Testing and Analysis of Web Applications using Page Models

Snigdha Athaiya
Indian Institute of Science
Bangalore, India
snigdha.athaiya@csa.iisc.ernet.in

Raghavan Komondoor
Indian Institute of Science
Bangalore, India
raghavan@csa.iisc.ernet.in

ABSTRACT

Web applications are difficult to analyze using code-based tools because data-flow and control-flow through the application occurs via both server-side code and client-side pages. Client-side pages are typically specified in a scripting language that is different from the main server-side language; moreover, the pages are generated dynamically from the scripts. To address these issues we propose a static-analysis approach that automatically constructs a "model" of each page in a given application. A page model is a code fragment in the same language as the server-side code, which faithfully over-approximates the possible elements of the page as well as the control-flows and data-flows due to these elements. The server-side code in conjunction with the page models then becomes a standard (non-web) program, thus amenable to analysis using standard code-based tools. We have implemented our approach in the context of J2EE applications. We demonstrate the versatility and usefulness of our approach by applying three standard analysis tools on the resultant programs from our approach: a concolic-execution based model checker (JPF), a dynamic fault localization tool (Zoltar), and a static slicer (Wala).

CCS CONCEPTS

• **Information systems → Web applications**; • **Software and its engineering → Automated static analysis**; **Dynamic analysis**; *Functionality*;

KEYWORDS

JSP; Web Applications; Static Analysis

ACM Reference format:
Snigdha Athaiya and Raghavan Komondoor. 2017. Testing and Analysis of Web Applications using Page Models. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092734

1 INTRODUCTION

Web applications belonging to fields such as banking, e-commerce, health management, etc., have widespread uses, and constitute the key focus for a large part of the software development community. Yet, the problem of automated code-based analysis of web applications remains a challenge, for the following key reasons: (1) Client-side pages and server-side code are both responsible for providing the overall functionality of the application, but are typically implemented using different languages and technologies. (2) Client-side pages are generated dynamically by server-side code based on the contents of server-side state, yet are responsible for key control-flow data-flow links within the application overall. In other words, client-side pages conceptually resemble self-modifying code, which is very hard to analyze. (3) User interactions need to be accounted for.

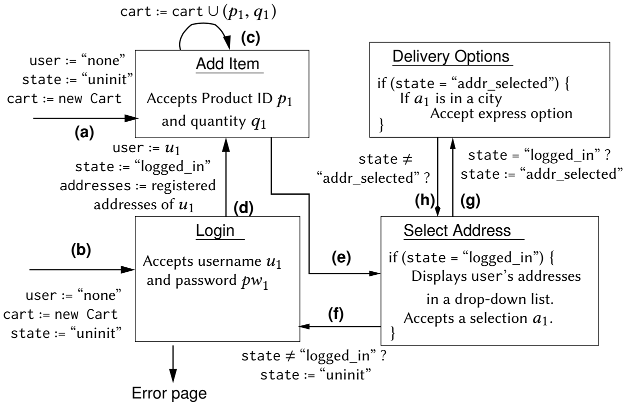

Figure 1: An illustrative web application

1.1 Motivating Example

Figure 1 shows schematically a simplified e-commerce web application that we use as a running example throughout this paper. Each rectangle represents a page, with its title (underlined) at the top, and with its main responsibility described in English after the title. Edges in the figure represent traversal paths between pages, either via clickable links or via form submissions.

Each "Accept" verb in a page description is shorthand for a textbox or a drop-down list. Subscripted symbols such as p_1, q_1, pw_1, etc., denote values that are passed by each page to the server in the form of *request parameters* whenever an edge out of the page is traversed.

The variables in typewriter font, namely, `user`, `state`, `addresses`, and `cart`, are *session attributes*. The server maintains a separate copy of these variables for each user's current session, with the lifetime of these variables being the entire duration of the session.

The annotations on some of the edges in the figure indicate the server-side *actions* that are performed when these edges are

ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092734

```
1  <c:choose>
2    <c:when test=“${state==‘logged_in’}”>
3      <c:forEach var=“addr” items=“${addresses}”>
4        <a href = “DeliveryOptions.jsp?address=${addr}”>
5           Select ${addr} </a>
6      </c:forEach>
7      <c:set var=“state” value=“addr_selected”/>
8    </c:when>
9    <c:otherwise>
10    <a href=“Login.jsp”> You are not logged in </a>
11    <c:set var=“state” value=“uninit”/>
12   </c:otherwise>
13 </c:choose>
```

Figure 2: SelectAddress.jsp

traversed; e.g., when the user traverses from the “Login” page to the “Add Item” page, the session attribute `user` gets set in the server to the user-name u_1 that the user entered in the Login page's user-name text box (u_1 gets passed to the server by the Login page as a request parameter).

Some of the edges have a *guard* on them; e.g., the link (g) from “Select Address” has the guard ‘`state` = “logged_in”’. What this means is that the form/link via which this edge is traversed is placed by the server in the “Select Address” page only if this guard evaluates to *true* on the server-side during the (dynamic) generation of this page.

In this paper, we address the problem of code-based (i.e., white-box), end-to-end analysis of web applications. For instance, a developer may want to use a *property checking* approach such as *concolic* execution [14] to check whether the application could ever violate the property that the address a_1 that is used in the “Delivery Options” page will always be a registered address of the currently logged-in user. Note that if a bug in the application can cause some other user's address to appear here, then it would be an erroneous outcome – the items could get shipped to a user who did not order the product unless the user happens to notice the mistake on the web page. Another developer may want to use the *static program slicing* [31] operation to determine the statements throughout the application's code that affect the address being passed to the “Delivery Options” page.

1.2 The Code Structure of Web Applications

Web applications typically use a traditional programming language to implement server-side business logic and to manage the server-side state. They also use HTML-based *page specifications* to specify the layout of client pages. Page specifications are especially useful when pages need to be *dynamic* in nature; i.e., the layout and functionality of the same page does not remain static across visits to the page, and needs to vary as per the server-side state. To make things even more complicated, the page specifications can, in general, also contain embedded non-layout operations that are meant to update server-side state.

For instance, say, our running example (which was summarized in Figure 1) is implemented using *J2EE* technology. In this framework, server-side logic would be implemented in Java, while the content of dynamic pages could be specified using the HTML-based scripting language JSP (Java Server Pages). Figure 2 depicts a part of the JSP page specification for the page titled “Select Address” in Figure 1. This JSP specification is processed at the server side whenever the user clicks the link (e) in Figure 1. We describe below informally the meaning of the JSP specification, and the steps taken at the server-side to process it.

