JSPChecker: Static Detection of Context-Sensitive Cross-Site Scripting Flaws in Legacy Web Applications

Antonín Steinhauser
Oracle Labs, Brisbane
Faculty of Mathematics and Physics
Charles University in Prague
steinhauser@d3s.mff.cuni.cz

François Gauthier
Oracle Labs, Brisbane
francois.gauthier@oracle.com

ABSTRACT

JSPChecker is a static analysis tool that detects context-sensitive cross-site scripting vulnerabilities in legacy web applications. While cross-site scripting flaws can be mitigated through sanitisation, a process that removes dangerous characters from input values, proper sanitisation requires knowledge about the *output context* of input values. Indeed, web pages are built using a mix of different languages (e.g. HTML, CSS, JavaScript and others) that call for different sanitisation routines. Context-sensitive cross-site scripting vulnerabilities occur when there is a mismatch between sanitisation routines and output contexts.

JSPChecker uses data-flow analysis to track the sanitisation routines that are applied to an input value, a combination of string analysis and fault-tolerant parsing to approximate the output context of sanitised values, and uses this information to detect context-sensitive cross-site scripting vulnerabilities. We demonstrate the effectiveness of our approach by analysing five open-source applications and showing how JSPChecker can identify several context-sensitive XSS flaws in real world applications with a precision ranging from 96% to 100%.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Protection Mechanisms

Keywords

Cross-site scripting, security, string analysis, data-flow, web application

1. INTRODUCTION

Cross-site scripting (XSS) flaws continue to plague web applications despite all the efforts academia and industry have invested to eradicate them. According to the OWASP Top 10 [5], XSS flaws are the third-most common type of flaws in web applications. The consequences of a XSS attack might vary from website defacing to identity theft.

Web applications can prevent XSS attacks through a process known as input sanitisation, where potentially malicious code in user input is removed or escaped in such a way as to prevent its execution. Unfortunately, sanitiser placement in web application code is still a highly manual and error-prone process [22, 20], making it difficult for developers to fully protect their code from XSS attacks. Indeed, one missing sanitiser is often sufficient to make an application vulnerable to XSS attacks.

Initial research on XSS prevention therefore focused on detecting missing sanitisers by using static and dynamic analysis strategies [16, 17, 24, 35, 14]. While they differ in their design and implementation, a common denominator to all these approaches is *taint analysis*, which identifies execution paths in a program where malicious inputs can reach sensitive instructions without being sanitised.

Taint analysis is, however, insufficient to fully protect an application against XSS attacks. Ensuring that user inputs are sanitised before reaching security-sensitive instructions is necessary but not sufficient to prevent XSS flaws. Indeed, one also has to ensure that sanitisers match the *output context* of the values being sanitised.

In web applications, the output context of a value refers to the programming language and syntactic construct in which a value is rendered. A web page is usually composed of several snippets of code, written in different programming languages (e.g. HTML, JavaScript and CSS) that the browser will parse, interpret and render to produce the final page that is presented to the user.

XSS prevention is complicated by the fact that different output contexts require different sanitisation routines. For example, while the snippet: `javascript:alert("Hacked")` would trigger a pop-up if it was rendered in a URI, it would be printed as-is if it was rendered in HTML text. Proper XSS prevention thus requires context-sensitive sanitisers.

Manual placement of context-sensitive sanitisers is, however, highly challenging and error prone because developers must ensure that all user inputs are sanitised, and determine the output context in which the input value will be rendered.

The problem of context-sensitive sanitisation is not new. In the past, several static and dynamic solutions have been proposed [29, 31, 36, 32, 7] to detect context-sensitive cross-site scripting vulnerabilities. We discuss existing approaches extensively in Section 8.

The motivation for our work stems from the fact that no existing approaches address the major challenges we face when analysing large legacy web applications. Legacy code-

PLAS'16, October 24 2016, Vienna, Austria

ISBN 978-1-4503-4574-3/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2993600.2993606

bases are typically characterised by their huge maintenance costs, slow evolution and lack of extensive tests and documentation [12, 8, 11, 33]. For such legacy codebases, purely static approaches are thus usually preferred to dynamic approaches that rely on existing test suites. Static approaches that do not require code modifications are also preferred to approaches that require re-factoring of existing code to work.

In this paper, we present JSPChecker, a static analysis tool to detect context-sensitive sanitisation flaws in legacy JEE applications. As its name suggests, JSPChecker analyses Java Server Pages (JSPs), a commonly used technology to generate HTML pages in legacy JEE applications.

1.1 Contributions

This paper makes the following contributions:

Detection of context-sensitive XSS flaws: We introduce our tool, the JSPChecker, that detects context-sensitive sanitisation flaws in legacy web applications without requiring any change to the application or the runtime environment. We show how JSPChecker implements a novel algorithm based on data-flow analysis, static string analysis, and fault-tolerant parsing to detect inconsistent uses of sanitisers that can lead to context-sensitive XSS flaws. We further extend JSPChecker with different HTML page generation strategies that allow users to adjust the trade-off between scalability and precision.

Fault-tolerant browser model: We show how our approach supports all major languages used in modern browsers and emulates encodings and decodings of code as they happen in real-world browsers. JSPChecker can recover from HTML syntactic errors that are common in legacy web applications, and produce useful results in practice.

Empirical evaluation: In Section 6, we experimentally evaluate JSPChecker on five real-world open-source applications with 2,500 to 230,855 lines of code, and 10 to 760 JSP files. We show how JSPChecker was able to detect between 1 and 168 context-sensitive XSS flaws in the investigated applications, achieving a precision ranging from 96% to 100%.

2. MOTIVATING EXAMPLE

In this section, we walk the reader through a simple example of context-sensitive XSS flaw and introduce some of the concepts that will be used later in the paper. Following examples show snippets of Java codes that were generated from a Java Server Page (JSP). The output of a JSP page is a string that is embedded in the body of the HTTP response that is sent back to the user. All calls to `out.print` effectively append strings to the body of an HTTP response.

```
 1  public void _jspService(
 2    HttpServletRequest req,
 3    HttpServletResponse resp) {
 4     String val = req.getParameter("id");
 5     ...
 6     out.print("<td>\n");
 7     out.print("    <a href=\"javascript:
           appendText( ’");
 8     out.print(Encoder.jsEncode(val));
 9     out.print("’ )\">\n");
10     out.print("</td>\n");
11  }
```

Listing 1: The `value` variable is printed in a JavaScript URI, but is only sanitised for the JavaScript context through a call to the `jsEncode` function.

In Listing 1, the `val` variable is retrieved from the request at line 3, and must be considered as potentially malicious. Indeed, observe that the `val` variable is sanitised with the `jsEncode` sanitiser before it is actually printed at line 8. While it seems that proper precautions were taken to prevent XSS attacks, this deceptively simple example actually hides a context-sensitive XSS vulnerability.

