The Journal of Systems and Software 83 (2010) 2263–2274

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Perturbation-based user-input-validation testing of web applications

Nuo Li [a], Tao Xie [a,*], Maozhong Jin [b], Chao Liu [b]

[a] Department of Computer Science, North Carolina State University, NC 27695, USA
[b] School of Computer Science and Engineering, Beihang University, Beijing 100083, China

ARTICLE INFO

Article history:
Received 20 April 2009
Received in revised form 30 May 2010
Accepted 2 July 2010
Available online 29 July 2010

Keywords:
Software testing
Web-application testing
User-input-validation testing

ABSTRACT

User-input-validation (UIV) is the first barricade that protects web applications from application-level attacks. Most UIV test tools cannot detect semantics-related vulnerabilities in validators, such as filling a five-digit number to a field that accepts a year. To address this issue, we propose a new approach to generate test inputs for UIV based on the analysis of client-side information. In particular, we use input-field information to generate valid inputs, and then perturb valid inputs to generate invalid test inputs. We conducted an empirical study to evaluate our approach. The empirical result shows that, in comparison to existing vulnerability scanners, our approach is more effective than existing vulnerability scanners in finding semantics-related vulnerabilities of UIV for web applications.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

User-input-validation (UIV) is the first barricade that protects a web application from application-level attacks (Beaver, 2006) such as buffer overflow, code-injection attack, hidden-field manipulation, and cross-site scripting. Attackers can launch these attacks by sending malicious inputs to a web application. As UIV protects a web application against these attacks by rejecting malicious inputs, improving the quality of UIV is a key means of enhancing a web application's security. Unfortunately, web-application developers usually forget to implement UIV, or implement defective UIV. As shown in a recent survey (Open Web Application Security Project, 2007), among the top 10 vulnerabilities of web applications, six vulnerabilities are induced by defective UIV. There is a strong need of an effective way to help improve the quality of UIV, thereby increasing web applications' security.

UIV testing is a common way in practice to improve the quality of UIV. There exist tools (Nikto2, 2008; Wikto, 2008; Acunetix Web Vulnerability Scanner, 2008; Fiddler, 2009; Burp Proxy, 2009; Tamperie, 2009) that test UIV of web applications. These existing tools can be classified into two major categories: crawler-based (Nikto2, 2008; Wikto, 2008; Acunetix Web Vulnerability Scanner, 2008) and proxy-based (Fiddler, 2009; Burp Proxy, 2009; Tamperie, 2009) UIV testing tools.

Crawler-based UIV testing tools retrieve HTML pages automatically, and submit predefined test inputs to the server through these HTML pages. However, using only predefined test inputs may not be suitable to be used for a particular input field.[1] For example, consider that an input field in a web application may require a year value to be between 1999 and 2003. To test this input field, we shall enter possible boundary values such as 1998 or 2004. These boundary values may not exist in the predefined test inputs; hence, it may not be possible to check whether the web application can deal with the boundary values properly. As a result, crawler-based testing tools cannot detect these semantics-related UIV defects. In this paper, we use semantic-related UIV defects to refer to defects that are induced due to the lack of checking the semantics of inputs, and semantic-related test inputs are test inputs that can detect semantic-related UIV defects.

Different from crawler-based UIV testing tools, proxy-based UIV testing tools allow developers to edit HTML requests directly. These tools basically provide a manual testing approach, which keeps the maximum flexibility without providing any help on test-input generation. Weber (2005), a senior security consultant, used Cross-Site Scripting (XSS) as an example to show how to test web applications for such vulnerabilities in practice using the proxy-based UIV testing technique. First, a developer finds some proxy tools that can intercept HTTP requests. Second, the developer maps the site and its functionality by discussing with other developers and project managers. Third, the developer identifies and lists input fields. Fourth, the developer writes test inputs manually. Finally, the developer starts testing with the proxy tools and adjusts test inputs. These manual steps are tedious, and the creation of

* Corresponding author. Tel.: +1 919 515 3772; fax: +1 919 515 7896.
E-mail addresses: nli3@ncsu.edu (N. Li), txie@ncsu.edu (T. Xie), jmz@buaa.edu.cn (M. Jin), liuchao@buaa.edu.cn (C. Liu).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.07.007

[1] In this paper, an input field is an HTML element, such as ⟨select⟩ and ⟨input⟩, through which users can send inputs to a web-application server.

