Bypass Testing of Web Applications

Jeff Offutt, Ye Wu, Xiaochen Du and Hong Huang
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
(+1) 703-993-1654 / 1651
{ofut,wuye,xdu,hhuang2}@ise.gmu.edu

Abstract

Web software applications are increasingly being deployed in sensitive situations. Web applications are used to transmit, accept and store data that is personal, company confidential and sensitive. Input validation testing (IVT) checks user inputs to ensure that they conform to the program's requirements, which is particularly important for software that relies on user inputs, including Web applications. A common technique in Web applications is to perform input validation on the client with scripting languages such as JavaScript. An insidious problem with client-side input validation is that end users can bypass this validation. Bypassing validation can reveal faults in the software, and can also break the security on Web applications, leading to unauthorized access to data, system failures, invalid purchases and entry of bogus data. We are developing a strategy called bypass testing to create IVT tests. This paper describes the strategy, defines specific rules and adequacy criteria for tests, describes a proof-of-concept automated tool, and presents initial empirical results from applying bypass testing.

1 Introduction

The World Wide Web gives software developers a new way to deploy sophisticated, interactive programs with complex GUIs and large numbers of back-end software components that are integrated in novel and interesting ways. Web applications are constructed from heterogeneous software components that interact with each other and with users in novel ways. Web software components are distributed across multiple computers and organizations, are often created and integrated dynamically, are written in diverse languages and run on diverse hardware platforms, and must satisfy very high requirements for reliability, availability and usability. These characteristics have the dual advantage of offering powerful new abilities and presenting new problems to software developers.

Analyzing, evaluating, maintaining and testing these applications present many new challenges for software developers and researchers. Most Web applications are run by users through a Web browser and use HTML to create graphical user interfaces. Users enter data and make choices by manipulating HTML forms and pressing submit buttons. Browsers send the data and choices to the software on the server using HTTP *requests*. An important point to note is that HTTP is a "stateless" protocol, that is, each request is independent of previous requests and, by default, the server software does not know whether multiple requests come from the same or different users.

The type of HTTP *request* determines how the user's data is packaged when sent to the server. Although HTTP defines a number of request types, this paper only considers GET and POST requests. GET requests package the data as parameters on the URL that are visible in the URL window of most browsers (for example, `http://www.buyit.com?name=george`). POST requests package the data in the data packets that are sent to the user.

A common activity of Web applications is to **validate** the users' data. This is necessary to ensure that the software receives data that will not cause the software to do bad things such as crash, corrupt the program state, corrupt the data store on the server, or allow access to unauthorized users. This type of input validation is crucial for Web applications, which are heavily user interactive, often serve a very large user base, have very high quality requirements, and are always publicly accessible [14]. Because of the fundamental client-server nature of Web applications, input validation is done both on the client and the server.

HTML pages (whether static `html` files or dynamically created) can include scripting programs that can respond to user events and check various syntactic attributes of the inputs before sending the data to the server. User events that JavaScript can respond to are defined by the HTML document object model (DOM) and include mouse over events, form field focus events, form field changes, and button presses, among others. Client-side checking is used to check that required fields are filled in, inputs conform to certain restrictions on characteristics such as length, characters used, and satisfaction of syntactic patterns (such as email addresses). Client-side checking can be done as soon as a user event is triggered or after the user clicks on a submit button but before the data is submitted. Doing input validation on the client avoids the need for a trip to the server and allows the checking to be defined within the input form.

Server side checking is done by programs on the server such as CGI/Perl, Java servlets, Java Server Pages, and Active Server Pages. Server side checking can perform all of the checks that client-side checking can, but not until after the user presses the submit button. Server side checking cannot respond to user events, but has access to the state of the file system and database on the server. When a high level language such as Java is used on the server, server side checking also provides more robust and flexible ways to check inputs and respond to invalid user inputs than does a scripting untyped language such as JavaScript.

1.1 Running Web Application Tests Through HTML Forms

HTML forms expect users to type their values and make their choices by using the keyboard and mouse. However, it turns out to be easy for users to bypass the HTML to send values directly to the server software. For example, if the GET request is expected, the users can simply type the parameters into the URL box in their browsers. If the POST request is expected, a simple program can be written on the client that creates and submits the request. There are two reasons for bypassing HTML forms. One is for convenience, if a Web application is used a lot it might be more convenient to skip the relatively slow FORM interface. Another reason is for automation. When running multiple tests on a Web application, the test execution can be automated by bypassing the forms.

This ability to bypass form entry allows another strategy to be used. If the Web application uses client-side input validation, then the users **can bypass the validation**. This technique is sometimes used by hackers. Our suggestion in this research is to utilize the ability to bypass client-side checking to create tests, thereby supplying invalid inputs to the software to test for robustness and security.

An additional ability that is available when bypassing HTML forms is to override hidden form fields. HTML allows data to be placed into a page with the tag “`<INPUT Type="Hidden" ...>`”. These fields are not shown to the users in the browsers, but data in the fields are submitted to the server. Bypassing forms allow the additional ability to change or remove the contents of hidden form fields.

