Identifying Inter-Component Control flow in Web Applications

William G. J. Halfond

University of Southern California
halfond@usc.edu

Abstract. As web applications become more complex, automated techniques for their testing and verification have become essential. Many of these techniques, such as ones for identifying security vulnerabilities, require information about a web application's control flow. Currently, this information is manually specified or automatically generated using techniques that cannot give strong guarantees of completeness. This paper presents a new static analysis based approach for identifying control flow in web applications that is both automated and provides stronger guarantees of completeness. The empirical evaluation of the approach shows that it is able to identify more complete control flow information than other approaches with comparable analysis run time.

1 Introduction

Modern web applications have become increasingly sophisticated, interweaving complex interactions and combining data from multiple sources. As web applications become more complex, automated testing and verification techniques specifically tailored for web applications have risen in importance. Many of these techniques require detailed information about the control flow of a web application. For example, to identify multi-module vulnerabilities [4], access control vulnerabilities [23], or eliminate navigation errors [11]. For early web applications, identifying this control flow was as simple as following the links embedded in each of the application's web pages. However, control flow in modern web applications is more complex and limitations of techniques for identifying control flow have meant that testing and verification techniques have to rely on less accurate methods. Unsurprisingly, two control flow related vulnerabilities, "Failure to Restrict URL Access" and "Unvalidated Redirects and Forwards" have made the infamous OWASP Top 10 Web Application Security Risks list.

The architecture of modern web applications makes their control flow more complex than that of traditional (e.g., desktop) software. On the server-side of a web application, modules of code, called *components*, can expose methods that allow them to be directly executed over the web. Examples of components include Java Servlets and PHP pages. In turn, a component may generate data or *object programs*, code that is intended to be interpreted and executed in another context. Object programs can be written in "web" languages, such as JavaScript and HTML, or with protocols, such as HTTP, which allow developers to control the behavior of an end user's browser (e.g., using a redirect or authentication request). In web applications, not only is there control flow within each component, such as jumps, loops, branching, and method calls, written in the application's general purpose programming language, there is also inter-component control flow. This additional control flow is defined by the combination of the semantics of the object programs, which themselves can interact with other components and users. Control flow models that fail to account for all of these additional types of control flow may be incomplete and represent only a subset of the potential runtime behaviors of a web application.

Researchers and web application developers have recognized the importance of checking web application control flow. In early web applications, errors in control flow typically manifested themselves as dead links. Researchers found that web crawlers [2, 15, 24], were very effective at detecting these types of errors. However, as web applications became more dynamic, the effectiveness of these approaches was reduced. Subsequent work did not directly address this problem, but did provide a diverse set of techniques that could be used to verify and enforce correctness and security properties of web applications [1, 11, 13, 16, 21]. Unfortunately, their primary assumption is that there exist techniques for accurately specifying control flow. In practice, the approaches rely on either the developer to manually specify the control flow, which is time-consuming and error-prone, or on web crawlers, which cannot provide strong guarantees of completeness. Several automated static analysis techniques (e.g., [4, 5, 20, 23, 26]) have attempted to derive more complete control flow models, but do not account for all possible types of control flow.

This paper presents a new approach for statically identifying control flow in web applications that addresses the limitations of previous approaches. The approach is based on static analysis of the code of a web application. The static analysis identifies control flow related constructs defined by the generated HTML and JavaScript of the web application and by server-side commands generated by the general purpose language of the web application. This information is combined with traditional control flow information to provide a more complete model of a web application's control flow. As shown in the evaluation, the approach is more complete than other approaches and its running time is fast enough to allow it to be used to generate control flow information for a range of testing and verification tasks [11–13, 22, 23, 25, 27].

2 Web Applications Control Flow

Web application control flow includes both traditional control flow constructs, such as jumps, branches, and loops, as well as the following new types of control flow that are specific to web applications:

Dynamically Generated HTML Web pages are displayed in a user's browser and provide the user with the ability to interact with the web application. Certain types of HTML tags can also affect the control flow, either automatically or when they are clicked on by the end user. These tags include: `<a>`, `<form>`, `<img>`, `<meta>`, `<frame>`, and `<script>`.

JavaScript is a scripting language widely used to write programs that are embedded in HTML pages and that can interact with the web page's document object model (DOM) and the end-user. The embedded programs can perform a range of functions that can affect the inter-component control flow of a web application. These can be done by referencing the location property of the DOM and accessing the navigation functionality of the browser. Note that inter-component control flow related to *JavaScript* does not include control flow within the *JavaScript* programs, only the actions that cause a user to navigate from one component to another.

HTTP Commands Components communicate among themselves and with the end user's browser by sending messages using the Hyper-Text Transport Protocol (HTTP). A component issues HTTP commands by calling a special API-based command and passing it an HTTP response code and a message parameter. Certain HTTP response codes cause the browser to redirect to the location specified in the HTTP message. These codes are 300, 301, 302, 303, and 307. For all of these, the message is specified in a special HTTP header field that is defined by a message of the form "Location: *target*" where *target* is a URL that indicates where the user should be redirected. A lesser known HTTP refresh header can also be used in much the same way as the HTML `<meta>` tag.

Component Inclusion In many web application frameworks, it is possible to issue a command that imports the contents of another component. For example, both PHP and JSP have a variant of the `include` command. These commands import the contents of the target component at the point of the command. This can be done either statically via precompilation or dynamically at runtime. These commands affect control flow since the imported components are themselves executable code.

