

# A Case Study on Bypass Testing of Web Applications

Jeff Offutt, Vasileios Papadimitriou, and Upsorn Praphamontripong  
Software Engineering  
George Mason University  
Fairfax, VA, USA  
offutt@gmu.edu, vpapadim@gmail.com, uprapham@gmu.edu

*Accepted for publication, Empirical Software Engineering journal, 20-June-2012*

## Abstract

*Society's increasing reliance on services provided by web applications places a high demand on their reliability. The flow of control through web applications heavily depends on user inputs and interactions, so user inputs should be thoroughly validated before being passed to the back-end software. Although several techniques are used to validate inputs on the client, users can easily **bypass** this validation and submit arbitrary data to the server. This can cause unexpected behavior, and even allow unauthorized access. A test technique called bypass testing intentionally sends invalid data to the server by bypassing client-side validation. This paper reports results from a comprehensive case study on 16 deployed, widely used, commercial web applications. As part of this project, the theory behind bypass testing was extended and an automated tool, AutoBypass, was built. The case study found failures in 14 of the 16 web applications tested, some significant. This study gives evidence that bypass testing is effective, has positive return on investment, and scales to real applications.*

## 1 Introduction

Users expect web applications to be accessible from anywhere and anytime. This research considers web applications to be user interactive software applications deployed on the Web [4]. They usually are built with heterogeneous software components that may be distributed across computers or organizations. Not only are web software components developed and integrated dynamically with diverse techniques [22], but they are also triggered and governed by users' interactions. Hence, the control flow of a web application depends on the inputs the users provide, making web applications extremely vulnerable to invalid inputs that compromise functionality and security. In the 2003-2004 Common Vulnerability and Exposure report [42], Xu *et al.* reported that approximately 40% of web application security problems originate from implementation errors that allow attackers to inject malicious code into carefully crafted inputs. In 2011, the CWE/SANS Top 25 report [7] reports that a principle attack vector is input validation. Furthermore, inappropriate user inputs often cause functional failures. Therefore, it is necessary to validate user inputs to ensure that they will not cause the web applications to fail, corrupt data on the server, or provide unauthorized access to sensitive information [25, 26].

Web applications use various technologies to validate users' inputs and control users' interactions with the software interfaces. Some validation is done on the client's computer and some on the server. Client-side input validation uses HTML tags and attributes to impose constraints and uses scripts that run inside the browsers to perform semantic checks on the inputs. Server-side input validation relies on a variety of programming languages and resources to identify invalid input requests. The study presented in this paper uses bypass testing to skip the client-side input validation and systematically explore invalid portions of the input space.

Web application interfaces use browser events and scripting languages to react to user's actions and block submission of data that do not meet input requirements. This is called *input validation*, which this research views as a guard, or a barrier, that only allows valid data to pass. However, two factors prevent web servers from completely preventing invalid inputs. First, HTML source and scripts are available to the users and can be modified and resubmitted to the server, that is, *bypassing*

the guard [25]. Second, the Hyper Text Transfer Protocol (HTTP) is *stateless*, making it easy for clients to disguise the fact that multiple requests are related, or make it seem as if requests from different computers are related. Thus, despite the constraints in the user interface, arbitrary requests and inputs are allowed from users.

Bypass testing [25] was proposed to provide invalid inputs directly to the server as an attempt to evaluate software's behavior in the presence of invalid inputs. The goal of this paper is to evaluate whether bypass testing can be successful in practical situations. Can a robust tool be built to support bypass testing? Can bypass testing scale to full commercial applications? Can bypass testing find failures in real software?

This paper has several contributions. First, the paper presents significant refinements and extensions to the theory of bypass testing. Second, the paper presents a tool that supports bypass testing. AutoBypass runs as a web application that accepts a URL to another web application under test and generates test inputs based on the subject's HTML form fields. Third, the tool, AutoBypass, was used in a case study. The paper presents results from 16 commercial web applications, many of which are widely used.

This paper is organized as follows. Section 2 describes the concepts behind bypass testing, including the constraints for input validation, rules for test data generation, and the new theoretical ideas that were developed as part of this project. An automated bypass testing tool, AutoBypass, is presented in Section 3. Results are presented in Section 4. Section 5 provides an overview of related research and Section 6 concludes with suggestions for further research.

## 2 Bypass Testing

Input data to web applications are subjected to checks at various places, including the input form, software running on the client (JavaScript), the presentation layer on the server, and during detailed processing of the data on the server. Bypass testing views these checks as imposing *constraints* on the input space. This section presents specific, common constraints for client input validation, which have been substantially revised and extended from Offutt *et al.* [25]. The focus of these constraints is on client-side validation as implemented in HTML and JavaScript in the users' browsers. These constraints are then used to define rules for test data generation. The rules are formulated as *test requirements*, which direct test design.

### 2.1 Types of Client Input Validation

Client-side input validation uses three elements to evaluate the inputs or check for required fields before a form is submitted: HTML form field elements, their attributes, and scripts that can access the Document Object Model (DOM). Client-side input validation is separated into two general types, HTML validation and scripting validation.

**HTML Validation:** HTML validation uses elements in the HTML language [28] that define input fields, or *controls*, to implicitly limit user inputs. HTML has five types of input validation:

1. *Length constraints* specify the maximum number of characters allowed in a text field using the HTML attribute *maxlength*. For instance, the maximum number of digits in the US social security number is 9 (or 11 if dashes are included), and text inputs for US bank routing and account numbers are restricted to 9 characters. If too many characters are used when wiring money, the result will be an unknown bank or an unknown account. In some cases, money may be transferred to a different bank account as the length of bank routing and account numbers may differ for non-USA banks; as an example, Thai banks use 10 digits for bank account numbers. However, users can save and modify the HTML, removing or modifying the *maxlength* attribute, then load the modified HTML in their browsers and submit the server.
2. *Value constraints* specify sets of specific values that the user can submit for an input selection. For example, the values from a drop-down list or a collection of check boxes are limited by the predefined values in the HTML. Thus these controls explicitly constrain the set of values available to the user. Also, a *hidden* form field is an HTML input element that has a single predefined value and that the browser does **not** display to the user. When the user submits, however, that value is sent to the server. Web programmers sometimes use hidden form fields to pass state information, for example user IDs, from one user request to the next. This research considers the following types of controls that define value constraints:
  - *Hidden controls*, normally created with the HTML *input* element, signify predefined values outside of the user's knowledge and explicitly dictate the values available to the form.

- *Menu controls* include the *select* element, the *option group*, and the *option* elements. These controls supply users options from which to choose. The selections may be single or multiple.
- *Boolean controls* are input elements of type checkbox or radio button whose values are toggled by the user. While checkboxes allow multiple selections, radio buttons allow mutually exclusive selection.
- *Read-Only controls* are *textarea* elements and text controls with the *readonly* attribute set, and hence their values cannot be changed.

However, it is very important to note that, unlike with traditional GUIs, users can save and modify HTML pages (screens)<sup>1</sup>, thus submitting data that are not in the original list of values.

3. *Transfer mode constraints* control how data are transmitted from the web browser to the web server. HTTP defines nine *transfer modes* (or *request methods*): HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, and PATCH. The most commonly used in web applications are GET and POST, and thus are the only ones considered in this research. A GET request transfers information in Unicode as parameters appended to the URL separated by a question mark (?). Their values are visible to the user in the URL. Most browsers have a maximum URL length, which limits the number of characters that can be transmitted by GET requests. A POST request is constructed as an HTTP message to the action URL and does not limit the amount of data transmitted. However, users can change the transfer mode at will.
4. *Field element constraints* determine which data fields are submitted to the application. In general, the fields submitted to the server application are predefined, and the values are either explicitly assigned in controls that render on HTML screens or implicitly in non-rendered fields (such as hidden fields). Thus, web applications constrain users to submit values for only a pre-defined collection of data fields. However, unlike with traditional GUIs, web users can submit values for other fields that do **not** appear in the form being used.
5. *Target URL constraints* control the URLs to which the users are allowed to navigate from the current screen. However, users can modify the target URLs by modifying the HTML or by altering the string in the URL form in their browsers.

Scripting Validation: HTML screens may contain scripts to respond to user events and validate data inputs before the forms are submitted. By associating actions with form elements, scripts can validate form data in all the ways that HTML syntax can, as well as more. For example, client scripts can enforce application-specific requirements and apply semantic restrictions, such as all characters in a string must be numeric. Several scripting languages are in use, but JavaScript is the most common, so this work focuses on JavaScript and emphasizes semantic limits on data entry. Bypass testing considers five types of scripting validation:

1. *Data type constraints* specify the appropriate types of data to be transmitted to the server. For example, an age field should only accept integers. To ensure that the server software component receives appropriate values, a scripting function can verify the appropriate data by checking the input control and, if necessary, asking the user to correct invalid inputs.
2. *Data format constraints* specify the appropriate format of data to be submitted to the server. For instance, in the US, a zip code is a string of five digits and a phone number is typically comprised of the area code and the local number (10 digits). Other types of values that are often checked include the format of money, personal identification numbers, email address, and URLs.
3. *Data value constraints* specify the appropriate data values when fields have semantic restrictions. As an example, an age field should not only prohibit non-integer values (as a data type constraint), but also limit the value of the integer to an appropriate range (for example, 0-150).
4. *Inter-value constraints* describe fields that are related in the sense that they **all** need valid values before being submitted. For instance, when payment information is required, a choice of a credit card payment should require the input of credit card type, account number, and an expiration date. As another example, if requesting a direct deposit for tax return, two inputs are required: a bank routing number and account number. Omitting any of the related values makes the transaction incomplete or invalid; therefore, scripts are used to verify that all required fields are completed before the form data are submitted.

---

<sup>1</sup>To emphasize that this research tests programs, not static web sites, we use the term "screens" when the web page is used as part of the web application.

5. *Invalid characters constraints* indicate invalid strings that can cause reliability or security threats to the server. As an example, several types of invalid characters such as XML tags (<, >) and directory separators (../) can crash web applications. Not only do inputs with invalid characters affect the security of the applications, they also influence their robustness. To avoid this vulnerability, scripts verify the input data against predefined unacceptable characters.