On the one hand, at runtime, the browser performs URI decoding on every URI string, be it a regular URI or a JavaScript URI. On the other hand, the `jsEncode` sanitiser does not escape the `%` sign that is used in URI encoding. As a consequence, if the attacker injects a URI-encoded payload in the `id` parameter, the payload will be left untouched by the sanitiser and be decoded in the victim's browser before being executed by the JavaScript interpreter. For example, injecting the URI-encoded payload `%27)%3Balert(%27Hacked!` would effectively result in `’);alert(’Hacked!` after URI-decoding by the browser.

```
public void _jspService(
  HttpServletRequest req,
  HttpServletResponse resp) {
   String val = req.getParameter("id");
   ...
   out.print("<td>\n");
   out.print("    <a href=\"javascript:
       appendText( ’");
   out.print(URLEncoder.encodeForURL(
       Encoder.jsEncode(val)));
   out.print("’ )\">\n");
   out.print("</td>\n");
}
```

Listing 2: Wrapping the jsEncode sanitiser in the encodeForURL sanitiser eliminates the XSS flaw.

The snippet in Listing 2 shows the secure version of the same example. Observe how the `jsEncode` sanitiser is now wrapped in the `encodeForURL` sanitiser.

Generally, successful sanitisation against XSS requires the sequence of sanitisers to match the sequence of output contexts of an output value. In the example above, the value is printed in a JavaScript context that is itself nested in a URI context. This nesting needs to be reflected in the sanitiser sequence. Using *insufficient* sanitisers leads to context-sensitive XSS flaws.

3. OVERVIEW

In this section, we present an overview of JSPChecker, and highlight how it addresses limitations of previous approaches to enable analysis of legacy web applications.

Conceptually, our approach to detect context-sensitive XSS flaws can be formulated as an extension of static taint analysis for web applications [35, 6].

Following taint analysis terminology, statements that assign tainted data to a variable are called *sources*, security-sensitive statements are called *sinks*, and statements that remove taintedness are called *sanitisers*. Taint analysis reports tainted flaws in the form of data-flow paths that start in a source, and end in a sink without going through a sanitiser. In the context of XSS flaws detection, sinks are typically statements that print data to an HTML page.

Taint analysis thus aims at reporting XSS flaws that arise from data-flow paths that *miss* a sanitiser. To detect such paths, taint analysis will typically track data from sources to sinks, and stop when it encounters a sanitiser. On the other hand, context-sensitive XSS flaws occur on data-flow paths that go through at least one *insufficient* sanitiser. JSPChecker thus tracks data flows *from* sanitisers and only stops when it reaches a sink. Only analysing paths that contain at least one sanitiser saves computation time.

JSPChecker uses the SOOT static analysis framework [19] to track data flows in JEE applications. SOOT translates analysed applications into the Jimple intermediate representation, implements various analyses (e.g. call graph construction and points-to analysis), and provides an API to build static analyses. JSPChecker tracks data from sanitisers to sinks through a forward sparse data-flow analysis implemented in the SOOT framework. Starting from each sanitiser in a program, JSPChecker uses a forward data-flow analysis to propagate sanitised values to output statements, recording sanitisers that are encountered on each path from a sanitiser to an output statement.

JSPChecker then uses the Java String Analyser (JSA) [9] to generate HTML document approximations. Given a string expression, JSA computes a finite-state automaton that provides an over-approximation of the string values that may be generated at runtime. Detailed implementation of JSA is described in [9]. Given that JSPs generate HTML documents by printing strings into an HTTP response, JSPChecker uses JSA to over-approximate the strings produced by print statements in JSPs, and to generate HTML document approximations.

Finally, JSPChecker uses a suite of parsers to evaluate generated HTML documents and to approximate the output context of sanitised values. The suite of parsers emulates the way documents are processed in web browsers. As a result, the analysis determines the output statements that print sanitised values, and the sequences of sanitisers that are applied to each output value. The next step consists of inferring the output context of sanitised values.

For example, JSPChecker decodes the code, as would a browser, when control is passed from one parser to another. Through this parsing and decoding process, JSPChecker emulates the behaviour of the browser and keeps track of the browser output context. Figure 1 illustrates the four parsers that JSPChecker uses together with the triggers that induce switches between different parsers and the type of decoding that occurs when control is handed from one parser to another.

Finally, whenever a parser encounters a sanitised value, it determines whether the current output context matches the sequence of sanitisers associated with the value. If it detects a mismatch, JSPChecker reports a potential context-sensitive cross-site scripting flaw.

4. JSPCHECKER APPROACH

This section details the JSPChecker approach to detect context-sensitive XSS flaws.

4.1 Requirements and assumptions

Requirements: JSPChecker requires the user to only supply a mapping from sanitisers to browser output contexts. This mapping is usually specified once by a security expert and can be reused across different applications that share the same sanitisers.

Assumptions: JSPChecker makes a few assumptions in order to be scalable and precise. First, JSPChecker models user inputs, public fields, and external values as empty strings. This assumption is in line with previous work [13] and enables JSPChecker to generate *practical* HTML document approximations. Second, JSPChecker generally assumes that sanitisers always receive tainted data or the output of another sanitiser as input. This assumption speeds up and simplifies JSPChecker analysis significantly by allowing JSPChecker to analyse only execution paths that contain at least on sanitiser.

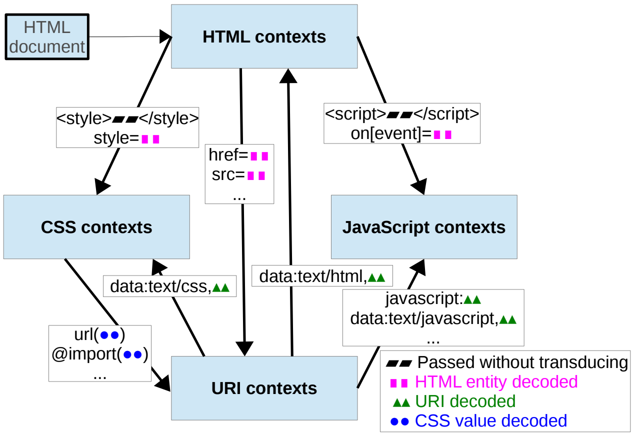

Figure 1: Transitions and decoding between supported parsers. Black arrows connect two parsers if the later can be invoked by the former. Labels on arrows indicate the syntactic constructs that trigger the transfer of control between two parsers. Decoding operations on transitions are represented with coloured shapes.