Direct Entry Users are able to enter the URL of a web component directly in a web browser's location bar. This action causes the root method of the target component to execute. *Direct Entry* can occur even if a link to the target component has not been exposed and the developer did not intend for the component to be an entry point into the application. In the security literature, *Direct Entry* control flow is also known as "forced browsing" and can lead to workflow vulnerabilities [4]. *Direct Entry* control flow can be further refined by the type of encoding used by the HTTP requests, such as GET or POST.

3 Motivating Example

Figure 1 shows a partial listing of a web application component, `Login.jsp`, that manages a user logging in to a web application. The component is implemented as a servlet in the Java Enterprise Edition (JEE) web application framework.

The input to `Login.jsp` is a `Request` object for accessing the HTTP message sent to the component and a `Response` object for sending content to the end user. At line 2, `Login.jsp` accesses a name-value pair, `session`, that is used by the application to track the logged-in status of the end user. If the session corresponds to a valid session, an HTTP command is issued at line 4 that causes the end user to redirect to the default servlet. This is done by calling `sendHttpCmd` and passing in three parameters, the first is the `Response` object, the second is the intended HTTP response code, and the third is the URL of the component to which the end user will be redirected. If the user does not have a valid session, but the session variable is equal to "login," then the servlet accesses the supplied username and password (lines 6 and

```
void service(Request req, Response resp)
    JspWriter out = resp.getOutputStream();
    String session = req.getParam("session");
    if (isValidSession(session)) {
     sendHttpCmd(resp, 302, "Default.jsp");
    } elsif (session.equals("login")) {
     String login = req.getParam("uname");
     String password = req.getParam("pword");
     if (isClean(login) && isClean(pword)) {
      if (loginOK(uname, pword)) {
       sendHttpCmd(resp, 302, "Default.jsp");
      }
     } else {
      sendHttpCmd(resp, 303, "Error.jsp");
     }
    } else {
     out.print("<html><body>");
     out.print("<script language='JavaScript'>");
     out.print("function goBack() {");
     out.print("window.location.href="Index.jsp";
     out.print("}");
     out.print("</script>");
     out.print("<h1>Login Page</h1>");
     out.print("<form method=POST" + " action=`Login.jsp'>");
     out.print("<input type=hidden value=" +  "`login' name=session>");
     out.print("User:<input type=text name=uname>");
     out.print("Password:<input type=" + "password name=pword>");
     out.print("<input type=submit value=`Login'>");
     out.print("<input type=submit value=`Back'" + " onClick=`goBack()'>");
     out.print("</form>");
     out.print("<a href='Reset.jsp'>" + " Reset password</a>");
     out.print("</body></html>");

void sendHttpCmd(Response resp, int code, String msg)
    String location = "Location: ";
    location += urlEncode(msg);
    location += "\n\n";
    resp.sendHttpMessage(code, location);
```

Fig. 1. Implementation of servlet, `Login.jsp`.

7) and checks the credentials at lines 8 and 9. If the login is successful, then at line 10 a redirect command is executed that allows the end user to proceed to the default servlet; otherwise, the user is redirected to an error page at line 13. Finally, if neither condition at line 3 or 5 applies, then the servlet prints a web form that allows the user to submit a username and password (lines 15–31). This web form, when submitted, sends the username and password back to the `Login.jsp` servlet. Alternatively, the user can click on the link generated at line 30 if they have forgotten their password or go back to the previous page that triggered the login request by clicking on the button generated at line 28. Clicking this button triggers the execution of the JavaScript function generated at lines 18–20.

`Login.jsp` illustrates several types of inter-component control flow. These include *HTTP Commands* at lines 4, 10, and 13; *Dynamically Generated HTML* via the HTML page produced at lines 16–31; and *JavaScript* via the script tag at lines 17–21 and called at line 28. The control flow links the `Login.jsp` servlet to `Index.jsp`, `Default.jsp`, `Error.jsp`, and `ResetPassword.jsp`. A web crawler could miss several of the control flow links during a crawl of the example application. For example, it is likely a web crawler could not accurately guess the constraints on the user input imposed at line 8. Therefore, the crawler would find the error page referenced at line 13, but not the default page referenced at line 10.

4 Approach

The goal of the proposed approach is to identify inter-component control flow in web applications. The proposed approach has four steps that together account for the different types of control flow. The first step targets control flow defined by the components' generated object programs, *Dynamically Generated HTML* and *JavaScript*, and is explained in Section 4.1. The second step, explained in Section 4.2, identifies server-side API-based control flow, which includes *HTTP Commands* and *Component Inclusion*. Section 4.3 details the third step, which identifies *Direct Entry* related control flow. The fourth step, in Section 4.4, combines the identified control flow into an Inter-Component Control Flow Graph (ICCFG).

4.1 Control flow in Generated HTML Pages

Control flow due to *Dynamically Generated HTML* and *JavaScript* is defined in the HTML output of the components of a web application. To identify this control flow, the approach builds on prior work in web page string analysis [8, 10] to compute the set of HTML pages that each component can generate at runtime. To do this, the approach analyzes each method of the component and computes a parameterized summary of the HTML that could be generated by the method. The methods are analyzed in reverse topological order with respect to the component's call graph to ensure that a method's summary is computed before those of calling methods. All methods that are part of a recursive call are analyzed together as one "super method." When the analysis of the component terminates, the summary of the root method represents all of the possible HTML pages that could be generated by the component.

$$\texttt{Gen}[n] = \begin{cases} \{\{\}\} & \text{if } n \text{ is method entry} \\ \{n\} & \text{if } n \text{ generates output} \\ \{n\} & \text{if } n \text{ is a callsite} \\ & \text{and target}(n) \text{ has a summary} \\ \{\} & \text{otherwise} \end{cases}$$

$$\texttt{In}[n] = \bigcup_{p \in pred(n)} \texttt{Out}[p]$$

$$\texttt{Out}[n] = \{p | \forall i \in \texttt{In}[n], p \leftarrow \texttt{append}(i, \texttt{Gen}[n])\}$$

$$\texttt{summary}(m) = \left\{ p | \forall s \in \texttt{Out}[exit(m)] \prod_{n \in s} \texttt{resolve}(n) \right\}$$

Fig. 2. Data-flow equations for identifying generated HTML [8].

