Automated Detection of Client-State Manipulation Vulnerabilities

ANDERS MØLLER and MATHIAS SCHWARZ, Aarhus University

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

We present a static analysis for the widely used frameworks Java Servlets, JSP, and Struts. Given a web application archive as input, the analysis identifies occurrences of client state and infers the information flow between the client state and the shared application state on the server. This makes it possible to check how client-state manipulation performed by malicious users may affect the shared application state and cause leakage or modifications of sensitive information. The warnings produced by the tool help the application programmer identify vulnerabilities before deployment. The inferred information can also be applied to configure a security filter that automatically guards against attacks at runtime. Experiments on a collection of open-source web applications indicate that the static analysis is able to effectively help the programmer prevent client-state manipulation vulnerabilities. The analysis detects a total of 4,802 client-state parameters in the 10 applications, whereof 4,437 are classified as safe and 241 reveal exploitable vulnerabilities.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Languages, Verification

Additional Key Words and Phrases: Web application security, information flow analysis, static analysis

ACM Reference Format:
Anders Møller and Mathias Schwarz 2014. Automated detection of client-state manipulation vulnerabilities. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 29 (August 2014), 30 pages.
DOI: http://doi.acm.org/10.1145/2531921

1. INTRODUCTION

Errors in web applications are often critical. To protect web applications against malicious users, the programmers must be aware of numerous kinds of possible vulnerabilities and countermeasures. Among the most popular guidelines for programming safe web applications are those in the OWASP Top 10 report that covers "the 10 most critical web application security risks" [Open Web Application Security Project 2010]. Many security properties depend on the flow of untrusted data in the programs. This flow is often not explicit in the program code, so it can be difficult to ensure that sensitive data is properly protected. Although tool support exists for detecting and preventing some risks, manual code review and testing remain crucial to ensure safety.

This work was supported by Google, IBM, and the Danish Research Council for Technology and Production.
Authors' address: Department of Computer Science, Aarhus University, Aabogade 34, Aarhus, Denmark.

DOI 10.1145/2531921 http://doi.acm.org/10.1145/2531921

However, code review and testing are tedious and error-prone means, so it is desirable to identify classes of vulnerabilities that are amenable to tool support.

As an example, consider the category *A4 - Insecure Direct Object References* from the 2010 OWASP Top 10 list. A direct object reference is a reference to an internal implementation object, such as a database record, that is exposed to the user as a form field or a URL parameter in an HTML document. Such references are examples of *client state*, which is used extensively in web applications to work around the statelessness of the HTTP protocol, for example, to store session state in the HTML documents at the clients.

Figure 1 shows two excerpts of source code from a web application named *JSPChat*[1]. Part (a) shows a JSP page containing an HTML form for saving personal information in a chat service, and part (b) shows the servlet code that is executed when the form data is submitted by the user. The first thing to notice is that the `nickname` form field on line 20 in the JSP page functions as a direct object reference that refers to a `ChatRoom` object and a `Chatter` object stored on the server. As the application programmer cannot trust that the user does not modify such references in an attempt to access resources that belong to other users, it is important to ensure that object references are protected. This can be done, for example, using a layer of indirection (i.e., using a map stored on the server from client-state values to the actual object references), via cryptographic signatures or encryption of the client state, or by checking that the user is authorized to access the resources being referenced in the requests. This is, however, easy to forget when programming the web application. In the example, the servlet reads the `nickname` parameter, stores it in a field in the servlet object, and then uses it – without any security measures – to look up the corresponding `ChatRoom` and `Chatter` objects in the shared application state on lines 47 and 49. Obviously, a malicious user could easily forge the parameter value and thereby access another person's data. (The careful reader may have noticed another vulnerability in the program code; we return to that in Section 7.)

As documented in security alerts and reports by, e.g., ISS [Internet Security Systems 2000], MSC [Brussin 1998], Advosys [Advosys Consulting 2000], and Sanctum [ZDNet 2001], vulnerabilities of this kind have been known—and exploited—for more than a decade. A study of several hundred penetration test reports conducted by Imperva between 2000 and 2003 placed it as the most common kind of web application vulnerability [Cerf and Shulman 2004]. However, it remains widespread, as evident from the 2010 OWASP report. A recent study shows that web application developers are still unaware of common classes of related vulnerabilities, despite awareness programs provided by, for example, OWASP, MITRE, and SANS Institute [Scholte et al. 2011]. A notable recent example of client-state manipulation is the attack on the Citigroup website that allowed hackers to disclose account numbers and transaction history for 200,000 credit cards [Moran 2011].

According to OWASP, "automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe". Nevertheless, in this article we show that it is possible to develop automated tools that can detect many of these flaws. Our approach is based on a simple observation: *Vulnerability involving client-state manipulation is strongly correlated to information flow from hidden fields or other kinds of client state to operations involving the shared application state on the server.* This approach is along the lines of previous work on static taint analysis [Livshits and Lam 2005; Tripp et al. 2009], however, with crucial differences in how we characterize the sources and sinks of the information flow. We discuss related work in Section 8.

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

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

(a) editInfo.jsp

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

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

(b) SaveInfo.java

Fig. 1. A simplified version of the *JSPChat* web application.

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

— Our starting point is a characterization of *client-state manipulation vulnerabilities* (Section 2) that has considerable overlap with category A4 from the OWASP 2010 list of the most critical risks. In particular, we describe safety conditions under which the use of client state is likely not to cause vulnerabilities.
— Based on this characterization, we present an automated approach to detecting occurrences of client state in a given web application and to checking whether the safety conditions are satisfied (Sections 3–6).
— Through experiments performed on 10 open-source web applications with a prototype implementation of our analysis, we show that the approach is effective for helping the programmer detect client-state manipulation vulnerabilities (Section 7). On a total of 1,575 servlets, JSP pages, and Struts actions, our tool identifies 4,802 possible occurrences of hidden fields and other client-state parameters. After customization, the tool classifies 4,437 of these occurrences as safe. Of the 365 warnings being produced, 241 cases reveal exploitable vulnerabilities involving 59 different field names.

The static analysis that underlies our automated approach to detecting client-state manipulation vulnerabilities consists of three components. The first component (Section 4) infers the dataflow between the individual servlets and pages that constitute the application in order to identify the *client-state parameters*. This requires a static approximation of the dynamically constructed output of the servlets and pages and extraction of relevant URLs and parameter fields in forms and hyperlinks. The second component (Section 5) analyzes the program code to find out which objects represent *shared application state*, i.e., server state that is persistent or shared between multiple clients, as opposed to session state or transient state. The third component (Section 6) performs an information flow analysis to identify the possible flow of user-controllable input from client-state parameters to shared application state objects. The vulnerability warnings being produced by the tool are useful for guiding the programmer to apply appropriate countermeasures by modifying the application source code. Alternatively, the information obtained by the analysis can be used for automatically configuring a security filter that guards against client-state manipulation attacks at runtime (Section 8.2).

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

2. CLIENT-STATE MANIPULATION VULNERABILITIES

In a web application, *client state* comprises information that is stored within a dynamically generated HTML document in order to be transmitted back to the server at a subsequent interaction, for example, when a form is submitted. Since the HTTP protocol is stateless, client state is widely used for keeping track of users and session state that involve multiple interactions between each client and the server. Storing session state at the client instead of at the server can have several benefits. Most importantly, it decreases the load on the server and avoids the need for a session-state expiration mechanism. Client state appears as hidden fields in HTML forms (as in the example in Section 1) and as URL query parameters in hyperlinks. Such state is not intended to be modified by the user, but nothing prevents malicious users from doing so, and this is easy to forget when programming web applications. A related situation occurs

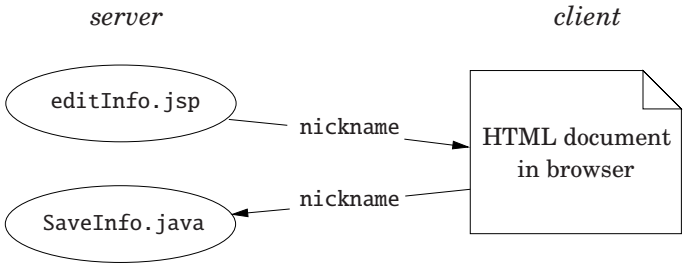

Fig. 2. Dataflow of client state from a JSP page to a servlet via an HTML document. The `nickname` parameter is sent from `editInfo.jsp` to `SaveInfo.java` via the HTML document.

with HTML select menus, radio buttons, and checkboxes, which also contain fixed sets of values that the user is intended to choose from. We commonly refer to HTTP GET/-POST request parameters that contain such state as *client-state parameters*. Cookies provide a related mechanism; in this article, we focus on ordinary HTTP request parameters, but our approach in principle also works for cookies.

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

Figure 2 illustrates the dataflow of client state for the *JSPChat* example. The value of `nickname` is passed as client state from a JSP page, `editInfo.jsp`, to a servlet, `SaveInfo.java`, using a hidden field in the HTML document.