4.2 Extracting and tracking sanitiser sequences

The starting point of JSPChecker analysis is with the sanitisers. JSPChecker starts by following inter-procedural def-use chains and aliases, available in SOOT/JSA, from each sanitiser s to output statements o, keeping track of other sanitisers encountered along the way. This process results in a set of sanitiser sequences starting in a sanitiser, ending in an output statement, and containing an arbitrary number of sanitisers in between: $SS = \{(s_1, \ldots, o_1), \ldots, (s_m, \ldots, o_n)\}$.

Each sanitiser sequence $ss \in SS$ ends in an output statement o that prints sanitised values. Hence, each $ss \in SS$ is a potential context-sensitive XSS flaw that must be analysed to check that the output context matches the sanitiser sequence that is applied to o.

In order to match sanitiser sequences to the output context, however, JSPChecker must be able to track the exact place in the HTML page where the sanitised output value was printed. JSPChecker achieves this tracking by mapping sanitiser sequences to unique placeholders that can be retrieved from the HTML page. Consider the following example:

```
1  String link = "location.replace(\"" +
      Encoder.jsEncode(request.
      getParameter("name")) + "\");";
2  out.println(link);
```

In this case, JSPChecker would first identify that the `jsEncode` sanitiser at line 1 and the output statement at line 2 are in def-use relation, yielding the following sanitiser sequence: $(jsEncode, print)$. Then, JSPChecker would map this sequence to a unique placeholder, yielding: $(jsEncode, print) \rightarrow ph1$.

JSPChecker supports nested sanitisers in a similar manner. Consider the following example:

```
1  String link = "location.replace(\"" +
       Encoder.jsEncode(request.
       getParameter("name")) + "\");";
2  link = "<a href='javascript:" +
       URLEncoder.urlEncode(link) + "'>Go
       to file</a>";
3  out.println(link);
```

In this case, JSPChecker would start by replacing the output of the `jsEncode` sanitiser at line 1 with $ph1$. Then, following def-use chains in SOOT/JSA, JSPChecker would detect that $ph1$ is further sanitised with the `urlEncode` sanitiser at line 2, before being printed at line 3, yielding: $(jsEncode, urlEncode, print) \rightarrow ph1$

In the following section, we detail the way JSPChecker uses string automata to approximate the HTML output of JSP pages and how it uses placeholders to keep track of sanitised values in the generated HTML pages.

4.3 Building web page approximations

Given a sanitiser sequence $ss \in SS$, its output statement o and its associated placeholder $\mathcal{PH}$, JSPChecker uses JSA to generate string automata that will later be used to generate HTML pages and, ultimately, check that the sanitiser sequence matches the output context. To enable this check, however, JSPChecker must ensure that o appears and can be easily identified in the generated HTML pages. JSPChecker achieves this goal using placeholders.

In general, a JSP page can theoretically produce an infinity of HTML pages, not all of which contain o. Algorithm 1 shows how JSPChecker uses placeholders to build string automata that are guaranteed to generate an output value of interest.

Algorithm 1 High-level HTML page generation algorithm

```
Input: ($\mathcal{CFG} = (\mathcal{V}, \mathcal{E})$, $\mathcal{A} : v \rightarrow DFA_v$, $\mathcal{SS} : (s_1, \ldots, s_n, o)$, $\mathcal{PH}$)
Output: HTML page approximations for a JSP
  function GenerateHtmlForJsp
      $\mathcal{A}(s_1) = \mathcal{PH}$
      $Start = \mathcal{V}_{start}$, $End = \mathcal{V}_{end}$
      $P_{Start}$ = SimpleSemiPaths($Start$, $o$, $(\mathcal{V}, \mathcal{E})$)
      $P_{End}$ = SimpleSemiPaths($o$, $End$, $(\mathcal{V}, \mathcal{E})$)
      $Pages = \emptyset$
      for each $p_{start} \in P_{Start}$ and $p_{end} \in P_{End}$ do
          $Path = p_{start} \cup p_{end}$
          $Pages = Pages \cup$ PathCoverage($Path$, $\mathcal{A}$, $o$)
      end for
      return $Pages$
  end function
```

Algorithm 1 receives as parameters the control-flow graph $\mathcal{CFG} = (\mathcal{V}, \mathcal{E})$ of a JSP, as produced by SOOT, a mapping from vertex $v \in \mathcal{V}$ to Deterministic Finite Automata (DFA), as produced by JSA [9], a sanitiser sequence $\mathcal{SS} : (s_1, \ldots, s_n, o)$, and its associated placeholder $\mathcal{PH}$.

The algorithm starts by replacing the automaton associated with s_1 with an automaton that recognises only the placeholder $\mathcal{PH}$. In the next step, JSPChecker identifies the entry and exit points of the JSP page in the control-flow graph. Then, JSPChecker attempts to build the minimal set of simple paths (paths without cycles) from the entry node to the exit node that go through o and that include every node reachable from o in the control-flow graph. JSPChecker achieves this goal by first building all simple semi-paths from the entry node to o and all simple semi-paths from o to the exit node. The call to SimpleSemiPaths performs this operation using a modified depth-first search algorithm that is not illustrated here.

JSPChecker then loops over all semi-paths and pairs them deterministically until all semi-paths have been paired at least once. JSPChecker then generates HTML pages for each simple path by calling a PathCoverage algorithm, and finally returns the set of HTML pages.

In the next section, we introduce the three path coverage algorithms we developed and tested in this study.

4.4 Path coverage strategies

Given an execution path $P = (entry, \ldots, o, \ldots, exit)$, from the entry to the exit of a JSP page that goes through a given output statement o, the task of path coverage algorithms is to generate HTML pages for that execution path. For this study, we designed and implemented three strategies to achieve this goal: *Exhaustive Path Coverage*, *Minimal Path Coverage* and *Shortest Path Coverage*.

Algorithm 2 Exhaustive path coverage algorithm

```
Input: ($P = (entry, \ldots, o, \ldots, exit)$, $\mathcal{A} : v \rightarrow DFA_v$, $\mathcal{PH}$)
Output: HTML page approximations for a path
  function ExhaustivePathCoverage
      $Pages = \emptyset$
      $DFA = \emptyset$
      for each $v$ in $P$ do
          if $v$ is a print statement then
              $DFA = DFA \circ \mathcal{A}(v)$
          end if
      end for
      $DFA = DFA \cap (.* \circ \mathcal{PH} \circ .*)$
      for each $simplePath$ in SimplePaths($DFA$) do
          $Pages = Pages \cup$ GetString($simplePath$)
      end for
      return $Pages$
  end function
```

Exhaustive Path Coverage: This coverage is the most complete strategy and is presented in Algorithm 2. The *Exhaustive Path Coverage* algorithm expects as parameters an execution path $P = (entry, \ldots, o, \ldots, exit)$, a mapping from vertex $v \in \mathcal{V}$ to DFA, and a placeholder $\mathcal{PH}$.