Within each method, the approach uses iterative data-flow analysis to compute the set of HTML pages. This analysis computes the fixed point solution to the equations shown in Figure 2. As shown in the figure, each node n in the method is assigned a `Gen` set based on whether it directly generates HTML data, calls a function that then generates HTML data, or does not contribute at all to the generated HTML. A node can be identified as generating HTML content based on the signature of its target invocation method. The `In` and `Out` sets propagate this information. The general intuition behind the equations is that nodes that can directly or indirectly generate HTML content are appended together, so the `Out` set of the exit of a method is a set of ordered sets (representing paths) of nodes that can generate HTML.

Once the `Out` set of the method's exit node reaches a fixed point, the approach uses the `resolve` function to convert each node to a set of strings that represents its generated HTML. The `resolve` function handles two general cases. (1) If the node directly generates output, `resolve` computes a finite state automata (FSA) representation of its strings. For nodes that print a constant string or variable defined without string operations (e.g., no concat or insert), the approach identifies the reaching definitions of the string values. For variables defined using a string expression, the approach uses the Java String Analysis (JSA) to compute an approximation of the possible string values [3]. If `resolve` encounters the use of a string that is defined as one of the formal parameters to the enclosing method or external to the method, then the `resolve` function leaves a placeholder in the returned results. (2) If the node represents a call site to a method with a summary, `resolve` replaces any placeholders in the target's summary with the call site's corresponding arguments. The result of calling `resolve` on each node in each ordered set is appended together to create an FSA based representation of the HTML content that could be generated along each path in the application.

When the analysis terminates, each component is associated with a set of FSA based representations of its potentially generated HTML. The approach traverses the generated representation using standard parsing techniques, and identifies strings that define control flow related tags and JavaScript. The approach analyzes the contents of these tags to identify their control flow information.

The relevant tags include: <a> allows for the creation of hyperlinks that can be clicked on by the user to move from the current page to the target of the hyperlink; <form> allows the user to submit data in a web form, which causes control flow to transfer from the current page to the target component specified by the tag; <img> contains a URL-based attribute that defines its source. In some cases the source points to a component that dynamically generates an image and causes execution of the target component. To prevent inclusion of uninteresting control flow that would be identified in this case, the approach uses the heuristic of not including control flow generated by the <img> tag if its source URL has a suffix that corresponds to a static image type; <meta> can define a "refresh" attribute, which redirects the end user to another URL after a certain amount of time has elapsed; <frame> and <iframe> include a URL attribute that indicates a target component is to be executed and then displayed in a portion of the HTML page.

To identify *JavaScript* related inter-component control flow, the approach first identifies JavaScript contained by <script> tags, embedded as event handlers, and included from external files. Next, the approach parses the JavaScript to identify statements that implement *Component Inclusion*, *Dynamically Generated HTML*, or *HTTP Commands* control flow. For example, statements that reference the `location` property, use the Document Object Model's (DOM) navigation model to send the user to another URL, generate HTML, or load additional JavaScript. Since each of these statements requires an argument specifying the target of the operation, the approach attempts to identify the values that could be referenced at each statement. This is done using an algorithm almost identical to Algorithm 1, where the identified statements are the analyzed command points. The difference between this analysis and the one explained in Section 4.2 is that these `resolve` functions return string values based on reaching definitions and only model the effect of concatenation used at the identified statements to join reaching string values. This more limited `resolve` function is used because there is not yet a JSA equivalent for JavaScript. Once the potential values at each of the identified statements have been calculated, the approach parses the arguments to identify the HTML tags used to define *Dynamically Generated HTML* control flow, the targets of *HTTP Commands* related control flow, or the code included by *Component Inclusion* constructs.

Example In `Login.jsp`, only one of the four paths generates HTML content, the path that follows the false branch at line 5. The HTML page generated along this path is comprised of the string data generated at lines 16–31. The output of these nodes is appended together to form one string that contains the HTML output generated by the corresponding nodes. This string is then analyzed by an HTML and JavaScript parser to identify control flow information. This analysis identifies three control flow edges: a form that directs control flow back to `Login.jsp`, a hyperlink that directs control flow to `ResetPassword.jsp`, and a call to a JavaScript function that redirects the user to `Index.jsp`.