First, the session attribute `state` is checked in Line 2 in the JSP specification to see if it has value ‘`logged_in`’. If this check passes, it indicates that the current user has logged in successfully previously. (When the user would have previously successfully logged in, then, as depicted next to link (d) in Figure 1, the server-side code would have set `state` to value ‘`logged_in`’ and would have also populated `addresses` with the addresses of the just logged-in user. For brevity, we have not shown this server-side code.) Reverting back to the page specification in Figure 2, the constructs within “${” and “}” are called *EL* expressions. These are used in JSP to refer to session attributes, and, to test conditions and to compute expressions over session attributes as well as JSP local variables. Next, the loop in Lines 3-6 emits dynamically a set of HTML links, with each link containing an assignment of the request parameter ‘address’ to one of the addresses in `addresses`. The idea is that the user who views the “Select Address” page can click the link that contains the address to which they want to schedule delivery. (In practice, a form with a drop-down list containing all the addresses would have been used in this scenario rather than a set of links.) Line 7 uses the JSP tag c:**set** to set the session attribute `state` to contain the value “addr_selected”. On other hand, if the check in Line 2 does not pass, lines 9-12 specify that a link to the “Login” page is required (instead of links to the “Delivery Options” page); in this case, Line 11 sets `state` to “uninit”. Finally, after all the updates to the session-attributes and generation of HTML corresponding to the “Select Address” page as mentioned above are completed at the server-side, the thus generated HTML page is sent to the user's browser.

1.3 Illustration of Challenges in Analysis

The example above illustrates a few features of pages and page specifications that pose challenges to any approach that aims to perform end-to-end whitebox analysis of web applications:

1.3.1 Dynamic Web Pages. Page specifications describe *dynamic* pages. As seen in Figure 2, the layout of the “Select Address” changes dramatically based on the value of the session attribute `state` at the time this page is generated on the server side.

1.3.2 Cross-Language Dataflows. Besides layout, page specifications also affect the intra-page, inter-page and client-server dataflows in a web application. For instance, from Lines 3-4 in Figure 2, it is clear that this page specification is responsible for picking up addresses that were earlier stored in the session attribute `addresses` by the server-side code, and for placing these addresses as request parameter values in the links in the HTML of the “Select Address” page. Subsequently, when the user clicks one of the links in this page, the address that is embedded as the request parameter value in the clicked link is sent back to the server-side. In other words, page specifications implicitly cause data flows from some parts of server-side code to other parts of server-side code via to-and-forth flows via the client.

In other words, to automatically perform analyses such as the ones mentioned at the end of Section 1.1, the control-flows and data-flows due to page specifications would need to be taken into account. It is also follows from this discussion that standard whitebox analysis tools like slicers and property checkers that target single-language programs cannot be used in an off-the-shelf manner on web applications.

1.4 Previous Approaches for Web Applications

The literature mentions several end-to-end analysis approaches that specifically target (multi language) web applications [3, 7–10, 12, 16, 19, 26]. However, even these approaches do not fully account for dataflows due to pages generated dynamically from page specifications. To illustrate this, we use the following (toy) JSP fragment:

```
1   <a href = "paymentsPage.jsp?stateP=${state}"></a>
2   <c:set var = "state" value = "payment_received" />
```

Say *P* is the server-side method that executes before the page specification above is processed. Note that the request parameter *stateP* in the link that is generated from the specification above will have the value that was placed in the session attribute `state` by *P*. After the page generated from this specification is sent to the client, say the user clicks the link, and say server-side method *Q* handles this click. Therefore, when *Q* retrieves the value of the request parameter *stateP*, it will get the value that was earlier placed in `state` *P*. Therefore, a slicing operation from this retrieval point in *Q* should pull in the code in *P* that assigns to `state`. The recent approach by Nguyen et al. [12] for cross-language static slicing does not identify this sequence of dataflows correctly, because this approach does not accommodate session attributes at all.

There is another challenge that is illustrated by the example above. Any direct references to `state` in *Q* are supposed to see the value placed into this session attribute during the processing of Line 2 in the page specification above, and *not* the value written by *P*. This is true even though Line 2 appears *after* the specification of the link that takes control to *Q*. It is not clear that any simple extension to Nguyen et al.'s approach to accommodate session attributes can handle this *inversion* of control-flow order correctly.

Other analysis approaches mentioned above such as that of Artzi et al. [3] perform precise forms of end-to-end property checking, even in the presence of session attributes. However, they do not represent dataflows explicitly, and hence cannot be extended readily to perform static analyses such static slicing.

1.5 Our Proposal: Page Models

Our objective in doing the work described in this paper is to enable a variety of end-to-end, whitebox analyses of web applications. Our key proposal to enable this is a novel notion of a *page model*, which is a code fragment in the same language as the server-side code. The page model of each page specification is generated automatically by our approach, and captures the underlying computational semantics of the page across all possible dynamic instances of this page. Subsequently, the server-side code in conjunction with the page models (which replace the original page specifications) can be treated as a standard program (in the language of the server side) for analysis purposes.

```
1  public class SelectAddress_jsp {
2   public void pageModel() {
3    if(state.equals("logged_in")) {
4      for(String addr:addresses)){
5        if(coinFlip()) {
6          address = addr;
7          state = "addr_selected";
8          DeliveryOptionsServlet.doGet(null, null);
9        }}}
10   else {
11     ... similar code for Login page ...
12   }}}
```

Figure 3: Simplified page model of the page "SelectAddress.jsp"

Figure 3 shows a simplified page model for the page specification SelectAddress.jsp that was depicted in Figure 2. The JSP page is converted into a Java class containing a "root" method *pageModel*, which receives and processes a request. The links are modelled as method calls to suitable targets; e.g., in Figure 3, Line 8 is a call to the "doGet" method of 'DeliveryOptionsServlet', which receives the request when the link that is generated due to Line 4 in Figure 2 is clicked. Note that the servlet that is supposed to receive each request can be obtained from the *deployment descriptor*, which is typically a mapping from URLs to servlet names.

In a web page, the user has a choice of which link to click. The page model encodes this choice mechanism using the boolean method "coinFlip()", which returns *true* or *false* non-deterministically.

Session attributes as well as request parameters are modeled as global variables in the page model. The former are denoted in Figure 3 using `typewriter` font. Request parameters are denoted using *italics* font – there is only one request parameter that is passed from this page, namely, *address*, to DeliveryOptionsServlet. Line 7 in Figure 2, which sets the session attribute `state`, becomes an assignment to the corresponding global variable `state` in Line 7 in Figure 3. Similarly, references to session attributes in EL-expressions are modeled as references to the corresponding global variables. The JSP control-flow constructs like **forEach** and **choose-when-otherwise** are translated faithfully into equivalent Java constructs in the page model.

A notable point is that the global variable `state` is updated in Line 7 in Figure 3, which is *before* the call in Line 8, even though in the page specification in Figure 2 the corresponding lines (i.e., Line 7 and Line 4) appear in the *opposite* order. This is to account properly for the inversion of control-flow that we had discussed in Section 1.4.

The page models of the pages in an application need to be integrated with the server-side code in order to yield a standard (single language) program that is behaviorally faithful to the original web application. To do this, our approach also applies certain code changes in the server side code. For instance, all references to session attributes are replaced with references to the corresponding global variables. And "forwards" that are in present in Java code to page specifications are replaced with calls to the *pageModel* methods in the corresponding page models.

