabilities that cannot be detected by ZAP or HTML5Sec repositories. One example is shown below.

```
<div style="height: <%= escapeHtml(input) %>px; "> </div>
```

The following attack string generated by my approach can detect this vulnerability.

```
;background-image:url('javascript:atk()');
```

Attack repositories in ZAP and HTML5Sec rely on contributions from pen-testing experts. My approach systematically derives attack strings based on a set of grammar rules modeling the behavior of browsers interpreting JavaScript programs.

6.4 Computational performance

We looked at the performance of XSS unit testing using experiments performed on a desktop Mac with a 2.7 GHz Intel core i5 with 8GB RAM. My attack generator produced 223 attack strings, which were applied to each unit test. For iTrust, it takes 17 seconds on average to evaluate a XSS unit test. A JSP file may contain multiple branches of execution paths but only those containing sinks with tainted variables will be tested. My evaluation of 235 JSP pages in iTrust shows that on average a JSP file leads to 29 XSS unit tests. On average, if a JSP page contains no vulnerabilities, my approach will take 493 seconds or 8.2 min to complete all the unit tests. Generation of XSS unit tests is much faster than running all the tests. Because each JSP file can be tested independently, this approach lends well for parallel processing. Overall, I believe the approach I described in this paper may scale well for large applications.

6.5 Auto-Repair

We applied the described auto-fixing mechanism to all 24 vulnerabilities found in iTrust. My approach is able to automatically fix all of these vulnerabilities. Figure 6.1 shows an example of a vulnerability on line 1 and its repaired version on line 2. Line 1 shows a vulnerability due to incorrect use of a HTML encoder (*escapeHtml()*) for JavaScript context (*onclick* event attribute) for untrusted variable *tempName*. This vulnerability can be exploited using an attack script like **'+ attack() + '**.

```
1  <a onclick= "fn('<%= escapeHtml(user) %>')" > ... </a>
2  <a onclick= "fn('<%= escapeJavascript( escapeHtml(user )) %>')"
   >...</a>
```

Figure 6.1: Vulnerable Code and Repaired Version

All iTrust vulnerabilities are of the pattern (a) and (b) in Figure 5.4. The time required to evaluate each candidate encoder is the same as the time required to evaluate the unit tests for the vulnerability detection phase. On average it takes testing for two candidate encoders before a fix is found.

Because the evaluation of encoders and their combinations is sequential, the order of candidate encoders used to find the proper encoder determines the time required to fix each vulnerable unit test. The effectiveness of this order is sensitive to the context (HTML, JavaScript, CSS, URI) of vulnerable points. Based on examining different orders of encoders to repair the iTrust vulnerabilities, I observed that if the combinations of JavaScript encoder is placed at the top of the candidate encoders list, it would lead to find the repair solutions with minimum repair time. This repair time is proportional to the time required to evaluate each unit test against all attack vectors. I could find the proper encoders in the second effort of code modification on average and because each unit test evaluation took 17 seconds in my case (depending on computing environment) the average time of auto-fixing was 2*17 =34 seconds per unit test. This reasonable repair time is because all of the encoder placement vulnerabilities found in iTrust are due to misuse of the HTML encoder for Javascript contexts such as tag attributes (e.g., onclick). Thus, if I examine the JavaScript encoders first, it leads to find the proper encoder sooner. This forms a reasonable rule that in order to reduce the repair time the list of candidate encoders should be shaped by prioritizing the candidate encoders with respect to the majority of context of vulnerabilities. This order can be different for different applications.

CHAPTER 7: Summary and Future Works

In this work I introduced the automatic unit-testing to detect vulnerabilities in web applications. This approach aims to discover XSS vulnerabilities in web applications by combining static and dynamic analysis techniques in a unit-testing environment leading to reducing the false positive and false negative rates. Finding cross-site scripting vulnerabilities due to encoding mistakes is the research focus. These vulnerabilities are because of difficulties in inferring the correct context of the web application to apply the proper output encoder by the developers. Section 2.1 highlights issues that make effective output encoding a challenging problem for developers. There is not a general purpose data encoding function to be used in different contexts of HTML, JavaScript, CSS and URI. To handle encoding requirements of these different contexts, various vulnerability prevention and detection mechanisms have been studied. In section 2.2 I explained the vulnerability prevention mechanism including secure coding guidelines and auto-sanitization techniques (based on type-inference methods) to avoid the XSS vulnerabilities. Next, in section 2.3 I focused on vulnerability detection mechanisms based on static and dynamic analysis techniques.