Client/Server Asynchronous Validation: Recent years have seen the advent of technologies collectively called Ajax (“Asynchronous JavaScript and XML”), which allow data to be transmitted back and forth between the client and the server in an asynchronous manner, that is, without involving explicit user submissions or the standard HTTP request/response cycle [39]. Ajax is not normally used for data validation and although AutoBypass can parse and test web sites that use Ajax, explicit generation of values to test the use of Ajax is currently out of scope for this research. Ajax is discussed explicitly as future work in Section 6. The AutoBypass tool can test web application interfaces that use Ajax, but does not generate values specifically to test the Ajax interaction.

## 2.2 Rules for Defining Test Inputs

Bypass testing generates test inputs that violate the constraints described in Section 2.1. The constraints usually describe a set of allowable values, so the complementary set of disallowed values is often infinite. Thus, a notion of completeness is not practical. This research gets values from three sources: (1) values that appear elsewhere in the HTML for similar parameters, (2) values that were created for other tests for similar parameters, and (3) values that are close to the boundary of the set of allowed values. In addition, the tester can (but is not required to!) include domain knowledge of the program in the tests by providing additional values. For example, the tester may need to supply login values, or specific strings if the application requires a specific format for values. The rules for test input generation are grouped into two categories corresponding to the constraints: rules for violating HTML validation constraints and rules for violating scripting validation constraints.

Rules for Violating HTML Validation Constraints: The following rules are used to generate test inputs that violate HTML client validation. Bypass testing looks at the HTML and embedded JavaScript that is transmitted to the client, but does not use any source that runs on the server.

1. *Length constraint violations* submit a string value with length =  $maxlength + 1$ . The *maxlength* attributes are obtained directly from the HTML source, excluding disabled controls. Test input values are obtained by searching other values used in the same web application, starting with values assigned to the current parameter and continuing to values assigned to other parameters. Values from this *input domain* are used if they violate the length constraint. If values are found, a random string value of length  $maxlength + 1$  is generated to replace the default value.
2. *Value constraint violations* submit values outside a predefined set of values such as empty strings and modified values (that is, adding new values or value omission). The violation rules for the preset values are:
  - Replace existing values with an empty string.
  - Replace the original value of a hidden control with a modified value.
  - Omit values of the *readonly* attribute of a read-only control.
  - For single-value controls, such as selection menus, radio button groups, and checkboxes that do not belong in a group, the original value is replaced with a modified value.
  - For multi-value controls (including checkboxes that are grouped by using the same identifier and selection menus that allow multiple selections), a combination of valid values is partially replaced with modified values. Combinations of the possible valid-invalid sets are avoided due to the potential for generating an enormous number of tests. Future systems could use a more formal approach, perhaps a combination strategy such as base choice [1].
  - Image buttons cause submission of the form data appended with the control’s name and its x and y pixel coordinates relative to the image location. The existing values (that is, coordinates x and y) are replaced by different values. Hence, an image will function as a different reference.
  - For file upload input controls, the original file is replaced with corrupted files, files with different extension, and malicious code.
3. *Transfer mode violations* change the method of HTTP transfer, from GET to POST and POST to GET.

4. *Field element constraints violations* submit forms that include a set of controls that are different from the predefined selection. For example, an HTML form that consists of a user name and a password input has a defined set of two inputs. This constraint is violated by adding or removing fields. Specifically, sets of controls may be modified by:
  - *Control omission* removes controls from forms. For a form with  $n$  controls, the number of possible modified forms that can be generated is the sum of all possible combinations of the controls. That is, the potential number of tests is  $\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \dots + \binom{n}{n-1} + \binom{n}{n}$ . To avoid creating such a large number of tests, only one form control is removed at a time.
  - *Control addition* adds new controls into forms. For effective testing, the added controls must exist somewhere in the web application being tested. To violate the field element constraints, a form is modified by adding disabled controls or other buttons defined in the form that are not originally used for submission.
  - *Control number alteration* changes the number of values sent to the server for a control. Form controls can have single values or multiple values. For example, *radio* buttons only allow one value and *select* buttons allow multiple values. All successful controls in a form are transmitted as name-value pairs to the server, so changing a radio button to a checkbox will still be syntactically valid on the server. This rule changes the number of values that are sent from single-value to multiple-value and from multiple-value to single-value.
5. *Target URL violations* alter the parameters and their values by dropping parameters and using alternative URL paths. By altering the parameters and their values, test inputs are created that interact with the state of the server as unexpected requests are received. However, the alternative URLs must exist in the web application under test, pointing to different components in the same application domain, to be effective.

Rules for Violating Scripting Validation Constraints: Two rules are used to generate test inputs that bypass the client scripting (JavaScript) validation:

1. For general situations, the scripts are parsed to detect error states. The most common error state is represented by an *alert box* statement in functions that are triggered upon form submission. Then, input restrictions are identified from the conditions under which the error states are reached. Finally, tests that include values violating the interface's validation are generated. As an example, assume that an input field exists to enter the user's age and a JavaScript checks that the value is between 0 and 150. To violate this constraint, data of invalid type or out of range (for example, *-1*, *230*, or *someText*), are created. So invalid characters (including empty strings, commas, directory paths, ampersands, strings that start with forward slash, strings starting with a period, control characters, characters with high bit set, and XML tag characters) are created.

The general problem of detecting all error states and generating values to reach them is generally undecidable. The approach in this research is classical engineering; to do as well as can be done in reasonable time to ensure good quality tests are created. Specifically, HTML alert boxes and obvious error messages (messages that are added to the HTML and include specified strings such as "error" and "invalid") are located and values are generated that reach the code that display the alert boxes or print the error messages. This is done using simplified versions of the dynamic domain reduction algorithm developed for automatic test data generation [23, 5]. In brief, a path from the beginning of a JavaScript function to an alert box is identified, then the decisions along the path are used to construct values that reach the alert box.

2. Inter-value constraints are violated by (i) submitting related, required fields with no values, or (ii) submitting values that violate other constraints. For instance, consider a web application about cars; a form may include a drop down list for users to select a vehicle make and another drop down list to select a vehicle model. The selection on the car make list will dynamically affect the values displayed on the model list.

### 3 Automating Bypass Testing

AutoBypass is a web application that was built to demonstrate the feasibility of bypass testing and to support empirical studies. Tests are designed using the rules in Section 2 and automated in HttpUnit [10]. HttpUnit is a Java API that allows the tester to supply a URL, a transfer mode (GET, POST, etc.), and a list of name-value pairs, then uses HTTP to send the request to the URL independently of a browser. HttpUnit was designed to help automate the testing of web applications and to ease the creation of web crawlers.

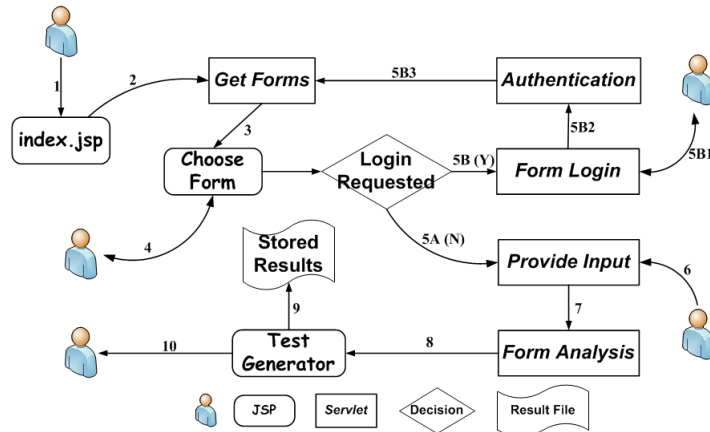
AutoBypass accepts a URL of a web application to be tested, called the *subject*. No server-side source code is required. HTML screens are parsed to identify all HTML form fields.

To explore the entire application, AutoBypass uses the initial URL as a *root* screen, identifies screens that can be reached from the root URL, then searches those screens. This process is repeated recursively in a breadth-first search in an attempt to discover all web screens in the application that can be reached from the root URL.

It is possible that some screens can only be reached if certain specific values are supplied; for example, in a login screen the account and password must be valid to authenticate the user. Since this information cannot be known or guessed, AutoBypass allows the tester to supply values if desired.

An interesting challenge in testing dynamic web applications is that the same URL can produce different forms with different elements at different times, depending on the parameters supplied, state on the server, characteristics of the client, and other environmental information [24]. In extreme situations, some links will not appear on a screen unless very specific values were chosen on a previous screen. This means that the problem of finding all screens associated with a web application can be reduced to the halting problem—thus is formally undecidable [11]. This is a limitation for all web application testing tools that do not access the source. Again, AutoBypass allows the tester to supply additional inputs if desired.

Another problem with automatically generating test inputs for web applications is that many fields are plain text fields on the client, and the data are sent as plain text, but they are interpreted to have a more strict syntax on the server. Without more semantic information than the HTML contains, arbitrary string values are sometimes not useful. AutoBypass addresses this problem by allowing the tester to define values when desired. That is, testers can be partially involved in the test design and generation. This mechanism is described in more detail below.



Sequence	Event
1	The tester provides a URL for the application to be tested
2	All anchors, forms, and script elements are obtained and made available to the application
3	The request is forwarded to a component that presents the forms and the URLs found
4	The tester is presented with a list of forms and URLs available to select for testing
5A	No authentication is required (proceed to Seq. 6)
5B	The tester chooses to test password protected components
5B1	The tester provides authentication information for the application under test
5B2	Submit login name and password to the application. A session is established by the WebConversation object and the subject application (return to Seq. 4)
6	The tester provides test inputs (not required)
7	Bypass rules are applied to create tests
8	Submit bypass values to TestGenerator, which automates and runs the tests
9	Test responses are stored to a file
10	Result summary presented to the tester

Figure 1. AutoBypass Flow of Execution

Figure 1 shows the sequence flow of execution in AutoBypass along with interactions of the user with the various components of the tool. When using AutoBypass, the tester enters a URL of the web application to be tested. The tool checks whether the given URL is valid and provides feedback to the tester if not. HTML forms and controls are discovered and recorded for later test input generation. From the HTML forms discovered, the tester can choose a particular form for which to generate test inputs. If authentication is required, the tester can choose to test password protected components and can provide the authentication information. If the tester desires to test for a specific purpose, he or she may supply test inputs such as additional HTML form elements and their values. The tool then applies the bypass rules from Section 2.2, along with the tester's test inputs, to automatically create test values. Test results are reported in a collection of navigable web pages.