The following characterization is our key to automating detection of the vulnerabilities we consider: A web application is vulnerable to *client-state manipulation* if users, by modifying client state, can gain additional capabilities to access or change shared application state. Note that by this definition, if all parts of the application state that can be accessed or changed by modifying client state can also be accessed by other means, for example, via another page in the application, it is not considered vulnerable to this kind of attack.

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

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

1) The client-state parameter value stored in the HTML document is encrypted using a key that is private to the server. The server then decrypts the value when it is returned. A variant is to leave the value unencrypted but add an extra hidden field or URL parameter containing a digital signature (or MAC, message authentication code) computed from the client-state value and the server's private key. The server then verifies that the client-state value is unaltered by checking the

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

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

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

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

Fig. 3. Weakness categories from MITRE and OWASP that are related to client-state manipulation.

signature when the form data is returned. To prevent against replay attacks and impersonation attacks, a timestamp and a client ID can be included in the encryption or signature generation. A drawback of this approach is that extra work is needed when producing and receiving the client state.

2) The client state entirely consists of large random values that are practically impossible to predict by attackers. A typical example is the use of *session IDs*: in many web applications, all session state is stored on the server, and the only client state being used consists of session IDs, that is, references to the session state on the server. A drawback of this approach is that it requires extra space on the server to store the session state.
3) An indirection is used. The client state consists of, for example, only numbers between 1 and some small constant, and these numbers are then mapped to the actual application state on the server. This approach is particularly useful for select menus, radio buttons, and checkboxes. In this way, client-state manipulation cannot provide access to data beyond what is accessible from this map. A drawback of this approach is the burden involved in maintaining the indirection map.
4) The client-state parameter is treated as untrusted input, no different from other kinds of parameters, and any access to application state involving the given client-state value is guarded by an authorization check.
5) Finally, a sufficient condition for safety according to the preceding definition is that all the shared application state that can be accessed through client-state manipulation is already available by other means, that is, the information should not be considered sensitive.

We see uses of several of these techniques in the web applications *Hipergate*, *Pebble*, and *JWMA* that we study in Section 7. OWASP's ESAPI[4] library contains support for implementing techniques 1–4. As an example, to apply the ESAPI encryption approach to the *JSPChat* web application from Figure 1, the programmer would wrap the expression `chatter.getName()` on line 21 into a call to `ESAPI.httpUtilities().encryptHiddenField(...)` and insert a matching call to `decryptHiddenField` on line 39. The `decryptHiddenField` method throws an `IntrusionException` if tampering is detected. In .NET, the `LosFormatter`[5] class pro-

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

vides support for MAC protection of client state (or view state, as it is called in .NET). The use of encryption and signatures to prevent manipulation of hidden form fields was originally suggested by MSC [Brussin 1998] and Advosys [Advosys Consulting 2000]. Thus, the countermeasures are well known; the goal of our analysis is to detect when they are applied inadequately.

3. OUTLINE OF THE ANALYSIS

We adapt the well-known approach to static information flow analysis for identifying the possible dataflow from *sources* to *sinks* that does not pass through *sanitizers* [Huang et al. 2004; Livshits and Lam 2005; Tripp et al. 2009; Wassermann and Su 2008; Xie and Aiken 2006; Jovanovic et al. 2010].

— The sources in our setting are the locations in the code where client-state parameters are read. With common web application frameworks, such as Java Servlets, JSP, and Struts, it is not explicit in the application source code which parameters contain client state, so we need a static analysis to infer this information.
— The sinks are the operations in the source code that involve shared application state. We conservatively assume that this application state is not accessed by other means (cf. condition 5 in Section 2). This assumption may lead to false positives, which we consider experimentally in Section 7. As is it not explicit in the source code which operations involve shared application state (compared to session state or transient state), we need another static analysis component to extract this information.
— The sanitizers correspond to the various kinds of protection described in Section 2. For example, decrypting an encrypted client-state value is one kind of sanitization.

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

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

It is, in principle, possible for a programmer to specify incorrect customization rules, which may affect soundness of the analysis output. However, we trust that the customization rules are correct. The purpose of our analysis is to alert the programmer to potential vulnerabilities, not to formally verify correctness of, for example, low-level operations for digital encryption or MAC checking. Moreover, the customization mechanism is easy to use, as we shall see in Sections 6 and 7.

One approach to remedy vulnerabilities detected by the analysis is that the programmer manually incorporates appropriate countermeasures into the web application source code, as discussed in Section 2. Another option is to feed the vulnerability report to a security filter, which we describe in Section 8.2, for automatic protection. Our experiments (Section 7) indicate that the burden of the customization step is manageable. However, we note that a fully automatic approach to protect against client-state manipulation can be obtained by omitting customization entirely and applying the security filter without having eliminated false positives. Compared to the more manual approach involving customization, the price is a modest runtime overhead incurred by the security filter, since it may protect some client state unnecessarily.

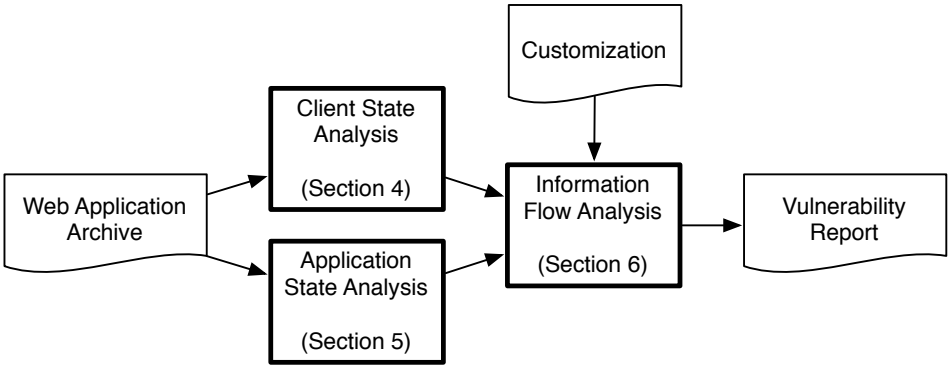

Fig. 4. Structure of the analysis.

The following sections explain how we identify client-state parameters and application state and perform the information flow analysis. The structure of the combined analysis is illustrated in Figure 4.

4. IDENTIFYING CLIENT STATE

When a page p reads an HTTP parameter, for example, on lines 39 and 41 in Figure 1, the only way we can find out whether that is a client-state parameter is to analyze every page q of the application that dynamically constructs HTML documents with links or forms referring to p. Specifically, we need to recognize the construction of the hidden field named `nickname` on line 20 in `editInfo.jsp`, and via the `action` attribute of the `<form>` element in `editInfo.jsp`, establish the connection from `editInfo.jsp` to line 39 in the servlet `SaveInfo.java`.

The goal of this phase is to analyze every page in the application to find out which client-state parameters appear in the generated HTML documents and which references exist to other pages. For each page q in the web application, we first generate a context-free grammar G_q that conservatively approximates the set of HTML documents that may be generated by q. This can be done as in our previous work on analysis of dynamically generated HTML documents [Møller and Schwarz 2011]. To find the client-state parameters in the HTML documents generated by q, we identify all elements that define hidden fields, select boxes, radio buttons, and links in G_q and collect the corresponding parameter names. Since these names may be generated dynamically in the program, we approximate them conservatively by a regular language $C_{out}(q)$. In the *JSPChat* example from Figure 1, this step identifies `nickname` as the only client-state parameter originating from `editInfo.jsp`, thus $C_{out}(\texttt{editInfo.jsp}) = \{\texttt{nickname}\}$. We also infer the references between the pages by identifying `href` attributes in `<a>` elements and `action` attributes in `<form>` elements in G_q. This results in a map S that holds the set of possible successor pages for each page. For example, $\texttt{SaveInfo.java} \in S(\texttt{editInfo.jsp})$. Section 4.1 explains in more detail how to infer this information from the web application code.

Combined with information extracted from the deployment descriptors (`web.xml` in Servlets and `struts.xml` in Struts), this results in a *page graph* in which nodes correspond to pages and edges correspond to S, describing the possible links and form actions. In the example in Figure 1, this step identifies the edge from `editInfo.jsp` to `SaveInfo.java`. Figure 5 shows the page graph for the six pages in *JSPChat* that involve flow of client state.

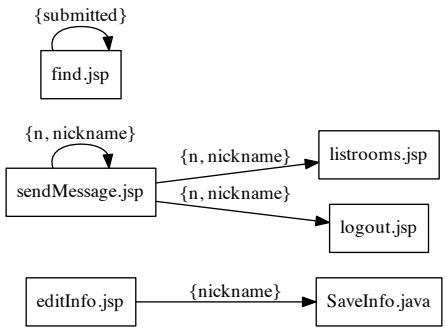

Fig. 5. The automatically constructed page graph for the parts of *JSPChat* that involve client state.