All of the prevention and detection mechanisms are compared based on "Context Sensitiveness", compatibility with "Legacy Applications" and "Code Coverage" criteria. **Context Sensitiveness** is the ability of a detection(prevention) technique to consider the context (grammar) in which an encoding function has been used to report(prevent) a vulnerability. For example, if a HTML encoder has been used for a JavaScript context (e.g., inside a <script> tag) but that encoder usage is not reported by a vulnerability detection technique as an encoding mistake, that detection

Table 7.1: Summary of Approaches

Approach Criteria	Context Sensitiveness	Legacy Applications Support	Code Coverage
Auto Sanitization	Yes (in a new language)	No	Yes
Static Analysis	No	Yes	Yes
Dynamic Analysis	Yes (Depends on attack vectors)	Yes (Code Coverage Problem)	No

technique is not context-sensitive.

Legacy Applications focuses on the ability of a vulnerability detection/prevention technique to be applied to legacy applications without requiring any rewriting of the code completely or partially. Many of the prevention mechanisms that use type-inference techniques (e.g., AngularJS) need developer to rewrite the existing codes in a new language to activate the security features of that mechanism. This can lead to noticeable time and cost for legacy applications.

Code Coverage is the ability of a detection/prevention mechanism to cover all potential vulnerable points of a web application. Proper code coverage means lower false negatives due to missing vulnerable points in the source code.

In Table 7.1 I compared these approaches based the above mentioned criteria.

Using auto-sanitization techniques to prevent vulnerabilities provide context sensitiveness because of using language features or annotations that help detection of context of the encoding functions. These techniques can be applied to the whole source code (by rewriting the code) ,and thus have complete code coverage. However rewriting the source code makes them an expensive approach for legacy applications.

Table 7.2: Summary of Approaches

Approach Criteria	Context Sensitiveness	Legacy Applications Support	Code Coverage
Auto Sanitization	Yes (in a new language)	No	Yes
Static Analysis	No	Yes	Yes
Dynamic Analysis	Yes (depends on attack vectors)	Yes (Code Coverage Problem)	No
Unit-Testing Vuln. Detection	Yes	Yes	Yes

Static analysis techniques can be easily applied to the whole source code to find all potential vulnerable points using source code scanning tools. This means they can be easily applied to existing source code of legacy applications. However, these techniques only check the existence of encoding functions (during source code scanning), and not their effectiveness, leading to noticeable false positives in the reported vulnerabilities.

Dynamic analysis techniques are context-sensitive because they really execute the applications against the attack vectors. Therefore the reported vulnerability which are based on runtime reaction of the applications are automatically context-sensitive. This runtime evaluation can be used for legacy applications as well because it does not require any changes on source code. However, the attack scripts (and other test inputs) used in dynamic analysis of applications can not reach all parts of an application, leaving some of the vulnerable points unchecked. This limitation in code coverage introduces false negatives for this category of techniques.

Considering the advantages and limitations of the above mentioned techniques, I introduced a new technique to combine the static and dynamic analysis methods to reduce their false positives and negatives rates. This new technique can have full code coverage due to using static analysis and no false positives because of really executing the application against attack scripts. Moreover, it does not need to modify the source code of legacy applications which makes it an affordable approach for such applications.

I used a unit-testing technique as a base mechanism to integrate static and dynamic analysis. This novel technique has been explained in chapters 3 (Unit Test Extraction and Evaluation) and 4 (Attack Script Generation). I also introduced a mechanism to automatically fix the reported vulnerabilities in chapter 5. This automatic repair of reported vulnerabilities is based on code refactoring of vulnerable encoding functions. The correctness of the suggested repairs will be validated by applying the same evaluation mechanism used for vulnerability detection leading to no false positives. These advantages of the introduced unit-testing vulnerability detection method are expressed in the last row of Table 7.2. This technique is composed of unit testing extraction and evaluation, attack generation and vulnerability, and auto-fixing components that are summarized below.