One of the most common ways to violate data security is through “SQL injection.” Many Web applications use client-supplied data in SQL queries. However, if the application does not strip potentially harmful characters, users can add SQL statements into their inputs. This is called SQL injection, and Anley [2] claims that despite being simple to protect against, many production systems connected to the Internet are vulnerable. SQL injection vulnerability occurs when an attacker inserts an SQL statement into a query by manipulating data inputs.

2 Types of Client-side Validation

Input validation can check both the syntax and the semantics of inputs. Client-side input validation can be done by using the HTML input boxes to restrict the size or contents of inputs (syntactic restrictions only), by writing programs such as JavaScripts to evaluate the values before submission (syntactic and semantic restrictions).

2.1 Semantic Input Validation

In an initial attempt at categorization, we have identified three types of *semantic* data input validation.

1. Data type conversion. Most inputs to HTML form elements are plain strings that may be converted to other types on the server. The client can check whether the string can be converted correctly. For example, if the input is an integer, the client can check to ensure that all characters are numeric digits.

2. Data format validation. There are many more restrictive constraints on inputs that can be checked, and

this is one of the most common ways to validate input on the Web. This includes checking the format of money, phone numbers, personal identification numbers, email address, and URLs.

3. Inter-value constraint validation. There are often constraint relationships among input values. For example, when providing payment information, a check payment should include a bank routing number and a bank account, whereas a credit card payment should include a credit card number and an expiration date. The combination of a bank routing number and a credit card number should not be allowed.

2.2 Syntactic Input Validation

HTML can also be used to impose several types of *syntactic* restrictions, all of which can be avoided by bypass testing:

1. Built-in length restriction. Text boxes can include a "maxlength" attribute to restrict the length of text inputs. In the following text input box, only three characters will be accepted:

```
<INPUT Type=text Name=Age Maxlength=3>
```

2. Built-in input value restriction. HTML can use select boxes, check boxes, and radio boxes to restrict the user to a certain pre-defined set of inputs.

3. Built-in input transfer mode. HTML forms define the type of request (GET or POST). Because of the differences in these requests, this is effectively a way to restrict the user's input. HTML links always generate a GET request.

4. Built-in data access. Web browsers manage two types of data, cookies and hidden form fields. Hidden form fields can be viewed if the users look at the source, but are normally not shown. Cookies are automatically managed by the browsers and server software and are sent to the server automatically. Cookies can also be viewed in most browsers (for example, in Mozilla by "Tools-Cookie Manager-Manage Stored Cookies"). A major difference is that cookies persist across multiple requests, whereas hidden form fields are transient data items that only appear in individual HTML pages.

5. Built-in input field selection. An HTML form has a pre-defined set of form fields that users can select values for. Other values are normally not allowed, and client-side scripting can also disable certain input fields by making them unavailable or hidden.

6. Built-in control flow restriction. HTML pages allow the user to transfer to a certain, fixed set of URLs. These are defined by `Action` attributes in `FORM` tags and by HTML links.

Illegal Character	**Symbol**
Empty String	
Commas	,
Directory paths	.. ../
Strings starting with forward slash	/
Strings starting with a period	.
Ampersands	&
Control character	NIL, newline
Characters with high bit set	decimal 254 and 255
XML tag characters	$<, >$

Table 1. Characters that sometimes cause problems for Web applications

2.3 Generalizing to Input Validation

Some of the vulnerabilities (both on client and server) are due to the server not checking input and data from the client; but it would be a mistake to assume checking data is all that is necessary. By considering penetration techniques such as SQL injection, cross-site scripting, buffer overflow, embedded script attack, and shell escape vulnerabilities, Wheeler [18] gives a general solution to user input validation from a security perspective. Any input accepted from user must be validated and any illegal input data should be filtered out. Here are some general rules that should be considered.

1. Filters: Set up filters in the Web application to prevent illegal characters from reaching the server's data store. Table 1 lists some specific characters that can be problems for Web applications.

2. Numeric limits: Limit all numbers to the minimum (often zero) and maximum allowed values.

3. Email addresses: A full email address checker should be enforced. A *full email* includes a username and valid domain name. A complete email check should also ensure that the email contains all expected information, including subject, and recipient addresses.

4. URLs: URLs (and more generally, URIs) should be checked to ensure that they have a valid form and the destination exists.

5. Character patterns: When possible, legal character patterns need to be identified. They can often be expressed as regular expressions. Inputs that do not match the pattern should be rejected.

2.4 Feasibility Study: CyberChair

As an initial feasibility study, we applied some of the bypass testing techniques to CyberChair, a Web-based paper submission and reviewing system [17] that is used by a number of conferences, including ISSRE. It has been in use since 1996, and was opened as free software for downloading in 2000. The CyberChair web site (www.cyberchair.org) listed 242 users in April 2004.