The names of the incoming client-state parameters to a page p can now be expressed as $C_{in}(p) = \cup_{q_i} C_{out}(q_i)$ for each page q_i where $p \in S(q_i)$. For the example, we get $C_{in}(\texttt{SaveInfo.java}) = \{\texttt{nickname}\}$. This tells us that when `SaveInfo.java` reads the `nickname` parameter, as in `request.getParameter("nickname")` on line 39 in Figure 1, that should be treated as a source in the subsequent information flow analysis, in contrast to the `email` parameter that is read on line 41. Section 4.2 explains more generally how to use C_{in} for locating operations in the application code that read client-state parameters.

The result of these steps is a set of method calls in the application code that will serve as sources of client-state values in the information flow analysis in Section 6. All of the steps can be done soundly in the sense that every call to `getParameter` and related operations that may return client state is always included in the statically inferred set of client-state value sources. In our experiments, we never observe any imprecision of this phase.

4.1. Analyzing HTML Output

As previously outlined, we analyze the source code of the web application to find the hidden fields, URL parameters, links, and form actions that may appear in the generated HTML pages. With Java Servlets, output is generated by printing string fragments to an output stream. JSP and Struts compile to Servlets, so we can handle each of these frameworks by focusing on Servlets. We assume that the HTML documents being generated do not use JavaScript code in ways that interfere with forms and links. Starting from the Java bytecode of the web application, the first step is to construct an *output stream flow graph*, which is a representation of the program that abstracts away everything not directly relevant for generating output to the output stream. The notion of output stream flow graphs originates from previous work [Kirkegaard and Møller 2006; Møller and Schwarz 2011]; here we give a more formal description, including a precise description of the connection to context-free grammars.

An output stream flow graph F is a directed graph given as a tuple (N, E, C, L).

— N is a finite set of nodes, divided into three disjoint subsets.
  — N_{append} are *append nodes* representing instructions that print strings to the output stream.
  — N_{invoke} are *invoke nodes* corresponding to method calls.
  — N_{return} are *return nodes* corresponding to method returns.
— $E \subseteq (N_{\mathsf{append}} \cup N_{\mathsf{invoke}}) \times N$ is a set of *intra-procedural* edges.
— $C \subseteq N_{\mathsf{invoke}} \times N$ is a set of *call* edges.
— $L : N_{\mathsf{append}} \to \mathcal{R}$ gives a regular string language (represented by a regular expression or a finite-state automaton over the Unicode alphabet) for every append node.

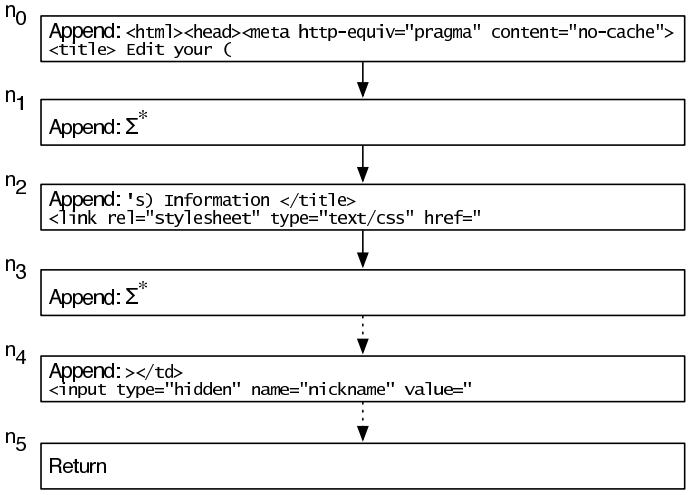

Fig. 6. Excerpt from the output stream flow graph for `editInfo.jsp`. Dashed edges indicate places where additional nodes exist in the full graph.

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

Our implementation constructs output stream flow graphs from Java bytecode using the Soot program analysis framework [Vallee-Rai et al. 1999] and the JSA string analysis tool [Christensen et al. 2003; Feldthaus and Møller 2009]. From Soot, we use the Jimple intermediate representation, the built-in class-hierarchy analysis for obtaining call graphs, and the Spark points-to analysis to find the operations that affect the HTTP output stream. JSA gives us the regular string languages for the invoke nodes. The resulting output stream flow graph has one entry node in F for each page in the web application.

Figure 6 shows a part of the output stream flow graph for `editInfo.jsp`. Since there are no control-flow structures in the source code of the page, the graph becomes linear and contains an append node for each operation that may emit output to the client. The hidden field `nickname` is generated by the append node n_4. Also notice that the analysis approximates the possible value of `chatter.getName()` from line 8 in Figure 1 as any string, Σ^*. In the subsequent analysis, we choose to ignore the fact that this may in principle alter the syntactic structure of the HTML document, which may cause invalid HTML as well as other kinds of vulnerabilities.

A *context-free grammar* G is a tuple (V, Σ, s, P) where the following hold.

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

$v_0 \rightarrow$ <html><head><meta http-equiv="pragma" content="no-cache">
<title> Edit your (v_1
$v_1 \rightarrow \Sigma^* v_2$
$v_2 \rightarrow$'s) Information </title>
<link rel="stylesheet" type="text/css" href="$\ldots v_3$
$v_3 \rightarrow \Sigma^* v_4$
$v_4 \rightarrow$ ></td>
<input type="hidden" name="nickname" value="$\ldots v_5$
$v_5 \rightarrow \epsilon$

Fig. 7. Excerpt of the context-free grammar for `editInfo.jsp` corresponding to the output stream flow graph from Figure 6.

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

We now construct a family of context-free grammars $\{G_{q_1}, \ldots, G_{q_k}\}$, one for each page in the web application, from the output stream flow graph F. All the grammars have the same nonterminals, terminals, and productions; only the start nonterminal differs. The nonterminals are the nodes from F, that is, $V = N$, and Σ is the Unicode alphabet. The start nonterminal for G_q is the entry node of the page q. The productions are constructed such that $\mathcal{L}_{G_q}(n) = \mathcal{L}_F(n)$ for all $n \in N$. Although output stream flow graphs have an operational flavor and context-free grammars are a declarative formalism, this construction of the productions is straightforward:

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

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

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

We now have a family of context-free grammars where $\mathcal{L}(G_q)$ is an over-approximation of the set of strings that the web application page q may possibly produce as output. As an example, Figure 7 shows an excerpt of the context-free grammar for `editInfo.jsp`.

Recall that the goal of this phase is to identify specific elements and attributes that hold client state, in particular, names of hidden fields, and URLs that point to other application pages. As the context-free grammars we have produced work at the level of individual characters, we need to analyze each G_q to produce an annotated grammar G'_q that shows how the characters group into HTML elements and attributes.

To see how this can be done, consider the way an HTML parser performs a left-to-right scan through the characters of an ordinary HTML document. The parser is initially in a state *contents*. When it encounters a < character, it switches to another

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

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

The least solution can be computed using a simple fixpoint algorithm. Intuitively, the first constraint ensures that generated strings start in the *contents* state, and the other constraints apply the transition function δ from left to right on the characters in the strings to find the possible parse states at the different positions.

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

The last step of the static analysis of the dynamically generated HTML output consists of a simple traversal through the annotated grammar G'_q, looking for specific elements and attributes. To construct C_{out}, that is, the required information about names of hidden fields and URL parameters, we look for the `name` attributes in `<input>` elements that have a `type` attribute with value `hidden` and for `href` attributes in `<a>` elements. To construct the page graph edges S we look for the `action` attributes in `<form>` elements and for `href` attributes in `<a>` elements.

After annotating the grammar from Figure 7, we can determine that the production on v_4 can generate an element named `input` with attributes `type="hidden"` and `name="nickname"`, so C_{out}(`editInfo.jsp`) contains `nickname`.

4.2. Analyzing Input Parameters

Parameter values in the Java Servlet framework are read using the `getParameter` method of the `HttpServletRequest` object. We conservatively assume that all objects of type `HttpServletRequest` are relevant. The Servlet framework instantiates all request objects and provides no implementation of the `HttpServletRequest` interface to the programmer, so this assumption is unlikely to result in false positives in practice. Since the request parameter name that is given as an argument to this method may

not be a constant in the source code, we approximate for each call to `getParameter` the possible values as a regular language. We obtain this information using the JSA string analysis tool as in Section 4.1. If the language overlaps with $C_{in}(p)$, we mark the method call as a *client-state value source*. This step will mark the method call on line 39 in Figure 1 as such a source. The call on line 41 will not be marked, since `email` is not in C_{in}(`SaveInfo.java`).