Table 1
Comparison of UIV testing approaches for web applications.

Features	Crawler-based	Proxy-based	PIUIVT
Input fields discovery	Yes	No	Yes
Edit test inputs	No	Yes	Yes
Blind SQL injection	Yes	No	Yes
Predefined XSS	Yes	No	Yes
Context-related UIV test	No	No	Yes
Generate invalid inputs based on valid inputs	For user login	No	Yes
Test oracle	For SQL injection and XSS	No	Yes

test inputs heavily depends on developers' knowledge and experience.

In this paper, we propose a new approach, called Perturbation-based Interactive UIV Testing (PIUIVT), to improve the quality of a web-application UIV with robustness testing of web applications against invalid inputs. Table 1 shows the features of different UIV testing approaches for web applications. PIUIVT combines the automation of crawler-based UIV testing and the flexibility of proxy-based UIV testing. To automatically generate semantics-related test inputs, PIUIVT analyzes the client-side information of a web application to collect input-field information that is helpful to generate valid inputs. PIUIVT associates each input field with a regular expression that defines valid-input constraints for the input field. PIUIVT next perturbs the regular expression to generate invalid test inputs[2] based on the perturbed regular expressions. Similar to crawler-based UIV testing tools, PIUIVT allows testers to manually modify the automatically generated test inputs. To automatically assess the test results (pass or fail), PIUIVT compares structural similarity among the original HTML page, the response page of an invalid input, and the response page of a valid input (which is automatically generated based on the regular expressions that defines valid inputs).

This paper makes the following main contributions:

- An approach based on regular expressions for generating UIV test inputs for web applications. Our approach also includes different types of perturbation operators.
- A strategy based on comparing HTML structures to assess UIV test results (pass or fail), and a solution for the longest common subsequence problem to evaluate similarity between HTML pages.
- A prototype and a set of evaluations of PIUIVT with open-source web applications. Our results show that PIUIVT is effective in web-application UIV testing: PIUIVT detects 80% semantic-related defects that we injected into a web application, while Wikto and Paros detect 25%; compared with other similarity measurement algorithms, our algorithm measures the similarity among HTML pages more precisely.

The rest of this paper is structured as follows. Section 2 introduces background of UIV testing. Section 3 illustrates our approach through an example. Sections 4 and 5 explain the PIUIVT approach and its implementation, respectively. Section 6 presents an empirical evaluation of the approach. Section 7 discusses our limitation. Section 8 discusses related work. Section 9 concludes the paper.

[2] A *valid* test input is a test input that should be accepted by a web application with defect-free UIV, and an *invalid* test input is a test input that should be rejected by a web application with defect-free UIV.

```
<a href=http://www.contoso.com/req.asp?name=
  <FORM action=http://www.badsite-sample-13.com/data.asp
  method=post id="idForm">
      <INPUT name="cookie" type="hidden">
  </FORM>
  <SCRIPT>
       idForm.Cookie.value=document.cookie;
       idForm.submit();
  <SCRIPT>>
Click here!
</a>
```

Fig. 1. An XSS example.

2. Background

Existing technologies such as anti-virus software and network firewall offer comparatively secure protection at host and network levels, but not at the application level (Huang et al., 2004). Application-level attacks are more difficult to detect than attacks at host and network levels. These attacks can come from any on-line user — even authenticated ones (Tipton and Krause, 2006). As UIV checks inputs from any on-line user, UIV is an effective means to protect a web application from application-level attacks. Here, we give a brief introduction of several vulnerabilities in web applications to show how attacks can happen at the application level because of defective UIV.