CyberChair has multiple phases to support a conference. Authors submit abstracts in the first phase, and then full papers in the second. We manually tested the submission page of the second phase. Tests were performed on the ISSRE 2004 conference server. We did not have access to the source code and did not download CyberChair. We started with a user id and access code from an abstract submission in the first phase. After logging in, CyberChair returns an HTML page with a form to submit papers. To implement bypass testing, we saved the page and then modified it.

In this early feasibility study, our test creation process was not formalized. We broke the inputs into three levels; the control flow level, parameter level, and value level. At the control flow level, we attempted to submit a paper without logging in. At the parameter level, we removed some parameters from the form and then submitted. At the value level, we tried various values for parameters, including values that are normally not expected by the software. This process revealed five types of faults, all of which are potential security holes.

1. <u>Submission without authentication</u>: After correctly logging in to CyberChair, a submission form is returned. We decided to attempt to use that form to submit without a valid login. We saved the page locally, and changed the `Action` attribute on the `FORM` tag from a relative URL to a complete URL. (A relative URL does not include a domain name and only works within a single session.) Then we copied the modified form to a second computer, and used it to submit a file. The submission was allowed, implying that the semantics of a login is to send the submission page, **not** to only allow authenticated users to submit. Whereas we used a valid login to find the submission page, it would not be difficult for someone to find or guess a valid URL for the submission, particularly since CyberChair is an open-source program.

2. <u>Unsafe use of hidden field</u>: The submission page uses a hidden field to track the user. We customized the submission form by changing the value of the hidden form field and were still able to submit the paper. This allows the possibility of overwriting another user's submission.

3. <u>Disclosing information</u>: We also tried removing the hidden field and setting its value to empty. In these cases, the software failed and returned messages that indicated in which file and which line of code the program failed. This kind of information is confusing to valid users and potentially unsafe to show to malicious hackers.

4. <u>No validation for parameter constraint</u>: The server does not check if the selected file type and the file submitted really match. For example, it is possible to select the file type to be pdf, but submit an rtf file instead. This lack of constraint checking can damage the state on the server.

5. <u>No data type or data value validation</u>: CyberChair asks the user to submit the number of pages, which should be an integer value between 1 and some fairly small number such as 10 or 15. We tried submitting non-integer values, negative numbers, and extremely large numbers, none of which were detected by the software. Similar problems are also found in other fields.

Although this experience is anectodal, and our process was fairly ad-hoc, it does demonstrate that bypass testing can be effective on software that has reasonably wide use. The rest of this paper provides a first attempt to formalize these ideas.

3 Modeling HTML Input Units

Web application include static HTML files and programs that dynamically generate HTML pages. HTML pages can include more than one form, and each form can include many input fields. For example, we identified 169 HTML hyperlinks and 20 forms on amazon.com's home page. This makes automatic input validation difficult to manage by hand, thus we take a first step toward automation by constructing a formal model for HTML client-server inputs.

Each HTML page, whether a static file or dynamically generated, can have zero or more HTML links and forms that let users interact with the server. An *input unit* $IU = (S,\ D,\ T)$ is the basic element of interaction on the client side. The inputs are sent to a software component on a server, S, and includes a set of input elements D. D is a set of ordered pairs, $(n,\ v)$, where n is a name (parameter) and v represents the set of values that can be assigned to n. The set of values may be unlimited, as in a text box, or finite, as with a selection input. It is sometimes convenient to think of these sets of values as defining a *type*. T is the HTTP transfer mode (GET, POST, etc.).

There are two types of input units. A *form input unit* is an HTML form that specifies the server component as the `Action` attribute within the `Form` tag, and the input data corresponds to all the input fields within the form. The transfer mode is specified within the `Method` attribute of the `Form` tag.

A *link input unit* is an HTML link in an `<A>` tag. A link input unit's server target can either be a static HTML file or a server program such as a servlet, and the server target is specified as the `HREF` attribute of the `<A>` tag. Link input units always generate GET requests and only have input elements when URL rewriting is used. In the HTML link `<A HREF="prog?val=1"`, S is `prog` and D is $\{(val, 1)\}$.

As an example, consider the screen shot of STIS in Figure 1. STIS helps users keep track of arbitrary textual information. The main part of the screen in Figure 1 contains two form input units and 12 link input units, plus the menu bars on the top and the bottom have 5 more link input units apiece. Key portions of the HTML for the search form input units and the delete link input units are shown in the callout bubbles.

3.1 Composing Input Units

Some Web pages may have large numbers of inputs, which can be difficult to manage. For example, it is common to have identical forms for things like searching and logging in. It is easy to eliminated this redundancy in static HTML pages, but harder for dynamically generated pages. Because the number of potential unique dynamically generated pages is arbitrary, and which pages are generated depends on inputs and processing on the server, the problem of identifying all input units from a client without access to the program source is undecidable.

When possible, the following composition rules are used to eliminate redundancy. To simplify the discussion, the following definitions assume two input units, each of which contains only one parameter: $\{iu_1 = (S_1,\ D_1,\ T_1),\ iu_2 = (S_2,\ D_2,\ T_2)\}$, and $D_1 = \{(n_1,\ v_1)\}$ and $D_2 = \{(n_2,\ v_2)\}$. All three composition rules require the two input units to have the same server component.