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

5. IDENTIFYING SHARED APPLICATION STATE

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

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

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

Finding all expressions in the code that may yield internal application state according to these rules can be done with a simple iterative fixpoint algorithm combined with an alias analysis, such as the points-to analysis provided by the Soot tool that we also used in Section 4.1. We first define a set of abstract locations $K = \mathit{Field} \cup \mathit{Local}$, where Field and Local denote the fields in classes and the local variables and method parameters, respectively, in the application code. For a field $f \in \mathit{Field}$, the abstract location f corresponds to the set of fields named f in objects at runtime. Similarly, a local variable or method parameter $x \in \mathit{Local}$ corresponds to all occurrences of x at runtime. Every name in Field and Local is implicitly qualified by the signature of the surrounding class and method, respectively, to distinguish between variables of the same name in different contexts. We assume that nested expressions have been linearized by Soot using extra local variables, and the keyword `this` is treated as a local variable. The points-to analysis gives us a may-alias equivalence relation $\sim \subseteq K \times K$ such that $k_1 \sim k_2$ if k_1 and k_2 may point to the same object at runtime. We now find the internal application state by computing a subset of the abstract locations $A \subseteq K$ as the least solution to the following constraints, where $x, y \in \mathit{Local}$, $f \in \mathit{Field}$, c is a class, and m is a method:

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

$$\texttt{this} \in A$$
$$t_1 \in A$$
$$t_1 \in A \Rightarrow \texttt{roomList} \in A$$
$$\texttt{roomList} \in A \Rightarrow \texttt{chatRoom} \in A$$
$$\texttt{chatRoom} \in A \Rightarrow \texttt{chatter} \in A$$

Fig. 8. Constraints for computing the internal application state for the class `SaveInfo`. The local variable t_1 corresponds to the sub-expression `getServletContext()` on line 44 in Figure 1.

The first condition corresponds to rule (1) from before, the next two correspond to rule (2), and the two after those correspond to rule (3). The last condition takes aliasing into account. This computation of A captures all internal application state, although obviously as an approximation. As already mentioned, we may conservatively classify some session state or transient state as application state. The coarse heap abstraction and alias analysis, as well as the lack of, for example, flow- and context-sensitivity may also contribute to imprecision. Nevertheless, the experiments described in Section 7 indicate that this simple analysis is sufficient.

Continuing the *JSPChat* example from Figure 1, the variables `nickname`, `email`, `session`, and `contextPath` in `SaveInfo.java` are fields in the servlet class, so their values are correctly classified as internal application state. (That is, however, presumably not intended by the programmer, which we return to in Section 7.) The `roomList` variable gets its value from an attribute in the servlet context object using the method calls `getServletContext().getAttribute(...)`, so its value is also classified as internal application state. In contrast, the variables `request` and `response` are not included as internal application state. The constraints being generated are shown in Figure 8. The alias analysis is not needed in this simple example.

We also find the *external* application state stored in files and databases. Such state is read and written using special API functions. The analysis treats all parameters to all methods from the standard Java libraries as sinks, except for a built-in collection of method parameters that have a special meaning for the information flow. We describe these exceptions and a customization mechanism in Section 6.

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

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

6. INFORMATION FLOW FROM CLIENT STATE TO SHARED APPLICATION STATE

As outlined in Section 3, we use an information flow analysis to identify flow of the client-state values in the program to the shared application state. In general, information flow analysis considers two kinds of flow: *explicit* and *implicit* flow [Denning and Denning 1977]. Explicit flow is caused by assignments and parameter passing. Other forms of explicit flow may be described using customized derivation rules, as described next. Implicit flow arises when the value of a variable depends on a branch condition involving another variable. Other work involving information flow in web applications typically disregards implicit flow [Livshits et al. 2009; Tripp et al. 2009]. According to Tripp et al. [2009], "experience shows that attacks based on control dependence are rare and complex, and thus less important than direct vulnerabilities." To simplify our analysis, we also choose to consider only the explicit flow.

Information flow analysis requires a characterization of sources, sinks, and sanitizers. The sources in our analysis are the client-state value sources that were identified in Section 4. The sinks are operations in the code where the application writes to fields of internal application state objects or invokes methods that involve external application state, which we found in Section 5.

Sanitizers can be methods that determine whether a given client-state value is safe, for example, by performing access control or MAC checking, and methods that convert unsafe values to safe ones, for example, by decrypting the values. An example is the ESAPI method `decryptHiddenField` mentioned in Section 2. As sanitizers are highly application specific, they are provided through customization, and none are built into the analysis.

The information flow analysis we use is a simple whole-program dataflow analysis. It is flow sensitive, meaning that different information is obtained at different program points. It is context sensitive using one level of call-site sensitivity. The state abstraction uses the same definition of abstract locations, K, as the analysis in Section 5. Each abstract state provides a set of client-state parameter names for each abstract location. For example, at the program point after the assignment on line 39 in Figure 1, the abstract state maps the `email` field of the servlet class to the singleton set {`nickname`}, and all other locations are mapped to the empty set. Our implementation uses Soot, as in the other analysis components, with class-hierarchy analysis for call-graph construction. The analysis scales well since it only tracks client-state parameters, and relatively few fields and variables involve client state in typical web applications. Another important factor is that the analysis skips library code.

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

The customization rules can be given either as annotations in the code or in a separate file. Application-specific rules can be added by the user of the analysis. Examples of such customizations are presented in Section 7. Additionally, we provide a collection of predefined rules for the Java standard library. Figure 9 shows some examples. The first four rules involve operations on strings that propagate client-state information from parameters to return values or to the base value. For the `add` method on a `List` object, the `List` object is marked as client-state if the object being added to the list has that status. All `Iterator` objects being produced from such `List` objects also become marked as client state, and similarly for objects that are returned from the `next` method on these `Iterator` objects. This accounts for the common pattern of information flow to and from `List` containers. Other containers, such as `HashMap` objects, are treated similarly. The last rule shown in the list tells the analysis that creating a `File` object is harmless—in fact, such objects are often used in authentication checks (cf. condition 4 in Section 2)—so the `File` constructor should not be treated as a sink. However, we specify information flow from the parameter to the constructed object, since that object may later be used for constructing, for example, `FileWriter` objects, which are treated as sinks.

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

Fig. 9. Examples of predefined derivation rules for the information flow analysis.

For the example in Figure 1, the information flow analysis finds out how the hidden field values appearing at the source on line 39 may affect the application state, which triggers a vulnerability warning. This is explained in more detail in Section 7.1.

7. EVALUATION

Our prototype implementation, WARLORD[6], reads in a Java web archive (.war) file containing a web application built with Java Servlets, JSP, or Struts, together with an analysis customization file, and performs the analysis described in Sections 3–6. As mentioned in previous sections, the implementation is based on the Soot analysis infrastructure [Vallee-Rai et al. 1999], the JSP compiler from Tomcat[7], and our tools for HTML grammar analysis [Møller and Schwarz 2011] and string analysis [Christensen et al. 2003; Feldthaus and Møller 2009]. With this implementation, we aim to answer the following research questions:

Q1: Is the analysis precise enough to detect client-state vulnerabilities with a low number of false positives? Specifically, can it identify the common uses of client state, and is it capable of distinguishing between safe and unsafe uses of client state in the sense described in Section 2?
Q2: Are the warning messages produced by the tool useful to the programmer for deciding whether they are false positives or indicate exploitable vulnerabilities?
Q3: In situations where the programmer decides that a vulnerability warning is a false positive, is it practically feasible to exploit the customization mechanism to eliminate the false positive?
Q4: Is the analysis fast enough to be practically useful during web application development?

[6] http://www.brics.dk/WARlord/
[7] http://tomcat.apache.org/tomcat-7.0-doc/jasper-howto.html

	Frameworks	Pages	Client-state parameters	Unique names
JSPChat	Servlets, JSP	16	19	3
Hipergate	JSP	760	2,680	264
Takatu	JSP, Struts	558	1,840	31
Pebble	Servlets, JSP	122	22	11
Roller	JSP, Struts	53	86	27
JWMA	Servlets, JSP	26	40	10
JsForum	Servlets, JSP	10	14	8
JavaLibrary	JSP	20	92	35
BodgeIt	Servlets, JSP	9	6	6
WebGoat	Servlets	1	1	1

Fig. 10. List of benchmarks. The 'Frameworks' column shows which web frameworks are used in each benchmark; 'Pages' is the total number of JSP pages, servlet classes, and Structs action classes; 'Client-state parameters' is the number of client-state parameters inferred by the analysis, and 'Unique names' is the number of distinct names of such parameters.

To answer these questions, we experiment with a collection of web applications. For each application, we go through the process suggested in Section 3: We first run the WARLORD tool on the application with no customization. After a manual study of the warnings being produced, appropriate customization is added, if possible, to address the false positives.

If any exploitable vulnerabilities are found after running the analysis again, this time with the new customization, we fix them manually using one of the techniques mentioned in Section 2.