The resultant standard program that results from our approach is amenable to any single-language analysis tool. For instance, in our running example, a backward slice from the statement in 'DeliveryOptionsServlet' that reads the value of the request parameter *address* will pull in Lines 3-6 in Figure 3, and also, transitively, the code in other parts of the server that populated the list `addresses`. And a (single language) property checker will be able to traverse execution paths through server-side code and through the page models to determine, for e.g., that 'DeliveryOptionsServlet' necessarily uses a valid address of the currently logged-in user.

Page models result in the following advantages compared to previous approaches:

- Page models are generally agnostic to the analysis, and support, in principle, any code-based analysis, whether it is based on static analysis, dynamic analysis, symbolic execution, etc. Whereas, previous white-box approaches [2, 3, 8, 9, 12, 22, 26, 27] that analyzed both page specifications as well as server-side code each targeted a specific analysis problem. Therefore, page models could potentially form the foundation of a generic web application analysis infrastructure.
- Developers can directly leverage the rich set of existing single-language tools on the standard program that results from integrating the page models with the server-side code.
- Page models enable static analysis to conservatively track dataflows via request parameters and via session attributes simultaneously. Previous static analysis approaches [9, 12, 26] did not address this combination.

1.6 Our Contributions

Our primary contribution is the novel notion of page models to enable a variety of end-to-end whitebox analyses of web applications. The other contributions of this paper are as follows:

- An instantiation of our ideas in the context of J2EE applications, by describing in detail a translation algorithm that translates dynamic JSP (Java Server Pages) page specifications into corresponding page models in Java (which is the language of server-side code). Conceptually, however, our core ideas apply to any technology that uses HTML-based scripting for page specifications, such as PHP, Ruby on Rails, Struts, etc.
- An implementation of our approach.
- A discussion of our experiences from applying a variety of off-the-shelf white-box analysis tools, such as static slicing by Wala [29], concolic execution by Java Path Finder [14], and dynamic-analysis based fault localization by Zoltar [5].

The rest of this paper is structured as follows. Section 2 provides background on J2EE and JSP. Section 3 introduces in greater depth our notion of the page model, while Section 4 discusses our automated translation approach that generates page models from page specifications. Section 5 provides implementation details, while Section 6 provides empirical results. We discuss related work in Section 7, and conclude the paper in Section 8.

2 BACKGROUND ON J2EE AND JSP

```
1 if (((String)ssn.getAttribute("state")).equals(
      "logged_in")){
2   for (String addr: ssn.getAttribute("addresses"){
3     out.write("<a href=\"DeliveryOptions.jsp?address="+
       addr+"\">\n");
4     out.write("Select"+addr+"</a>");
5   }
6   ssn.setAttribute("state","addr_selected");
7 }
```

Figure 4: compiled form of lines 2-8 of SelectAddress.jsp.

The basic building block of a J2EE web application is a *servlet*, which is a Java program that executes within the web server. Whenever a HTTP *request* comes to the server, this request is sent to an appropriate servlet by invoking the 'doGet(request, response)' method of the servlet. It is the responsibility of the servlet to process the request, and to emit and return a (HTML) *response* page. Servlets can be written purely in Java. However, emitting HTML via "write" statements is tedious and error-prone from the programmer's perspective. To alleviate this, HTML-based scripting languages are typically used to specify client pages using a more natural, view-oriented syntax. JSP (Java Server Pages) is the standard client-scripting that is part of J2EE. Like other similar languages, JSP is based on HTML, with certain extra tags included to enable the specification of dynamic pages. The JSP compiler converts a JSP file into a (pure Java) servlet, which contains "write" statements that emit the HTML fragments in the JSP file.

A programmer can also adopt a hybrid approach, where in a programmer-written Java servlet receives the request, does some initial processing, and then "forwards" the request to a JSP-derived servlet whose responsibility it is to emit and return the response HTML page. Throughout this paper, whenever we refer to "server-side code", we mean only the programmer-written Java servlets. Therefore, in our view, a web application is a combination of server-side code and JSP page specifications.

The semantics of JSP becomes more clear if we look at sample Java output produced by the JSP compiler. Figure 4 depicts a simplified version of the Java code produced by the compiler for Lines 2-8 in Figure 2. It is interesting to note that this entire Java code executes before the generated page is sent to the client. This results in the inversion of control-flow issue, which poses a challenge to sound analysis as discussed earlier in Sections 1.4 and 1.5.

3 PAGE MODELS

Before we discuss page models further, we discuss why it would be preferable not to instead work with the Java code produced for each JSP file by the JSP compiler. (1) To enable any form of precise static analysis of the web application, the format of the HTML emitted by "write" statements would need to be inferred. This requires complex static analysis techniques such as *string analysis* [8] to be applied on the JSP scripts, whose precision may not always be high. (2) Even if the format of the emitted HTML can be inferred, dataflows from the server to the page and back would be hard to infer. For instance, it would be hard to infer that the address in variable 'addr' in line 2 in Figure 4 flows from the server into the HTML page, and then back to the server as a request parameter when the user eventually clicks the generated link.

Our key proposal, on the other hand, is to directly and faithfully encode the control-flow and data-flow semantics of page specifications into page models, which would then enable a variety of analyses to be applied on the resultant application.

3.1 Challenges in the Construction of a Correct Page Model

We had introduced the basic ideas behind page models in Section 1.5. The simplified page model in Figure 3 is indeed an acceptable page model for the page specification in Figure 2. However, in the presence of more complex page specifications, a page model having the simple form as shown in Figure 3 will not suffice.

The first issue is the required reordering of statements that update session attributes, in order to account for inversion of control flow. The page model in Figure 3 incorporates this reordering (as discussed in Section 1.5). Let us consider a more complex page specification that has a loop, with both state updates and link specifications inside the loop. Now, in the page model, all state-update statements would need to appear before all the calls that represent the links. This would require splitting the loop into two loops in a semantics-preserving manner, which is in general a complex transformation. Such code duplication becomes even more complicated if sequencing, loops, conditionals, etc., are simultaneously used in page specifications.

Secondly, the true semantics of a link-click is non-returning. Whereas, a call that represents a link appears to permit control to come back to the page, and then for another link to be clicked, etc. This issue cannot simply be fixed by placing a "goto" after each call-site to the end of the page model, because this would make it appear that the first link to be encountered in any execution of a page model will always be the one that is taken.

3.2 A Practical and Correct Modeling Approach

In order to illustrate our actual proposal, we present in Figure 5 the page model that is generated by this approach for the page specification in Figure 2. The notable features of this page model are as follows.

Lines 6-18 are the core of the page model. This part of the code essentially has the same control-flow structure as the page specification, but with certain local transformations as mentioned below.

At the location of each link, instead of having a call corresponding to the link, a local variable 'whichClicked' is set to record the URL of the link. For instance, Lines 9 and 16 in Figure 5 correspond to the link specifications in Lines 4 and 10, respectively, in Figure 2.

Lines 19-27 contain what we call a **finalBlock**. This is basically a "switch" statement that checks the value of 'whichClicked', and dispatches to the appropriate servlet. This way, calls representing links essentially become non-returning.

The local variable 'anyClicked' is used to ensure that at most one link is clicked in any path through a page model.