1. *Identical input units composition.* Two input units $iu_1 = (S_1,\ D_1,\ T_1)$ and $iu_2 = (S_2,\ D_2,\ T_2)$ are identical iff $S_1 = S_2$, $D_1 = D_2$ and $T_1 = T_2$. The two units are merged into a new input unit $iu = (S_1,\ D_1,\ T_1)$. For example, it is common for a Web page to have the same search form in two different places on the page.

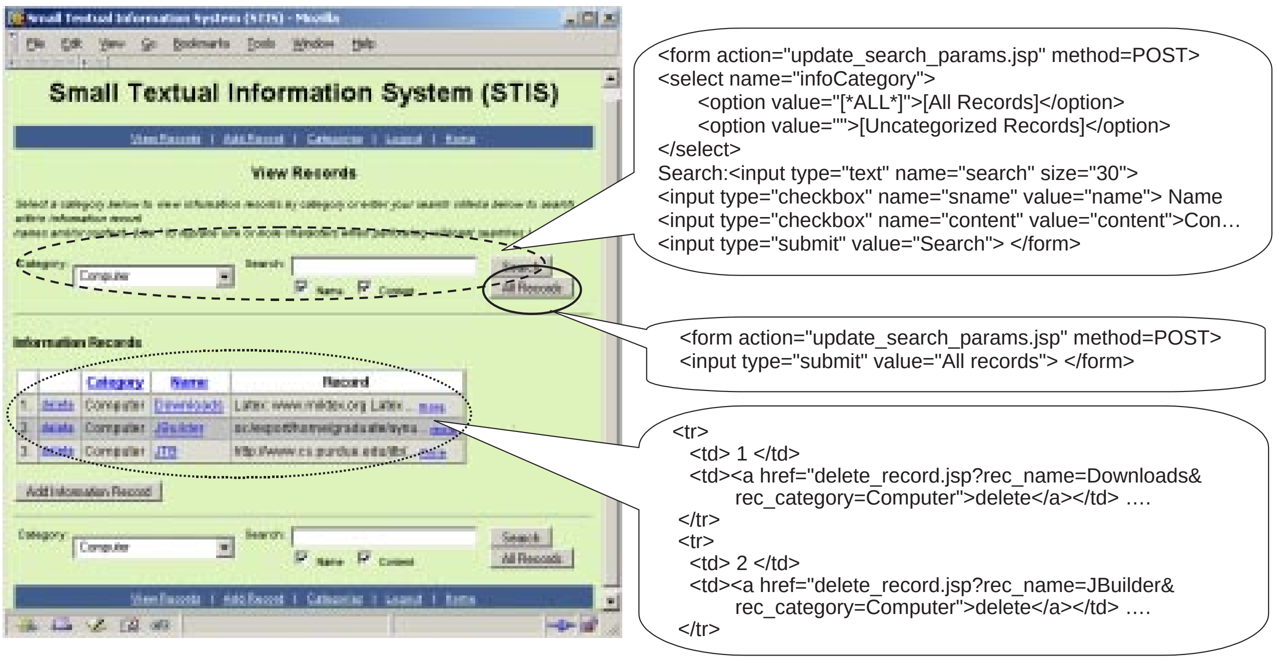
2. *Optional input element composition.* Two input units $iu_1 = (S_1,\ D_1,\ T_1)$ and $iu_2 = (S_2,\ D_2,\ T_2)$ have optional elements if $S_1 = S_2$, $T_1 = T_2$ and one input unit has an input element name that is not in the other. That is, there exists $(n_1,\ v_1) \in D_1$ such that there is no v_2 where $(n_1,\ v_2) \notin D_2$ or $(n_2,\ v_2) \in D_2$ such that there is no v_1 where $(n_2,\ v_1) \notin D_1$. The two input units are merged, forming $iu = (S_1,\ D^{'},\ T_1)$ where $D^{'} = \{D_1 \cup D_2\}$. This happens when a dymanically generated page sometimes includes different input elements, for example, if an order entry form sometimes includes an input box to enter discount coupon code.

3. *Optional input value composition.* Two input units $iu_1 = (S_1,\ D_1,\ T_1)$ and $iu_2 = (S_2,\ D_2,\ T_2)$ have optional input values if $S_1 = S_2$, $T_1 = T_2$ and there exists $(n_1,\ v_1) \in D_1$ and $(n_2,\ v_2) \in D_2$, such that $n_1 = n_2$ but $v_1 \neq v_2$. Then the two input units are merged, forming $iu = (S_1,\ D^{'},\ T_1)$ where $D^{'} = \{D_1 - (n_1,\ v_1)\} \cup \{D_1 - (n_2,\ v_2)\} \cup \{(n_1,\ (v_1 \cup v_2)\}$. This happens when a dymanically generated page sometimes includes different input values, for example, in an online grade entry form at our university, undergraduate courses have fewer choices for grades than graduate courses do.