Our experiments are based on 10 open-source web applications found on the web: *JSPChat*[1] (the small chat application mentioned in Section 1), *Hipergate*[8] (a customer resource management application written entirely in JSP), *Takatu*[9] (a large tax administration system), *Pebble*[10] (a widely used blogging application), *Roller*[11] (another blogging application), *JWMA*[12] (a web mail application), *JsForum*[13] (a forum application), *JavaLibrary*[14] (a book library management application), *BodgeIt*[15] (a web shop written to demonstrate common security problems in web applications), and *WebGoat*[16] (another web application that has been made to demonstrate typical security problems, written by OWASP). The benchmarks were selected as the first 10 applications we encountered that use some form of client state and are based on Java Servlets, JSP, or Struts. Our prototype supports Struts 2 but not version 1, so we do not include the full list of benchmarks from Stanford SecuriBench [Livshits 2005]. The benchmarks on

[8] http://hipergate.sourceforge.net/
[9] http://takatu.sourceforge.net/
[10] http://pebble.sourceforge.net/
[11] http://roller.apache.org/
[12] http://jwma.sourceforge.net/
[13] http://sourceforge.net/projects/jsforum/
[14] http://sourceforge.net/projects/javalibrary/
[15] http://code.google.com/p/bodgeit/
[16] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

```
Write of client-state value 'nickname' to application state
on line 23 of sukhwinder.chat.servlet.SaveInfoServlet
Trace:
 sukhwinder.chat.servlet.SaveInfoServlet:
  void doGet(HttpServletRequest,HttpServletResponse)
```

Fig. 11. Output from the WARLORD tool for the *JSPChat* benchmark.

our list cover a variety of application kinds of different size, they are written by different programmers, and they use different web frameworks. The *Takatu* and *JsForum* projects do not appear to be active but represent interesting snapshots of incomplete web applications. Some characteristics of the benchmarks are listed in Figure 10. The column 'Client-state parameters' shows the total number of client-state parameters computed as $\sum_p |C_{in}(p)|$ for all pages p. Although $C_{in}(p)$ may in principle be infinite, each of the sets is finite and usually small. Note that client-state values appear in all the benchmarks. The number of distinct names of the parameters, $|\bigcup_p C_{in}(p)|$, shown in the last column gives an indication of how many different kinds of client state that occur.

7.1. Experiments

JSPChat. The analysis identifies uses of 19 client-state parameters, and only 1 warning is produced about potential client-state manipulation vulnerability. The single warning is shown in Figure 11: as hinted in Section 1, the application is prone to a timing attack since the values of the request variables are stored in fields on the servlet object, which the analysis reveals. Since this is indeed shared application state, such a vulnerability falls within our characterization of client-state manipulation vulnerabilities. Notice that the analysis output includes a trace from the source to the sink, which can make it easier to confirm or dismiss the error by manual inspection. If we manually correct this error by changing the field into a local variable, the analysis finds another error: the application is also prone to a classical client-state manipulation attack, since a malicious user may change the `nickname` request parameter and consequently change the information for another user. This error can be corrected by fetching the nickname from the session instead of a client-state parameter. After also correcting this error, WARLORD gives no more warnings. A manual inspection confirms that the remaining occurrences of client-state parameters are indeed safe. No customization is necessary for this application.

Hipergate. Client-state parameters are used massively in this web application; in fact, all client-specific values are passed around using hidden fields. Running the analysis yields 197 warnings. With 14 customizations, this number is brought down to 132 warnings, almost all of which are caused by client-state parameter values that flow into parameterized database queries without any checks. We have inspected all of the warnings, and many of them correspond to code that is vulnerable to attacks, as explained in the following. The main source of false positives originates from a use of randomly generated ID strings for database rows. Such strings are hard to guess and we do not consider this as vulnerable. If we exclude warnings involving these random strings, 71 warnings remain.

All in all, 40 of the warnings reveal exploitable client-state manipulation vulnerabilities. One of the warnings reveals that a file can be read from the disk using a file name originating from a client-state parameter in `wb_style_persist.jsp`. This parameter can be exploited to change files on the disk. Although the programmer has carefully inserted authorization checks to ensure that the user should be granted ac-

cess to the page in question, no checks are made for any of the client-state parameters, and they can therefore be manipulated by the client. The tool also gives a warning on the page `docrename_store.jsp`, which can be exploited to rename files. The programmer has inserted a check to ensure that the user has rights to rename the files, but this is performed on another parameter than the one holding the file name, and an attacker can therefore create an exploit that changes only the file name. Furthermore, the tool emits 4 warnings for the page `reference.jsp` where parameters can be injected into an SQL string. 1 warning on the page `catusrs_store.jsp` reveals that a client-state parameter can give access to update permissions for any user, and 2 warnings reveal a similar problem for `catgrps_store.jsp`. Similarly, 31 warnings in 18 other pages reveal places where client-state values give direct access to the database. In all cases, data is queried or changed in the database using a client-state parameter.

For the remaining 31 warnings, we found that they could not be exploited. In three cases, the parameters control settings for querying the database without affecting the result, for example, the number of rows queried at a time. In additional three cases, the values are references to objects that are owned by the user and changing these values does not give access to new information. In the remaining cases, values flow to the database API, but the queries are only used for logging client actions or for retrieving data that is used for access control. The current customization mechanism is not able to express the precise behavior of SQL expressions that are executed through calls to the JDBC API, and therefore the analysis considers all such calls as sinks. The analysis is able to classify 2,548 out of 2,680 uses of client-state parameters as safe.

Takatu. The analysis identifies 1,840 client-state parameters. 184 warnings are issued, all but 14 are caused by reading from the database using IDs that come from hidden fields. These IDs are used for querying objects from the database. After manually inspecting the warnings, we can see that 162 of them can be exploited to change or read data on the server. Other 8 warnings indicate places where values are read from the database in ways that are not vulnerable, for example, for searching for values in the database. The remaining 14 warnings indicate places where a client-state parameter holds the value of a log flag that is used to query the database but none of them can be exploited. No customization is required for this application.

Interestingly, this web application at multiple places asks the user to confirm the deletion of an object. The ID of the object is stored in a hidden field that is not protected, so the client can delete any object of the same type by modifying the ID used as object reference. The errors are easily corrected, for example, by signing the vulnerable parameters and checking the signature when the parameter is sent back to the server.

Pebble. WARLORD identifies 22 uses of client-state parameters and initially produces 4 warnings. This web application uses a dispatcher, so all requests except those to JSP pages go through a single servlet. The number of client-state parameters seems small because of this structure, but the classes being dispatched to make heavy use of the client-state parameters.

The web application stores files on the disk such that each blog has its own directory, and it uses the value of a parameter from a hidden field to determine the name of the file to save to, which is the cause of 2 warnings. However, each value used this way is verified to be a child of the blog folder, so the folder structure ensures that users cannot overwrite each other's files. The two first customization rules shown in Figure 12 handle this check of the parent folder.

Only 1 warning is produced after the customization. It is caused by the page where a new blog is added. This page uses an `id` parameter originating from a hidden field to set the database ID of the newly created blog and to create a directory for the files belonging to the blog. The `id` parameter is verified to only contain letters, and another

```
net.sourceforge.pebble.util.FileUtils.underneathRoot(File,File):
   Sanitizer for arg 2
net.sourceforge.pebble.domain.FileManager.isUnderneathRootDirectory(File):
   Sanitizer for arg 1
net.sf.ehcache.Element.get(Serializable):
   Not a sink
net.sourceforge.pebble.index.StaticPageIndex.getStaticPage(String):
   Not a sink
net.sourceforge.pebble.util.FileUtils.getContentType(String):
   Not a sink
```

Fig. 12. Customization rules for the *Pebble* benchmark.

check ensures that the ID is not already in use. Together, these two checks mean that there are no exploitable vulnerabilities related to the 4 warnings. The safety depends on a subtle invariant about the directory structure where files are stored on the disk. While this invariant is beyond what we can express with the customization mechanism, extracting the relevant code into a separate method would make the code easier to read, less prone to become vulnerable as a result of future changes, and it would become expressible as a sanitizer using the customization mechanism.

Roller. The developers of this web application have systematically reviewed the code for the class of vulnerabilities we are trying to detect. All client-state parameters are protected with authorization checks that are well documented in the code. Running WARLORD initially results in 53 warnings on the 53 pages. We added 14 customization rules, which mainly describe information flow for a few string manipulation functions and information about queries of public information such as blog comments. Those functions are part of the Apache Commons API, so these rules are generally useful in all applications that use this API.

Only 1 warning remains after adding these rules. That warning refers to a page that allows blog comments to be deleted using a client-state parameter to identify the blog comments. All comments belong to a blog, and user rights are defined for each blog. The page checks whether each comment belongs to the blog and refuses any attempt to delete comments on other blogs in a way that cannot be modeled with our customization mechanism. However, if the code were rewritten slightly to use a separate method to check the ownership directly, this method could be marked as a sanitizer. That would also make it possible to check that future changes to this code would not create vulnerabilities, and it would make the code more readable.

JWMA. This web application acts as a frontend for an email server using the Java Mail API, and it stores almost all data in the session state. It has little shared application state, but it does use client state.

The HTML view is generated through JSP pages and form data is handled using servlets. The behavior of the receiving servlet is determined by one of two hidden fields, `acton` and `todo`. The behavior depends only on implicit information flow from these two parameters, and no warnings are issued in relation to them. Inspecting the use of the parameter values manually does not reveal any vulnerabilities either.

With no customizations, WARLORD produces 3 warnings. Two of them are spurious warnings related to reading and using the values of the client-state parameters `paths` and `contact.id` in the servlet `JwmaController`. Request parameters are read using a method on the class `JwmaSession` and WARLORD is unable to analyze this precisely enough to determine that these two parameters are not read by `JwmaController`.

The third warning relates to the client-state parameter `numbers`, which is used for moving and deleting messages in `JwmaController`. Through manual inspection we have found that this parameter is not vulnerable, since it only allows manipulation of data in the client's own folder.