AutoBypass only uses HTML and JavaScript that is embedded in the HTML. It cannot test Flash applications, Java applets, and other web applications that do not use HTML as the front end. However, it does not use any server-side source, so can test applications written in J2EE, .Net, PHP, and a variety of other web application development frameworks.

AutoBypass has four main steps; interface parsing, interface analysis, test input selection, and test input generation. These steps are described in the following subsections.

### 3.1 Interface Parsing

AutoBypass first recursively parses the URL of the web application under test to discover all HTML screens. Each HTML screen is analyzed to obtain all link anchors, forms, form elements and script elements. The initial screen<sup>2</sup> of AutoBypass where a tester can submit a URL for the web application under test is shown in Figure 2.

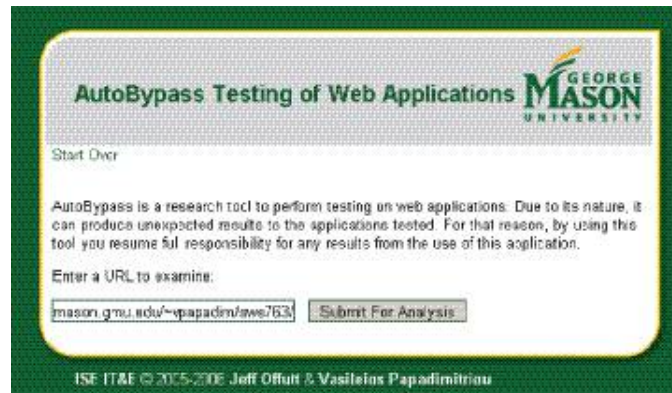


Figure 2. AutoBypass–Main Screen

### 3.2 Interface Analysis

Given a valid URL for the subject application, AutoBypass's frontend forwards the request to a component that presents the forms and the URL found. A sample output of this stage is displayed in Figure 3. The tester is presented with a list of available forms and asked to select the form to generate test inputs for.

At this point, the application finds login fields, which are found based upon the fields names (such as login and user name) and upon password controls. (In HTML, "<INPUT Type = "password" ..." indicates the field is a password and the contents should be "masked," that is, not displayed on-screen.) If the tester chooses to test through password protected screens, he or she can provide authentication information (login name and password). The *HttpUnit.WebConversation* object sends the login information to the subject application, which establishes a session. AutoBypass then forwards the request back to the step where the new output of the tested application is analyzed. The tester is then presented with a new set of screens and forms that can be analyzed. Furthermore, new fields are available, that is, the tester has an option to test a different area of the web application that was not accessible prior to authentication.

After this stage, the selected form is analyzed and all controls are identified. Figure 4 shows the AutoBypass screen based on the subject's form fields. From this screen, the tester may skip all inputs and let AutoBypass select all values based purely

<sup>2</sup>The screenshot has a typo. "resume" should have been "assumes."



Figure 3. AutoBypass–Form and URL selection for testing

on the bypass rules. If the tester wishes to provide some specific values, for example, to ensure specific valid strings are used, they may be provided through this screen. The tester can provide parameter-value pairs that will be used in the tests. It is sometimes helpful for the tester to provide one complete set of valid values to result in a known valid response conforming to the constraints of the HTML interface. The tester may also wish to augment AutoBypass’s rules by providing specific invalid values. These user inputs, although not required, allow the tester to apply domain knowledge that cannot be included automatically.

### 3.3 Test Input Generation

AutoBypass applies the bypass rules described in Section 2.2 to create test inputs, generates invalid requests, and submits them to the subject application.

After tests are generated, the tester is presented with a test result summary as illustrated in Figure 5. The tester can review a table of all the tests generated and review each test case’s result through links to the response screens. The tester can also access logs to view further information, including test generation information, the input domain map, default values, additional controls specified, invalid characters selected for each control, and the buttons used.

Even though tests are generated automatically, the response screens must be reviewed manually. This is because the bypass methodology produces a variety of invalid requests whose outputs are difficult to anticipate and encode as expected values. Since invalid requests may produce different responses and different responses may be of different levels of acceptance, the results cannot be simply compared with predefined expected outputs. Hence, the results must be reviewed by a human who can use domain knowledge to determine whether the response from the application is appropriate.

The subject’s response, or the *test response*, follows a specific structure with four parts. A screen shot of a test response for a test to the Google search engine is displayed in Figure 6. The first part includes the URL of the subject application that was tested (<http://google.com>) and a link to the form under test that produced the response. The second part consists of the description of the test case with the rule information (indicating the rule used for test input generation, *transfer* mode), the test case number (3), the related control (*FORM METHOD*), the original values of the control (*GET*), and the modified values (*POST*). In this example, AutoBypass examines the subject’s response when a transfer mode constraint is changed. The rule used to generate this test case may include additional information such as the form control’s *maxlength* length constraint violations. The third part consists of the modified form control list along with their values as submitted. For the example shown, the transfer mode is changed from GET to POST while other values of the form controls remain the same. Finally, the last part of the result output is the actual HTML response from the subject application. The subject responded to the invalid request with the generic error message “*The server is unable to process your request.*” Some supporting elements, such as





Figure 4. AutoBypass–Test Input

images and external script files, are excluded from the result to avoid display problems that may make results evaluation harder and lead to mistakes in the evaluation.


The modified form allows the tester to verify the result (the automated response) by accessing the subject with a browser. Figure 7 shows the HTML version of the modified form for the test response in Figure 6. The modified form can be accessed with a browser and resubmitted manually.

### 3.4 Selecting Test Inputs

AutoBypass generates many tests for each screen and loads them automatically for every HTML form element. The tester may manually provide additional inputs by selecting a category of invalid characters, including empty strings, commas, directory paths, ampersands, strings starting with forward slash, strings starting with a period, control characters, characters with the high bit set, and XML tag characters. Additional values can be entered in the box in the bottom of Figure 4 in the following format:

*name1:value1; name2:value2a\$value2b; name3;:*

where *name1*, *name2*, and *name3* are names of parameters from the form and *value1*, *value2a*, and *value2b* are values from the parameters. The dollar sign (\$) separates multiple values for the same parameter and semi-colons (;) separate parameter name-value pairs. This is the same syntax used by the HTTP GET request.



## AutoBypass Testing of Web Applications

Start Over

[Test Log](#) | [Input Domain](#) | [Default Values](#) | [Additional Controls](#) | [Invalid Character](#) | [Buttons Used](#) | [\[-\] Hide Logs](#)

Testing: **TEST\_Form**, Found at: <http://mason.gmu.edu/~vpapadim/swe763/testForm.html>, Test Set No: **1145000827740** View All Results in one HTML file

download the worksheet for this set:

Test No	Test Rule	Related Control	Original value	New value	Mutant URL	Server Response URL
1	Length Violation (length limit of 20 characters)	testControlName	testControlValue	agb8oockb8oockzovjeeko	MutantNo_1	ResponseNo_1
2	Transfer Mode	FORM METHOD	GET	POST	MutantNo_2	ResponseNo_2
3	Field Element Violation - control omission	testControlName	testControlValue	n/a	MutantNo_3	ResponseNo_3
4	Target URL Violation - Parameter Omission	p1	n/a	[*1]	MutantNo_4	ResponseNo_4
5	Target URL Violation - Parameter Omission	p2	n/a	[*2]	MutantNo_5	ResponseNo_5

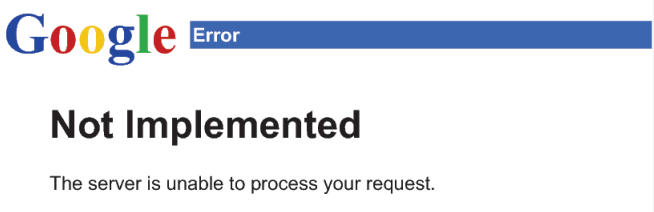
ISE IT&E © 2005-2006 Jeff Offutt & Vasileios Papadimitriou

Figure 5. AutoBypass–Test Results

Testing: <http://google.com>  
 Mutant URL: [The Mutant URL](#)

**RULE : [Transfer Mode]**  
 Test No: [3]  
 Related Field/Attribute: [FORM METHOD]  
 Original Value: [GET]  
 New Value: [POST]

FORM name: [f] action: [<http://www.google.com/search>] method: [POST]  
 name :[btnG] value: [Google Search]  
 name :[hl] value: [en]  
 name :[q] value: [some search]



**Not Implemented**

The server is unable to process your request.

Figure 6. AutoBypass–Example Test Response

```
theForm.getBaseURL(): http://www.google.com/
theForm.getAction() :/search
```

btnG:

hl:

q:

[Submit This Mutant](#)

Figure 7. AutoBypass–Example Modified Form

### 3.5 Interpreting Server Responses—What is a Failure?

An unusual aspect of bypass testing is that the tester is supplying **invalid** values to the application and exceptional outputs are **expected**. Thus, what is usually normal behavior is probably incorrect. Failure behavior can include exception reports that should **not** be shown to the user, storing invalid data on the server, and security access violations. Server responses to invalid data are categorized into three types: valid responses (*V*), faults and failures (*F*), and exposures (*E*). Valid responses (*V*) occur when invalid inputs are adequately processed by the server. A *V* response may be an error message or a request to modify the input data (as in Figure 6, for example). Faults and failures (*F*) include invalid inputs that cause abnormal server behavior. The server’s automatic error handling mechanisms should catch these errors. Exposures (*E*) occur when invalid input is not recognized by the server and abnormal software behavior is exposed to the users. The consequences of exposures can range from annoying (a minor usability problem), to confusing (a worse usability problem that can cause the user to switch to a competitor’s product), to providing malicious users with information that can be used to compromise the system (a security problem).

Expected responses to invalid inputs are seldom defined, so expected output cannot always be included in bypass tests. Instead, the responses must be evaluated by using semantic domain knowledge to decide if the application handled the invalid inputs appropriately. A potential research issue would be to automate all or part of response evaluation.

Of course, some specific error messages are functionally appropriate, although questionable from a usability standpoint. For example, server response messages such as “*The zip code you entered is invalid,*” “*We cannot process your request,*” and “*Your browser sent an invalid request*” are functionally appropriate responses (the server does not process the data). However they are problematic from a usability standpoint; in particular, the second and third are non-specific and do not tell the user what to do to fix the problem [4]. This research is concerned with functional behavior, thus distinguishing usability problems is out of scope.