Algorithm 1 Identify API-based control flow

```
Input: $C$: web application component
Output: set of edges identified in $C$
 1: $methods \leftarrow$ methods of $C$ in RTO
 2: for all $m \in methods$ do
 3:     for all $stmt \in m$ do
 4:         if $stmt$ is a CP then
 5:             $codes \leftarrow \{FP, CI, 301, 302...\} \cap$ resolveCodes($stmt$)
 6:             if $codes \neq \emptyset$ then
 7:                 $targets \leftarrow$ resolveTarget($stmt$)
 8:                 $linenumber \leftarrow$ getLineNumber($stmt$)
 9:                 summary($m$) $\leftarrow$ summary($m$) $\cup$ createEdges($C, targets, codes$)
10:             end if
11:         else if isInvoke($stmt$) $\wedge$ summary(getInvkTarget($stmt$)) $\neq \emptyset$ then
12:             $target \leftarrow$ getInvkTarget($stmt$)
13:             $mappedEdges \leftarrow$ map(summary($target$), $stmt$)
14:             for all $e \in mappedEdges$ do
15:                 $targets \leftarrow$ resolveTarget($e$)
16:                 $codes \leftarrow$ resolveCodes($e$)
17:                 $linenumber \leftarrow$ getLineNumber($e$)
18:                 summary($m$) $\leftarrow$ summary($m$) $\cup$ createEdges($C, targets, codes$)
19:             end for
20:         end if
21:     end for
22: end for
23: return summary(root method of $C$)
```

4.2 Server-side API-based Control flow

The approach identifies server-side API-based control flow using a static analysis based technique. The general intuition behind this approach is to first identify the application's command points (CP) – points in an application where commands are issued to perform either *HTTP Commands* or *Component Inclusion* based control flow. The approach analyzes the CP to identify the possible values of their arguments. This is done by analyzing the chain of definitions and uses of the arguments. The identified values are parsed to extract control flow information. For CPs related to *HTTP Commands*, parsing the arguments identifies the HTTP response codes and the value of the `Location` header. For CPs related to *Component Inclusion*, the parsing identifies the component to be included at that point.

The algorithm for identifying server-side API-based control flow is shown in Algorithm 1. The input to the algorithm is a web component C, and the output is a set of edges, which contains the control flow defined in C. Each edge in $edges$ is a tuple of the form $\langle source, destination, code, linenumber \rangle$. The first element of the tuple, $source$, is the name of the component from which the edge is originating. In most cases, this is the canonical name of C. The second element of the tuple, $destination$, represents the target component to which control flow is redirected. The third element, $code$, is the HTTP response code that is part of the message or the value "CI" for edges re-

Algorithm 2 CreateEdges: helper function for creating edges.

```
Input: $C$: web application component; $targets$: set of edge targets; $codes$: set of codes for the edges
Output: $edges$: set of edges created
1: for all $target \in targets$ do
2:     for all $code \in codes$ do
3:         edges $\leftarrow$ edges $\cup$ (nameOf($C$), target, code, linenumber)
4:         if $code = CI$ then
5:             edges $\leftarrow$ edges $\cup =$(target, nameOf($C$), code, linenumber)
6:         end if
7:     end for
8: end for
9: return edges
```

lated to *Component Inclusion*. The last element, $linenumber$, is the line number in the component where the command is issued. This is used in the fourth step (Section 4.4) when the inter-component control flow information is combined with the CFGs of the individual components of the web application.

The analysis begins by creating a list of the methods of C. As in Section 4.1, the methods are analyzed in reverse topological order with respect to the component's call graph to ensure that a method's summary is computed before those of calling methods (line 1). All methods that are part of a recursive call are analyzed together as one "super method." Then for each method, m, each statement, $stmt$, is processed (lines 2–22).

If the statement is a CP (line 4), then `resolveCodes` is called on the statement to identify the possible HTTP codes that could be used at that point (line 5). The function `resolveCodes` is a simplified version of the `resolve` function introduced in Section 4.1, but is for resolving integer instead of string values. The set of values returned by calling `resolveCodes` is intersected with a set that contains HTTP control flow related codes, the symbol that is used to denote a placeholder for formal method parameters "FP," and the symbol for *Component Inclusion*, "CI" (line 5). If the intersection is non-empty (line 6), then `resolveTarget` is called to determine the value of the message used at the CP (line 7). The `resolveTarget` function is similar to the `resolve` method, but is customized to extract information related to *Component Inclusion* and the "Location" header. Then the line number that corresponds to $stmt$ is identified (line 8). A helper function, `CreateEdges`, shown in Algorithm 2 is called to generate the inter-component control flow edges. In `CreateEdges`, the algorithm creates a new edge for each code and target. If the code relates to *Component Inclusion*, then an additional control-edge is added that shows control flow returning from the target back to the source. This reflects that *Component Inclusion* includes the target component's control flow and then continues execution within the original component.

If the statement is not a CP, then it is checked to see if it invokes a method that has a summary associated with it (line 11). If this is the case, the target of the invocation is identified (line 12). Next, function `map` takes the statement and summary of the target method and replaces any placeholders in the method summary with the corresponding argument provided at the invocation call site (line 12). For each mapped edge, the corresponding resolve calls are performed to identify the possible locations and codes that could be executed at that point (lines 15 and 16). This is done since the substituted argument for either the location or code could be resolvable in the current method context or may itself be defined by a parameter to the current enclosing method. With the potential codes and locations discovered by the calls to the resolve functions, new edges are created by calling `createEdges` (line 18).

Once each of the methods have been processed, the summary of the root method of the component contains all of the *Component Inclusion* and *HTTP Commands* control flow edges that can be generated by the component at runtime. If there are any placeholders remaining in the summary, these edges are noted as being defined by external input to the component. The root method's summary is returned as the output of the algorithm.

Example To illustrate the second step of the approach, consider the example servlet, whose implementation is shown in Figure 1. Analysis of `Login.jsp` begins by analyzing method `sendHttpCmd`, since it is the first method in reverse topological order. The algorithm examines each statement of the method and identifies the CP at line 36 of `Login.jsp`. Next it attempts to resolve the value of the `code` variable that specifies the HTTP response code that will be issued at that point. The definition-use chain (DU) leads back to the second formal parameter of the method, so a placeholder (FP_2) that specifies this relationship is generated and returned by the call to `resolveCode` at line 6 of the algorithm. The formal placeholder is in the set of valid control flow related codes at line 7 of the algorithm, so the algorithm next tries to resolve the message sent at line 36 of `Login.jsp`. The DU chain leads back to the third formal parameter, so here again a placeholder is generated and returned (FP_3). No other statements in `sendHttpCmd` match either of the conditions at lines 5 or 20 of the algorithm, so a single edge of the form $\langle$`Login.jsp`, FP_3, FP_2, 36$\rangle$ is added to `sendHttpCmd`'s summary.