The algorithm starts by identifying every print statement on the path P and concatenating corresponding automata into DFA. Then, it intersects DFA with the $.*\mathcal{PH}.*$ automaton so that paths in DFA that do not print $\mathcal{PH}$ are removed. Finally, the algorithm loops over all the simple paths from the start state to the stop state of DFA that are returned by the call to SimplePaths and calls GetString to generate an HTML page for each path.

Depending on the complexity of the original JSP page, the exhaustive path coverage strategy can be very costly in practice. To address this problem, we designed the *Minimal Path Coverage* algorithm.

Minimal path coverage: This coverage was designed to circumvent the practical limitations of the *Exhaustive Path Coverage* strategy while maintaining good context-sensitive XSS flaws detection power. Algorithm 3 details the *Minimal Path Coverage* strategy.

Algorithm 3 Minimal path coverage algorithm

```
Input: $(P = (entry, \ldots, o, \ldots, exit)$, $\mathcal{A} : v \rightarrow DFA_v$, $\mathcal{PH})$
Output: HTML page approximations for a path
  function MinimalPathCoverage
      $Pages = \emptyset$
      $DFA = \emptyset$
      for each $v$ in $P$ do
          if $v$ is a print statement then
              $DFA = DFA \circ \textsc{MergeEdges}(\mathcal{A}(v))$
          end if
      end for
      $DFA = DFA \cap (.* \circ \mathcal{PH} \circ .*)$
      for each $path$ in $\textsc{CoverAllEdges}(DFA)$ do
          $Pages = Pages \cup \textsc{GetString}(path)$
      end for
      return $Pages$
  end function
```

Context scope	Special character classes
HTML	< > & " ' = ! - \ / [:white:]
CSS	" ' ( ) [ ] : ; < > = \ ! [:white:]
JavaScript	" ' \
URI	: ; , /

Table 1: Special characters that have the potential to induce a change of output context in different languages

20 syntax-equivalency character classes				
'<'	'\"'	'-'	'['	':'
'>'	'\\'	'/'	']'	';'
'&'	'='	'{'	'('	All white characters
"''	'!'	'}'	')'	All other characters

Table 2: Character equivalence classes

The main intuition behind the *Minimal Path Coverage* strategy is that only certain characters have the potential to induce a change of output context. Table 1 shows those special characters that have the potential to induce a change of output context in HTML, CSS, JavaScript, and URI. The first step of the *Minimal Path Coverage* algorithm is thus to reduce the number of edges in $\mathcal{A}(v)$ by performing a projection of its alphabet Σ to a reduced alphabet Σ' where all non-special characters are merged in one equivalence class. Table 2 shows the equivalence classes that are included in Σ'. The call to MergeEdges in Algorithm 3 performs the projection, and adjusts the edges of $\mathcal{A}(v)$ accordingly.

Apart from reducing the size of the alphabet, Algorithm 3 also applies another heuristic. Instead of generating one HTML page per simple path in DFA, the call to CoverAllEdges guarantees only that each edge in DFA is traversed at least once. The intuition behind this heuristic comes from our observations of open-source and industrial applications. From our experience, and as evidenced by our empirical results, it appears that this strategy does not alter the context-sensitive XSS flaw-detection power of JSPChecker compared to the exhaustive strategy. Following these observations, we designed another, even simpler path coverage algorithm called *Shortest Path Coverage*.

Algorithm 4 Shortest path coverage algorithm

```
Input: $(P = (entry, \ldots, o, \ldots, exit)$, $\mathcal{A} : v \rightarrow DFA_v$, $\mathcal{PH})$
Output: HTML page approximations for a path
  function MinimumPathCoverage
      $Pages = \emptyset$
      $DFA = \emptyset$
      for each $v$ in $P$ do
          if $v == o$ then
              $DFA = DFA \circ \textsc{MergeEdges}(\mathcal{A}(v))$
          else if $v$ is a print statement then
              $DFA = DFA \circ \textsc{ShortestMatch}(\mathcal{A}(v))$
          end if
      end for
      $DFA = DFA \cap (.* \circ \mathcal{PH} \circ .*)$
      for each $path$ in $\textsc{CoverAllEdges}(DFA)$ do
          $Pages = Pages \cup \textsc{GetString}(path)$
      end for
      return $Pages$
  end function
```

Shortest Path Coverage: The *Shortest Path Coverage* algorithm is detailed in Algorithm 4. It builds on the intuitions behind the *Minimal Path Coverage* algorithm and implements even more aggressive heuristics. This algorithm simply generates and concatenates the shortest match of every automaton $\mathcal{A}(v)$. An exception is made for $\mathcal{A}(o)$, the automaton of the initial output statement, for which this algorithm uses the same strategy as the *Minimal Path Coverage* algorithm. As shown in Section 6, this strategy works surprisingly well in practice.

4.5 Parsing HTML outputs

Each HTML approximation that is produced by a path coverage algorithm is then parsed by a suite of fault-tolerant parsers to determine the output context of sanitised values, represented as placeholders. These parsers simulate a web browser, and support transductions between HTML, JavaScript, CSS and URI snippets. In the context of our approach, two specific situations require special handling by the parsers: transfer of control to a sub-parser and context-switching derivations.

Transfer of control to a sub-parser occurs when the current parser encounters a snippet of code in a different language. For example, the value following an `href` attribute in HTML must be parsed by a URI parser. In web browsers, transfer of control between parsers is usually preceded by a transducing step, where the original code is transformed before the control is transferred. For example, the value following an `href` attribute will be HTML-entity-decoded before it is parsed by the URI parser. JSPChecker reproduces the transduction steps that happen in the browser. See Figure 1 for an overview of parser interactions and their associated decoding routines in JSPChecker.

Context-switching derivations occur when the parser derives a production rule that introduces a new security-sensitive context. To keep track of the current context, parsers in JSPChecker perform a simple syntax-directed translation where a new context is pushed on the beginning of a context-switching derivation and popped when the derivation is complete.

4.6 Detecting context-sensitive XSS flaws

Whenever a parser encounters a placeholder, it checks that the current output context sequence matches the sequence of sanitisers represented by the placeholder. If the set of safely sanitised output contexts includes the current output context, sanitisation is correct. Otherwise, a flaw is reported.

One situation, however, requires deeper investigation. When the set of safely sanitised output contexts includes only a suffix of the current output context, further analysis is needed. Consider the example in Listing 3.

```
1  foo() {
2     String link = "location.replace(\""
          + request.getParameter("link") +
           "\");";
3     printLink(link);
4  }
5
6  printLink(String link) {
7     link = "<a href='javascript:" +
          URLEncoder.urlEncode(link) + "'>
           Go to file</a>";
8     out.println(link);
9  }
```

Listing 3: The urlEncode sanitises only a suffix of the (JavaScript, URI) context.