Some test inputs can produce messages that are more generic. For example, “*Server is unavailable, try later*” and “*Server error*” indicate the server recognized the error, but this kind of generic message is not useful to users. The example in Figure 6 is also in this category.

Web applications sometimes ignore or do not recognize invalid data, and proceed to process the request. This results in a valid response.

Other tests produce responses that ignore the invalid data and do not process the input. An example is when a request is submitted and the server simply returns the original web screen (ignoring the invalid request). We define four types of valid responses:

1. **V1:** The software acknowledges the invalid request and provides an explicit message regarding the violation
2. **V2:** The server produces a generic error message
3. **V3:** The software apparently ignores the invalid request and produces an appropriate response
4. **V4:** The software apparently ignores the request

Although we defined these categories as unambiguously as possible, formal definitions are impossible. Thus the human must use domain knowledge and judgement.

This case study did not have access to the data on the servers (databases, user session data, etc.). In another testing scenario, this information might be available, but this case study could not use it. Thus, during this testing we could not be certain whether the state of the web application was corrupted by an invalid request. It seems unlikely that responses in categories V1, V2 and V4 would corrupt the state—the web application recognized the invalid data, so it probably did not change the state. On the other hand, a tester can be less certain about a V3 response. The web application may or may not have recognized that the data was invalid, so it is possible that part of the application’s state was corrupted. When possible, a wise tester would want to explore the behavior of a web application further on invalid tests that resulted in V3 responses.

## 4 Case Study: Using Bypass Testing on Commercial Web Applications

We applied AutoBypass to a mix of commercial, open source, and proprietary web applications. This evaluation highlights a major advantage of the nature of this tool—it can be applied to arbitrary web applications. The response screens, in HTML, are easily available on the client, and the source from the server is not needed.

This section describes how we applied AutoBypass, introduces the subject applications, discusses the test inputs generated, summarizes the responses from the application servers, and discusses the testing cost in terms of dedicated time and effort to perform the testing.

## 4.1 Empirical Plan

The new concepts in Section 2 presented extensions and refinements of the bypass testing theory. The AutoBypass tool, described in Section 3, demonstrates that bypass testing can be automated in a usable way. The purpose of the case study was to assess whether bypass testing can be effective at finding failures in real web applications.

To ensure practical application and scalability, we chose subject web applications that are in wide use and of varying sizes, including some with very large user bases. Although AutoBypass generates most input values automatically, we added additional values when needed. All testing and additional value selection was done by the second author. The results were evaluated by the second author with a post-decision verification by the first author. The subjects were selected for diversity in terms of implementation language on the backend, size, number of uses, and purpose. Subjects were only used if the experimenters had enough knowledge of the domain to evaluate the results.

As this was a case study on real programs, the empirical plan was very simple. The experimenter provided the initial URL to the “root screen,” and AutoBypass saved all response screens. After testing was completed, the experimenter looked at each response screen and determined whether it represented a failure, exposure, or one of the four types of valid responses. The first author subsequently checked these decisions. All failures found were naturally occurring and bypass testing was not compared with other test techniques. Neither do we have any way of knowing how many faults were not found by AutoBypass. We assume the web applications had been subjected to the companies’ standard test practices, but those are unknown and it seems likely the amount and type of testing would be quite varied.

Of course, this kind of case study is open to many threats to validity. As with almost all software engineering study, the subjects may not be representative. We tried to limit the effect by making the subject pool as diverse as possible. Bypass testing has many test generation rules, as defined in this paper, however, any given tool will by necessity make many decisions in applying these rules. It is possible that different interpretations in a different tool would yield different results. Another researcher may make different decisions on some of the response screens. Having two people look at the screens limited this issue. The fact that our goal was to determine whether bypass testing can be effective in practical situations means that the case study does not have to find a precise number of failures.

Further details on some of these effects are given in the following subsections.

## 4.2 Evaluating the Technique

Although it would have been possible to apply bypass testing by modifying **all** fields with **all** possible values with **all** the bypass rules, this would result in a very large number of tests, many of which we believe would be redundant. Instead, AutoBypass first found default values for all input fields that were valid according to all input constraints. Then a version of *base choice* testing [1] was applied. For each invalid value generated with bypass testing rules from Section 2.2, that value was combined with the default values for all the other input fields.

As stated in the introduction, the goal of this project was to validate bypass testing’s ability to be successful in practical situations. Thus, the AutoBypass tool was used in a case study of commercial software, so a formal comparison is not possible (except by default with the companies’ standard, but unknown, test practices). It is well known that modern web applications should be designed and built to be reliable, usable, and secure [22], thus we assume that applications already deployed on the web have been tested. The testing methods of the subject developers are not known. However, the fact that the subject applications are being used suggests that they have been subjected to some reasonable amount of testing. The case study used subjects that have been used by many users. These were chosen specifically to support the assumption that the applications studied have already been tested.

## 4.3 Subject Selection

The web application subjects were selected to provide a range of size, complexity, size of user base and backend technology. The complexity was loosely estimated by looking at the number of input screens and inputs in the application. Backend technologies included J2EE, PHP, ASP, and .Net. Web interfaces that use scripting to extensively modify the DOM were not tested. Neither were interfaces based on Macromedia Flash or Java applets.

We used open source applications as well as proprietary commercial web applications. All of the commercial applications have a large user base, yet some are significantly larger than others. We did not inform the companies of our study and did not have access to source or server-side data. Not having access to server-side data means that some tests may have caused failures that were not detected.

We applied AutoBypass to 16 subject applications, which collectively had 30 components. More details about these applications as well as detailed data from our study can be found in Papadimitriou’s MS thesis [27].

## 4.4 Detailed Results

This section presents results of bypass testing on four of the 16 subjects. Bypass testing found failures in 14 of the 16 applications studied. The results of all 16 subjects are summarized in Table 8 in subsection 4.5. Because giving details on all 16 subjects would take a lot of space and is quit tedious, this subsection gives details on four of the more interesting subjects. For each, we describe the application, highlight interesting aspects, summarize the results for that application in tabular forms, and discuss interesting aspects of the results.

### 4.4.1 Bank of America

Bankofamerica.com is an online banking portal. Even though AutoBypass provides the ability to test password protected screens and potentially can be used to assess the correctness of online banking features, only public components of the applications providing information were tested. We chose not to provide login information because we experienced severe corruption of data while applying AutoBypass to some small applications, and we were not willing to risk compromising our bank accounts. Such tests would require careful preparation and cooperation with the institutions to avoid unexpected results (such as financial losses).

The Automated Teller Machine (ATM) and Branch Locator tools are part of the main screen in the subject’s portal. Behind the simple appearance of the interface used to access this feature, many hidden input controls are used to transmit data to the application. Analysis of the interface’s HTML source revealed that the form action URL points to a separate domain. Specifically, bofa.via.infonow.net hosts the application that returns the results of this action. By submitting a request, infonow.net acts as an extension of the Bank of America’s portal, presenting results of the search using the bank’s identity. This is not apparent to a user unless she or he carefully reads the URL.

The application correctly handled 66% of the test cases, returning valid responses and explicitly acknowledging the invalid input in 4% of the tests. Invalid parameters were apparently ignored 62% of the time and produced appropriate responses in which default locations were presented. Specifically, locations in California (headquarters of the bank) were shown. The application server handled exceptions generated by invalid inputs for 14% of the test cases. For instance, the error “[ServletException in:/jsp/layouts/ApplicationTemplate.jsp] Error - Tag Insert: No value found for attribute metaText.” was returned from one test. Finally, the remaining 20% of the tests exposed errors. Typically, the invalid requests caused the application to produce responses referring to the default locations (CA) while the originating request included a zip code for Virginia. In addition, the links provided for driving directions are incorrect. An example is shown in Figure 8. The value violation rules produced most of the invalid responses (99.9%), with one produced by a field element violation rule. Results for this module are presented in Table 1. A legend is shown in this first table, but is not repeated in subsequent tables to save space.

The second module tested was the site search tool in the main portal screen. All invalid requests produced appropriate responses. The application ignored most of the invalid requests (68%) and created an appropriate response. The application ignored the request completely for 31% of the tests and responded with the original interface. Finally, two tests (1%) produced generic messages (for example, “The Bank of America page you are trying to reach: http://www.bankofamerica.com/stateerror is not available”), which were considered correct (V2). Results for this module are presented in Table 2.

### 4.4.2 Barnes and Noble

Barnesandnoble.com sells and buys books, stationery, toys, music, and video. Only public sections of the web application (*Shopping Cart* and *Book Search*) were tested; no authentication was required to access these screens.

As each product detail is displayed on the application’s interface, the *add to shopping cart* form allows a user to add the current item to the collection that she or he wants to purchase. In the video collection, AutoBypass generated 176 tests, 98.9% of which resulted in valid responses and 1.1% resulted in exposure of errors, as presented in Table 3.