The algorithm then analyzes method `service`. The condition at line 20 of the algorithm is true for nodes 4, 10, and 13, since all call method `sendHttpCmd` and this method has a summary. The algorithm maps the statement's argument to the method's summary. For node 4, this creates $\langle$`Login.jsp`, `Default.jsp`, 302, 4$\rangle$; for node 10 this creates the edge $\langle$`Login.jsp`, `Default.jsp`, 302, 10$\rangle$; and for node 13 the edge $\langle$`Login.jsp`, `Error.jsp`, 303, 13$\rangle$ is created. The edges generated for each of the statements are added to the summary of `service`. No other statements in `service` match either of the conditions at line 5 or line 20 of the algorithm, so processing of `service` is finished and its summary is returned as the output of the algorithm.

4.3 Control flow Based on Direct Entry

The third part of the approach identifies control flow related to *Direct Entry*. There are two steps in this identification. The first identifies if a component is able to receive requests directly from an end user. This is done by analyzing the component to determine if it meets the necessary conditions specified by the web application framework

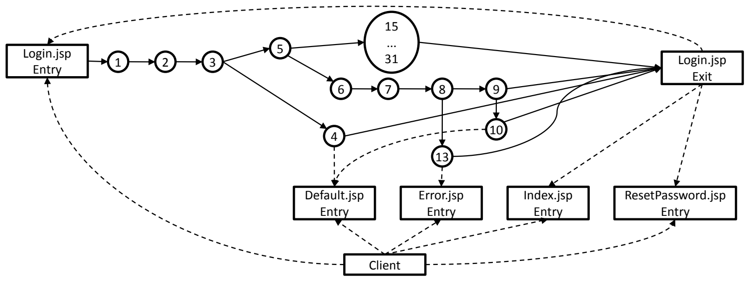

Fig. 3. Inter-component control flow graph for servlet `Login.jsp` shown in Figure 1.

to receive requests. For example, in the Java Enterprise Edition (JEE), a component must implement one of a set of specific interfaces. These conditions can be checked via static analysis. If the component satisfies the conditions, an inter-component control flow edge that originates from the user and connects to the component entry points is identified. Once an edge is identified, the approach attempts to refine the information by determining the type of HTTP request method encoding (e.g., GET or POST) required to access the component. In certain frameworks, such as JEE, the HTTP request method indicates which procedure will be treated as the root method. For example, the presence of a `doPost` method implies that the component can handle POST requests. In PHP, the name of the global variable used to access input parameters indicates the expected request method. If it is possible to identify the request method, then the added edges are updated to include an annotation specifying the request method.

Example The example presented in Figure 1 has several edges related to *Direct Entry* control flow. The first of these is an edge annotated with the "POST" request method that runs from the user to the entry node of `Login.jsp`. This is created because `Login.jsp` implements the JEE servlet interface method "doPost." This edge is of the form $\langle$User, `Login.jsp`, "Direct", 0$\rangle$, where the linenumber is not defined. Additionally, there are four other components identified as targets in in Sections 4.1 and 4.2: `Default.jsp`, `Error.jsp`, `Index.jsp`, and `ResetPassword.jsp`. For the purpose of illustration, we assume that these are analyzable by the approach and also implement the necessary entry points, which leads to edges being added from the user to each of their entry points.

4.4 Combining Control flow Information

The Inter-Component Control-Flow Graph (ICCFG) includes inter-component and intra-component control flow. Intra-component control flow can be identified by standard techniques for building control flow graphs. Inter-component control flow edges that have a non-zero line number associated with them are added by specifying that the source of the edge is the node that corresponds to the line number in the source component and connecting it to the edge's destination. Destination edges are connected to the entry node of the target component. In the case of inter-component control flow edges for which there is a zero or undefined line number, their source is the exit point of the source component and the destination is determined the same as edges with defined line numbers. For any edge with the user as the source, a “Client” node is created in the graph and all such edges’ source runs from this node to the target component. Control flow edges that are derived from *Component Inclusion* are handled slightly differently in terms of their connection to specific nodes in the traditional control flow graph. The reason for this is that the naive approach to connecting these edges leads to the generation of a cycle at the node that performs the *Component Inclusion*. This would happen because both the outgoing edge and returning edge would be connected to the same node in the source component. To address this, the source node n is split into two corresponding nodes n_{call} and n_{return}. The outgoing edge to the included component is connected at its source to n_{call} and its destination to the entry node of the included component. The corresponding return edge is connected at its source to the exit of the included component and its destination to n_{return}.

The ICCFG for servlet `Login.jsp` is shown in Figure 3. This graph includes the traditional intra-component control flow of the component as solid lines and the inter-component control flow edges as dotted lines. Due to space constraints the control flow edges that go from nodes 4, 10, and 13 to method `sendHttpCmd` are omitted. Additionally, the control flow graphs for `Default.jsp`, `Error.jsp`, `Index.jsp`, and `ResetPassword.jsp` are not included in this example.

Table 1. Subject applications, analysis time, and inter-component edge count. Techniques are Crawler (C), HTML Only (H), MiMoSA (M), SXS (S), and ICE (I).