The "unit test extraction" component aims to extract unit tests from each web page for XSS vulnerability detection. The server side code of each web page (e.g., JSP pages) contains multiple flow control statements, such as *if/else*, leading to multiple execution paths. Each unit test corresponds to one execution path and can be requested independent of other unit tests like its parent web page. This way, all the source code will be used to achieve the maximum code coverage leading to very low false negatives. To confirm having a vulnerability in each unit test, I need the unit test evaluation component.

The "unit test evaluation" component aims to run the extracted unit tests in a real environment to confirm having vulnerabilities. This confirmation is done through a unit test evaluation process which runs unit tests in an execution framework like JUnit. This prevents false positives in the reported vulnerabilities. To execute unit tests in unit testing frameworks (e.g., JUnit), test drivers are required. I use a test driver that invokes and renders unit test pages using a headless browser against all the attack scripts I generated using the attack generation component.

The "attack generation" component focuses on synthesizing the attack scripts using a grammar-based approach. The grammar describing the pattern of attack scripts is based on internal behavior of the web browsers composed of different interpreters like HTML, CSS, JavaScript and URI. Upon detecting and parsing special tokens in web pages the browsers switch control from one interpreter to another one. I call this interpreter switching as context-switching. Because the main goal of each XSS attack script is to run a JavaScript payload, the proposed technique focuses on combining specific aspects of each grammar (context) that can transfer control to another interpreter to finally reach the JavaScript interpreter. This way an attack script can be a list of context-switching tokens to finally force the browser to run the payload in a JavaScript context leading to a successful attack.

I also explored the possibility of automatically fixing XSS vulnerabilities by trying each of the possible encoders to replace vulnerable encoders and use the attack evaluation mechanism described in section 3.2 to verify if the replacement produces a safe solution. This repair strategy is computationally feasible for most program structures due to the limited number of candidate encoders.

There are several contributions of this work. I minimized false positives by confirming vulnerabilities via execution in a real browser. I also minimized false negatives by ensuring path coverage for unit tests as well as systematically generating attack strings using grammars based on modeling how browsers interpret JavaScript programs. Moreover, the proposed auto-fixing mechanism can fix many XSS vulnerabilities.

7.1 Future Works

This work can be extended in a number of ways. First, extending this work to handle security sinks in **client-side code** that use asynchronous calls to web services (AJAX technology) and JSON-based communications. This is important because many of the current web applications especially single-page applications heavily use these technologies and tools. Considering client-side vulnerabilities requires applying static analysis to the JavaScript code and libraries which by default have weak type-checking capabilities.

Second, extending the "auto-repair" component to include **code restructuring**, addressing one of the limitations of my current auto-fixing approach, can also be a topic for improvements in future. This extension includes cases in which a variable has been used in two different contexts and finding a proper encoding solution entails refactoring the code to introduce a new variable for one of the contexts.

The third extension can be applying the introduced technique to find and repair vulnerabilities in **hybrid mobile applications** as well. These applications are completely based on HTML and JavaScript and thus are potentially vulnerable to the same XSS attacks as regular web applications.

REFERENCES

[1] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *European Symposium on Research in Computer Security*, pp. 150–171, Springer, 2011.

[2] H. Joh and Y. K. Malaiya, "Defining and assessing quantitative security risk measures using vulnerability lifecycle and cvss metrics," in *The 2011 international conference on security and management (sam)*, pp. 10–16, 2011.

[3] "Dom based xss prevention cheat sheet." https://www.owasp.org/index.php/DOM-based-XSS-Prevention-Cheat-Sheet.

[4] "Acunetix web application vulnerability report 2016." https://www.acunetix.com/acunetix-web-application-vulnerability-report-2016/.

[5] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 587–600, ACM, 2011.

[6] "Closure templates." http://developers.google.com/closure/templates/.

[7] "Yahoo secure handelbars." https://yahoo.github.io/secure-handlebars/.

[8] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 332–345, IEEE, 2010.

[9] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Ligre: Reverse-engineering of control and data flow models for black-box xss detection," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 252–261, IEEE, 2013.

[10] Www-03.ibm.com, "Ibm - software - ibm security appscan." http://www-03.ibm.com/software/products/en/appscan, 2016.