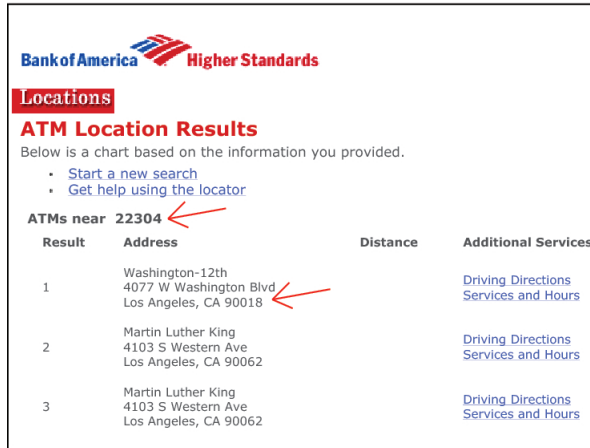


Figure 8. Exposure at Bankofamerica.com, ATM & Branch Locator

Table 1. Results for Bankofamerica.com, ATM & Branch Locator by infonow.net

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Validation	1	1	0	0	0	0	0	1
Value Violation & Invalid Characters	66	3	0	63	0	17	25	108
Transfer Mode	1	0	0	1	0	0	0	1
Field Element Violation	17	1	0	16	0	1	0	18
<b>Total</b>	<b>85</b>	<b>5</b>	<b>0</b>	<b>80</b>	<b>0</b>	<b>18</b>	<b>25</b>	<b>128</b>
<b>%</b>	<b>66.4%</b>	<b>3.9%</b>	<b>0.0%</b>	<b>62.5%</b>	<b>0.0%</b>	<b>14.1%</b>	<b>19.5%</b>	

Legend	
<b>V</b>	<b>Valid response (includes V1, V2, V3, V4)</b>
<b>V1</b>	Valid response with server producing explicit error message
<b>V2</b>	Valid response with server producing generic error message
<b>V3</b>	Valid response with server ignoring invalid request
<b>V4</b>	Valid response with server ignoring and not processing request
<b>F</b>	<b>Faults and Failures</b>
<b>E</b>	<b>Exposure</b>

Table 2. Results for Bankofamerica.com, Web Site Search

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Validation	1	0	0	0	1	0	0	1
Transfer Mode	1	0	0	0	1	0	0	1
Field Element Violation	4	0	0	0	4	0	0	4
Target URL Violation	204	0	2	142	60	0	0	204
<b>Total</b>	<b>210</b>	<b>0</b>	<b>2</b>	<b>142</b>	<b>66</b>	<b>0</b>	<b>0</b>	<b>210</b>
<b>%</b>	<b>100.0%</b>	<b>0.0%</b>	<b>1.0%</b>	<b>67.6%</b>	<b>31.4%</b>	<b>0.0%</b>	<b>0.0%</b>	

(See legend in Table 1.)

**Table 3. Results for Barnesandnoble.com, Shopping Cart**

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation & Invalid Characters	123	0	1	122	0	0	2	125
Transfer Mode	1	0	0	1	0	0	0	1
Field Element Violation	10	0	1	9	0	0	0	10
Target URL Violation	40	0	5	35	0	0	0	40
<b>Total</b>	<b>174</b>	<b>0</b>	<b>7</b>	<b>167</b>	<b>0</b>	<b>0</b>	<b>2</b>	<b>176</b>
<b>%</b>	<b>98.9%</b>	<b>0.0%</b>	<b>4.0%</b>	<b>94.9%</b>	<b>0.0%</b>	<b>0.0%</b>	<b>1.1%</b>	

(See legend in Table 1.)

**Table 4. Results for Barnesandnoble.com, Book Search**

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation & Invalid Characters	132	13	0	119	0	3	0	135
Transfer Mode	1	1	0	0	0	0	0	1
Field Element Violation	6	0	0	6	0	0	0	6
<b>Total</b>	<b>139</b>	<b>14</b>	<b>0</b>	<b>125</b>	<b>0</b>	<b>3</b>	<b>0</b>	<b>142</b>
<b>%</b>	<b>97.9%</b>	<b>9.9%</b>	<b>0.0%</b>	<b>88.0%</b>	<b>0.0%</b>	<b>2.1%</b>	<b>0.0%</b>	

(See legend in Table 1.)

Valid responses mainly consisted of output suggesting that the application refused the invalid parameters; this output was considered to be valid. Yet some of the valid responses resulted in generic error messages that are confusing to the users. Two specific messages were:

- Message A:

*“We are sorry... Our system is temporarily unavailable due to routine maintenance. We apologize for any inconvenience during the outage and expect our systems to result shortly. Thank you for your patience.”*

This is especially interesting because it is so obviously incorrect! (Not to mention grammatically wrong.)

- Message B:

*“Sorry, we cannot process your request at this time. We may be experiencing technical difficulties, or you may need to adjust your browser settings. If you continue to have problems, please click [here](#) to send a message to customer service.”*

By examining the output captured by AutoBypass, test inputs that generated message B when they were tested using a regular browser resulted in an empty text file with the phrase *“The parameter is incorrect.”* Obviously, this response is not very helpful as it does not specify which parameter or how the error might be solved. The different responses may be due to browser compatibility issues; for example, desktop browsers and hand-held device browsers render HTML quite differently. AutoBypass uses HttpUnit to bypass the browser, thus it might not have been recognized as one of the common web browsers, causing a simplistic error message to be generated.

The small percentage of test cases that cause exposures were due to invalid inputs that violate the cart display function. Even though an error message specified that the item could not be added to the shopping cart, the screen also displayed an empty cart with the title *“You just added this item to your Cart.”* In addition, the cart display used JavaScript, which (due to the lack of references in the cart) printed undefined values (for example, NaN). Furthermore, all other sections of the screens (such as *Wish list*, *Suggested items*, and *Special offers* were empty. This response confused the user and could decrease the usability of the interface.

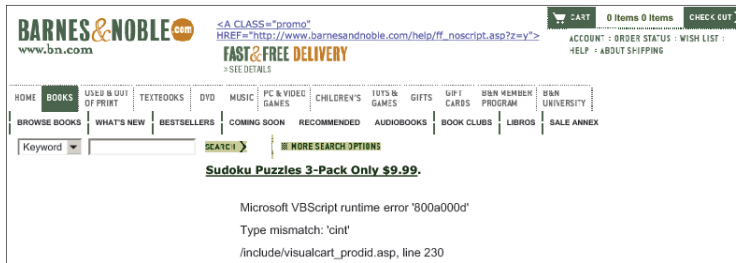


Figure 9. Fault/Failure at Barnesandnoble.com, Book Search component

Table 5. Results for Joomla CMS, Poll Administration module

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation & Invalid Characters	218	0	0	5	213	0	104	322
Transfer Mode	2	0	0	0	2	0	0	2
Field Element Violation	82	0	0	1	81	0	4	86
Target URL Violation	16	0	0	0	16	0	0	16
<b>Total</b>	<b>318</b>	<b>0</b>	<b>0</b>	<b>6</b>	<b>312</b>	<b>0</b>	<b>108</b>	<b>426</b>
<b>%</b>	<b>74.7%</b>	<b>0.0%</b>	<b>0.0%</b>	<b>1.4%</b>	<b>73.2%</b>	<b>0.0%</b>	<b>25.4%</b>	

(See legend in Table 1.)

The second component tested was the *Book Search* tool. AutoBypass generated 142 tests and got mostly valid responses (97.89%), while the remaining tests (2.11%) produced faults and failures. Most of the invalid requests (88%) received valid responses that appeared to ignore the invalid inputs and process the request with no apparent error. A few tests (9.9%) produced valid responses with explicit messages regarding the invalid requests. Finally, three failures were detected when a hidden input control (“*TXT*”) was violated with a string starting with a percent sign (%). An example of such a failure is shown in Figure 9, which shows a VBScript error message. Results for this module are presented in Table 4.

#### 4.4.3 Joomla

Joomla is a Content Management System (CMS) used by thousands of users to create and administer online portals [15]. A demo server is available for trial, which was used to perform the tests on password protected screens. All Joomla modules are accessed through a common interface. By passing parameters in the GET requests, the application renders different component management interfaces through the same form and uses JavaScript to hide parts of the interface, depending on the module administered. Two modules were tested, *Poll Administration* and *Online User Information*.

AutoBypass generated 426 test cases for *Poll Administration*. Most of the tests resulted in valid responses (74.65%) while the rest produced exposure of the application. The errors found in this module were primarily caused by parameters used to control the screen’s navigation. The application allows arbitrary inputs to be passed as part of the navigation menus. The remaining invalid responses created a new poll, despite the fact that the submission was intended to update an existing poll. In particular, by modifying the parameter *id*, which apparently corresponds to the *id* of the poll, the application created a new poll without checking if this action was legitimately requested or if the *id* value was within the appropriate range.

With respect to the rules used to generate tests, 33% of value violations caused exposures, and only 9.3% of the field element violations caused exposures. Transfer mode and target URL violations produced only valid responses. Finally, in 99% of the responses that were classified as valid, it appeared that the server ignored the invalid input and did not process the request (V4) since the original screen was reprinted. Results for this module are presented in Table 5.

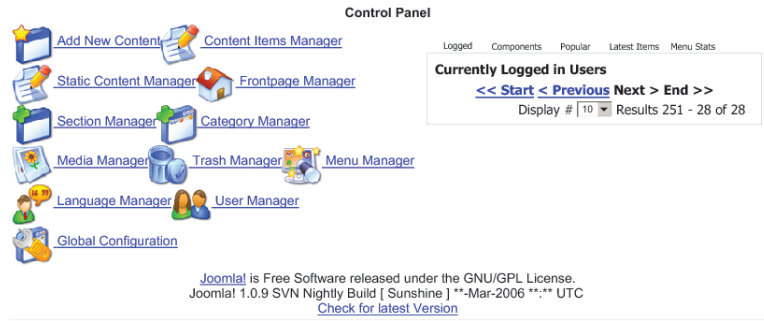
The online user information is displayed on the main screen of the CMS administrator and provides access to all users who are currently logged on. An administrator may force logout a user or change the user’s account settings. The tests generated resulted in 68.6% valid responses and 31.4% exposure of application errors as shown in Table 6. By modifying hidden parameters, the test requests caused an invalid argument to be passed in a control loop, resulting in errors on the



**Table 6. Results for Joomla CMS, Online User Information module**

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation & Invalid Characters	52	0	0	52	0	0	27	79
Transfer Mode	1	0	0	1	0	0	0	1
Field Element Violation	6	0	0	6	0	0	0	6
<b>Total</b>	<b>59</b>	<b>0</b>	<b>0</b>	<b>59</b>	<b>0</b>	<b>0</b>	<b>27</b>	<b>86</b>
<b>%</b>	<b>68.6%</b>	<b>0.0%</b>	<b>0.0%</b>	<b>68.6%</b>	<b>0.0%</b>	<b>0.0%</b>	<b>31.4%</b>	

(See legend in Table 1.)



**Figure 10. Exposure at Joomla CMS, Online User Information Module**

display. In this particular version of the demo, script errors were output to the response and thus detectable. If the error output was disabled, this exposure could not be identified. Some other invalid requests exposed the wrong number of current users. For instance, Figure 10 shows the application output “*display: results 251-28 of 28*” while no users are listed.