JsForum. This web application uses a combination of JSP pages for generating the HTML view and Servlets for updating data in the database. The database connection uses the standard JDBC API for accessing a MySQL database.

Client-state parameters are primarily used for storing database identifiers. WARLORD reveals that the programmers have not protected the application against client-state manipulation attacks. Without customization, WARLORD produces 12 warnings, all of which relate to the use of database identifiers. In the servlet `AddThread`, the analysis warns that the client-state parameters `lastThread_id` and `forum_id` are stored in an application state object. This happens because the servlet generates an SQL query based on these parameters and stores the query string in a field reachable from the servlet class. The methods are not synchronized and another request might therefore override the value before it is sent to the database. Other warnings reveal that clients can change the values of `forum_id` and `lastThread_id` to post to a different forum and to manipulate the identifier of a newly created thread. Furthermore, the servlet `AddThread` allows the client to post as a different user by changing the value of the hidden field named `user`.

Further inspection of the other pages reveals similar vulnerabilities in the servlets `ChangeMessage`, `AddReply`, and `AddForum`. In `DeleteForum`, however, the application code checks that the client is an administrator before deleting a forum. We therefore do not consider that servlet to be vulnerable. The customization mechanism is not able to express such a property. Of the 12 warnings, 11 corresponded to actual client-state manipulation vulnerabilities. No customization was used.

JavaLibrary. This is a small JSP application for managing book reservations and lists of users with varying levels of privileges. WARLORD detects 92 client-state parameters in the application and deems 65 of them safe. Of the remaining 27 parameters, 22 are read by the servlet `FormProcess`.

JavaLibrary uses a bean for representing all values related to users. This bean is updated from client-state values when a user is added or edited. The JSP page `user_form.jsp` is used for creating and editing users. Depending on the rights of the user, the page prefills the HTML form with hidden fields. The `FormProcess` does not check for client-state manipulation, and it is therefore possible to modify many of these parameters to gain privileges similar to that of an administrator when creating or editing users. This accounts for 7 of the 22 warnings. Furthermore, client-state manipulation through other forms can be exploited to change reservation dates and due dates for borrowed books and to borrow books for other users. All of the 22 parameters can be exploited for attacks.

The remaining 5 warnings that are not related to `FormProcess` result from client-state in JSP pages. According to comments in the code, state is saved in these fields to allow the client to return to the page later and complete the data entry. Similarly to *JSPChat*, this creates a possibility of a timing attack, and in this application, it also allows clients to read values entered by other clients.

No customization was necessary for this web application.

BodgeIt. This web application was written as a benchmark for penetration testing tools. It contains what the authors call "hidden (but unprotected) content" and "insecure object references", which are within our definition of client-state vulnerability. It therefore serves well as a test for our static analysis.

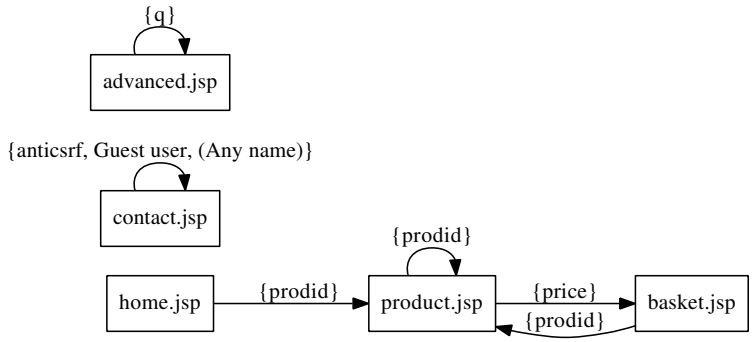

Fig. 13. An excerpt of the page graph for *BodgeIt* showing the nodes and edges that involve client state.

```
com.thebodgeitstore.util.AES.hexStringToByteArray(java.lang.String):
    Flow from parameter 1 to return value
com.thebodgeitstore.util.AES.decryptCrt(java.lang.String):
    Not a sink (the method returns a safe value)
```

Fig. 14. Customization rules for the *BodgeIt* benchmark.

Figure 13 shows the part of the page graph for *BodgeIt* that involves client state. Before customization, WARLORD reports 5 warnings. We added the two customization rules shown in Figure 14. They relate to the use of an encrypted token in `contact.jsp` for protecting against cross-site request forgery. One warning refers to a parameter named `prodid` in the JSP page `product.jsp`. The value originates from a URL parameter and is used to query the database for a product with the corresponding database ID. While this page demonstrates the possibility for client-state manipulation, changing the parameter does not give the client access to additional information. Consequently, there is in fact no vulnerability in this case. Another warning, which originates from the URL parameter `typeid`, is also used for a database query. Manipulating this parameter does not give access to new information either.

In the JSP page `contact.jsp`, WARLORD detects a hidden field. Rather unusually, the name of this hidden field can be arbitrary, because it is set to the name of the current user. This causes WARLORD to consider all parameters in the successor page, which is `contact.jsp` itself, as client-state parameters, which results in a false positive when storing the value of the `comments` parameter. Although this warning does not indicate a possible client-state manipulation vulnerability, it reveals the possibility of a name clash if a user is registered with the name "`comments`".

The JSP page `basket.jsp` page places an item in a shopping basket along with the price of the item. WARLORD gives a warning for the `productid` parameter. The client is able to arbitrarily change this parameter to add any item to the basket, however, we classify this as another false positive because the client is already able to add any item to the basket without client-state manipulation. The item price is stored in a hidden field called `price`, but this field is never read, so the user cannot gain any extra privileges by changing its value, and WARLORD correctly yields no warning in this case.

In conclusion, we find that there are, surprisingly, no exploitable client-state manipulation vulnerabilities in this web application.

WebGoat. We have analyzed a single servlet in this web application. The purpose of the servlet, which uses a single hidden field, is to demonstrate vulnerabilities of exactly the kind we want to detect. Unlike the other benchmarks, this application generates output using a custom DOM-like framework and we decided to manually create the set of parameters that may hold client-state values.

Perhaps surprisingly, our tool reports 0 warnings for this application. The reason is that the servlet does not use the input variable for anything else than selecting a message to send back to the client. This usage does not violate any of the safe usages presented in Section 2 and we therefore conclude that while the illustrative servlet of course mimics the behavior of a vulnerable piece of server code, it is actually not vulnerable to any attack. A manual inspection of the code confirms that the client is indeed not able to change the shared application state in any way by changing the value of the hidden field.

As an additional experiment related to Q1, we test whether the client-state analysis component (Section 4) is really necessary. If we disable that component and run WARLORD in a mode where all parameters are conservatively assumed to contain client state, we observe a drastic increase in the number of false positives, even with all customization enabled. As an example, in *JSPChat* from Figure 1, this will treat the `email` parameter as client state, such that line 41 becomes a source for the information flow analysis and a spurious vulnerability warning is triggered on line 50 (in the version where the timing vulnerability has been fixed by changing the fields into local variables, as discussed in Section 7.1). For the other benchmarks, hundreds of additional warnings appear without revealing any new actual client-state manipulation vulnerabilities, which demonstrates that the client state analysis component is useful.

7.2. Summary of Results

Figure 15 summarizes the benchmark results from the previous section. The first column, 'Client-state parameters', is the same as in Figure 10. The next columns show the number of warnings before customization, the number of customization rules, and the number of warnings after customization. The tool produces at most one warning for each of the client-state parameters from the first column (however, each warning may contain multiple traces from sources to sinks). The next column, 'Exploitable', shows how many of the warnings we could manually verify to be exploitable by malicious clients performing client-state manipulation attacks. The column 'Safe client-state parameters' shows the number of client-state parameters that the analysis after customization determines not to be vulnerable. The final column shows the time spent for the full analysis. The numbers in parentheses show the results after grouping together data that involve parameters of the same name, which indicates that the warnings and vulnerabilities involve many different uses of client state.

The tests have been performed on a 2.4GHz Core i5 laptop running OS X. The JVM was given 1GB of heap space for each benchmark. The time and memory was primarily used by the Soot framework for loading classes and performing the pointer analysis.

With this, we are able to answer the research questions:

Q1: A manual inspection of the application code confirms that the client-state analysis succeeds in finding all client-state value sources. This amounts to a total of 4,802 client-state parameters. The analysis determines that 4,437 (92%) of those parameters are safe, that is, they are not present in any warnings. Moreover, after customization, 241 (66%) of the 365 warnings that are produced in total reveal exploitable vulnerabilities. The false positives are not evenly distributed among the benchmarks, and they are concentrated on a small number of different parameter

	Client-state parameters	Warnings before customization	Customization rules	Warnings after customization	Exploitable	Safe client-state parameters	Time
JSPChat	19 (3)	1 (1)	0 (0)	1 (1)	1 (1)	18 (2)	30 s
Hipergate	2,680 (264)	197 (99)	14 (14)	132 (75)	40 (30)	2,548 (189)	116 m
Takatu	1,840 (31)	184 (10)	0 (0)	184 (9)	162 (9)	1,656 (22)	20 m
Pebble	22 (11)	4 (4)	5 (5)	1 (1)	0 (0)	21 (10)	10 m
Roller	86 (27)	53 (1)	14 (14)	1 (1)	0 (0)	85 (26)	4 m
JWMA	40 (10)	3 (3)	0 (0)	3 (3)	0 (0)	37 (7)	3 m
JsForum	14 (8)	12 (7)	0 (0)	12 (7)	11 (7)	2 (1)	1 m
JavaLibrary	92 (35)	27 (22)	0 (0)	27 (22)	27 (22)	65 (13)	2 m
BodgeIt	8 (8)	5 (5)	2 (2)	4 (4)	0 (0)	4 (4)	2 m
WebGoat	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	30 s

Fig. 15. Summary of experimental results.

names. The experiments also demonstrate that the client state analysis component is critical for the analysis precision.