Since JSPChecker tracks values from sanitisers to output statements, it would track only the output of `urlEncode` at line 7 to the `println` statement at line 8, yielding the sanitiser sequence: (*urlEncode*). The (*urlEncode*) sanitiser sequence can safely sanitise the (URI) output context only, which is a suffix of the actual output context ($JavaScript, URI$).

In this case, further analysis is needed to check whether a path exists from a source to `urlEncode` that does not go through a JavaScript sanitiser. JSPChecker uses a state-of-the-art, on-demand, backward taint analysis, to perform such checks.

5. EXPERIMENTAL SETUP

Our current experimental setup is based on the Java String Analyser [9] and the SOOT static analysis framework [19]. While we focused our analysis on JSP files, the techniques that are presented are not tied to this specific technology.

Our experimental setup currently uses the Tomcat compiler [1] to translate JSP files into Java files. In order to speedup computations, we also use Java dependency analysis (JDeps) [3] to identify dependencies of JSPs. Compiled JSPs and their dependencies are then analysed by SOOT/JSA.

JSPs are elementary units of our analysis. If multiple JSPs are to be analysed, JSPChecker analyses them in separate processes using the master-slave paradigm. Multi-process parallelisation is needed because SOOT is not thread-safe.

5.1 JSA configuration

By default, JSA generates string automata that overapproximate the set of strings that can be produced at a given point in a program. Hence, arguments and return values of external methods as well as arguments to public methods and public fields are converted to `.*` automata. In practice, the imprecision introduced by the use of `.*` automata often quickly spread through the whole document, making the analysis useless. To circumvent this problem, JSPChecker replaces public and external values with automata representing with the empty string.

Empty automata are also a major source of imprecision in JSPChecker because concatenation of the empty automaton with any other automaton always results in the empty automaton. Empty automata can be produced in cases where the string value is `null`. JSPChecker replaces empty automata with automata that recognise the empty string.

5.2 Parser configuration

Our current experimental setup uses an extended version of the Jsoup HTML parser [4]. The parser was extended to keep track of HTML contexts, and delegate analysis of embedded languages (CSS, URI and JavaScript) to appropriate sub-parsers, as detailed in Figure 1.

For CSS parsing, JSPChecker uses an extended version of the CSS Parser [2] that keeps track of CSS contexts, and delegates URI parsing to the URI lexer.

JSPChecker currently uses a simple lexer to process URIs. For the purpose of identifying context-sensitive XSS flaws, it is indeed sufficient to detect whether the URI starts with the `javascript:` or `data:` keyword. To our knowledge, other common protocols (e.g. `ftp`, `mailto`, etc.) cannot induce a transition from the URI context to either the CSS, HTML or JavaScript context (see Figure 1) and are therefore irrelevant to our analysis.

JSPChecker currently has minimal support for JavaScript. It uses a lexer that supports three contexts: double-quoted string, single-quoted string, and JavaScript code. Analysing the JavaScript code itself is beyond the scope of this paper and left as future work.

The inputs to our parsers and lexers are static approximations of JSPs, and they might contain syntactic errors. The errors can be caused by the approximation process itself or by programming errors in the original document. While it is possible to refine our HTML approximation process to eliminate some errors, it is much more difficult to correct hard-coded errors in legacy applications. Unfortunately, syntax errors are quite common in legacy web applications because server-side technologies that are typically used to generate HTML pages (e.g. ASP, PHP, JSP, and others) do not enforce syntactic correctness of the produced HTML output.

Historically, browser vendors have circumvented this limitation by developing fault-tolerant parsers that correct or ignore faulty segments of an HTML document. As a consequence, HTML pages produced by legacy web applications are often syntactically invalid. Any approach that deals with legacy web application must be fault tolerant. JSPChecker handles syntactic errors by using fault-tolerant parsers that can recover from common syntactic errors.

6. EVALUATION

We evaluated our approach on five open source JEE projects that use the JSP technology. Table 3 shows characteristics of analysed applications. In order to highlight the prevalence of context-sensitive XSS flaws, we also ran a state-of-the-art, static taint analysis tool, built on top of the Parfait framework [10], that can detect XSS flaws due to *missing* sanitisation. Table 4 shows results from Parfait and JSPChecker.

	Clinportal	**Hipergate**	**iTrust CS427**	**NENU Contest service**	**Free WIS portal**
Number of JSP files	97	760	207	40	8
Number of JSP files with sanitisers	7	75	181	26	1
Lines of JSP code	25 136	93 246	27 700	4 592	1 004
Lines of Java code	73 137	137 609	54 170	7 296	1 548

Table 3: Characteristics of analysed applications

	Clinportal	**Hipergate**	**iTrust CS427**	**NENU Contest service**	**Free WIS portal**
True positives	19	334	230	6	0
False positives	3	75	0	0	3
Execution time (mm:ss)	1:39	1:14	5:15	0:59	0:07

Table 4: Parfait taint analysis results

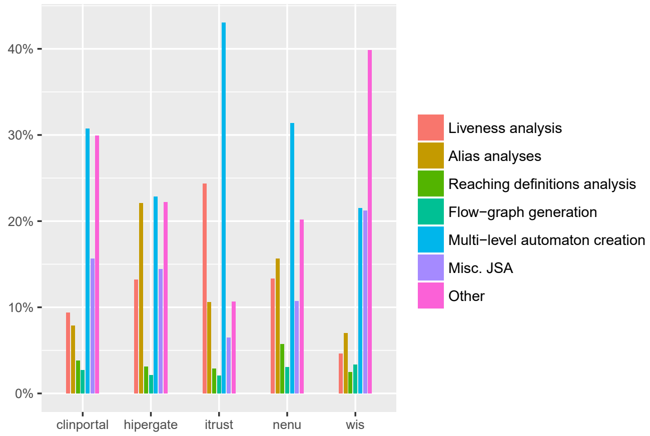

Figure 2: Analysis time breakdown for shortest path coverage analysis on benchmark applications.

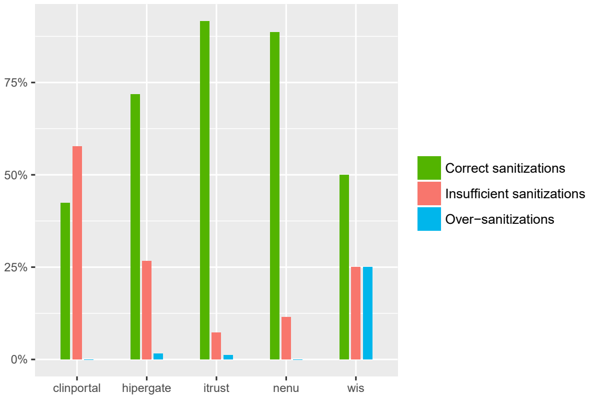

Figure 3: Proportions of correct, insufficient and over-sanitisations that were detected by JSPChecker in benchmark applications.