**4.4.4 New York Times**

Nytimes.com is an online edition of the New York Times. The interface tested was in the business section (marketwatch.nytimes.com) which was built using Active Server Pages (ASP) technology. This component provides an overview of the stock markets and allows users to search for specific trade symbols. Table 7 presents the results for this subject. The invalid requests resulted in only 55.4% valid responses, 1.42% faults and failures, and 43.18% exposures.

Most valid responses appeared to ignore the invalid request parameters and produce an appropriate response. In a few cases (1.1%), a generic message was received. Upon invalidating the parameter guid, the application responded with a generic message “*story not available.*” Faults and failures were caused by 1.4% of the test cases; the application did not respond at all. The use of advertising may have caused these failures.

Many of the invalid requests (44.6%) exposed faults. All test cases that resulted in exposure were generated using the value violation rules. By altering values in hidden input controls, the application referenced images that do not exist in place of the graphs for each market (that is, an empty image was rendered). A second type of error was the violation of specific parameters, namely screen and exchange, resulting in the omission of the last part of the output. Part of the output was replaced by the ASP error report.

**4.5 Results Summary**

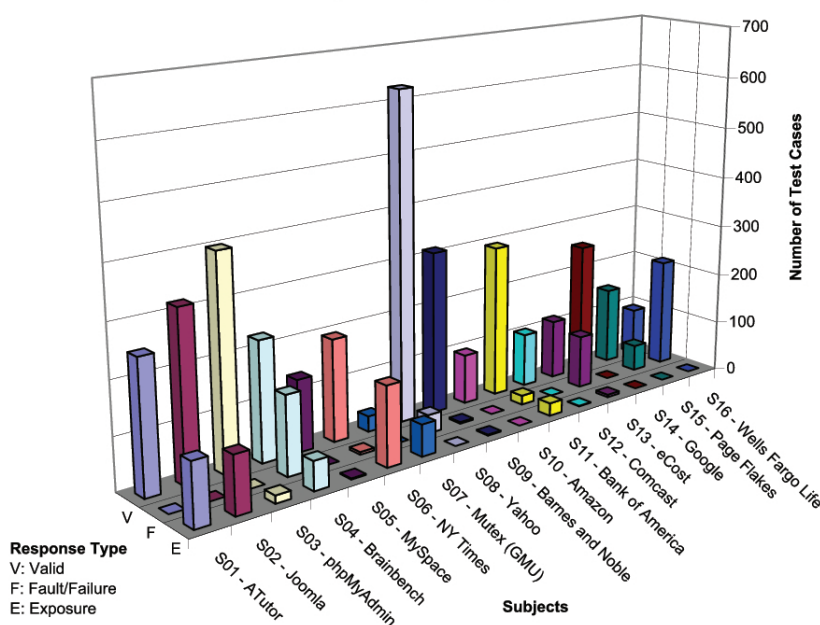
Table 8 summarizes the results of bypass testing on all 16 web applications. More than one component of some applications were studied, so they have multiple rows in the table (for example, amazon and bankofamerica). In total, AutoBypass generated 4812 tests and 23.9% were inappropriately handled (F+E), 12.0% generated faults and failures (F), and 11.9% resulted in exposure (E). AutoBypass found failures in all but two subjects, amazon.com and google.com, and the highest percentage of failing tests was 69.7%. The types of invalid responses from the subjects were recorded and summarized in Table 9.

**Table 7. Results for NYtimes.com, Market Watch**

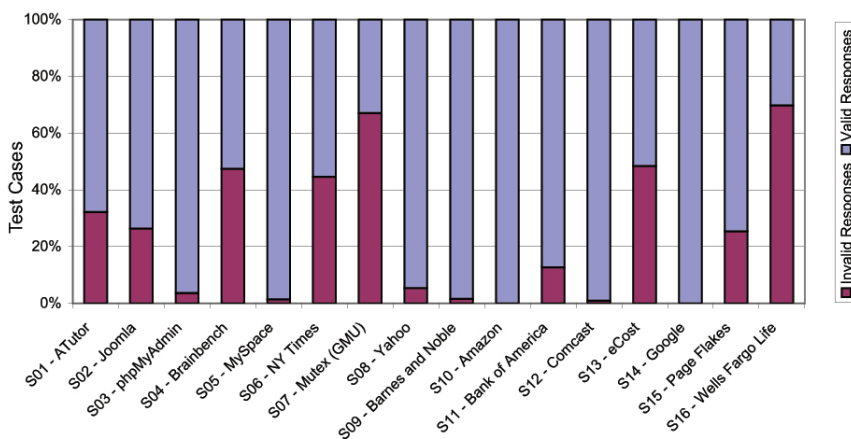
Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation & Invalid Characters	40	0	0	39	1	2	147	189
Transfer Mode	1	0	0	1	0	0	0	1
Field Element Violation	9	0	0	9	0	0	5	14
Target URL Violation	145	0	4	0	141	3	0	148
<b>Total</b>	<b>195</b>	<b>0</b>	<b>4</b>	<b>49</b>	<b>142</b>	<b>5</b>	<b>152</b>	<b>352</b>
<b>%</b>	<b>55.4%</b>	<b>0.0%</b>	<b>1.1%</b>	<b>13.9%</b>	<b>40.3%</b>	<b>1.4%</b>	<b>43.2%</b>	

(See legend in Table 1.)

Graph A: Response Types



Graph B: Valid vs. Invalid Responses



**Figure 11. Result Summary Graphs**

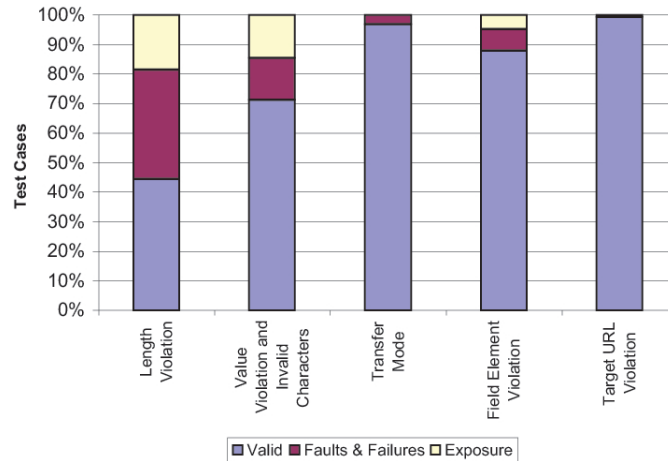
**Table 8. Results Summary**

Application	V	V1	V2	V3	V4	F	E	Total
amazon	63	0	0	0	63	0	0	63
	33	0	0	0	33	0	0	33
atutor.ca	249	3	0	216	30	0	118	367
bankofamerica	85	5	0	80	0	18	25	128
	29	0	2	142	66	0	0	210
barnes-noble	174	0	7	167	0	0	2	176
	139	14	0	125	0	3	0	142
brainbench	0	0	0	0	0	150	1	151
	227	79	0	148	0	0	54	281
comcast	104	16	85	0	3	0	1	105
ecost	17	0	0	15	2	41	5	63
	98	0	0	97	1	62	0	160
google	148	0	0	148	2	0	0	148
	107	0	0	107	0	0	0	107
joomla	318	0	0	6	312	0	108	426
	59	0	0	59	0	0	27	86
mutex	30	30	0	0	0	0	61	91
myspace	77	0	0	77	0	0	0	77
	60	0	27	33	0	0	2	62
nytimes	195	0	4	49	142	5	152	352
pageflakes	150	88	0	0	62	51	0	201
phpmyadmin	54	10	0	0	44	0	0	54
	87	1	0	0	86	0	15	102
	133	2	0	0	131	0	0	133
	126	0	0	125	1	0	0	126
wellsfargolife	92	0	19	71	2	212	0	304
yahoo	98	0	0	0	98	0	0	98
	470	0	0	0	470	0	0	470
	1	1	0	0	0	36	0	37
	59	0	0	59	0	0	0	59
<b>Total</b>	<b>3663</b>	249	144	1722	1548	<b>578</b>	<b>571</b>	<b>4812</b>
<b>%</b>	<b>76.1%</b>	5.2%	3.0%	35.8%	32.2%	<b>12.0%</b>	<b>11.9%</b>	

Legend	
<b>V</b>	<b>Valid response (includes V1, V2, V3, V4)</b>
<b>V1</b>	Valid response with server producing explicit error message
<b>V2</b>	Valid response with server producing generic error message
<b>V3</b>	Valid response with server ignoring invalid request
<b>V4</b>	Valid response with server ignoring and not processing request
<b>F</b>	<b>Faults and Failures</b>
<b>E</b>	<b>Exposure</b>

**Table 9. Types of Invalid Responses**

<b>Application</b>	<b>Type of Invalid Responses</b>
<b>Amazon</b>	NONE
<b>ATutor</b>	Reference of non existing files within main frame of interface Corrupted output (sound files) Incorrect parameter reference for output
<b>Bank of America</b>	Exceptions caught by web server (ATM search) Incorrect results (ATM search)
<b>Barnes &amp; Noble</b>	Incorrect output VB runtime errors
<b>Brainbench</b>	Exceptions caught by web server (request info) Requests for information were submitted with bogus data Invalid new user requests were allowed Violation of specific parameters produced irrelevant errors
<b>Comcast</b>	Incorrect result
<b>Ecost</b>	VB runtime errors
<b>Google</b>	NONE
<b>Joomla</b>	Invalid characters appeared in navigation bar New polls created upon invalid parameters on existing pools Current user display was incorrect Invalid character caused errors caught by runtime engine
<b>Mutex</b>	Invalid requests were redirected to nonexistent screens
<b>MySpace</b>	Screen number on results was incorrect
<b>NYtimes</b>	Application referenced images that did not exist Parts of the output were omitted VB runtime errors
<b>Pageflakes</b>	.NET runtime errors
<b>phpMyAdmin</b>	Invalid characters caused errors caught by runtime engine Invalid input (character set) reached database queries
<b>Wellsfargolife</b>	VB runtime errors
<b>Yahoo</b>	Invalid requests resulted in empty response (desktop search reminder)



**Figure 12. Responses by Violation Rule**

Figure 11(A) summarizes the responses for each subject in a bar chart, grouped by valid, failure and exposure. The number of tests heavily depend on the number of form input elements, the constraints used, and the inputs supplied by the tester. Figure 11(B) abstracts this to show the percentage of tests that resulted in invalid (faults, failures and exposures) and valid responses. The percent of invalid responses ranged from 0 to almost 70%.