[11] "Zap, owasp zed attack proxy project." https://www.owasp.org/index.php/OWASP-Zed-Attack-Proxy-Project.

[12] "Owasp enterprise security api." https://www.owasp.org/index.php/Category:OWASP - Enterprise-Security-API.

[13] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, 2008.

[14] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, "Current state of research on cross-site scripting (xss)–a systematic literature review," *Information and Software Technology*, vol. 58, pp. 170–186, 2015.

[15] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, pp. 152–156, IEEE, 2012.

[16] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 33–39, IEEE Computer Society, 2009.

[17] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, pp. 372–382, ACM, 2006.

[18] L. K. Shar and H. B. K. Tan, "Auditing the xss defence features implemented in web application programs," *IET software*, vol. 6, no. 4, pp. 377–390, 2012.

[19] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," in *Proceedings of the 18th international conference on World wide web*, pp. 561–570, ACM, 2009.

[20] "Xss (cross site scripting) prevention cheat sheet - owasp." https://www.owasp.org/index.php/XSS-Cross-Site-Scripting-Prevention-Cheat-Sheet.

[21] M. Graff and K. R. Van Wyk, *Secure coding: principles and practices.* " O'Reilly Media, Inc.", 2003.

[22] M. Johns, C. Beyerlein, R. Giesecke, and J. Posegga, "Secure code generation for web applications," in *International Symposium on Engineering Secure Software and Systems*, pp. 96–113, Springer, 2010.

[23] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 601–614, ACM, 2011.

[24] G. Agosta, A. Barenghi, A. Parata, and G. Pelosi, "Automated security analysis of dynamic web applications through symbolic code execution," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, pp. 189–194, IEEE, 2012.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[26] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Ligre: Reverse-engineering of control and data flow models for black-box xss detection," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 252–261, IEEE, 2013.

[27] M. K. Gupta, M. Govil, and G. Singh, "Static analysis approaches to detect sql injection and cross site scripting vulnerabilities in web applications: A survey," in *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, pp. 1–5, IEEE, 2014.

[28] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 6–pp, IEEE, 2006.

[29] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications.," in *NDss*, 2010.

[30] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317–331, IEEE, 2010.

[31] L. K. Shar and H. B. K. Tan, "Auditing the xss defense features implemented in web application programs," *IET software*, vol. 6, no. 4, pp. 377–390, 2012.

[32] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *ACM Sigplan Notices*, vol. 44, pp. 87–97, ACM, 2009.

[33] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th international conference on Software engineering*, pp. 171–180, ACM, 2008.

[34] J. Wilander, "Modeling and visualizing security properties of code using dependence graphs," in *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, pp. 65–74, 2005.

[35] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 652–661, IEEE Press, 2013.

[36] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: a learning approach to web security testing," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 347–357, ACM, 2013.

[37] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *Proceedings of the 20th USENIX conference on Security*, pp. 1–1, USENIX Association, 2011.

[38] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 387–401, IEEE, 2008.

[39] A. Armando, R. Carbone, L. Compagna, K. Li, and G. Pellegrino, "Model-checking driven security testing of web-based applications," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 361–370, IEEE, 2010.

[40] A. Avancini and M. Ceccato, "Grammar based oracle for security testing of web applications," in *Proceedings of the 7th International Workshop on Automation of Software Test*, pp. 15–21, IEEE Press, 2012.

[41] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pp. 85–94, IEEE, 2011.

[42] A. Aydin, M. Alkhalaf, and T. Bultan, "Automated test generation from vulnerability signatures," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pp. 193–202, IEEE, 2014.

[43] P. Bisht and V. Venkatakrishnan, "Xss-guard: precise dynamic prevention of cross-site scripting attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43, Springer, 2008.

[44] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in *Proceedings of the 9th International Workshop on Automation of Software Test*, pp. 1–7, ACM, 2014.

[45] J. Bozic and F. Wotawa, "Xss pattern for attack modeling in testing," in *Automation of Software Test (AST), 2013 8th International Workshop on*, pp. 71–74, IEEE, 2013.

[46] R. Sekar, "An efficient black-box technique for defeating web application attacks.," in *NDSS*, 2009.

[47] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pp. 253–262, IEEE, 2012.