Statements that update session attributes remain at the same corresponding locations as in the page specification; see Lines 12 and 18 in Figure 5, which correspond to Lines 7 and 11 in Figure 2. This way, all state-updates occur before all the calls (which are in the **finalBlock**).

```
1  public class SelectAddress_jsp{
2    public void pageModel(){
3      boolean anyClicked = false;
4      String whichClicked = “none”,
5      String address_shadow = null;
6      if(state.equals(“logged_in”){
7        for(String addr: addresses){
8          if ((!anyClicked) && (coinFlip())) {
9            whichClicked = “DeliveryOptions.jsp”;
10           anyClicked = true;
11           address_shadow = addr;}}
12       state = “addr_selected”;
13     }
14     else{
15       if ((!anyClicked) && (coinFlip())) {
16         whichClicked = “Login.jsp”;
17         anyClicked = true;}
18       state = “uninit”;}
19     address = null; address = address_shadow;
20     switch (whichClicked) {
21       case “none”: break;
22       case “DeliveryDetails.jsp”:
23         DeliveryOptionsServlet.doGet(null, null);
24         break;
25       case “Login.jsp”:
26         LoginServlet.doGet(null, null); break;
27     }}}
28
```

Figure 5: Final page model for SelectAddress.jsp

Finally, at the original link locations, values of request parameters are captured into newly introduced *shadow variables* (see Line 11 in Figure 5). Later, just before the "switch" statement in the **finalBlock**, these shadow variables are copied into the corresponding global variables that represent the request parameters in the modeled program (see Line 19). This way, examples such as the one in Section 1.4, where a session attribute is first used in a link specification and then updated, will be handled correctly. Also, at the beginning of the **finalBlock** all global variables representing all request parameters of all pages are set to null, to ensure that a value sent as a request parameter to a page does not end up reaching some other subsequent page.

4 TRANSLATION ALGORITHM

In this section we present our algorithm for translating a JSP file into a page model. The algorithm is a syntax-directed translation based on the DOM structure of JSP pages. A JSP page is similar in structure to an HTML page, with additional tags. Thus, it can be viewed as a DOM tree. The translation algorithm descends from the root of this tree, in a top down manner, and translates different JSP/HTML tags into corresponding Java code.

The algorithm is presented, in Figure 6, as a set of syntax-directed translation rules. The algorithm maintains an *environment*, which is a map from local variables as well as global variables (i.e., session attributes and request parameters) to their types. Types are nothing but Java classes. Each translation rule in the algorithm is expressed in the form $\langle e, C\rangle \twoheadrightarrow \langle e', S\rangle$. This means that under the environment e, (a) the JSP page element C (which could be a JSP tag, HTML tag, EL-expression, etc.) is to be translated to the Java code fragment S,

$$\frac{\mathsf{S}_1 =_{def} \textbf{shadowInitBlock},\quad \langle e, C\rangle \twoheadrightarrow \langle e_1, \mathsf{S}_2\rangle,\quad \mathsf{S}_3 =_{def} \textbf{finalBlock}}{\langle e, <\ root\ >\ C\ <\ /root\ >\rangle \twoheadrightarrow \langle e_1, \texttt{public void pageModel() \{}\mathsf{S}_1\ \mathsf{S}_2\ \mathsf{S}_3\texttt{\}}\rangle}\ [\text{T-Root}]$$

$$\frac{\langle e, C_1\rangle \twoheadrightarrow \langle e_1, S_1\rangle,\ \langle e_1, C_2\rangle \twoheadrightarrow \langle e_2, S_2\rangle}{\langle e, C_1 C_2\rangle \twoheadrightarrow \langle e_2, \mathsf{S}_1\ \mathsf{S}_2\rangle}\ [\text{T-Seq}] \qquad \langle e, \mathit{PlainText}\rangle \twoheadrightarrow \langle e, \epsilon\rangle\ [\text{T-Text}] \qquad \frac{\exists t \in \mathit{Type}.\ (\mathit{var}, t) \in e}{\langle e, \$\{\mathit{var}\}\rangle \twoheadrightarrow \langle e, \mathtt{var}\rangle}\ [\text{T-ELVar}]$$

$$\frac{\langle e, \$\{E_1\}\rangle \twoheadrightarrow \langle e, \mathsf{E}_1\rangle,\quad \langle e, \$\{E_2\}\rangle \twoheadrightarrow \langle e, \mathsf{E}_2\rangle}{\langle e, \$\{E_1\ op\ E_2\}\rangle \twoheadrightarrow \langle e, \mathsf{E}_1\ \mathtt{op}\ \mathsf{E}_2\rangle}\ [\text{T-ELBin}] \qquad \frac{\langle e, \$\{E_1\}\rangle \twoheadrightarrow \langle e, \mathsf{E}_1\rangle}{\langle e, \$\{E_1.\mathit{property}\}\rangle \twoheadrightarrow \langle e, \mathsf{E}_1.\mathit{getProperty}()\rangle}\ [\text{T-ELProp}]$$

$$\langle e, <\ \mathit{input}\ \ \mathit{type} = \text{“}\mathit{text}\text{”}\ \ \mathit{name} = \text{“}xyz\text{”}/\ >\rangle \twoheadrightarrow \langle e, \texttt{xyz\_shadow = read();}\rangle\ [\text{T-Input1}]$$

$$\frac{\langle e, E\rangle \twoheadrightarrow \langle e, \mathsf{E}\rangle}{\langle e, <\ \mathit{input}\ \ \mathit{type} = \text{“}\mathit{text}\text{”}\ \ \mathit{name} = \text{“}\mathit{xyz}\text{”}\ \ \mathit{value} = \text{“}E\text{”}/\ >\rangle \twoheadrightarrow \langle e, \texttt{xyz\_shadow = coinFlip()?E : read();}\rangle}\ [\text{T-Input2}]$$

$$\frac{\langle e, E\rangle \twoheadrightarrow \langle e, \mathsf{E}\rangle\ \ \langle e, C\rangle \twoheadrightarrow \langle e_1, \mathtt{S\_1}\rangle}{\begin{array}{l}\langle e, <\ \mathit{form}\ \ \mathit{action} = \text{“}E\text{”}\ >\ C\ <\ /\mathit{form}\ >\rangle \twoheadrightarrow \langle e_1, \texttt{if(!anyClicked \&\& coinFlip())\{}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{whichClicked = E;}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{anyClicked=true;}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{S\_1 \} }\rangle\end{array}}\ [\text{T-Form}]$$

$$\frac{\begin{array}{c} P \equiv \mathit{paramMap}(E),\ \ E_1 \equiv \mathit{target}(E), \langle e, E_1\rangle \twoheadrightarrow \langle e, \mathtt{E\_1}\rangle,\ \ \langle e, C\rangle \twoheadrightarrow \langle e_1, \mathtt{S\_C}\rangle \\ \mathtt{S\_P} =_{def} \mathsf{S}_1\ \mathsf{S}_2\ \mathsf{S}_3 \cdots, \text{where } \mathsf{S_i} =_{def} \text{‘}\mathtt{shadow\_p_i = v_i}\text{’}, (p_i, v_i) \in P,\ \ \langle e, v_i\rangle \twoheadrightarrow \langle e, \mathsf{v_i}\rangle \end{array}}{\begin{array}{l}\langle e, <\ a\ \ \ \mathit{href} = \text{“}E\text{”}\ >\ C\ <\ /a\ >\rangle \twoheadrightarrow \langle e_1, \texttt{if(!anyClicked \&\& coinFlip())\{}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{whichClicked = E\_1;}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{anyClicked=true;}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{S\_P; S\_C \} }\rangle\end{array}}\ [\text{T-Link}]$$