Application	LOC	Classes	Servlets	Time (s)					Edge Count				
				C	H	M	S	I	C	H	M	S	I
Bookstore	19,402	28	27	440	248	660	660	660	192	118	118	368	415
Classifieds	10,702	18	18	3,389	156	464	464	464	104	78	78	174	198
Daffodil	18,706	119	70	9	1,088	1,482	1,482	1,482	31	92	96	96	101
Employee Dir.	5,529	11	9	246	105	282	282	282	10	35	35	51	65
Events	7,164	13	12	288	248	346	346	346	51	46	46	79	91
Filelister	8,671	41	10	6	90	191	191	191	7	19	19	19	160
Portal	16,089	28	27	2,486	262	755	755	755	294	116	116	491	517
Webmail	17,078	81	24	2,822	1,373	1,900	1,900	1,900	56	59	59	59	76

5 Evaluation

This section presents the results of an empirical evaluation of the approach. For the evaluation, the author implemented the analysis in a prototype tool, ICE (Inter-component Control-flow Extractor). The accuracy and runtime cost of ICE was compared against web crawling and three static analysis based approaches. The research questions are: **RQ1:** How long does it take to analyze the subject applications using each of the evaluated approaches? **RQ2:** What is the precision of each approach? **RQ3:** What is the recall of each approach?

5.1 Experiment Setup

Subject Applications For the evaluation, a set of eight subjects were analyzed. Details of the applications are shown in Table 1. All of the applications are available as open source. They were chosen because their implementations are a mix of static HTML, JavaScript, Java servlets, and regular Java code.

Implementation of Analyses ICE was compared against web crawling and three static analysis based techniques. The static analysis based techniques were reimplemented for the evaluation since their original implementations were for PHP based web applications or were still at a prototype stage. Although none of the considered approaches were originally intended for control flow identification, they extract similar types of information and represent the most closely related approaches known to the author.

ICE: The implementation of ICE is in Java and leverages several previously developed program analyses: Soot, for generating control flow graphs; Indus, for data flow information; and JSA for evaluating string expressions. HTMLParser was used to parse the string representing the web pages and JavaScript was analyzed using Rhino.

Crawler: Approaches based on web crawling are well-known and widely-used for analyzing web applications. For this evaluation, two crawling based approaches were combined, CrawlJax [19], a state of the art crawler for AJAX based web applications, and a generic Spider based on VeriWeb [2]. Both approaches are used because preliminary results indicated that they were highly complementary; CrawlJax was better at finding control flow information related to JavaScript commands and Spider was better at finding control flow information that was not represented as a "clickable" unit in the web page. For both approaches, input specifications that allowed them to navigate login screens was provided. For CrawlJax a custom list of "clickable" elements was also provided for each subject application.

HTML Only: This technique is representative of several static analysis based approaches that analyze a web application and identify HTML output [5, 20, 26]. The computed HTML output is then parsed for control flow constructs. Although it is clear that these techniques will not be able to identify other forms of control flow, it is included since it represents a widely used approach.

SXS: Sun and colleagues propose a static analysis based technique for identifying access control vulnerabilities in PHP based web applications [23]. As part of this technique, they construct a sitemap that models several types of elements that are also relevant for constructing the ICCFG. Their approach handles some *HTTP Commands* related constructs, *Dynamically Generated HTML*, *Component Inclusion*, and *Direct Entry*, but not *JavaScript*.

MiMoSA: Balzarotti and colleagues propose a static analysis based technique for identifying multi-module vulnerabilities in PHP based web applications [4]. As with SXS, the authors must identify certain types of control flow in the web applications in order to discover these vulnerabilities. This control flow modeling includes limited support for *HTTP Commands*, heuristics for links generated by *JavaScript*, *Dynamically Generated HTML*, and *Direct Entry*. There is no support for *Component Inclusion*.

5.2 Experiments

To collect the experiment data, each of the five approaches was run on the eight subject applications. The machine used to run the results was an Intel Core i7@2.8Ghz with 8GB DDR3 RAM running Ubuntu 10.10 with 2GB RAM dedicated to the JVM heap. For each run, Table 1 shows the time for the analysis to execute ("Time") and the size of the edge set ("Edge Count"). Each approach is abbreviated by the first letter of its name.

5.3 Discussion of Results

The timing results in Table 1 show that analysis time varied significantly by application and analysis. Several of the worst run times were generated by Crawler. Even though web crawling is generally a fast technique, Crawljax loads and renders every crawled page, which incurs a high overhead, but is necessary to accurately model the effect of JavaScript. Full static analysis of daffodil and webmail was also very expensive. Investigation of the two applications showed that several large servlets represented pathological worst cases for the analysis, with almost every other line either a nested branch or output generating statement. Overall though, the results are positive for ICE. For 6/8 applications the runtime was under fifteen minutes and for 2/8 it was under thirty-two minutes. Note that SXS, MiMoSA, and ICE have the same runtime because they use the same implementations of the algorithms in Section 4.1 and 4.2.

Table 2. Recall (%) of the considered techniques: Crawler (C), HTML Only (H), MiMoSA (M), SXS (S), and ICE (I).