2.1. Hidden fields

Hidden fields refer to hidden HTML form fields, such as ⟨`input type = hidden name = hl value = "en"`⟩. In many web applications, developers use these fields to transfer values instead of presenting these values to users. Unfortunately, these fields are actually visible and manipulable to users. Malicious users could easily change the values of these fields in HTML source code and send the changed values back to the web application. If a web application uses a hidden field to hold merchandise prices, malicious users could purchase items at little or no cost. These attacks could be successful, because a web application may not validate whether the returning value of a hidden field is the same as its outgoing value, and accepts the illegally changed value.

2.2. Cross-Site Scripting

Cross-Site Scripting (XSS) flaws occur when a web application accepts user-supplied inputs that contain browser-executable scripts, and posts the inputs in an HTML page without validating or encoding. When another user accesses the HTML page, the web browser executes scripts posted in that HTML page. Through XSS, attackers could send an executable script to a victim's browser, and then possibly hijack user sessions, deface websites, introduce worms, etc.

Fig. 1 shows a typical XSS example, which is borrowed from the "Writing Secure Code" book (Howard and LeBlanc, 2003). Suppose that an attacker sends the code shown in Fig. 1 to a bulletin board, and then an innocent user opens that bulletin board and clicks the hyper link of "Click here!". As a result, this user's cookie would be stolen. Such attacks could be successful when the web application does not filter out or transform scripts included in users' inputs.

2.3. SQL injection

SQL injection flaws occur when user-supplied inputs are sent to an interpreter as part of a command or query. Attackers trick the interpreter to execute unintended commands via supplying specially crafted data (Open Web Application Security Project, 2007).

For example, consider a web application that authenticates a user by checking a database in this way:

```
SQLQuery = "SELECT * FROM Users WHERE (UserName='
"+ strUserName +"') AND (Password=' "+ strPass-
word +"');";
if GetQueryResult(SQLQuery) == 0
then authenticated = false;
else authenticated = true;
```

If an attacker enters `X' OR 'A' ='A` for UserName and `X' OR 'A' ='A` for Password and the web application executes the query on the database directly, the SQL statement at runtime becomes:

$$\texttt{SELECT} * \texttt{FROM Users WHERE}(UserName = 'X' OR 'A' = 'A')$$

$$AND(Password = 'X' OR 'A' = 'A');$$

In this way, the attacker bypasses the authentication and accesses all the user information in the *Users* table. Similar to XSS attacks, SQL-injection attacks could be successful if the web application does not filter or transform SQL commands included in users' inputs.

2.4. Unconscious mistakes

Besides the preceding malicious attacks, many users can enter invalid inputs unconsciously. For example, users may enter invalid characters, such as multiple blanks, `&`, and `null` accidentally. These characters may lead to a failure or even crash when they are used for database operations. Even though such inputs may not crash a web application, there can be a negative user experience. For example, when a user signs up for a service, a web application requires the user's email address, and sends an automatically generated password to that email address. If the user enters an invalid email address, the user does not get the password and the service sign up fails.

2.5. Solution

To avoid these vulnerabilities, a web application should validate a user input before using the input for further processing (Howard and LeBlanc, 2003). However, web-application developers often forget validating users' inputs, and UIV is often not correctly developed. As the manual process of testing UIV is tedious and strongly dependent on the experience of developers, we propose a new automatic approach, called PIUIVT, to improve the effectiveness of UIV testing for web applications.

3. Example

We next explain our approach with an example of testing an open-source web application, called Mvnforum (2006). MvnForum is a mature forum system built on the J2EE technology. It was started in 2001 and is continuously evolving. The latest version (1.1RC1) of MvnForum includes 159,409 lines of Java code (excluding Java code embedded in JSP files). The application's registration page contains 31 input fields of which nine input fields are of the type of selection list or check box. MvnForum accepts a user's inputs through these input fields, and inserts the inputs into a database on its server.

Consider an input field of email as an example. MvnForum may expect that a user enters a valid email address, and then insert this email address into a table that records the basic information of registered users in a database. However, the email address may be invalid, and MvnForum must validate the email address before processing it. To test whether MvnForum validates user-input email addresses properly, we submit invalid email addresses. A valid email address can be defined with a regular expression (RegEx1, shown in Fig. 2).

```
[\w-\.]+@([\w-]+\.)+[\w-]{2,4}
```

Fig. 2. RegEx1 an email regular expression.