The two search forms at the top and the bottom of the screen in Figure 1 are identical and are thus composed. The two search forms with buttons "Search" and "All Records" use the same server component, but have different input elements, thus can be merged under the optional input element composition rule. Finally, the three "delete" link input units all reference the same server component, so can be merged under the optional input value composition rule.

4 Bypass Testing

Most input validation focuses on individual parameters. This works well for traditional software, where the patterns of interaction between users and software are fixed and cannot be altered by the users. An interesting complexity is that the use of dynamic Web pages means that the same URL can produce different forms at different times, depending on the parameters supplied, state on the server, characteristics of the client, and other environmental information. Additionally, users of Web applications can not only change the values of input parameters, but can also change the number of input parameters and the control flow. This makes it easier to violate constraints among different parameters and between software components. This section describes a systematic approach to identify constraints among input parameters. Then rules are given to generate bypass test cases to test the Web application to ensure these con-



```
<form action="update_search_params.jsp" method=POST>
<select name="infoCategory">
    <option value="[*ALL*]">[All Records]</option>
    <option value="">[Uncategorized Records]</option>
</select>
Search:<input type="text" name="search" size="30">
<input type="checkbox" name="sname" value="name"> Name
<input type="checkbox" name="content" value="content">Con…
<input type="submit" value="Search"> </form>
```

```
<form action="update_search_params.jsp" method=POST>
<input type="submit" value="All records"> </form>
```

```
<tr>
  <td> 1 </td>
  <td><a href="delete_record.jsp?rec_name=Downloads&
        rec_category=Computer">delete</a></td> ….
</tr>
<tr>
  <td> 2 </td>
  <td><a href="delete_record.jsp?rec_name=JBuilder&
        rec_category=Computer">delete</a></td> ….
</tr>
```

Figure 1. STIS initial screen

straints are adequately evaluated. According to the classification of input validation types from Section 2 2, our bypassing testing will be conducted at three levels, as discussed in the following subsections.

4.1 Value Level Bypass Testing

This type of bypass testing tries to verify whether a Web application adequately evaluates invalid inputs. It addresses data type conversion, data value validation, and built-in input value restriction. This testing is based on the restrictions described in Section 2. Given a single input variable, invalid inputs can be generated according to the 14 types of input validation that are specified in Section 2:

- *Data type and value modification.* HTML inputs are initially strings, but they are often converted to other data types on the server. Data type conversion testing uses values of different types to evaluate the server-side processing, including general strings, integers, real numbers, and dates.

- *HTML built-in length violation.* The HTML tag *input* has an attribute *maxlength*, as described in Section 2. Invalid values are generated to violate these restrictions.

- *HTML built-in value violation.* Pre-defined input restrictions from HTML select, check and radio boxes are violated by modifying the submission to submit values that are not in the pre-defined set.

- *Special input value.* When data is stored into a database or XML document, and under certain kinds of processing, some special characters, as defined in Table 1, can corrupt the data or cause the software to fail. This data is often validated with client-side checking, but sometimes with server-side checking. Thus, following Wheeler's suggestions [18], values for text fields are generated with special characters such a commas, directory paths, slashes and ampersands.

4.2 Parameter Level Bypass Testing

This type of bypass testing tries to address issues related to built-in input parameter selection, built-in data access and inter-value constraints.

It is relatively easy to enumerate possible invalid inputs for an input parameter. However, the restrictive relationships among different parameters are hard to identify, hard to validate and are thus often ignored during testing. There are many kinds of relationships. One type is *invalid pair*, where two parameters cannot both have values at the same time. For example, it is not reasonable to have a checking account number and a credit card expiration date in the same transaction. Another type is *required pair*, where if one parameter has a value, the other must also have a value. For example, if we have a credit card number, we must also have an expiration date. *Parameter level bypass testing* tries to test Web application by executing test cases that violate restrictive relationships among multiple parameters.

These relationships are very often difficult to obtain statically and must be identified dynamically. They are sometimes described in English-language instructions, and sometimes simply assumed. Nevertheless, if we can identify and follow all possible ways to send parameters to a server program, we can ensure conformance to the restrictive relationships, and then find values to violate the restrictive relationships. Thus, we define the *input pattern* to be the set of parameters that are sent to a server. If an optional input element composition has been applied to create iu, the input patterns in iu will correspond to the parameters from the original input units.

The following algorithm is designed to derive all possible input patterns in a Web application. As with finding input units, this is generally an undecidable problem without access to the server program source. Thus, this algorithm creates an approximation that is limited by the data that is supplied to existing forms in Step 2. The input patterns created by the algorithm are used to generate parameter level bypass tests.

Algorithm:	Identify input patterns of web applications
Input:	The start page of a web application, S
Output:	Identifiable input patterns

Step 1 : Create a stack *ST* to retain all input units that need to be explored. Initialize *ST* to *S*. Create a set *IUS* to retain all input units that have been identified. Initialize *IUS* to empty.