True and false positives were determined through manually inspection. We contacted the developers of each investigated applications to share our findings.

As shown in Table 4 and Table 5, context-sensitive XSS flaws are about as common as missing sanitisation XSS flaws, highlighting the relevance of JSPChecker analysis.

Results also show, however, that the runtime of JSPChecker is typically much higher than of a traditional static taint analysis.

As Figure 2 shows, apart from WIS portal, building of the string automata by JSA always consumes the most time. In the figure, all the code we implemented in JSPChecker falls into the "Other" category. Depending on the application, 10% to 40% of the time is spent in JSPChecker code.

Results in Table 5 highlight how, in the context of the investigated applications, the *Shortest Path Coverage* strategy surprisingly outperforms the *Minimal Path Coverage* and *Exhaustive Path Coverage* strategies both in terms of runtime and precision. Indeed, the *Shortest Path Coverage* strategy detects all the context-sensitive sanitisation flaws that are detected by the other two approaches while producing fewer false positives.

False positives were either due to unknown strings (e.g. a string retrieved from a database) or syntactically invalid HTML constructs that caused our parsers to report wrong output contexts. Fixing the values of unknown strings and enhancing the error-recovery mechanisms in our parsers would thus help reduce the number of false positives reported by JSPChecker.

Table 5 also reports *over-sanitisations* that were detected by JSPChecker. Over-sanitisations occur when sanitisers safely but incorrectly encode characters for a given output context. For example, encoding the `<` character to `%3C` in a URL is safe and correct. In an HTML context however, such encoding would be considered safe, but incorrect, because the browser would not decode it back to `<` at rendering time. Over-sanitisation typically leads to loss of functionality and does not induce security vulnerabilities. Figure 3 shows the proportions of correct, insufficient and over-sanitisations that were reported by JSPChecker on different systems.

6.1 Example context-sensitive XSS flaws

In this section, we show concrete examples of context-sensitive XSS flaws that were reported by JSPChecker.

		Clinportal	Hipergate	iTrust CS427	NENU Contest service	Free WIS portal
SPC	True positives	30	51	168	25	1
	False positives	0	2	0	0	0
	Over-sanitisation	0	3	27	0	1
	Execution time (mm:ss)	21:35	70:54	133:23	13:03	1:04
MPC	True positives	30	51	168	25	1
	False positives	0	7	0	0	0
	Over-sanitisation	0	3	27	0	1
	Execution time (mm:ss)	22:04	74:47	137:23	14:54	1:24
EPC	True positives	29	44	167	24	1
	False positives	0	8	0	0	0
	Over-sanitisation	0	3	27	0	1
	Execution time (mm:ss)	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	1:42

Table 5: JSPChecker context-sensitive analysis. Results are grouped in three sections: SPC (Shortest Path Coverage), MPC (Minimal Path Coverage) and EPC (Exhaustive Path Coverage). TIMEOUT indicates that the timeout of three hours was reached.

```
1  String participantId=escapeHtml(request
       .getAttribute("participantId"));
2  ... // 53 lines of code
3  out.write("\t<script>\t\r\n");
4  ... // 150 lines of code
5  out.print(participantId);
6  ... // 428 lines of code
7  out.write("\t</script>\r\n");
```

Listing 4: Example from Clinportal where HTML-escaped value is printed inside JavaScript code.

Listing 4 shows an example from Clinportal where an input value is HTML-escaped and printed in JavaScript code.

At line 1, an input value is HTML-escaped using the `escapeHtml` sanitiser. At line 3, the JSP switches context from HTML to JavaScript with the introduction of the `<script>` tag. At line 5, the HTML-escaped input is printed and the JavaScript block is closed at line 7.

An attacker can therefore inject a JavaScript payload like:

```
document.location = String.fromCharCode
    (65,66,67)
```

to redirect the victim to a malicious website. This attack is possible because the HTML sanitiser at line 1 is does not escape the `(`, `)`, and `.` characters and is therefore insufficient to sanitise values for the JavaScript context.

Listing 4 also illustrates how manual placement of context-sensitive sanitisers can be difficult and error-prone due to the amount of code that needs to be reviewed. Indeed, observe that 631 lines were omitted in Listing 4 to make the example readable.

```
1  out.write("<script>\n");
2  out.write("function setURL() {\n");
3  out.write("document.location = \"
       company_listing.jsp?selected=");
4  ...
5  out.write("&find=");
6  out.print(URLEncode(request.
       getParameter("find")));
7  ...
8  out.write("</script>\n");
```

Listing 5: Example from Hipergate where URL-encoded value is printed inside JavaScript code.

Listing 5 shows an example from Hipergate where an input value is URL-encoded before being printed in JavaScript code.

At line 1, the JSP switches from the HTML context to the JavaScript context. At line 2, the `setURL` function is defined. At line 3, the `document.location` attribute is assigned a URL that is dynamically built in the JSP. Assigning a URL to `document.location` in JavaScript effectively redirects the browser to the assigned URL. At line 6, the `find` parameter is URL-encoded and inserted in the dynamically built URL. The expected outcome is thus that a call to `setURL` will redirect the user to a URL that was built during the rendering of the JSP.

In this application, the `URLEncode` routine does not encode the \ character. As a consequence, an attacker can inject an hex-encoded payload through the `find` parameter at line 6 to bypass the URL encoder and override any of the remaining URL parameters.

The snippet in Listing 5 illustrates another reason why manual placement of context-sensitive sanitisers is difficult. Looking at line 6, where the `find` parameter is printed, it is easy to be misled by the fact that the parameter is printed in a URL string. The difficulty stems from the fact that the developer has to reason about the sequence of contexts in which the URL string will be evaluated and sanitise it accordingly. In this case, the `find` parameter should have been sanitised with URLEncode first and then with a JavaScript sanitiser.

```
1  out.write("<a href=\"javascript:
       removeRep('");
2  out.print(escapeHtml(p.getMID()));
3  out.write("')\">Remove</a>\n");
```

Listing 6: Example from iTrust where HTML-escaped value is printed inside a JavaScript URL.

Listing 6 shows an example from iTrust where an input value is HTML-escaped before being printed in a JavaScript URL.

At line 1, an HTML anchor is created where the `href` parameter is set to a JavaScript URL. At line 2, a value is HTML-escaped and printed in the JavaScript URL. The anchor is closed at line 3.