Application	Recall				
	C	H	M	S	I
Bookstore	40	27	27	88	100
Classifieds	50	37	37	87	100
Daffodil	5	91	95	95	100
Empl. Dir.	14	52	52	77	100
Events	56	48	48	86	100
Filelister	1	10	10	10	100
Portal	54	21	21	95	100
Webmail	32	78	78	78	100
Average	32	46	46	77	100

The edge count results in Table 1 show that ICE, followed by SXS, consistently had the highest discovered edge count. The Crawler and the two other static analysis approaches each had higher edge counts for half of the subjects. The primary contributing factor to this was whether the application made extensive use of *Component Inclusion*. For Crawler it was trivial to discover these edges because they were present in the crawled page. However, HTML and MiMoSA did not consider the semantics of *Component Inclusion* in their analysis. Bookstore, Classifieds, Events, Employee Directory, and Portal used *Component Inclusion*, which led to Crawler having a higher edge count for four of them. Employee Directory made very limited use of *Component Inclusion* and many of these edges were actually redundant with edges already in the including page.

Table 2 shows recall results for the approaches. ICE had perfect recall; of the remaining approaches, SXS had the highest recall followed by a tie between HTML and MiMoSA, and Crawler last. To explain these results, the distribution of the edges over the different control flow types was analyzed. The primary differentiators between ICE and SXS was that ICE could handle all types of HTTP requests, whereas SXS could only handle HTTP requests related to the 302 response code. The differentiator between SXS and HTML/MiMoSA was primarily SXS' ability to handle *Component Inclusion*. Although MiMoSA is able to handle limited forms of *JavaScript* and *HTTP Commands* control flow, neither of the specific constructs it could handle were prevalent in the applications' code, which explains why there was very little difference between the two approaches. Lastly, there were several reasons for Crawler's low recall. The primary reasons was that many pages required the Crawler to interact with it in specific ways in order to reveal additional response behaviors. Since the Crawler could not randomly guess this, it was generally unable to access these pages. Filelister was particularly low for all approaches, except ICE, because almost 88% of its inter-component control flow was done via *HTTP Commands*. For all applications, all approaches were able to achieve 100% precision.

Overall, the results were very positive for ICE. It was able to discover a more complete set (i.e., higher recall) of control flow than all other approaches, and the runtime of ICE was comparable to the other approaches.

5.4 Threats to Validity

External validity is concerned with whether the results of this evaluation could generalize to other web applications. The primary threat to this validity is that the subject applications do not use as much JavaScript as AJAX based web applications. This threat is mitigated by the fact that most of the JavaScript complexity in AJAX applications is *intra* and not *inter* component control flow. Regardless, most of this control flow would still be discovered by the technique outlined in Section 4.1 for *JavaScript*. The presence of this type of control flow would not change the results of ICE versus the other static analysis approaches, but could increase the relative number of edges found by Crawler. Also, even though the subject applications are all written in Java, the approach would be generally applicable to other web application frameworks and languages, such as PHP, Perl, or .NET, since all of these provide analogous APIs for sending HTTP commands and generating HTML content.

Internal validity addresses whether the conclusions about ICE's performance can be made based on the experiment design. The primary threat is that the techniques compared against were reimplemented for the study. To reduce this threat, the author made optimistic assumptions about the capability of the other techniques. Case in point, it was assumed that HTML, SXS, and MiMoSA could compute HTML pages as well as ICE and could be extended to handle the broader range of HTML constructs identified in this paper. This means that the performance of SXS and MiMoSA is higher than would be expected in practice because neither technique could properly handle object oriented code in PHP, and MiMoSA, as defined in the original paper, cannot safely handle dynamically generated HTML or HTTP messages.

6 Related Work

Early approaches for identifying control flow were based on the use of web crawlers that traversed the links of a web page, discovering web pages as they went [2, 15, 24]. More recent approaches add support for the interpretation of client-side JavaScript [18]. However, since they only interact with the web application via its generated HTML pages, they cannot offer any guarantees of completeness with regard to server-side control flow, such as *HTTP Commands* or *Component Inclusion*. Furthermore, it is common for web applications to only display certain pages after interactions that meet specific constraints. Therefore, it is likely the approaches could be incomplete with respect to *Dynamically Generated HTML* and *JavaScript* as well. Other approaches have proposed the use of captured user session data to build models of the target web application [7, 14]. However, they can only model portions of the code that have been exercised by users, and would be incomplete with respect to the complete behavior of the web application.

Another large group of approaches uses specifications provided by the developer [1, 11, 13, 16, 21]. These specifications are typically provided using a formal language, such as UML or state-based models. These approaches allow developers to capture the *intended* control flow semantics of web applications. The drawback of manual specification is that the intended and actual control flow can differ. Furthermore, the development of complete and precise manual specifications for large web applications can be very time-consuming. Other approaches have proposed the development of new languages and frameworks that make much of the implicit control flow of web applications explicit in the structure and semantics of the language [6, 17], but require developers to learn a new language and web application framework.

Other researchers have also proposed the use of static analysis to identify elements of web applications related to control flow. Deng, Frankl, and Wang proposed an early technique that used static analysis to identify link targets and paths through a web application [5]. As compared to the proposed approach, their technique could only be used to discover control flow related to a subset of *Dynamically Generated HTML*. Tonella and Ricca proposed an approach that could identify dynamically generated object programs [26]. This information was used to build web application system dependence graphs that accounted for certain types of control and data flow. As compared to their approach, the proposed approach takes into account a larger set of control flow related constructs on the client and server-side, such as JavaScript and HTTP redirects, and has a more precise method of determining string values based on method summarization and string analysis.

There is also an extensive amount of research that uses control flow related information to verify and test web applications. The proposed approach complements this body of work by providing a more complete mechanism for identifying control flow. For example, several approaches use control flow models to verify or enforce web application behaviors [11–13, 27]. Security related approaches could also benefit from the automated control flow generation to more completely check properties related to session handling [6]. Lastly, other approaches use a web crawling based approach to build control flow models of web applications for testing and slicing [22, 25] and the use of the proposed approach could increase the effectiveness of these techniques.