[48] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 251–262, ACM, 2011.

[49] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, pp. 37–48, ACM, 2014.

[50] B. Garn, I. Kapsalis, D. E. Simos, and S. Winkler, "On the applicability of combinatorial testing to web application security testing: a case study," in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pp. 16–21, ACM, 2014.

[51] S.-K. Huang, H.-L. Lu, W.-M. Leong, and H. Liu, "Craxweb: Automatic web application testing and attack generation," in *Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on*, pp. 208–217, IEEE, 2013.

[52] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 15–pp, IEEE, 2006.

[53] Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 213–222, ACM, 2005.

[54] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th international conference on World Wide Web*, pp. 432–441, ACM, 2005.

[55] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 124–145, Springer, 2005.

[56] J. Thomé, A. Gorla, and A. Zeller, "Search-based security testing of web applications," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, pp. 5–14, ACM, 2014.

[57] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, "Xss vulnerability detection using model inference assisted evolutionary fuzzing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 815–817, IEEE, 2012.

[58] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 474–484, IEEE Computer Society, 2009.

[59] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *IEEE software*, vol. 25, no. 5, 2008.

[60] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*, pp. 148–159, ACM, 2003.

[61] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of sql injection and cross-site scripting attacks,” in *Proceedings of the 31st International Conference on Software Engineering*, pp. 199–209, IEEE Computer Society, 2009.

[62] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “Hampi: a solver for string constraints,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 105–116, ACM, 2009.

[63] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, “Dynamic test input generation for web applications,” in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 249–260, ACM, 2008.

[64] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, “Xss vulnerability detection using model inference assisted evolutionary fuzzing,” in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 815–817, IEEE, 2012.

[65] Y.-H. Wang, C.-H. Mao, and H.-M. Lee, “Structural learning of attack vectors for generating mutated xss attacks,” *arXiv preprint arXiv:1009.3711*, 2010.

[66] S. Rawat, F. Duchene, R. Groz, and J.-L. Richier, “Evolving indigestible codes: Fuzzing interpreters with genetic programming,” in *Computational Intelligence in Cyber Security (CICS), 2013 IEEE Symposium on*, pp. 37–39, IEEE, 2013.

[67] B. Garn, I. Kapsalis, D. E. Simos, and S. Winkler, “On the applicability of combinatorial testing to web application security testing: a case study,” in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pp. 16–21, ACM, 2014.

[68] “Noscript - javascript-java-flash blocker for a safer firefox experience!.” https://noscript.net.

[69] R. Pelizzi and R. Sekar, “Protection, usability and improvements in reflected xss filters.,” in *ASIACCS*, p. 5, 2012.

[70] R. Pelizzi and R. Sekar, “Protection, usability and improvements in reflected xss filters.,” in *ASIACCS*, p. 5, 2012.

[71] “Html5 security cheatsheet.” https://html5sec.org/.

[72] “Xssed | cross site scripting (xss) attacks information and archive.” http://xssed.com/.

[73] J. Portner, J. Kerr, and B. Chu, *Moving Target Defense Against Cross-Site Scripting Attacks.* Springer, Cham, Nov 2014.

[74] W. Halfond, A. Orso, and P. Manolios, “Wasp: Protecting web applications using positive tainting and syntax-aware evaluation,” *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 65–81, 2008.

[75] Ecma-international.org, “Ecmascript language specification - ecma-262 edition 5.1.” http://www.ecma-international.org/ecma-262/5.1/, 2016.

[76] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Automata theory, languages, and computation,” *International Edition*, vol. 24, 2006.

[77] M. Mohammadi, B. Chu, and H. R. Lipford, “Detecting cross-site scripting vulnerabilities through automated unit testing,” in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pp. 364–373, IEEE, 2017.

[78] “Xss (cross site scripting) prevention cheat sheet - owasp.” https://www.owasp.org/index.php/XSS-Cross-Site-Scripting-Prevention-Cheat-Sheet.

[79] Owasp.org, “Owasp antisamy project - owasp.” https://www.owasp.org/index.php/Category:OWASP$_AntiSamy_Project$, 2016.

[80] “itrust:role-based healthcare.” http://agile.csc.ncsu.edu/iTrust/wiki/doku.php.