In this application, the escapeHtml does not encode the % character. As a consequence, an attacker can pass a URL-encoded payload to exploit the fact that the browser will URL decode the value of the href attribute before interpreting it as JavaScript code.

For example, injecting the %27)%3Balert(%27Hacked payload in the JSP would result in the following HTML snippet:

```
<a href="javascript:removeRep('%27)%3
    Balert(%27Hacked')">Remove</a>
```

After URL decoding, the browser would generate the following snippet:

```
<a href="javascript:removeRep('');alert
    ('Hacked')">Remove</a>
```

Because the javascript: scheme is used, the resulting URI would then be interpreted as JavaScript code, resulting in a successful attack.

7. SOUNDNESS TRADE-OFFS

In [23], the authors present a manifesto in defense of *soundy* static analysis. Quoting the authors: "A soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability. In their manifesto, the authors also issue a call to the community to clearly identify unsoundness in static analyses so that others can understand and reproduce the results.

In this section, we discuss the soundness trade-offs we made in JSPChecker. Designed for the analysis of large legacy web applications, JSPChecker has been successfully used internally to analyse very large (millions of LOC) legacy JEE applications.

In Algorithm 1, we showed how, given an output statement of interest o, JSPChecker builds execution paths from the entry point of a JSP to the exit point by assembling semi-paths from the entry point to o and from o to the exit point.

A sound strategy would have been to build and analyse one over-approximate automaton per JSP. While theoretically sound, this approach is highly impractical on real-world applications. Indeed, our experiments revealed that JSA seems to hit a performance bottleneck when the string automata it builds reaches a certain level of complexity, and analysis timed out on most JSPs. In [9], the authors mention that translation from a multi-level automaton, internally used in JSA, to a DFA is worst-case doubly exponential.

We also detailed, in Algorithms 2, 3, and 4, path coverage strategies with decreasing level of soundness. While our primary intuition was that decreasing levels of soundness would speed up analysis, increase precision, and lower recall, results proved our intuitions wrong. In Table 5, results show that shortest path coverage, the most unsound of the three coverage strategies, not only runs faster, it also achieves the same precision and recall as the two other strategies.

While we cannot formally explain this behaviour, it seems that most JSPs are developed in such a way that the syntactic structure of HTML documents it produces is always more or less equivalent. In other words, it appears that JSPs usually produce fixed canvases with variable content. Because JSPChecker only reasons about the syntactic structure of HTML documents, generating more HTML approximations does not increase the number of reported flaws.

Finally, the parsers that are used internally by JSPChecker might not be faithful to that of the browser. Furthermore, it is well known that different browsers or different versions of the same browser behave differently on the same inputs. As a consequence, the reported true positives, false positives and over-sanitisation rates may vary in practice. Hooking JSPChecker into the parsers of various browsers would eliminate this limitation at the expense of a significant engineering effort.

8. RELATED WORK

8.1 Context-sensitive XSS sanitisation

This section presents previous approaches to detect and prevent context-sensitive cross-site scripting flaws.

In [32], the authors present a technology called SCRIPTGARD that focuses on automatic context-sensitive sanitisation for large-scale legacy web applications. SCRIPTGARD is a dynamic analysis approach that works in two phases: training and runtime monitoring. In the training phase, SCRIPTGARD builds a *sanitisation cache* that is a map from execution paths to correct sanitiser sequences. During the runtime monitoring phase, if SCRIPTGARD encounters a path that is already in its sanitisation cache, it applies the correct sanitiser sequence. Otherwise, the choice is left to the user to either block the request or log the path for later analysis. In that regard, the performance of SCRIPTGARD depends on the number of execution paths it can cover during its training phase.

Another approach to context-sensitive sanitisation, called CSAS (Context-sensitive auto-sanitisation engine), is presented in [31]. CSAS uses a templating language to produce HTML outputs. Given a template script, CSAS first tries to determine the context of output values using type inference. If successful, CSAS applies the correct sequence of sanitisers immediately. Otherwise, CSAS inserts checks in the application code that detect the runtime context and sanitise outputs accordingly. While this approach is very promising for applications that are developed from scratch using a templating language, the manual effort required to adapt legacy JEE applications to use CSAS is prohibitively high. Other framework-based approaches to prevent SQLi and XSS have been presented in [29, 21, 25].

On a similar line of thought, in [20], the authors also present an automated context-sensitive sanitisation approach. Given a user-specified security policy for each source and sink in an application, the presented approach provides optimal placement of sanitisers in the code. While the approach is highly promising from a sanitiser placement perspective, it fails to address the main cause for context-sensitive XSS flaws: manually determining the context of an output value is difficult and error prone.

A study of XSS sanitisation is presented in [37]. Common causes for context-sensitive XSS bugs, such as context-insensitive sanitisers, failure to handle nested contexts or browser transductions are presented and described.

8.2 String analysis for vulnerability detection

JSPChecker uses string analysis to produce static approximations of dynamically generated web pages that are further analysed to detect context-sensitive XSS flaws. This section presents previous approaches that also make use of string analysis to detect vulnerabilities in web applications.

Approach	Error tolerant?	Purely static?	No code modification?	Context-sensitive policy?
JSPChecker	✓	✓	✓	✓
SCRIPTGARD [32]	✓	×	✓	✓
CSAS [31]	×	×	×	✓
Wassermann et al. [36]	×	✓	✓	×
Saner [7]	×	×	✓	×

Table 6: Comparing of JSPChecker features to existing approaches

An approach to detect SQLi flaws is presented in [13]. The Java String Analyzer (JSA) is first used to model SQL queries as string automata. Then, context-free reachability [26] is used to find the type environment of each path in the query automaton and to perform type checking to detect invalid queries.

In [36], the authors present an approach, based on the string analyser in [27] to detect XSS flaws in PHP applications. Similarly to JSPChecker, their approach first builds a string automaton for each output statement that can print potentially tainted data. These output string automata are then intersected with an automaton representing a security policy, and a flaw is reported if the intersection is not empty. Unfortunately, restricting the security policy to the realm of regular languages prevents proper detection of context-sensitive XSS flaws.

In [7], the authors present Saner, a tool to analyse custom sanitisation routines in PHP applications. The goal of Saner is to identify erroneous custom sanitisation routines that can lead to XSS and SQLi flaws. Similarly to [36], their approach also produces string automata for output statements that print tainted data, and verify sanitiser correctness by intersecting the resulting automata with a regular expression representing a security policy. Whenever Saner identifies a potentially vulnerable sanitisation routine, it dynamically tests it using a set of predefined malicious inputs. Overall, Saner suffers from the same limitations as [36] due to the fact that security policies must be defined with regular expressions.

In [34], the authors present an approach to discover and verify sanitisers. It first translates the program into monadic second-order logic. Then, any method that takes a string as an argument and returns a string is considered a potential sanitiser. Finally, a method is considered a valid sanitiser if it never returns strings belonging to a predefined blacklist encoded as a regular expression.