Step 2 : While *ST* is not empty, pop an *input unit* (defined in Section 3) from *ST*, generate data for the input unit and send it to the server. When a reply is returned, analyze the HTML content. For each input unit *iu*:

- if iu is a link input unit, and iu does not belong to a different server, do **not** push iu onto the stack.
- if $iu \in IUS$ (it has already been found), do **not** push iu onto the stack.
- if there exists an input unit $iu^{'} \in IUS$ such that iu and $iu^{'}$ have optional input elements, update the possible value of iu. Do **not** push iu onto the stack.
- Otherwise, a new input pattern has been identified; add iu to *IUS* as an optional input unit, and then push iu onto *ST*.

After executing the algorithm, we have a collection of input units $IU = (S,\ D,\ T)$, where $D = \{P_1, P_2, ..., P_k\}$ and $P_i = \{(n_1^i, v_1^i), (n_2^i, v_2^i), ..., (n_a^i, v_a^i)\}$. Each P_i is a valid input pattern for the input unit IU. Based on the input patterns, we generate three types of invalid input patterns to test the restrictive relationships among parameters.

- The *empty input pattern* submits no data to the server component. Formally, $IU1 = (S,\ \phi,\ T)$. The empty input pattern will violate all *required pair* restrictive relationships.
- The *universal input pattern* submits values for all parameters that the server component knows about. Formally, $IU2 = (S,\ P_1 \bigcup P_2 \bigcup ... \bigcup P_k,\ T)$. The universal input pattern will violate all *invalid pair* restrictive relationships.
- The *differential input pattern* submits appropriate values for all parameters in one input pattern, plus a value for **one** parameter that is **not** in that input pattern. For each pair of input patterns P_i and P_j, generate an invalid input pattern in the following way. x is a parameter from $P_j - P_i$, chosen arbitrarily. $IU3 = (S,\ P^{'},\ T)$, where $P^{'} = P_i \bigcup \{x\}$. The intent of the differential input pattern is to make subtle changes that are not likely to be identified by checks other than invalid input checking.

Parameter level bypass testing focuses on relationships among different parameters, therefore, all values of input parameters are selected from a set of valid values.

4.3 Control Flow Level Bypass Testing

The previous two types of bypass testing assume users follow the control flow that is defined by the software. This is a safe assumption for traditional software applications. However, users of Web applications can alter the control flow by pressing the back button, pressing the refresh button, or by directly entering a URL into a browser. This ability adds uncertainty and threatens the reliability of Web applications.

Control flow level bypass testing tries to verify Web applications by executing test cases that break the normal execution sequence. As a first step, the "normal" control flow must be identified. The algorithm for finding input patterns in the previous section provides the needed information. The input units that were identified can be used to define all normal control flows from that unit. So we expand the algorithm to derive all normal control flow for all input units. In the algorithm, an input unit iu is popped form the stack, data is supplied, then sent to the server. All the input units that are returned from that submission are considered to be candidates for the next step in the control flow. Given that, control flow bypass testing is carried out according to the following two categories:

1. **Backward and forward control flow alteration**. Given a normal control flow $iu_1, iu_2, ..., iu_k$, each pair of input units (iu_i, iu_{i+1}) forms a transition. Model use of the back button by changing each transition (iu_i, iu_{i+1}) to (iu_i, iu_{i-1}). Model use of the forward button by changing each transition (iu_i, iu_{i+1}) to (iu_i, iu_{i+2}).

2. **Arbitrary control flow alteration**. Given a normal control flow $iu_1, iu_2, ..., iu_k$, for each input unit iu_i, $1 \leq i \leq k$, change the control iu_i to some arbitrary iu_t, such at $t \neq i$.

4.4 Summary of Bypass Testing

The three levels of testing in this section, value level, parameter level, and control flow level, can be used individually or combined together. Parameter level and control flow level bypass testing focus on interactions among different parameters and different server components, thus can be run independently of value level bypass testing.

5 Empirical Validation

As an initial validation, we applied bypass testing to a small but non-trivial web application, the Small Text Information System (STIS). STIS helps users keep track of arbitrary textual information. It stores all information in a database (currently mysql) and is comprised of 17 Java Server Pages and 5 Java bean classes. Eight of the JSPs process parameterized requests, *login.jsp*, *browse.jsp*, *record_edit.jsp*, *record_delete.jsp*, *record_insert.jsp*, *categories.jsp*, *category_edit.jsp* and *register_save.jsp*. We extensively tested these eight JSPs with bypass testing.

When a Web application receives invalid inputs, there are three possible types of server responses. (1) The invalid inputs **are recognized** by the server and adequately processed by the server. (2) The invalid inputs **are not recognized** and cause abnormal server behavior, but the abnormal behavior is caught and automatically processed by server error handling mechanism. (3) The invalid inputs **are not recognized** and the abnormal server behavior is **exposed** directly to the users. Abnormal server behavior includes responses like run time exceptions and revealing confidential information to unauthorized clients. A type 1 response represents proper server behavior, while type 2 and 3 responses represent inadequate server behavior and are considered to be failures.