Based on RegEx1, a valid email address must have a "@", and consists of letters, digits, minuses, and dots ("\w" is equivalent to "[a-zA-Z0-9]"). To test whether MvnForum can detect an invalid email address, a developer can enter a string without "@" or with some invalid characters, such as "$", "&", and "=". In addition, to test whether MvnForum throws exceptions for some invalid characters, a developer can try to enter an empty string to MvnForum.

Another example is that MvnForum allows a user to enter "Birthday" by selecting dates from three selection lists in the registration page. In this way, acceptable inputs of these input fields are restricted to the selectable options in the selection lists. Users are supposed to enter an input by selecting an option from the selection lists. However, inputs from this input field still need to be checked, as users can send inputs to a server through parameters of a URL instead of using selection lists. In this way, a user can enter unexpected inputs, such as numbers out of the boundary of valid day or month (e.g., 40 for day and 13 for month), to these input fields.

Furthermore, consider an input field of address in MvnForum. MvnForum expects a user to enter a string that stands for an address to this input field. However, a user may enter some strings containing SQL statements, such as `name of a street'; DROP TABLE mvnforumgroups; SELECT * FROM mvnforummember WHERE name LIKE '%'`. If MvnForum cannot detect the SQL injection, the successful execution of the query can delete the `mvnforumgroups` table and retrieve the private information of members from the `mvnforummember` table. (An attacker may know the table name from error messages shown by a web application when he launches attacks.)

With PIUIVT, we automatically parse the registration page to retrieve the 31 input fields and their surrounding text, and analyze the surrounding text to identify the type of each input field.[3] Each type is associated with a regular expression that defines valid inputs for the type of input field. For example, we associate "Email" with RegEx1. If an input field is a selection list, such as "Birthday", we automatically abstract its selectable elements, i.e., option values of the selection list, and rewrite them following the grammar of the regular expression. For input fields whose regular expressions are not predefined, testers can derive expressions based on the input field's surrounding text. Then, we use predefined perturbation operators (explained in Section 4.2) to perturb these expressions to generate invalid inputs, and automatically send the generated invalid inputs to the web application. The returned HTML page is saved on a local directory. Testers can continue this procedure on the same page by different valid or invalid inputs. To determine the test results (pass or fail), we compare the similarity among the returned pages of valid and invalid inputs (the strategy is discussed in Section 4.3).

4. Approach

The goal of PIUIVT is to generate invalid test inputs and assess test results of UIV testing for web applications from the client side. The input of PIUIVT is an HTML page; the output of PIUIVT is invalid test inputs and UIV test results. Fig. 3 presents three main components of PIUIVT. First, the input-field identifier identifies input

[3] The type of an input field refers to which kind of text is expected for the input field, such as email address, zip code, and credit-card number.

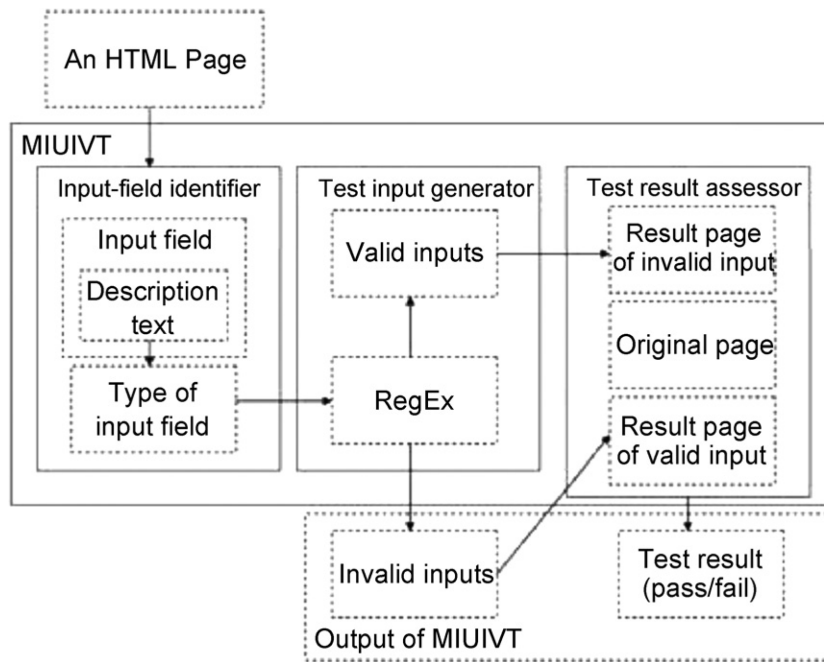

Fig. 3. Overview of PIUIVT.