$$\frac{\langle e, E\rangle \twoheadrightarrow \langle e, \mathsf{E}\rangle,\ \ \langle e, C\rangle \twoheadrightarrow \langle e_1, \mathsf{S}\rangle}{\langle e, <\ \mathit{if}\ \ \mathit{test} = \text{“}E\text{”}\ >\ C\ <\ /\mathit{if}\ >\rangle \twoheadrightarrow \langle e_1, \texttt{if(E)\{S\}}\rangle}\ [\text{T-If}] \qquad \frac{\langle e, E\rangle \twoheadrightarrow \langle e, \mathsf{E}\rangle,\ \ \exists t \in \mathit{Type}.\ (v, t) \in e}{\langle e, <\ \mathit{set}\ \ \mathit{var} = \text{“}v\text{”}\ \ \mathit{value} = \text{“}E\text{”}/\ >\rangle \twoheadrightarrow \langle e, \mathtt{v = E}\rangle}\ [\text{T-Set}]$$

$$\frac{\langle e, E\rangle \twoheadrightarrow \langle e, \mathsf{E}\rangle,\quad \mathsf{T} = \mathit{getTypeOfElement}(e, \mathsf{E}),\quad e_1 \equiv e[v \to \mathsf{T}],\quad \langle e_1, C\rangle \twoheadrightarrow \langle e_2, \mathsf{S}\rangle,\quad e_3 \equiv e_2 - \{(v, \mathsf{T})\}}{\langle e, <\ \mathit{forEach}\ \ \mathit{var} = \text{“}v\text{”}\ \ \mathit{items} = \text{“}E\text{”}\ >\ C\ <\ /\mathit{forEach}\ >\rangle \twoheadrightarrow \langle e_3, \texttt{for(T v : E)\{S\}}\rangle}\ [\text{T-ForEach}]$$

Figure 6: Translation Rules

and (b) the construct C results in an extended environment e'. Many of the rules have an antecedent (the part above the horizontal line), which basically inductively translates the components of C into their respective Java page model fragments. Note that for clarity we depict environments and JSP page elements in *italics* font, and Java page model fragments emitted by the rules in `typewriter` font. Due to space constraints, only the interesting translation rules are shown in the figure. The rules for a number of (simple) constructs have been omitted.

We assume the following for ease of presentation of algorithm: (a) The initial environment provided to the algorithm contains types for all session attributes and request parameters that occur across the application. (b) There are no name clashes among session attributes and among the set of all request parameters to all pages.

We describe in Section 5 how in our implementation we remove the first restriction mentioned above. The second restriction above can be handled easily, the details of which we omit.

We now go over the rules in Figure 6. The first rule, namely T-Root, works on the entire JSP file C. The result from this rule is the entire page-model method. The body of `pageModel()` method has three sections. The first section is the **shadowInitBlock**, which includes the initializations of all the record keeping variables - anyClicked, whichClicked, as well as the shadow variables corresponding to the request parameters (see Section 3.2 for the discussion about shadow variables). For instance, see lines 3-5 in Figure 5. The last section of this method's body is denoted as **finalBlock**. **finalBlock** first contains an assignment of null to all global variables that represent request parameters across all pages. It then contains assignments of the values of the shadow variables in the page model to the corresponding global variables representing the request parameters. In Figure 5, all these assignments occur in Line 19. It then contains the "switch" statement that contains the calls that represent the links (see lines 20-27 in Figure 5). The middle section requires the algorithm to translate the constituent elements of the JSP file (see the translation '$\langle e,C\rangle \twoheadrightarrow \langle e_1,\mathsf{S}_2\rangle$' in the antecedent).

Rule T-Seq says that the translation of a sequence of elements is the sequencing of the translations of the elements. Rule T-Text basically says that plain text (i.e., textual content that contains no tags within) need not appear in the page model.

Rules T-ELVar, T-ELBin, and T-ELProp convert EL-expressions into corresponding Java expressions. Rule T-ELVar puts variable/attribute references as-is in the page model. Rule T-ELBin replaces EL operators with equivalent Java operators. Rule T-ELProp encodes the EL-expression convention [13] that an EL-expression of the

```
<form action=“ab.jsp”>
  <input type=“text”
      name=“xyz”/>
  <input type=“text”
      name=“uvw”/>
</form>
```

```
if (!anyClicked &&
      coinFlip()) {
  whichClicked = “ab.jsp”;
  anyClicked = true;
  xyz_shadow = read();
  uvw_shadow = read();}
```

Figure 7: A sample HTML form, and its page-model translation

form $e.propX$ corresponds to a call to the getter method *getPropX* on the object that the expression e evaluates to.

Rules T-Input1 and T-Input2 handle two different kinds of HTML text boxes, the first kind not having any default filled-in value in the text box and the second kind having a default filled-in value E (which the user may edit). The translated Java code assigns to the corresponding shadow variable. `read()` is a special method whose return value models (arbitrary) user input.

Rule T-Form translates HTML forms. The left-side of Figure 7 shows a sample form. In a browser, whenever a request is sent by submitting a form, the values of input elements enclosed by the form body (in the example, the text boxes “xyz” and “uvw”) are basically sent as request parameters to the URL indicated in the “action” attribute of the form (in the example, “ab.jsp”). The right-side of Figure 7 shows the page-model translation of the sample form. Note that anyClicked and whichClicked need to be used, as was illustrated in Figure 5, to model the user's choice as to whether to submit this form or not (they could submit some other form or click some other link in the page instead). In the rule, S_1 denotes the translation of the body of the form, using the rules T-Input1 and T-Input2 and the sequencing rule.

The rule for links (Rule T-Link) is similar to, but somewhat more complicated than the rule for forms. Recall that we had earlier illustrated the translation of links: the link in lines 4-5 in Figure 2 gets translated as in lines 8-11 in Figure 5. In the antecedent of this rule, the utility function *paramMap*(E) syntactically extracts the set of (request-parameter,value) pairs from the link E; e.g., for the link in lines 4-5 in Figure 2, it would return a set containing the single pair (“address”, “${addr}”). The utility routine *target*(E) returns the URL-part of the link (i.e., the part before the ‘?’).

Rules T-If and T-Set are straightforward. In Rule T-ForEach, in the antecedent, first the type T of the items in the collection E is extracted from the environment e using the utility routine *getTypeOfElement*. Then, the body C of the loop is translated into Java code S in an extended environment $e[v \rightarrow \mathsf{T}]$.

Our page translation algorithm handles other features in page specifications, such as “forwards” and “redirects” in page scripts, attributes of different scopes (other than session scope) such as application scope, request scope, and page scope, input type elements other than text-boxes, such as drop-down lists, and scriptlets (i.e., arbitrary Java code fragments embedded in page scripts). Due to space constraints we omit the details of these (which are not technically very involved).