Some inputs had to be created by hand for bypass testing, including user names and passwords (STIS has two levels of access) and some very long invalid input strings. Other inputs were either automatically extracted from the HTML files or randomly generated. For comparison, we generated four levels of tests, for just the value level, the parameter level but not control, the control level but not parameter, and both the parameter and control level.

Table 2 summarizes the results. For each group of tests, the number of tests (T) and the number of tests that caused a failure (F) are shown. There were a total of 158 tests, 66 of which caused failures. Of these 158 tests, none of the parameter level or control level tests could be executed without bypass testing, and only 55 of the value level tests could be executed without bypass testing. These 55 tests only caused 9 failures.

6 Related Work

The bypass testing techniques are motivated by a combination of input validation and the category-partition method [15], a multi-step method to derive test frames and tests from specifications. The rest of this section discusses the most closely related test ideas, input validation testing and testing of graphical user interfaces.

6.1 Input Validation Testing

Input validation analysis and testing involves statically analyzing the input command syntax as defined in interface and requirement specifications and then gener-

Table 2. Failures found for each dynamic component
I: Value Level, No Parameter or Control
II: Parameter Level, No Control Level
III: Control Level, No Parameter Level
IV: Parameter Level and Control Level
T = number of tests, F = number of failures

Component	I		II		III		IV		Total	
	T	F	T	F	T	F	T	F	T	F
login	15	0	2	2	n/a		n/a		17	2
browse	7	4	1	0	1	1	1	1	10	6
record_edit	17	9	5	2	1	1	5	5	28	17
record_delete	5	0	2	0	1	1	2	2	10	3
record_insert	13	9	3	1	1	1	3	3	20	14
categories	12	2	2	0	1	0	2	0	17	2
category_edit	13	2	2	0	1	0	2	0	18	2
register_save	25	11	6	3	1	0	6	6	38	19
Total (#tests & #failures)	107	37	23	8	7	4	21	17	158	66

ating input data from the specification. Hayes and Offutt [6] proposed techniques for input validation analysis and testing for systems that take inputs that can be represented in grammars. Both IVT and bypass testing attempt to violate input specifications, so bypass testing could be viewed as a special kind of IVT that addresses concerns of Web applications.

6.2 GUI Testing

HTML forms can be considered to offer a graphical user interface to run software that is deployed across the Web. Memon has developed techniques to test software through their GUIs by creating inputs that match the input specifications of the software [13, 12]. This approach focuses on the layout of graphical elements and the user's interaction when supplying form data. Bypass testing relies on following the syntax of the GUI forms, but specifically finds ways to violate constraints imposed by the syntax. The two approaches are complementary, specifically, GUI testing could be used to develop values for bypass testing.

6.3 Web Application Testing

Most research in testing Web applications has focused on client-side validation and static server-side validation of links. An extensive listing of existing Web test support tools is on a Web site maintained by Hower [7]. The list includes link checking tools, HTML validators, capture/playback tools, security test tools, and load and performance stress tools. These are all static validation and measurement tools, none of which support functional testing or black box testing.

The Web Modeling Language (WebML) [4] allows Web sites to be conceptually described. The focus of WebML is primarily from the user's view and the data modeling. Our model derived from the software is complementary to the solutions proposed by WebML.

More recent research has looked into testing software from a static view, but few researchers have addressed the problem of dynamic integration. Kung et al. [9, 11] have developed a model to represent Web sites as a graph, and provide preliminary definitions for developing tests based on the graph in terms of Web page traversals. Their model includes static link transitions and focuses on the client side without limited use of the server software. They define *intra-object* testing, where test paths are selected for the variables that have def-use chains within the object, *inter-object* testing, where test paths are selected for variables that have def-use chains across objects, and *inter-client* testing, where tests are derived from a reachability graph that is related to the data interactions among clients.

Ricca and Tonella [16] proposed an analysis model and corresponding testing strategies for *static* Web page analysis. As Web technologies have developed, more and more Web applications are being built on dynamic content, and therefore strategies are needed to model these dynamic behaviors.

Benedikt, Freire and Godefroid [3] presented Veri-

Web, a navigation testing tool for Web applications. VeriWeb explores sequences of links in Web applications by nondeterministically exploring "action sequences", starting from a given URL. Excessively long sequences of links are limited by pruning paths in a derivative form of prime path coverage. VeriWeb creates data for form fields by choosing from a set of name-value pairs that are initialized by the tester. VeriWeb's testing is based on graphs where nodes are Web pages and edges are explicit HTML links, and the size of the graphs is controlled by a pruning process. This is similar to our algorithm, but does not handle dynamically generated HTML pages.