We were not able to assess the criticality of these faults. That would require more than knowledge of the domain, but also detailed knowledge of how the software was designed and built, thus only an engineer on the development project could make reasonable criticality assessments. Instead, we break the responses up by type of violation rule from the list in Section 2.2, as shown in Figure 12. The length violation rule caused more than half of the invalid responses (55.5%). However, that rule only generated 0.6% of the test cases. The value and the field element violations produced 28.8% and 12.1% invalid responses. The least effective rules in this experiment were the transfer mode and target URL rules.

#### 4.6 Testing Cost

An important aspect of bypass testing is the amount of time and effort needed. Software quality assurance and testing can account for more than 50% of the development cost [12, 13], which translates into hundreds or even millions of dollars. The amount of effort in our study, as shown in Table 10, was not high, especially given the extensive number of problems found. The highest cost was with Joomla, which was still tested by one person in just over a day. Most applications only took a couple of hours to test. The tester times given in Table 10 are all one person (the second author). Table 10 shows the number of tests in the second column. The third and fourth columns show the time the tool spent, in minutes, separating test generation from test submission and execution. The fifth and sixth columns show the time the tester spent, again in minutes, separating creating input parameters from reviewing responses. Most of the tester’s time, approximately 87%, was spent reviewing the responses. All testing techniques need the testers to spend some time reviewing results. An advantage of using a criteria-based technique is that criteria tend to generate fewer tests for the same benefit, leading to less effort needed by the human. The last two columns show the total time, both in minutes and hours.

These data provide evidence that bypass testing scales to real application. This testing was done by one person and only required a few hours per program (the maximum being 8.3 hours). Although we were not able to compare our testing time with the companies’ testing time, we were able to talk to one test manager (yahoo). One of us spent 6.4 hours testing yahoo, yet the equivalent testing effort at the company took **two** full-time people **three weeks** to test the same screens evaluated in this study—and was not as effective. Thus we consider this a very strong argument for the return on investment of criteria-based automated test data generation.

#### 4.7 Security Evaluation

The primary goal of bypass testing is to evaluate web applications’ ability to respond to invalid inputs, thus it should be considered primarily to be a functional testing approach. However, invalid inputs include penetration attacks, so bypass

**Table 10. Amount of Effort**

Application	Tool Processing Time *			Tester Time *		Total Time	
	Number of Tests	Test Generation	Test Submission **	Creating Input Parameters	Reviewing Responses	Minutes	Hours
amazon.com	63	1	3	5	32	41	0.7
	33	1	1	5	17	24	0.4
atutor.ca	367	1	41	5	184	231	3.8
bankofamerica.com	128	1	8	5	64	78	1.3
	210	1	5	10	105	121	2.0
barnesandnoble.com	176	1	2	5	88	96	1.6
	142	1	2	5	71	79	1.3
brainbench.com	151	1	2	5	76	84	1.4
	281	1	8	10	281	300	5.0
comcast.com	105	1	2	5	53	61	1.0
ecost.com	63	1	1	5	32	39	0.6
	160	1	6	10	80	97	1.6
google.com	148	1	1	10	74	86	1.4
	107	1	2	10	54	67	1.1
demo.joomla.org	426	1	5	10	426	442	7.4
	86	1	2	10	43	56	0.9
mutex.gmu.edu	91	1	1	2	46	50	0.8
myspace.com	77	1	3	5	77	86	1.4
	62	1	1	5	31	38	0.6
nytimes.com	352	1	5	10	176	192	3.2
pageflakes.com	201	1	5	10	101	117	1.9
phpMyAdmin	54	1	2	10	27	40	0.7
	102	1	3	15	51	70	1.2
	133	1	5	15	67	88	1.5
	126	1	7	5	63	76	1.3
wellsfargolife.com	304	1	3	10	152	166	2.8
yahoo.com	98	1	1	10	49	61	1.0
	370	1	15	15	235	266	4.4
	37	1	1	1	19	22	0.4
	59	1	2	2	30	35	0.6
<b>Total</b>	<b>4812</b>	<b>30</b>	<b>145</b>	<b>230</b>	<b>2798</b>	<b>3203</b>	<b>53.4</b>
<b>Average</b>		<b>1</b>	<b>5</b>	<b>8</b>	<b>96</b>	<b>110</b>	<b>1.8</b>

\* Time in minutes

\*\* Time it took the tool to submit the test cases over HTTP and capture responses

testing also contributes to evaluating the resistance of software to penetration attacks [20]. This study found several cases where invalid inputs were passed to the back-end software without validation. After careful study of this behavior, malicious users could exploit invalid input vulnerabilities and therefore compromise security.

A well known attack scenario is to pass server side scripting commands (for example, SQL) as part of invalid inputs. AutoBypass does not explicitly include such commands, so should not be considered a full service security evaluation tool.

One example of this behavior was the invalid inputs on the Joomla application, which passed through to the output for the interface's navigation. In phpMyAdmin, invalid input reached the database queries with no validation. Finally, session related values, such as the `VIEWSTATE` parameter of .NET applications, which is stored in hidden fields, are often passed without validation. Most of the applications we tested found values out of range with the exception handling mechanisms, but theoretically one can access the session of another user by manipulating the value of this parameter.

#### 4.8 Limitations and Bias

A strength of this study was that it used real software applications as subjects, some of which are widely used. This was possible because source was not needed. On the other hand, this brought in limitations. First, we were not able to observe back-end data stores, such as corrupted data stores on the server or incorrect values in-memory. Thus the faults and failure counts were conservative, and more faults may have been revealed. This is an issue of observability, widely recognized as a problem with web applications. In a different test scenario, bypass testing could be used by the developing organization to evaluate web applications before releasing. The large numbers of failures found in this study, however, indicates that bypass testing can be useful for end-users and customers trying to decide whether a web application can be trusted.

A second limitation is that a human had to judge whether responses were valid, failures, or exposures. This required domain knowledge of the programs. On the other hand, the amount of human effort needed (given in Section 4.6) indicates that this manual effort is reasonable and worthwhile.

Like any web application technology that does not use the source, AutoBypass cannot find all pages (screens) in the web application, particularly when some screens or particular input fields only appear when some very specific inputs are provided. Although the problem of finding all screens in a web application is known to be undecidable [11], AutoBypass allows the tester to partially overcome this limitation by using domain knowledge to provide additional inputs.

A fourth limitation of the tool is with web applications that use Ajax to perform input validation. AutoBypass is not currently able to bypass this validation, although it is theoretically feasible. Finally, this study only considered two transfer modes, GET and PUT.

### 5 Related Work

Many existing web application testing tools address web usability, performance, and portability issues [14]. These problems are out of the scope of the research in this paper, so are not discussed further.

Several papers look at testing the **static nature** of web sites based on graph representations [17] and the UML [29]. Kung *et al.* [16, 17] introduced a graph representation of web sites and analyzed the structure of a web site by traversing static links. Ricca and Tonella [29] proposed an analysis model based on the Unified Modeling Language (UML) that facilitated test case generation from static web pages. The model was then used to facilitate test input generation. Later, Ricca and Tonella [30] applied their UML model of the web applications [29] to support integration testing of web applications with a consideration of implicit and explicit states. These techniques need access to the server-side components, which bypass testing does not, and focus on static issues, whereas bypass testing evaluates dynamic behavior. None have been empirically evaluated on more than three or four small applications.

A few papers look at testing web applications from a **black-box** point of view, as does bypass testing. Eaton and Memon tested web applications to see how well they conformed to different browsers [6]; again, a different problem than addressed in this research. User session data [8, 9, 31, 33, 37] are derived from inputs from previous users, as captured in the server logs. Because the technique relies on previous users, user session data does not explore the input domain systematically and may not cover it well, as bypass testing does. However, user session data could be used as seed data with bypass testing, coupled with algorithms that modify the seed data to satisfy the bypass rules. Sampath *et al.* [32] showed that tests from usage data that cover different input names and values in web applications uncover more faults than test cases that cover just the base screens in the application. As opposed to this paper, which studied 16 widely used commercial applications, they used four relatively small subjects; a conference website (Masplas), an open-source, e-commerce bookstore, a course project manager (CPM), and a customized digital library. The same four subjects were studied by Sprenkle *et al.* [34].

Oracles and user-session-based testing frameworks have been proposed for web applications [35, 36]. Wang and Tang [40] proposed a methodology to model the flow graph of a web application based on the log files of the application. This technique also depends on how recent users have accessed the application. This requires more information than bypass testing does, and it would be useful to compare their performance empirically. Their model was used to estimate reliability, and the paper did not include an evaluation.

A closely related idea to bypass testing was presented by Nguyen-Tuong *et al.* [21], where input constraints violation was based on PHP and SQL commands as well as scripting. Nguyen-Tuong’s research was entirely focused on security, not testing functional aspects of the software. Tappenden *et al.* [38] introduced a tool using some of the concepts of bypass testing. Tappenden *et al.* extended the concepts of bypass testing to cookies and files being uploaded. The proposed architecture consists of several layers: the presentation layer, process/control layer, business entity layer, and data services layer. Tappenden *et al.* also considered whether the file’s name was too long, or the file size was too big for the system or web server; as well as whether the file was compatible with Base64 encoding. However, no rules or guidelines for testing are given explicitly (as our research does) and it is not clear how this extended bypass testing framework might cope with other constraints such as field element constraints, target URL constraints, or inter-value constraints. Neither of these papers had extensive empirical validation.

Bypass testing has also been **extended** to create “shields.” Mouelhi *et al.* [20] present a tool that analyzes the front end of web applications to discover the input validation rules (exactly like bypass testing does). But instead of creating tests, they create additional software on the server that duplicates the input validation. They narrowed the focus to be entirely on security issues, and evaluated on four small custom-built applications. Miller *et al.* [19] discuss a tool called InputValidator, which uses bypass testing to evaluate security. InputValidator poses test requirements as regular expressions, then solves them to create tests. They evaluated InputValidator on one small open-source system (JspCart).