4.1 Code Changes Required in the Server-Side

Our approach performs certain modifications on the server-side code as well. It replaces references to session attributes and request parameters in the server-side code with references to the corresponding global variables that we introduced earlier. It is notable that we have borrowed the idea of replacing session attributes references with global variables from the F4F [22]. However, that approach was directed specifically at taint analysis, and as such, analyzed server-side code alone, without using any page models.

Another change is to replace “forwards” with calls. For instance, ‘forward(“SelectAddress.jsp”)’ is translated to ‘(new SelectAddress_jsp()).pageModel()’. Also, a “return” statement is placed immediately after this call, to emulate the non-returning semantics of the forward.

Finally, the web application is converted into a “modified” single-language, executable, non-web application, by introducing a “main” method. This method will non-deterministically choose one of the initial pages of the application and will call the ‘doGet()’ method of the corresponding servlet.

4.2 Correctness

We argue correctness of our approach in this section, in an informal manner in the interest of space.

Theorem (correctness): Consider any two *corresponding* runs of the modified program that results from the approach, and the original web application. That is, the two runs start from the same initial states, visit the same sequence of pages, and use the same input choices (as well as text-box input values) in each pair of corresponding pages. The two runs satisfy the following properties.

(1) The same number of requests get passed from the client to the server. (2) Corresponding requests will be to the same URL, with the same values for the request parameters. (3) The server-side state at the time each pair of corresponding requests is received is identical (modulo attributes and request parameters are compared with their corresponding global variables). (4) This point is actually an implication of the previous point. Corresponding traces inside the server-side code will follow the same path, and will have identical states at corresponding points. □

From the theorem above it follows that any analysis that is done on the modified (single language) application that checks any property of the server-side state will generate the same answer as would be obtained if the analysis were to be applied on the original web application.

4.3 Assumptions and Limitations

Our translation rules (Figure 6) are devised in a syntax-directed fashion. This gives simplicity and efficiency. However, certain features cannot be accommodated in this framework. For instance, the name of a text-box cannot be anything other than a string literal (see rules T-Input1 and T-Input2). Also, our approach ignores “write” statements that emit HTML in the server-side code as well as in scriptlets. It is possible that certain previous string analysis approaches [8] can be integrated with our approach to overcome these limitations.

Similar to many other previous web application analysis approaches, we assume that users are non-adversarial. That is, we assume that users (a) start using the application from its designated entry page(s) only, (b) click links, but do not type URLs of inner pages directly into the browser, and (c) do not modify outgoing requests from the browser.

Our approach currently does not address Ajax or Javascript. An interesting future extension of our approach would be to handle these by conservatively incorporating into the page model DOM manipulations that can be performed by Javascript, and by replacing Ajax requests with (asynchronous) calls to the server-side code.

5 IMPLEMENTATION AND APPLICATIONS

We have a prototype implementation of our approach, which we describe briefly in this section. Subsequently, to demonstrate the versatility and usefulness of the modified applications generated by our approach, we describe how we applied three different kinds of off-the-shelf analysis tools on the modified applications: (1) A concolic execution tool (JPF) for deep property checking, (2) a dynamic analysis tool (Zoltar) for fault localization, and (3) a static slicing tool (Wala).

5.1 Our Tool

Our translation tool requires the following inputs: the entire server-side code of the application (Java code, and JSP specifications), and the list of all possible URLs in the application and their server side targets (i.e., the deployment descriptor). We use JSP2X [20] to parse JSP files, and Eclipse JDT to construct Java code for the page models. The server-side code changes (see Section 4.1) are also automated, except for one kind of change, which is the translation of "forwards" to calls in servlets. We currently do this change manually, but it should be automatable eventually. The output from our approach is the modified application, which is a standalone Java application and not a web application, as discussed in Section 4.1.

5.2 Analysis 1: Deep Property Checking Using JPF

A powerful analysis that is enabled by our translation is the detection of functional errors in the application, with complete consideration of dynamic client pages, server-side logic, and the interactions between these two sides. For this analysis, we use the Symbolic PathFinder (SPF) tool [14], which is the concolic execution variant of the widely used tool Java PathFinder (JPF) [4]. We sketch here the steps that we followed in applying SPF on the modified applications generated by our approach. Firstly, we performed three kinds of modifications in the generated page models in order to ensure *exhaustive* coverage of the application (rather than run through just one random sequence of pages): (1) The page models now use SPF APIs to choose *all possible* links/forms out of a page, one by one, whenever a page is visited. (2) Similarly, all values in a drop-down lists are tried one by one. (3) Text boxes are filled using an SPF API that returns a fresh symbolic value. We have performed these modifications in generated page models manually as of now; however, automation is easy in principle.

We also made two kinds of modifications in the server-side code: (1) SPF does not off-the-shelf precisely analyze database operations in applications. Therefore, we manually modified the benchmark applications that we used in our experiments to use plain Java collections instead of databases. (2) We inserted property-checking code into the benchmarks to actually check for violations of the desired properties. SPF can now be used to explore traces of the application in a systematic manner to look for violations.

Table 1: Benchmarks statistics

Name	JSP files	Java classes
Trainers Direct (TD) [28]	18	30
Royal Odyssey (RO) [21]	55	3
MusicStore (MS) [1]	31	34
Help Desk Wiki (HD) [11]	27	3
iTrust (IT) [17]	9	19

5.3 Analysis 2: Fault Localization Using Zoltar

Zoltar [5] is a dynamic analysis based fault localization tool. Given a set of passing and failing test cases, the tool analyzes the relative frequency of execution of all the statements in the program as per all the given test cases. It reports, for each statement in the program, a *suspicion* score indicating the likelihood that the statement is the cause of a test failure.

For this experiment, we first converted the set of benchmark web applications into modified (standard Java) programs using our approach. To be uniform across all our experiments, we used in this analysis and also in the next analysis (i.e., slicing) the versions of the modified programs in which database operations are replaced by collections operations (see Section 5.2). In the next step, we created a set of test-cases for each benchmark manually. A test-case is nothing but a sequence of pages to visit, along with a tuple of input choices within each page that is visited. All our test cases were passing test cases. We then seeded errors in the (modified) benchmark programs. To seed bugs in an unbiased way, we used a mutation testing tool, namely, PIT [15]. We randomly selected 20 mutations per benchmark from the ones suggested by PIT, taking care to select mutations that caused at least some test cases to fail. We thus obtained 20 buggy versions of each benchmark.

We then ran Zoltar on each mutated version of each benchmark, using the test cases for the benchmark (some of which are now failing). We then evaluated the effectiveness of Zoltar in identifying the mutated location in each buggy version of each benchmark.

5.4 Analysis 3: Static Slicing Using Wala

For this experiment, we applied static backward slicing on the modified programs generated by our approach. Since "full" slicing with Wala often does not scale to larger benchmarks, we computed "thin" slices [23], with the "object-type context sensitivity" option. Our objective was to see how large slices tend to be in practice, how frequently they cross boundaries of multiple pages, and whether slices could be helpful in detecting causes of bugs. Our other motivation for choosing this analysis is that slicing has been a frequent topic of study in the software engineering community, even specifically by researchers who have targeted web applications [12, 19, 26].