Elbaum, Karre and Rothermel [5] proposed a method to use what they called "user session data" to generate test cases for Web applications. Their use of the term user session data was nonstandard for Web application developers. Instead of looking at the data kept in J2EE servlet session, their definition of user session data was input data collected and remembered from previous user sessions. The user data was captured from HTML forms and included name-value pairs. Experimental results from comparing their method with existing methods show that user session data can help produce effective test suites with very little expense.

Lee and Offutt [10] describe a system that generates test cases using a form of mutation analysis. It focuses on validating the reliability of data interactions among Web-based software system components. Specifically, it considers XML based component interactions.

Jia and Liu [8] propose an approach for formally describing tests for Web applications using XML. A prototype tool, WebTest, based on this approach was also developed. Their XML approach could be combined with the test criteria proposed in this paper to express the tests in XML.

Andrews et al. use hierarchical FSMs to model potentially large Web applications. Test sequences are generated based on FSMs and use input constraints to reduce the state space explosion [1]. Finally, our previous work on modeling of Web applications has led to the development of *atomic sections*, which can be used to model dynamic aspects of Web applications [19]. This approach is at the detailed analysis level and relies on access to the code, unlike bypass testing.

7 Conclusions

This paper has presented four results. First, the concept of *bypass testing* was introduced to submit values to Web applications that are not validated by client-side checking. Bypass testing requires a detailed model for how to introduce inputs to server-side software components, thus we developed one. Third, this model supports more general input validation testing, and rules are defined for bypass and input validation. Finally, empirical results from an open-source conference management system and our own laboratory-built Web application were shown.

Bypass testing is a unique and novel way to create test cases that is available only because of the unusual mix of client-server, HTML GUI, and JavaScript technologies that are used in Web applications. It is also deceptively complicated. Although the concept is relatively simple, to submit inputs that violate client-side constraints, the distributed and heterogeneous nature of Web applications brings in many complexities. Not surprisingly, the most complicated part is handling inputs to dynamically generated HTML forms. The algorithm presented in Section 4.2 is a first attempt to approximate the kinds of input forms that can be generated dynamically.

The existence of bypass testing may motivate Web application developers to check data on the server, obviating much of the need for bypass testing. This may already be a trend in the industry. Five years ago, many books on Web software advocated checking inputs with JavaScript as a mechanism to reduce network traffic; modern books and instructors usually advocate doing input validation on the server. Nevertheless, major e-commerce and e-service sites still use client-side checking and hidden form fields. We found client-side checking on `amazon.com` and `netflix.com`, and the use of hidden form fields to store sensitive information on `fastlane.nsf.com`. The long history of buffer-overflow problems leads us to be somewhat pessimistic that developers will develop software well enough to make bypass testing completely obsolete.

A major advantage of bypass testing is that it does not require access to the source of the back-end software. This greatly simplifies the generation of tests and automated tools, and we expect bypass tests can be generated automatically. Our current plan is to build tools that parse HTML, discover and analyze the form field elements, parse the client-side checking encoded in the JavaScript, and automatically generate bypass tests to evaluate the server-side software.

References

[1] Anneliese Andrews, Jeff Offutt, and Roger Alexander. Testing Web applications. *Software*

and Systems Modeling, 2004. Revision submitted.

[2] Chris Anley. Advanced SQL injection in SQL server applications. online, 2004. http://www.nextgenss.com/papers/advanced _sql_injection.pdf, last access February 2004.

[3] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic Web sites. In *Proceedings of 11th International World Wide Web Conference (WW W'2002)*, Honolulu, HI, May 2002.

[4] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (WebML): A modeling language for designing Web sites. In *Ninth World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[5] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Portland, Oregon, May 2003. IEEE Computer Society Press.

[6] J. H. Hayes and J. Offutt. Increased software reliability through input validation analysis and testing. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 199–209, Boca Raton, FL, November 1999. IEEE Computer Society Press.

[7] Rick Hower. Web site test tools and site management tools, 2002. www.softwareqatest.com/qatweb1.html.

[8] Xiaoping Jia and Hongming Liu. Rigorous and automatic testing of Web applications. In *6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 280–285, Cambridge, MA, November 2002.

[9] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *Proc. of IEEE 24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, pages 537–542, Taipei, Taiwan, October 2000.

[10] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[11] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structural testing of Web applications. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 84–96, San Jose CA, October 2000. IEEE Computer Society Press.

[12] A. M. Memon, M. L. Soffa, and M. E. Pollack. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[13] Atif M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):90–91, August 2002.

[14] Jeff Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.

[15] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[16] F. Ricca and P. Tonella. Analysis and testing of web applications. In *23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34, Toronto, CA, May 2001.

[17] Richard van de Stadt. Cyberchair: A free web-based paper submission and reviewing system. online, 2004. http://www.cyberchair.org/, last access April 2004.

[18] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Published online, March 2003. http://www.dwheeler.com/secure-programs/, last access Feb 2004.

[19] Ye Wu, Jeff Offutt, and Xiaochen Du. Modeling and testing of dynamic aspects of Web applications. *Submitted for publication*, 2004. Technical Report ISE-TR-04-01, www.ise.gmu.edu/techreps/.