fields and descriptive text surrounding the input fields in HTML pages. Then, the input-point identifier determines the type of each input field based on analysis of key words in the descriptive text. Second, based on a user-defined mapping between an input-field type and the regular expression that defines valid test inputs for the input-field type, the test-input generator relates each input field to a regular expression that defines valid test inputs for the input field. The test-input generator next perturbs the regular expression to generate invalid test inputs. The test-input generator also uses the regular expression to generate valid test inputs, which are used by the test-result assessor. Third, the test-result assessor computes structural similarities among the result page of invalid test inputs, the result page of valid test inputs, and the original HTML page to produce test results (pass or fail).

4.1. *Input field identification*

Given a web application under test, the input-field identifier extracts input fields from HTML pages by accessing the web application through its URL and analyzing HTML tags. The extracted information of an input field includes the variable name and basic attribute values of the input field (e.g., an ⟨Input⟩ tag is visible or hidden).

After locating an input field (i.e., an HTML element that accepts user inputs), the input-field identifier identifies descriptive text surrounding the input field. When we access an HTML page through a web browser, we know which types of inputs are expected for an input field based on the text surrounding the input field, such as an email address or zip code. However, in HTML source code, the descriptive text surrounding an input field may not be physically adjacent to the input field (the input tags, text tags, and format tags are often intermixed in HTML source code). Therefore, the input-field identifier cannot determine the descriptive text of an input field based on what text is physically adjacent to the input field in HTML source code. To address the issue of locating input field description, we identify descriptive text of an input field based on the analysis of HTML Document Object Model (DOM). DOM is a tree-structure (a node tree) representation of a HTML document, with elements, attributes, and text (HTML DOM Tutorial, 2009). Fig. 4 shows an example of the DOM tree. To identify the descriptive text of the ⟨input⟩ tag, we prune the sub-tree belonging to the ⟨form⟩ tag, and then analyze the tags around the ⟨input⟩ tag in the sub-tree to identify which text is descriptive text of the ⟨input⟩ tag. In Fig. 4, the descriptive text of the ⟨input⟩ tag is identified to be the text ("search") of its brother node ⟨Text⟩.

Sometimes the descriptive text of an ⟨input⟩ tag is in a select box before it, or in the text node after it. If an input field is surrounded by a text node before, a text node after, and a selection list before or after, which node is the descriptive text of the input field? To answer this question, we analyzed the descriptive text location for 462 input fields in 50 popular websites. We randomly chose the 50 popular websites as our subjects, most of which are e-commercial websites and contain registration forms or search engines. The detailed information about the analyzed websites can be found in our project website (Perturbation-based User-Input-Validation Testing of Web Applications, 2010). We checked the source code of the web pages that contain the 462 input fields and found six different locations: the first text node before an input field, the first text node after an input field, default value of an input field, the first button after an input field, the first select list before an input field, and the first select list after an input field. We counted how many input fields' descriptive text locates in each of these six different kinds of locations. Table 2 shows our analysis results. Column 1 ("Locations") lists different descriptive text locations. For each location, Column 2 ("IF#") lists the number of input fields whose descriptive text appearing in the location. Column 3 ("Percentage") lists the percentage of each descriptive text location (i.e., for each row, Percentage =IF #/462 * 100%). Column 4 ("Priority") ranks the priority of descriptive text locations based on the percentage. From Table 2, we noticed that 90% input fields in these 50 websites set their descriptive text in the first text node before an input field. Therefore, the first text node before an input field has the highest priority to be the descritive text of an input field. Our input-field identifier identifies descriptive text of input fields based on the priority listed in Table 2, and then identifies the type of an input field by analyzing key words in the descriptive text. In addition, if an input field's type is file, password, hidden, or submit,