6 EXPERIMENTAL RESULTS

In this section we provide empirical results from applying our tool in conjunction with the three analysis tools that we had mentioned

Table 2: Results from deep property checking

Name	num. props.	PM		SE		RT
		num. viol.	false pos.	num. viol.	false pos.	num. viol.
TD	5	4	0	5	1	3
RO	3	0	0	1	1	0
MS	5*	2	1	2	2	1
HD	3	1	0	1	0	1
IT	3	1	1	2	2	0
Total	19	8	2	11	6	5

∗ SE was not able to check two out of these 5 properties

in Section 5, namely, JPF, Zoltar, and Wala. We have used 5 web applications as benchmarks in these experiments. Table 1 lists the number of JSP and Java files in each web application. The first three benchmarks are e-commerce applications. Help Desk Wiki (HD) is a help-desk application for tasks like entering complaint tickets, address book management, etc. iTrust (IT) is a medical records management application. Help Desk Wiki and iTrust are frequently downloaded applications (995 and 7292 downloads, respectively, since 2010). Due to the large size of the iTrust (223 JSP files and 365 Java files in total), we have analyzed only a small fragment of the application. For this benchmark, Table 1 shows the statistics only for the considered fragment. Also, some of the iTrust pages involve Javascript, which is ignored by our approach. Therefore, the precision and/or correctness of our analyses could have been impacted to a certain extent for this benchmark.

The machine we have used for all our runs has an Intel i7-6600 processor, and 32 GB of RAM.

6.1 Analysis 1: Property Checking Using JPF

We checked our benchmarks for various functional properties, e.g., "multiple registrations using the same email ID should not be allowed", "if the cart is empty then the total should be zero", and so on. We extracted the properties for iTrust from the requirements specification provided as part of the official iTrust documentation. The other benchmarks did not come with documentation; therefore, we requested graduate students who were not associated with our work to run the benchmarks and identify desirable properties. We obtained a total of 19 properties in this way. Table 2 summarizes this experiment.

Concolic execution is very expensive, and does not scale, in our experience, without manual intervention. Therefore, for each property that we checked, we manually identified pages that do not affect the property, and pruned away these pages from consideration. Also, we identified suitable bounds on path lengths, while ensuring that minimal-length page sequences that necessarily need to be traversed to even check the property are traversed by both approaches.

As seen in Table 2, JPF with page models (PM) reported that 8 of the 19 properties were violated in some run. Out of these, six were reproducible, while two (one each in IT and MS) turned out to be false positives. One of these false positives was due to a limitation of SPF, while the other was due to page models not expressing precisely the semantics when a form has multiple "submit" buttons.

The remaining 11 properties can be considered to be verified for paths of length up to the bounds considered. This is because of our approach of considering all choices exhaustively, and using symbolic values for all text boxes, as discussed in Section 5.2.

The running times we observed ranged from a few seconds to 7 minutes each for each of the 19 properties.

Baselines. To serve as baselines, we modified our approach to simulate certain previous approaches for concolic execution of web applications [16, 30] that treat incoming request parameters from pages as always being unconstrained symbols. This assumption can result in loss of precision; for instance, with the running example in Figure 1, the information that the incoming address a_1 into the "Delivery Options" page is a registered address of the currently logged-in user, would be lost.

The results from this experiment appear in the columns labeled SE in Table 2. These approaches reported 11 properties out of the 17 that they checked as being violated (they could not scale to the two remaining properties). 5 of these violations were not reported as violations by our approach. These 5 reports are basically false positives from their approaches, as our approach traversed the same path that these approaches flagged as violating, and our approach is sound with respect to the paths actually traversed.

Separately, we also modified our approach to simulate certain previous approaches that are based on large-scale random testing of web applications, such as the approach of Thummalapenta et al. [25]. These experiments are summarized in the columns labeled RT in Table 2. In these experiments, the surprising outcome was that 5 of the 6 reproducible property violations identified by concolic execution were also identified by random testing. Nonetheless, it is notable that concolic execution alone provides the guarantee that certain properties (13 of them) *cannot* be violated for *any* sequence of text-box inputs (modulo the path length bounds).

6.2 Analysis 2: Fault Localization Using Zoltar

From each run of Zoltar (on a buggy version of a benchmark), we first computed the suspicion score of each method as being the maximum suspicion score of any statement in the method. We then shortlisted those methods that had scores greater than or equal to the score of the method that contained the seeded error.

Across the 20 buggy versions of the benchmark TD, the mean number of shortlisted methods per version was only 3.27%. (Throughout this section we use geometric means only.) That is, intuitively, only 3.27% of methods in a buggy version would need to be inspected before the bug is found. The corresponding mean percentage of shortlisted methods for the other benchmarks were: 5.3% for RO, 4.9% for MS, 3.9% for HD, and 0.2% for IT. These strikingly low numbers indicate the usefulness of our page models in conjunction with the Zoltar tool to localize faults in our benchmark programs.

It was difficult for us to obtain a baseline for these experiments. Zoltar (or for that matter, most dynamic analysis tools) are not able to work on web applications directly because of the tiered architecture of the applications, and because of the server-side framework and libraries.

Table 3: Results from slicing

Name	Num. crit.	Mean pages	Mean %-age methods	%-age thru attr.	%-age thru req. param.
TD	29	4.0	11.3%	38%	65%
RO	45	8.0	8.4%	60%	80%
MS	43	1.7	4.9%	12%	56%
HD	48	8.0	14.1%	71%	71%
IT	34	3.8	17.4%	68%	77%

(a) All criteria

Name	Num. crit.	Mean pages	Mean %-age methods	%-age thru attr.	%-age thru req. param.
TD	4	8.5	24.3%	75%	100%
MS	1	0.0	0.3%	0%	0%
HD	1	20.0	63.8%	100%	100%

(b) Bug criteria

6.3 Analysis 3: Static Slicing Using Wala

In our first set of slicing experiments, we used all "output" expressions in the programs – i.e., expressions whose values get inserted into databases or into displayed pages – as criteria. Table 3, Part (a), provides statistics about the slices from all these criteria. The meanings of the first four columns in the table are as follows: (1) Name of the benchmark, (2) Total number of criteria (i.e., total number of slices taken), (3) Mean number of pages (i.e., distinct 'pageModel' methods) included in a slice, and (4) Mean percentage of methods in the application that are included in a slice. It is easy to see that slices in general cross a large number of page boundaries; in other words, a slicing tool for web applications definitely needs to account for dataflows through pages. Also, these slices tend to be small, and hence have the potential to be useful to developers.

Part (b) of Table 3 provides information about slices that we computed from criteria that are expressions that in some runs contain incorrect values. These criteria were identified from the property violations that were detected by JPF (see Section 6.1 above). Such slices could help identify root causes of the errors.