A few papers use source on the server to test web applications (**white-box testing**). FSM modeling [2, 3] models the implementation of web applications as finite state machines, then generates tests that cover the FSM. Lucca [18] proposed an approach to integrate existing testing techniques with state-based testing to discover possible inconsistencies caused by interactions with web browser buttons, such as reload, back and forward. Offutt and Wu [41, 24] dealt with both static and dynamic aspects of web applications by considering atomic sections of web pages constructed from static structure and dynamic contents. These techniques require access to the source and rely on very detailed analysis, making them less widely applicable than bypass testing. None have been evaluated empirically on more than one or two applications.

Most testing approaches either focuses the navigation by exploring static links or using the source to model web applications, and then tests functionality. With the exception of user-session-based testing, the other approaches rely on testers’ knowledge and experience to design test inputs. User-session-based testing overcomes the problem of inadequate test data but relies entirely on user sessions for test cases, thus parts of the input domain that users have not accessed may remain untested. For example, if users did not use certain inputs, then the corresponding input validation code will not be covered by test cases created from user session logs.

Thus, none of these approaches measure *how web software responds to different inputs* or *whether all input validation code has been exercised and tested systematically*. Bypass testing does both of these things, with the advantage of not needing access to the source. Moreover, after extensive search, we were not able to find any web application testing empirical study that used as many as five, let alone 16, commercially deployed web applications.

## 6 Conclusions and Future Work

This paper has three contributions. First, the paper greatly extends and refines the theory of bypass testing. New rules are defined, and significant ideas for automating bypass testing are presented. Second, a fully implemented tool, AutoBypass, is presented. AutoBypass implements bypass testing and runs as a web application itself. AutoBypass does not need access to the web application source and requires only two kinds of inputs, a URL to the web application under test, and any required login information to the web application. Third, results from a comprehensive case study on real, widely used, applications are presented. This is the most comprehensive case study we are aware of on testing web applications.

The rules from the previous bypass testing paper [25] were modified and expanded to more closely match the way HTML is used in web applications. These improvements make bypass testing more realistic and more effective.

We learned several things while building and using AutoBypass. Using HttpUnit allowed tests to automatically be submitted to the server from a standalone Java program rather than through a browser. A great deal of attention was given to the user interface. AutoBypass returns a lot of information from testing, which is carefully arranged in HTML pages that are connected with usable navigation links.



The experimental results indicate that the Transfer Mode and Target Rule Violation rules are not effective. We recommend against their use in future tools.

AutoBypass found hundreds of errors in 14 of the 16 web applications in our study. Although assessing security vulnerabilities is not the most important goal of bypass testing, several tests were able to pass invalid data to the back-end without authentication. Malicious users could exploit such vulnerabilities to compromise applications.

The number of problems found varied by application. Some of the well known web applications with millions of users (google.com and amazon.com) had the fewest problems. Some less widely used applications had lots of problems. This variation could indicate how much internal testing the different organizations used. Naturally, the applications vary greatly in their goals, criticality, safety, and overall need for quality to be profitable. These differences would certainly affect their development and testing.

The rapid and continuing advance in technologies for web applications provide ongoing challenges for future work in testing. On a purely engineering basis, bypass testing needs to be more effective at generating values that contradict semantic constraints implemented in JavaScript checking. Even without this, however, the number of software problems found in this case study is astounding.

AutoBypass currently tests each screen independently, one at a time. It might be desirable to test sequences of screens as paths through web applications. It would also greatly help bypass testing if the result analysis (whether a response is valid, a failure, or has exposure) could be automated. We currently have no solution to this problem.

Web application developers have recently been implementing more behavior on the client. The most common emerging technology is Ajax (“Asynchronous JavaScript and XML”), which allows web pages to access the server asynchronously, without the user explicitly sending a request. This allows developers to implement a richer user interface experience, however, the problem of identifying all possible screens and variations of those screens becomes more difficult. How to test the use of Ajax in web applications is a significant problem for researchers and developers.

We did not explicitly use cookies as part of our test cases, although they were created naturally when our tests included multiple requests. One type of security exploitation involves manipulating cookies to fool the server; for example, causing the server to think the user has authenticated, or causing the server to think a different user is accessing the system. This, and other type of security exploits, could be included into future bypass tests.

## References

- [1] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.
- [2] A. Andrews, J. Offutt, and R. Alexander. Testing Web applications by modeling with FSMs. *Software and Systems Modeling, Springer*, 4(3):326–345, August 2005.
- [3] A. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander. Scalability issues with using FSMWeb to test web applications. *Information and Software Technology, Elsevier*, 2009. DOI: 10.1016/j.infsof.2009.06.002, <http://dx.doi.org/10.1016/j.infsof.2009.06.002>.
- [4] A. Cooper and R. Reimann. *Designing for the Web, About Face 2.0: The Essentials of Interaction Design*. Wiley Publishing, 2003.
- [5] R. A. DeMillo and J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [6] C. Eaton and A. M. Memon. An empirical approach to testing web applications across diverse client platform configurations. *International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering*, 3(3):227–253, 2007.
- [7] S. (editor) Christey. 2011 CWE/SANS top 25 most dangerous software errors. online, September 2011. <http://cwe.mitre.org/top25>, last access April 2012.
- [8] S. Elbaum, S. Karre, and G. 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.
- [9] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.
- [10] R. Gold. Httppunit home. online: SourceForge, 2003. <http://htppunit.sourceforge.net/>, last access June 2005.

- [11] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the Foundations of Software Engineering*, pages 145–154, Dubrovnik, Croatia, 2007.
- [12] M. J. Harrold. Testing: A roadmap. In *International Conference on Software Engineering, Workshop on The future of Software engineering*, pages 61–72, Limerick, Ireland, June 2000.
- [13] T. Hilburn and M. Towhidnejad. Software quality across the curriculum. Proceedings of the 15th Conference on Software Engineering Education and Training, February 2002.
- [14] R. Hower. Web site test tools and site management tools, 2002. [www.softwareqatest.com/qatweb1.html](http://www.softwareqatest.com/qatweb1.html).
- [15] Joomla. The joomla project, 2007.
- [16] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, pages 537–542, Taipei, Taiwan, October 2000. IEEE Computer Society Press.
- [17] 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.
- [18] G. A. D. Lucca and M. D. Penta. Considering browser interaction in web application testing. In *5th International Workshop on Web Site Evolution (WSE 2003)*, pages 74–84, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.
- [19] J. Miller, L. Zhang, E. Ofuonye, and M. Smith. Towards automated bypass testing of web applications. In *Web Engineering Advancements and Trends*, pages 212–229. IGI, 2010.
- [20] T. Mouelhi, Y. L. Traon, E. Abgrall, B. Baudry, and S. Gombault. Tailored shielding and bypass testing of web applications. In *4th International Conference on Software Testing, Verification and Validation (ICST)*, pages 210–219, Berlin, Germany, March 2011. IEEE.
- [21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, Makuhari-Messe, Chiba, Japan, 2005.
- [22] J. Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.
- [23] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software-Practice and Experience*, 29(2):167–193, January 1999.
- [24] J. Offutt and Y. Wu. Modeling presentation layers of web applications for testing. *Software and Systems Modeling*, 9(2):257–280, April 2010. DOI: 10.1007/s10270-009-0125-4.
- [25] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of web applications. In *15th International Symposium on Software Reliability Engineering*, pages 187–197, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society Press.
- [26] J. Offutt, Y. Wu, X. Du, and H. Huang. Web application bypass testing. In *Workshop on Quality Assurance and Testing of Web-Based Applications; Associated with COMPSAC 2004*, pages 106–109, Hong Kong, PRC, September 2004.
- [27] V. Papadimitriou. Automating bypass testing for web applications. Master’s thesis, Department of Information and Software Engineering, George Mason University, Fairfax VA, 2006. Available on the web at: <http://www.cs.gmu.edu/~offutt/>.
- [28] D. Raggett, A. L. Hors, and I. Jacobs. *HTML 4.01 Specification - W3C Recommendation 24*. World Wide Web Consortium (W3C), Online: <http://www.w3.org/TR/html401/>, last access September 2007, December 1999.
- [29] F. Ricca and P. Tonella. Analysis and testing of web applications. In *IEEE 23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34, Toronto, CA, May 2001.
- [30] F. Ricca and P. Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14(1-4):93–114, December 2002.
- [31] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *19th IEEE International Conference on Automated Software Engineering*, pages 132–141, September 2004.
- [32] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock. Web application testing with customized test requirements-An experimental comparison study. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 266–278. IEEE Computer Society Press, November 2006.
- [33] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering*, 33(10):643–658, October 2007.

- [34] S. Sprenkle, C. Cobb, and L. Pollock. Leveraging user-privilege classification to customize usage-based statistical models of web applications. In *5th International Conference on Software Testing, Verification and Validation (ICST)*, Montreal, Canada, April 2012. IEEE.
- [35] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th International Conference of Automated Software Engineering*, pages 253–262, Long Beach, CA, USA, November 2005. ACM Press.
- [36] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Proceedings of the 18th International Symposium on Software Reliability Engineering*, pages 253–262, Trollhatten, Sweden, November 2007. IEEE Computer Society Press.
- [37] S. Sprenkle, S. Sampath, E. Gibson, A. Souter, and L. Pollock. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 587–596, September 2005.
- [38] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith. Agile security testing of web-based systems via HTTPUnit. In *Proceedings of the Agile Development Conference (ADC '05)*, pages 29–38, Denver, CO, July 2005. IEEE Computer Society Press.
- [39] A. van Kesteren. *The XMLHttpRequest Object*. World Wide Web Consortium (W3C), Online: <http://www.w3.org/TR/XMLHttpRequest/>, last access July 2009, April 2008.
- [40] W. L. Wang and M. H. Tang. User-oriented reliability modeling for a web system. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 293–304, Denver, Colorado, 2003. IEEE Computer Society Press.
- [41] Y. Wu and J. Offutt. Modeling and testing web-based applications. Technical report ISE-TR-02-08, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2002. [http://www.cs.gmu.edu/~tr\\_admin/2002.html](http://www.cs.gmu.edu/~tr_admin/2002.html).
- [42] W. Xu, S. Bhatkar, and R. Sekar. A unified approach for preventing attacks exploiting a range of software vulnerabilities. Technical Report SECLAB-05-05, Department of Computer Science, Stony Brook University, <http://seclab.cs.sunysb.edu/seclab1/pubs/papers/seclab-05-05.pdf>, August 2005.