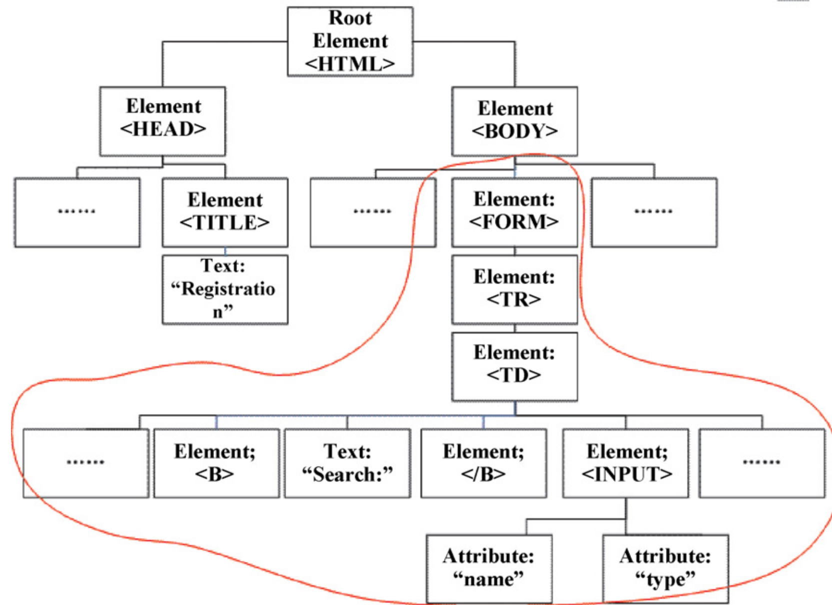

Fig. 4. DOM tree of an HTML page.

we can know the type of such an input field without analyzing its descriptive text.

As many HTML pages contain the same forms in a web application, it is unnecessary to generate test inputs for the same forms. To avoid redundant test-input generation, the input-field identifier combines the same forms. Two forms in a web application could be exactly the same, inclusive, approximately the same, or different as identified next. Formally, given two forms `f1` and `f2`, the value of their attributes (`action`, `method`, and `enctype`) are listed here: `f1.action=a1, f1.method=m1, f1.enctype=e1, and f2.action=a2, f2.method=m2, f2.enctype=e2`. The input fields belonging to `f1`, including inputs, selects, textareas and buttons, are the set `S1` while the input fields belong to `f2` are the set `S2`. If $(a1==a2) \wedge (m1==m2) \wedge (e1==e2)$, then `f1` and `f2` can be merged into one form whose input fields are $S1 \cup S2$.

4.2. Test-input generation

In the previous section, we described how to identify an input field and its descriptive text. In this section, we describe how to generate test inputs for an input field based on its descriptive text.

Given the descriptive text of an input field, we can know which types of input, such as email and zip code, this input field expects. Is it possible to automatically identify an input field's type based on its descriptive text? When we analyzed the descriptive text locations for the 462 input fields mentioned in the previous section, we also manually extracted a key word in the descriptive text for each input field. In summary, we extracted 68 different key words, such as email, phone, and address, which represent input types for 440 input fields. Among the 462 input fields, we could not extract key words for 22 input fields from their descriptive text, such as "How can we assist you?". In other words, we can identify the input types for over 95% input fields by extracting key words from their descriptive text.

Table 2
Statistics of descriptive text locations for 462 input fields.

Locations	IF #	Percentage	Priority
1st text node before IF	416	90.0	1
1st text node after IF	27	5.8	2
Default value	8	1.7	3
1st button after IF	6	1.3	4
1st select list before IF	4	0.9	5
1st select list after IF	1	0.2	6

IF: Input Field.