Q2: Based on the warnings given by the tool, especially the trace information, it was in each case possible for us to quickly determine whether it indicated a vulnerability or not. The entire process of classifying the warnings and adding customization rules for all 10 benchmarks took one person less than a day, despite having no prior knowledge of the benchmark code.

Q3: Adding customization rules in many cases reduced the number of spurious warnings considerably. As discussed for the individual benchmarks, the remaining cases typically involve subtle, undocumented invariants. Moreover, if allowing simple refactorings, such as extracting a safety check to a separate method, most of these cases could be captured within the existing customization framework. In the case of *Hipergate*, however, some uses of client state are safe for reasons that go beyond the current capabilities of customization. The decision mentioned in Section 3 that the analysis ignores condition 5 from Section 2 results in a few false positives in *BodgeIt*, *Hipergate*, and *Takatu*.

Q4: The tool analyzes between 10 and 200 pages per minute. Pages can be analyzed individually, so when a programmer is modifying the application, it is possible to run the tool only on pages that have changed.

8. RELATED WORK

Client-state manipulation vulnerabilities, in particular the kind involving hidden fields, have been known for many years, as described in Sections 1 and 2. Likewise, automated techniques for protecting against security vulnerabilities in web applications have a long history. Here we explain the connections between our approach and the most closely related alternatives that have been proposed.

8.1. Static Analysis of Web Applications

The first phase of our analysis that identifies the client-state parameters applies techniques from our earlier work on static analysis of HTML output of Java-based web applications [Kirkegaard and Møller 2006; Møller and Schwarz 2011]. The analysis tools WAM, WAM-SE, WAIVE, and ASCEND by Halfond et al. [2009; 2008; 2012] check that the names and possible values of HTTP request parameters in dynamically generated HTML documents are consistent with the program code that receives the parameters. As in WARLORD, this involves static analysis of the dynamically generated HTML documents and of the dataflow of HTTP request parameters in the server code; however, those tools do not identify which parameters contain client state, for example, originating from hidden fields.

An essential constituent of our approach is the observation that client-state manipulation vulnerabilities are correlated to information flow from client state to application state. Together with automatic inference of client state (Section 4) and shared application state (Section 5), this allows us to detect likely errors largely without requiring the programmers to provide any specifications. Some application specific customization is required though, as seen in Section 7. For future work, it may be interesting to apply probabilistic specification inference [Livshits et al. 2009] to automate this phase.

The WebSSARI tool by Huang et al. [2004] pioneered the use of static information flow analysis to enforce web application security, and numerous researchers have since followed that path, as discussed in the following. WebSSARI was designed for intraprocedural taint analysis of PHP programs without considering client-state manipulation vulnerabilities specifically. Related techniques for detecting injection vulnerabilities in PHP programs include the bottom-up analysis by Xie and Aiken [2006] and the top-down flow-sensitive dataflow analysis used in the Pixy tool by Jovanovic et al. [2010].

The PQL language by Martin et al. [2005] has been designed to support succinct specification of information flow queries. Their notion of derivation descriptors corresponds to our use of customization rules. Their static analysis is based on a context-sensitive but flow-insensitive pointer alias analysis using Datalog. In principle, the information flow that we consider is expressible within PQL, but we found it easier for our proof-of-concept implementation to use the simple information flow analysis described in Section 6. Livshits and Lam [2005] have used PQL for specifying and implementing a range of vulnerability analyses. Hidden field manipulation is among the list of vulnerabilities they consider. However, their techniques does not perform any client state analysis (Section 4) or shared application state analysis (Section 5); instead they use a fixed set of source and sink descriptors that cannot precisely identify the hidden fields or the operations that involve shared application state. PQL has also been combined with explicit state model checking using Java PathFinder for detecting injection vulnerabilities [Martin and Lam 2008; Lam et al. 2008]. The TAJ tool by Tripp et al. [2009] has similar goals as that of Livshits and Lam, but uses an alternative technique called hybrid thin slicing for obtaining a precise and scalable analysis.

Other related work uses string analysis for detecting web application injection vulnerabilities. The AMNESIA tool by Halfond and Orso [2005] applies string analysis to build models of legitimate SQL queries and then uses these models for runtime mon-

itoring. Wassermann and Su [2007] similarly use string analysis to reason about dynamically constructed SQL queries with fragments that originate from HTTP request parameters and then check whether those fragments may alter the syntactic structure of the queries, all using static analysis. In later work, Wassermann and Su [2008] use the same string analysis technique to detect whether untrusted input from HTTP request parameters may flow to the generated HTML documents and thereby inject JavaScript code. In comparison, our primary use of string analysis is for approximating the HTML output to be able to locate client-state parameters, whereas we track the HTTP request parameters with a simple information flow analysis without modeling their possible values.

Balzarotti et al. [2007] propose a pragmatic approach for finding stored SQL injection and XSS vulnerabilities that arise when a page stores user input in application state and another page later reads and uses this data. Application state is recognized using signatures of database API methods without considering potential aliasing. Their tool performs a simple scan of the application source code to detect links between pages and tracks the flow of user input values using a model-checking approach with a limited search space.

Detection of access control vulnerabilities with static analysis has been studied by Sun et al. [2011], among others. Their approach uses string analysis to statically construct sitemaps for different user roles, which are then compared to look for inconsistent access control. These sitemaps are reminiscent of our page graphs (Section 4) but do not consider client-state parameters.

Providing comprehensive support for diverse web application frameworks, such as Java Servlets, JSP, and Struts, is a challenging endeavor. A general framework, F4F, has been proposed by Sridharan et al. [2011]; however, we have found that it is not sufficiently flexible for our setting, in particular for the client-state identification phase. Still, the ideas in F4F may be adapted in future work to enable support for additional web application frameworks.

8.2. Dynamic Techniques for Vulnerability Detection

The approach of using MACs to protect against client-state manipulation attacks that we discussed in Section 2 can be implemented with a generic servlet filter that intercepts all HTML documents generated by the application at runtime and all HTTP requests that are sent by the clients, without modifying the web application code. For every use of client state in the HTML documents, an additional hidden field or query parameter containing the MAC is automatically inserted. Whenever an HTTP request is received from a client, the MAC check is performed on the appropriate request parameters. Scott and Sharp [2002] exemplified this as part of a more general security gateway. Given a manually constructed security policy, their gateway can, for example, automatically attach MACs to hidden fields. Using such a security filter can be viewed as an alternative or supplement to manually eliminating the vulnerabilities by appropriately patching the application source code. In contrast, the idea in our approach is to inform the programmer—using static analysis of the application source code—that protection may be inadequate.

It is possible to combine the filter and analysis approaches: A security filter needs to be configured with information about which fields and parameters contain client state that should not be manipulated, and this information is precisely what our static analysis can provide. It is of course important that the client-state analysis is precise enough to correctly distinguish between parameters that carry client state and ones that do not. It is less critical that the information flow analysis is able to correctly distinguish between safe and unsafe uses of client state. However, to avoid the overhead

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

Numerous other techniques have been developed for preventing web application vulnerabilities at runtime, in some cases leveraging static analysis to increase precision, although without focusing on client-state manipulation vulnerabilities in particular.

Dynamic taint analysis has been used for detecting various kinds of attacks, including hidden field tampering, for example, by Haldar et al. [2005]. The WASP tool by Halfond et al. [2008] uses a more fine-grained taint analysis to track strings that originate from client input and may be injected in SQL queries. This has been used together with the static analysis tool WAM that infers web application inferfaces to automate penetration testing [Halfond et al. 2011].

Another variant of attacks involving tampering of form parameters is to bypass client-side input validation or manipulate client state, which is the focus of several tools including NoTamper, WAPTEC, and TamperProof [Bisht et al. 2010; Bisht et al. 2011; Skrupsky et al. 2013]. NoTamper and WAPTEC dynamically infer constraints on HTTP parameters from the HTML and JavaScript code. Parameter values that violate these constraints are then submitted to the server, and the server-side input validation is then tested by comparing server response pages that are generated by submitting the tampered parameters to response pages that originate from benign input. TamperProof is a filter-based variant similar to Scott and Sharp's.