While the three previous approaches try to verify sanitiser correctness with respect to the outputs of a PHP script, BEK [15] is a tool that verifies sanitisers themselves. More precisely, BEK verifies the commutativity, idempotence and mutual equivalency of sanitisers in a system. This work is orthogonal and complementary to ours.

8.3 String analysis for syntax validation

String analysis was also used to validate the syntactic correctness of dynamically generated documents.

In [18], the authors present an approach to verify that JSPs always produce well-formed XML documents. Using JSA, their approach builds a grammar that approximates the output of JSPs. It then verifies that resulting grammar produces tag-balanced XML documents only. This approach does not support CSS, HTML, JavaScript or URI grammars.

A subsequent study [28] extends this work to support the syntactic validation of dynamically generated HTML documents against a DTD (Document Type Definition). However, their work supports only a subset of HTML and has no support for JavaScript, CSS or URI syntactic validation.

8.4 Context-sensitive XSS in legacy applications

JSPChecker was designed with the goal to detect context-sensitive XSS flaws in large legacy JEE applications. As mentioned in Section 1, large legacy applications are often characterised by huge maintenance costs and lack of extensive tests and documentation. Purely static approaches (no tests needed) that do not require code modifications (no maintenance overhead) are thus preferred in this context. Because legacy web codebases are also notorious for producing syntactically invalid HTML pages [30, 27, 28], analysis must be able to recover from common syntactic errors. Furthermore, context-sensitive XSS flaws stem from the fact that the composition of languages that form a web page results in a non-context-free language. Any approach that aims at detecting context-sensitive cross-site scripting flaws must track the context of output values. Table 6 summarises the main differences between JSPChecker and existing approaches, with respect to these aspects.

9. CONCLUSION AND FUTURE WORK

We have presented JSPChecker, a purely static, error-tolerant tool to detect context-sensitive XSS flaws in legacy web applications. Our technique is based on data-flow analysis, string analysis and syntax-directed translation, and it detects context-sensitive XSS flaws by matching sanitiser sequences to output contexts. We showed how our technique was able to identify several context-sensitive XSS flaws in five real world applications with a precision ranging from 96% to 100%.

As future work, we plan to extend JSPChecker to relieve users of the burden of identifying sanitisers and mapping them to safe output contexts. String analysis-based approaches to verify sanitisers have already been presented in [34, 15] and could be reused to achieve this goal.

10. REFERENCES

[1] Apache Tomcat. http://tomcat.apache.org. Accessed: 25-05-2016.

[2] CSS Parser. http://cssparser.sourceforge.net/. Accessed: 25-05-2016.

[3] Java Dependency Analysis Tool. https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool. Accessed: 25-05-2016.

[4] jsoup: Java HTML Parser. https://jsoup.org. Accessed: 25-05-2016.

[5] Open Web Application Security Project. https://www.owasp.org/. Accessed: 25-05-2016.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices (PLDI '14)*, 49(6):259–269, 2014.

[7] 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 *Symposium on Security and Privacy (S&P'08)*, pages 387–401. IEEE, 2008.

[8] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Conference on the Future of Software Engineering '00*, pages 73–87. ACM, 2000.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Static Analysis Symposium (SAS '03)*, pages 1–18. Springer, 2003. Available from `http://www.brics.dk/JSA/`.

[10] C. Cifuentes, N. Keynes, L. Li, N. Hawes, M. Valdiviezo, A. Browne, J. Zimmermann, A. Craik, D. Teoh, and C. Hoermann. Static Deep Error Checking in Large System Applications Using Parfait. In *European conference on Foundations of Software Engineering (FSE '11)*, pages 432–435. ACM, September 2011.

[11] J. R. Cordy. Comprehending Reality-Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *International Workshop on Program Comprehension (IWPC '03)*, pages 196–205. IEEE, 2003.

[12] M. Feathers. *Working Effectively With Legacy Code.* Prentice Hall Professional, 2004.

[13] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *International Conference on Software Engineering (ICSE '04)*, pages 645–654. IEEE, 2004.

[14] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Annual Computer Security Applications Conference (ACSAC'05)*, pages 311–319. IEEE, 2005.

[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium '11*, pages 1–1, 2011.

[16] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on World Wide Web (WWW '04)*, pages 40–52. ACM, 2004.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Symposium on Security and Privacy (S&P'06)*, pages 263–268. IEEE, 2006.

[18] C. Kirkegaard and A. Moller. Static Analysis for Java Servlets and JSP. *BRICS Report Series*, 2006.

[19] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot Framework for Java Program Analysis: A Retrospective. 2011.

[20] B. Livshits and S. Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Symposium on Principles of Programming Languages (POPL '13)*, pages 385–398. ACM, 2013.

[21] B. Livshits and Ú. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Workshop on Programming Languages and Analysis for Security (PLAS '07)*, pages 95–104. ACM, 2007.

[22] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. *ACM SIGPLAN Notices (PLDI '09)*, pages 75–86, 2009.

[23] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[24] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium '13*, volume 2013, 2005.

[25] Z. Luo, T. Rezk, and M. Serrano. Automated Code Injection Prevention for Web Applications. In *Workshop on Theory of Security and Applications*, pages 186–204. Springer, 2011.

[26] D. Melski and T. Reps. *Interconvertbility of Set Constraints and Context-Free Language Reachability*, volume 32. ACM, 1997.

[27] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on World Wide Web (WWW '05)*, pages 432–441. ACM, 2005.

[28] A. Moller and M. Schwarz. *HTML Validation of Context-Free Languages*, pages 426–440. Springer, 2011.

[29] W. K. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *USENIX Security Symposium '09*, pages 283–298, 2009.

[30] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *International Conference on Software Engineering (ICSE '12)*, pages 277–287. IEEE, 2012.

[31] M. Samuel, P. Saxena, and D. Song. Context-sensitive Auto-sanitization in Web Templating Languages Using Type Qualifiers. In *Conference on Computer and Communications Security (CCS '11)*, pages 587–600. ACM, October 2011.

[32] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In *Conference on Computer and Communications Security (CCS '11)*, pages 601–614. ACM, October 2011.

[33] H. M. Sneed. Risks Involved in Reengineering Projects. In *Working Conference on Reverse Engineering (WCRE '99)*, pages 204–211. IEEE, 1999.

[34] T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic

Second-order Logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM SIGPLAN Notices (PLDI '09)*, pages 87–97, 2009.

[36] G. Wassermann and Z. Su. Static Setection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE '08)*, pages 171–180, May 2008.

[37] 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 (ESORICS 2011)*, pages 150–171. Springer, 2011.