For common input types, such as email address, zip code, and SSN, users can define regular expressions for their valid test inputs. Based on a mapping between input-field type and the user-defined regular expressions, the test-input generator relates each input field to a regular expression that defines valid test inputs for the input field. Then, the test-input generator generates invalid test inputs based on perturbing the regular expression of valid test inputs for each input field. The invalid test inputs can detect UIV vulnerabilities in a web application under test, as the invalid test inputs can induce faulty behaviors of the web application, if there exist UIV vulnerabilities in the web application under test (explained in Section 4.3).

Before we illustrate our perturbation strategy, we define that a regular expression is a sequence of ordered sets and regular expression operators. Here, a set refers to a set of optional or mandatory constants in a regular expression, and the regular expression operators refer to the repetition time of selecting elements from a set, including "+", "*", "?", and "{min,max}". Suppose that a regular expression is a 4-tuple (C, S, O, Q) where C is the finite alphabet of characters, S is a finite set of characters, O is a set of the regular expression operators $(\{+|*|?|\{\min, \max\}\})$, and Q is an ordered

```
[\w-\.]+([\w-]+\.)+[\w-]{2,4}
@[\w-\.]+([\w-]+\.)+[\w-]{2,4}
[\w-\.]*@([\w-]+\.)+[\w-]{2,4}
[\d]+@([\w-]+\.)+[\w-]{2,4}
[\w-\.]+@([\w-]+\.)+[\w-]{2,4}
[\w-\.]+@([\w-]+\.)+[\w-]{2,4}' OR 'A' = 'A
```

Fig. 5. Perturbed RegEx1.

sequence of character sets and the regular expression operators, a subset of $S \times S \times O$.

For example, RegEx1 (shown in Fig. 2) contains four operators and contains four sets, one of which has two subsets. "\w-\." are constants in Set 1, and at least one of these constants should appear in the string. "@" is the only one element in Set 2, and it is mandatory. "[\w-]+" and "\." are two subsets in Set 3, and at least one string that consists of these two subsets should appear in the final string. "\w-" are constraints in Set 4, and at least two and at most four of these constraints should appear in the string.

After associating an input field with a regular expression that defined valid inputs for the input field, we generate test inputs for UIV testing by perturbing the regular expression of valid inputs. We perform six types of operations (MO1–MO6) on regular expressions.

MO1: Remove the mandatory sets from a regular expression.
MO2: Disorder the sequence of sets in a regular expression.
MO3: Change the repetition time of selecting elements from a set.
MO4: Select elements from complementary sets of sets in a regular expression, especially characters next to the boundary of the input domain.
MO5: Insert invalid and dangerous characters, such as an empty string, strings starting with a period, and extremely long strings, into a regular expression.
MO6: Insert special patterns, such as SQL queries and XSS segments, into regular expression.

Take RegEx1 shown in Fig. 2 for instance. If we apply each of the perturbation operators (MO1–MO6) on RegEx1 once, we can get six perturbed regular expressions shown in Fig. 5.

To better identify which UIV vulnerability exists in a web application, we apply perturbation operators individually, instead of in a mixed way. Even so, the number of test inputs is out of control. In order to reduce the cost of testing, we treat the elements belonging to the same set in a regular expression as equivalent.

After we apply MO1 and MO3–MO6, test inputs generated with the perturbed regular expressions cannot be matched by the original regular expression, because MO1 removes the mandatory sets from the original regular expression. MO4–MO6 insert some invalid string to the original regular expression. MO3 causes the number of elements selected from the same set to be different from the original one. However, a perturbed regular expression with MO2 may generate a string that is also matched by the original regular expression, as the two exchanged sets could have intersections. To avoid such repetition, we could compute the subtraction of the two sets before they are exchanged, and then exchange the subtraction. However, it is expensive to compute the subtraction of each pair of the exchanged sets. In our implementation, we check a string generated from the perturbed regular expression with MO2 to check whether it is matched by the original regular expression. If the string is matched by the original regular expression, this string is discarded.

4.3. Test-result assessment

PIUIVT automatically fills input fields in an HTML page with the test inputs generated based on perturbed regular expressions. Users can edit the test inputs manually and click a button to submit the test inputs to the web application under test. As defining a test oracle for each test input is tedious and sometimes hard, we develop a technique to assess the test result as pass or fail without the need of test oracles. We classify the behavior of a web application in the context of UIV testing into three types: defensive, insensitive, and crashing. If a web application rejects invalid test inputs, the web application is determined to be defensive, and the test result is pass. If a web application accepts invalid test inputs, the web application is determined to be insensitive or crashing, and the test result is fail. We next give the detailed definitions of these three types.