The black-box vulnerability scanning technique by Doupé et al. [2012] uses fuzzing based on dynamically inferred state machines, which resemble our use of page graphs. This approach can in principle also detect client-state manipulation vulnerabilities, although that is not explicitly the target of their work. The Waler tool by Felmetsger et al. [2010] uses dynamic execution to infer likely specifications of the intended program behavior followed by symbolic execution to detect violations of those specifications. Among their examples is a client-state manipulation vulnerability, which is found as a violation of a proposed invariant about the value of a hidden field. In comparison, we propose a fully static approach that directly considers the flow of client state in the application code.

Finally, we note that several commercial tools are capable of detecting security vulnerabilities in web applications. According to a 2007 IBM white paper [IBM 2007], the AppScan tool is capable of detecting vulnerabilities involving hidden field manipulation and parameter tampering. The latest version uses techniques from TAJ [Tripp et al. 2009]; however, we have been unable to perform a proper comparison and obtain further information about the techniques applied by AppScan. Microsoft's CAT.NET[17] also uses static information flow analysis, but it cannot detect client-state manipulation vulnerabilities without detailed specifications provided by the user. Other commercial tools include NTOSpider[18] from NT OBJECTives, WebInspect[19] from Fortify/HP, and CodeSecure[20] and HackAlert[21] from Armorize. To our knowledge, most of these tools (with the exception of CodeSecure, which is developed from WebSSARI) employ crawling [Doupé et al. 2010; Bau et al. 2010], not static analysis. We believe static analysis can be a promising supplement to dynamic approaches, as it may provide better coverage of the web application source code.

[17] http://blogs.msdn.com/b/securitytools/archive/2010/02/04/cat-net-2-0-beta.aspx
[18] http://www.ntobjectives.com/ntospider
[19] https://www.fortify.com/products/web_inspect.html
[20] http://armorize.com/index.php?link_id=codesecure
[21] http://armorize.com/index.php?link_id=hackalert

9. CONCLUSION

We have demonstrated that it is possible to provide tool support that can effectively help programmers prevent client-state manipulation vulnerabilities in web application code. The static analysis we have presented is capable of precisely identifying client state, in particular, state stored in hidden fields, and help distinguishing between safe and unsafe use of such state. With WARLORD, our prototype implementation of the analysis, we quickly discovered 241 exploitable weaknesses in 10 web applications. The analysis has high precision: after customization, 66% of the warnings revealed vulnerabilities.

Our experiments also indicate potential for improvements. Specifically, although analyzing the *Hipergate* benchmark revealed a large number of weaknesses, it also resulted in some false positives, which originate from a small group of client-state parameters. It appears that many of these false positives can be avoided if the analysis is extended to also infer the provenance of the client-state values, which can be a subject for future work. It may also be worthwhile to extend the technique to reason about client state stored in cookies.

REFERENCES

ADVOSYS CONSULTING. 2000. Preventing HTML form tampering. `http://advosys.ca/tips/form-tampering.html`.

BALZAROTTI, D., COVA, M., FELMETSGER, V., AND VIGNA, G. 2007. Multi-module vulnerability analysis of web-based applications. In *Proc. 2007 ACM Conference on Computer and Communications Security*.

BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. C. 2010. State of the art: Automated black-box web application vulnerability testing. In *Proc. 31st IEEE Symposium on Security and Privacy*.

BISHT, P., HINRICHS, T., SKRUPSKY, N., BOBROWICZ, R., AND VENKATAKRISHNAN, V. N. 2010. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proc. 17th ACM Conference on Computer and Communications Security*.

BISHT, P., HINRICHS, T., SKRUPSKY, N., AND VENKATAKRISHNAN, V. N. 2011. WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In *Proc. 18th ACM Conference on Computer and Communications Security*.

BRUSSIN, D. I. 1998. A white paper analyzing the MSC hidden form field web site vulnerability. Miora Systems Consulting.

CERF, M. AND SHULMAN, A. 2004. How safe is it out there? `http://www.klab.caltech.edu/~moran/files/safeout/`.

CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. 2003. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium*.

DENNING, D. E. AND DENNING, P. J. 1977. Certification of programs for secure information flow. *Communications of the ACM 20,* 7, 504–513.

DOUPÉ, A., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. 2012. Enemy of the state: A state-aware black-box vulnerability scanner. In *Proc. 21st USENIX Security Symposium*.

DOUPÉ, A., COVA, M., AND VIGNA, G. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

FELDTHAUS, A. AND MØLLER, A. 2009. *The Big Manual for the Java String Analyzer*. Department of Computer Science, Aarhus University.

FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. 2010. Toward automated detection of logic vulnerabilities in web applications. In *Proc. 19th USENIX Security Symposium*.

HALDAR, V., CHANDRA, D., AND FRANZ, M. 2005. Dynamic taint propagation for Java. In *Proc. 21st Annual Computer Security Applications Conference*.

HALFOND, W. G. J. 2012. Automated checking of web application invocations. In *Proc. 23rd IEEE International Symposium on Software Reliability Engineering*.

HALFOND, W. G. J., ANAND, S., AND ORSO, A. 2009. Precise interface identification to improve testing and analysis of web applications. In *Proc. 18th International Symposium on Software Testing and Analysis*.

HALFOND, W. G. J., CHOUDHARY, S. R., AND ORSO, A. 2011. Improving penetration testing through static and dynamic analysis. *Software Testing, Verification & Reliability 21,* 3, 195–214.

HALFOND, W. G. J. AND ORSO, A. 2005. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*.

HALFOND, W. G. J. AND ORSO, A. 2008. Automated identification of parameter mismatches in web applications. In *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

HALFOND, W. G. J., ORSO, A., AND MANOLIOS, P. 2008. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering 34,* 1, 65–81.

HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. 2004. Securing web application code by static analysis and runtime protection. In *Proc. 13th International World Wide Web Conference*.

IBM. 2007. The dirty dozen: preventing common application-level hack attacks. `ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_dirtydozen.pdf`.

INTERNET SECURITY SYSTEMS. 2000. Form tampering vulnerabilities in several web-based shopping cart applications. ISS E-Security Alert, `http://www.iss.net/threats/advise42.html`.

JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. 2010. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security 18,* 5, 861–907.

KIRKEGAARD, C. AND MØLLER, A. 2006. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium*.

LAM, M. S., MARTIN, M. C., LIVSHITS, V. B., AND WHALEY, J. 2008. Securing web applications with static and dynamic information flow tracking. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*.

LIVSHITS, B. 2005. Defining a set of common benchmarks for web application security. In *Workshop on Defining the State of the Art in Software Security Tools*.

LIVSHITS, V. B. AND LAM, M. S. 2005. Finding security vulnerabilities in Java applications with static analysis. In *Proc. 14th USENIX Security Symposium*.

LIVSHITS, V. B., NORI, A. V., RAJAMANI, S. K., AND BANERJEE, A. 2009. Merlin: specification inference for explicit information flow problems. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*.

MARTIN, M. C. AND LAM, M. S. 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proc. 17th USENIX Security Symposium*.

MARTIN, M. C., LIVSHITS, V. B., AND LAM, M. S. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

MØLLER, A. AND SCHWARZ, M. 2011. HTML validation of context-free languages. In *Proc. 14th International Conference on Foundations of Software Science and Computation Structures*.

MORAN, L. 2011. How Citigroup hackers broke in 'through the front door' using bank's website. `http://www.dailymail.co.uk/news/article-2003393/How-Citigroup-hackers-broke-door-using-banks-website.html`.

OPEN WEB APPLICATION SECURITY PROJECT. 2010. OWASP top 10. `https://www.owasp.org/`.

SCHOLTE, T., BALZAROTTI, D., AND KIRDA, E. 2011. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proc. 15th International Conference on Financial Crypto*.

SCOTT, D. AND SHARP, R. 2002. Abstracting application-level web security. In *Proc. 11th International World Wide Web Conference*.

SKRUPSKY, N., BISHT, P., HINRICHS, T., VENKATAKRISHNAN, V. N., AND ZUCK, L. D. 2013. TamperProof: a server-agnostic defense for parameter tampering attacks on web applications. In *Proc. 3rd ACM Conference on Data and Application Security and Privacy*.

SRIDHARAN, M., ARTZI, S., PISTOIA, M., GUARNIERI, S., TRIPP, O., AND BERG, R. 2011. F4F: Taint analysis of framework-based web applications. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

SUN, F., XU, L., AND SU, Z. 2011. Static detection of access control vulnerabilities in web applications. In *Proc. 20th USENIX Security Symposium*.

TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. 2009. TAJ: effective taint analysis of web applications. In *Proc. ACM Conference on Programming Language Design and Implementation*.

VALLEE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference*. IBM.

WASSERMANN, G. AND SU, Z. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

WASSERMANN, G. AND SU, Z. 2008. Static detection of cross-site scripting vulnerabilities. In *Proc. 30th International Conference on Software Engineering*.

XIE, Y. AND AIKEN, A. 2006. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symposium*.

ZDNET. 2001. New e-rip-off maneuver: Swapping price tags. http://www.zdnetasia.com/new-e-rip-off-maneuver-swapping-price-tags-21187583.htm.