Baselines. Previously reported dedicated approaches for slicing web applications happen to ignore either dataflows through session attributes [12], or dataflows through request parameters [9]. Hence, they can compute unsound slices in the presence of these features, respectively. The fifth column in the tables indicates the percentage of slices that involve dataflows through session attributes, while the last column indicates the percentage of slices that involve dataflows though request parameters. Clearly, a large proportion of slices happen to involve at least one of these two features, and hence are likely to be computed unsoundly by previous approaches.

7 RELATED WORK

In this section we sample certain closely related previous approaches for analysis of web applications.

Regarding static analysis of web applications, a seminal approach in this area is that of Ricca and Tonella [18]. They propose to use models constructed using human assistance for analysis and test-case generation. Subsequently, the same authors focused on the problem of (automated) static slicing of web applications [26], by conservatively approximating the formats and contents of dynamically created pages. In this approach dataflows due to session attributes are not handled. The approach of Liu [9], which also targets program slicing, identifies dataflows due to session attributes, but not dataflows due to request parameters.

The Taj approach [27] is about static taint analysis of server-side code, treating all incoming request parameters are tainted. Subsequent work by the same authors [22] builds on Taj; we adopt from this approach the idea of replacing session attributes with global variables to enable more precise analysis. Moller et al. [10] detect tainted dataflow from request parameters to session attributes, but do not detect the data flows from session attributes to request parameters. The approach of Kirkegaard et al. [8] is a static analysis on compiled JSP servlets, and aims to conservatively over-approximate the formats and contents of HTML pages generated by these servlets. The work by Halfond et al.[7] identifies inter-component control flows.

There is a substantial body of work on black-box techniques for crawling and testing web applications; these (and other topics) have been surveyed in a recent paper [6]. A recent work [24] targets automated generation of *page objects* from page specifications. Page objects communicate with the server via HTTP, and are useful primarily for black-box testing.

Several researchers have applied symbolic or concolic execution on web applications, for varying purposes. Wassermann et al. [30] generate test inputs to detect vulnerabilities, but they are not able to sequences of page visits. Weave [16] uses SPF [14] for checking deep functional properties over page sequences, but considers all request parameters as fresh symbolic values. The works by Artzi et al. [2, 3], are effectively the closest to our work. They use a combination of symbolic and explicit-state model checking, and target detection of vulnerabilities, HTML wellformedness, interface discovery, fault localization, etc., in PHP applications. Though their technique precisely tracks values from session attributes to request parameters during execution, it does not represent dataflows explicitly and hence is not amenable to static analysis. Recent work by Nguyen et al. [12] uses symbolic execution to perform static slicing of dynamic PHP applications. This approach is a precise one, but does not consider the effects due to session attributes. On the other hand, they are able to handle JavaScript.

8 CONCLUSIONS AND FUTURE WORK

Previous research on the analysis of web applications has yielded sophisticated techniques, but with each technique addressing a very specific kind of analysis problem. Perhaps as a result of this, there is a surprising lack of scalable, robust tools and platforms for fine-grained white-box analysis of web applications. Our approach using page models, in contrast, enables the the application of the plethora of robust single-language tools that exist out there on web applications. Furthermore, in comparison with previous static analysis approaches for web applications, ours is the first one to address simultaneous dataflows through request parameters and session attributes in a principled manner. Ideas for future work would include addressing cookies, Javascript, as well as other platforms such as Ruby on Rails and PHP.

REFERENCES

[1] 2016. MusicStore. https://github.com/cooervo/musicStore-Servlets-JSP-JPA. (April 2016).

[2] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Practical fault localization for dynamic web applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, Vol. 1. IEEE, 265–274.

[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D Ernst. 2010. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering* 36, 4 (2010), 474–494.

[4] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. 2000. Java PathFinder-second generation of a Java model checker. In *Proc. Workshop on Advances in Verification.* Citeseer.

[5] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 378–381.

[6] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. 2013. A systematic mapping study of web application testing. *Information and Software Technology* 55, 8 (2013), 1374–1396.

[7] William GJ Halfond. 2015. Identifying Inter-Component Control Flow in Web Applications. In *International Conference on Web Engineering*. Springer, 52–70.

[8] Christian Kirkegaard and Anders Møller. 2006. Static analysis for Java Servlets and JSP. In *Proc. Symposium on Static Analysis (SAS)*. 336–352.

[9] Chien-Hung Liu. 2006. Data flow analysis and testing of JSP-based Web applications. *Information and Software Technology* 48, 12 (2006), 1137–1147.

[10] Anders Møller and Mathias Schwarz. 2014. Automated detection of client-state manipulation vulnerabilities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 29.

[11] Gianni Moretti. 2017. Help Desk Wiki. https://sourceforge.net/projects/helpdeskwiki/. (May 2017).

[12] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2015. Cross-language program slicing for dynamic web applications. In *Proc. Foundations of Software Engineering (FSE)*. ACM, 369–380.

[13] Oracle. 2017. Expression Language Reference. http://download.oracle.com/otndocs/jcp/el-3_0-fr-eval-spec/index.html. (May 2017).

[14] Corina S Pasareanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. 15–26.

[15] PIT 2017. PIT Mutation Testing, http://pitest.org. (May 2017).

[16] Sreeranga P Rajan, Oksana Tkachuk, Mukul Prasad, Indradeep Ghosh, Nitin Goel, and Tadahiro Uehara. 2009. Weave: Web applications validation environment. In *International Conference on Software Engineering (ICSE) - Companion Volume*. 101–111.

[17] NCSU RealSearch Group. 2017. iTrust. https://sourceforge.net/projects/itrust/. (May 2017).

[18] Filippo Ricca and Paolo Tonella. 2001. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. 25–34.

[19] Filippo Ricca and Paolo Tonella. 2002. Construction of the system dependence graph for web application slicing. In *Proc. IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. 123–132.

[20] Hannes Schmidt. 2017. JSP pages to JSP documents conversion. https://code.google.com/archive/p/jsp2x/. (May 2017).

[21] Aamir Sheikh. 2017. Royal Odyssey, Travel and tourism management application. http://www.postslush.com/2014/01/ewheelz-travel-and-toursism-managemen.html. (May 2017).

[22] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint analysis of framework-based web applications. In *Proc. ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1053–1068.

[23] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 112–122.

[24] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2016. APOGEN: automatic page object generator for web testing. *Software Quality Journal* (2016), 1–33.

[25] Suresh Thummalapenta, K Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Swarup Chandra. 2013. Guided test generation for web applications. In *International Conference on Software Engineering (ICSE)*. 162–171.

[26] Paolo Tonella and Filippo Ricca. 2005. Web application slicing in presence of dynamic code generation. *Automated Software Engineering* 12, 2 (2005), 259–288.

[27] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective taint analysis of web applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 87–97.

[28] Itoro Udoh. 2017. Trainers Direct,. https://github.com/aityworld/JavaEE-e-Commerce-Application. (May 2017).

[29] Wala 2017. T. J. Watson Libraries for Analysis, http://wala.sourceforge.net. (May 2017).

[30] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic test input generation for web applications. In *Proceedings of the 2008 Int. Symposium on Software Testing and Analysis*. ACM, 249–260.

[31] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.