7 Conclusion

This paper presents a new technique for automatically identifying control flow in web applications. The technique is based on static analysis and analyzes each component of a web application to identify a wide range of of control flow types. The identified control flow is combined into a new representation, the Inter-Component Control-Flow Graph, which shows both traditional and inter-component control flow. The proposed approach was evaluated in terms of its runtime cost and accuracy of the identified control flow and compared against those achieved using a web crawling based approach and other static analysis based approaches. The results were positive; the proposed approach had a higher level of recall, and precision and runtime costs comparable to the other approaches. Overall, the results indicate that the approach is useful for accurately identifying web application control flow and, as such, could be used to help to improve testing and verification techniques for web applications that require control flow information.

References

1. Andrews, A.A., Offutt, J., Alexander, R.T.: Testing Web Applications by Modeling with FSMs. Software Systems and Modeling 4(3), 326–345 (July 2005)
2. Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically Testing Dynamic Web Sites. In: Proceedings the International World Wide Web Conference (May 2002)
3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Proceedings of the International Static Analysis Symposium. pp. 1–18 (June 2003)
4. D.Balzarotti, M.Cova, V.Felmetsger, G.Vigna: Multi-module vulnerability analysis of web-based applications. In: Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS). (October 2007)
5. Deng, Y., Frankl, P., Wang, J.: Testing Web Database Applications. SIGSOFT Software Engineering Notes 29(5), 1–10 (2004)
6. Desmet, L., Verbaeten, P., Joosen, W., Piessens, F.: Provable protection against web application vulnerabilities related to session data dependencies. IEEE Transactions on Software Engineering 34(1), 50–64 (2008)
7. Elbaum, S., Rothermel, G., Karre, S., II, M.F.: Leveraging User-Session Data to Support Web Application Testing. Transactions On Software Engineering 31(3), 187–202 (March 2005)
8. Halfond, W.G.J.: Automated checking of web application invocations. In: Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE) (November 2012)
9. Halfond, W.G.J., Anand, S., Orso, A.: Precise Interface Identification to Improve Testing and Analysis of Web Applications. In: Proceedings of the International Symposium on Software Testing and Analysis. pp. 285–296. (July 2009)

10. Halfond, W.G., Orso, A.: Automated Identification of Parameter Mismatches in Web Applications. In: Proceedings of the Symposium on the Foundations of Software Engineering (November 2008)
11. Halle, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010) (September 2010)
12. Han, M., Hofmeister, C.: Modeling and verification of adaptive navigation in web applications. In: In Proceedings of the 6th International Conference on Web Engineering. pp. 329–336. ACM Press (2006)
13. Han, M., Hofmeister, C.: Relating navigation and request routing models in web applications. In: Engels, G., Opdyke, B., Schmidt, D., Weil, F. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 4735, pp. 346–359.
14. Haydar, M.: Formal framework for automated analysis and verification of web-based applications. In: Proceedings of the 19th IEEE international conference on Automated software engineering. pp. 410–413. (2004),
15. Huang, Y., Huang, S., Lin, T., Tsai, C.: Web Application Security Assessment by Fault Injection and Behavior Monitoring. In: Proceedings of the International World Wide Web Conference. pp. 148–159 (May 2003)
16. Jia, X., Liu, H.: Rigorous and Automatic Testing of Web Applications. In: Proceedings of the International Conference on Software Engineering and Applications. pp. 280–285 (November 2002)
17. Licata, D., Krishnamurthi, S.: Verifying Interactive Web Programs. In: Proceedings of the International Conference on Automated Software Engineering. pp. 164–173 (September 2004)
18. Mesbah, A., Bozdag, E., van Deursen, A.: Crawling Ajax by Inferring User Interface State Changes. In: Schwabe, D., Curbera, F., Dantzig, P. (eds.) Proceedings of the International Conference on Web Engineering. pp. 122–134. (July 2008)
19. Mesbah, A., van Deursen, A.: Invariant-Based Automatic Testing of Ajax User Interfaces. In: Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers. pp. 210–220. (May 2009)
20. Minamide, Y.: Static Approximation of Dynamically Generated Web Pages. In: Proceedings of the International World Wide Web Conference. pp. 432–441 (May 2005)
21. Ricca, F., Tonella, P.: Analysis and Testing of Web Applications. In: Proceedings of the International Conference on Software Engineering. pp. 25–34 (May 2001)
22. Ricca, F., Tonella, P.: Web Application Slicing. Proceedings of the International Conference on Software Maintenance, (2001)
23. Sun, F., Xu, L., Su, Z.: Static detection of access control vulnerabilities in web applications. In: Proceedings of the USENIX Security Symposium (Aug 2011)
24. Tonella, P., Ricca, F.: Dynamic Model Extraction and Statistical Analysis of Web Applications. In: Proceedings of the Fourth International Workshop on Web Site Evolution. pp. 43–52 (October 2002)
25. Tonella, P., Ricca, F.: A 2-Layer Model for the White-Box Testing of Web Applications. In: Proceedings of the International Workshop Web Site Evolution. pp. 11–19. (2004)
26. Tonella, P., Ricca, F.: Web Application Slicing in Presence of Dynamic Code Generation. Automated Software Engineering 12(2), 259–288 (2005)
27. Yang, J., Huang, J., Wang, F., Chu, W.: Constructing Control-Flow-Based Testing Tools for Web Application. In: Proc. of the 11th Software Enginnering and Knowledge Enginnering Conference (SEKE) (1999)