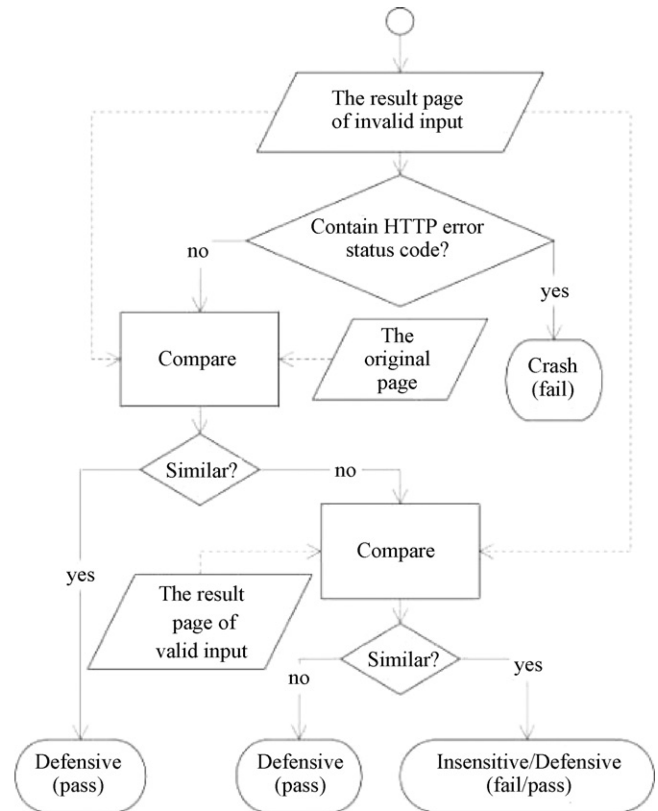

Fig. 6. Flowchart of test-result assessment.

In particular, a web application is defensive, if the web application can detect invalid test inputs to be invalid. There are three types of result pages returned by defensive web applications. The first one includes the same form as the original page, plus suggestive directions, such as "please enter a valid email address"; the second one includes no form and shows some suggestive directions; and the third one is a redirected result page. In the second type of result, the layout of the result page differs greatly from the original one and does not look like the result page of a valid input. The third type is hard to determine automatically because a web application can redirect the request to any other page on its server.

A web application is insensitive if the returned page of an invalid input is structurally similar to the returned page of a valid input. Two HTML pages are structurally similar if the two HTML pages have similar layout when we open them with a browser. It is not easy to automatically identify whether two HTML pages are structurally similar based on comparing their layout in browsers. To address this challenge, we extract HTML tag sequences from two HTML pages and compare the similarity between the two HTML tag sequences, as the structure of an HTML page depends on the HTML tag sequences of this page.

A web application is crashing, if the returned page contains HTTP error status code, such as "404 Not Found". In such cases, a web application accepts an invalid input and returns a page that leaks information about configuration or internal workings through error text. This information can be leveraged to launch or even automate powerful attacks (Open Web Application Security Project, 2007).

To classify the behavior of a web application, we use the following steps for classifications (shown in Fig. 6). At the beginning, we

determine whether the returned page of an invalid input contains any pre-defined HTTP status code (such as "404 Not Found"). If yes, the web application is determined to be crashing. If no, we compare the returned page of the invalid input with the original page and the result page of a valid input. If the result page of the invalid input is structurally similar to the original one, the invalid input is prevented, and the web application is defensive. If the result page of the invalid input is similar to the result page of the valid input, there are two possibilities: the web application is insensitive, because it accepts the invalid input wrongly; the web application is defensive, being robust enough, and it redirects to another HTML page with the valid input. In the case that the result page of the invalid input is different from both the result pages of the valid input and the original page, the web application is determined to be defensive.

A challenging task during the process of test-result assessment is to determine whether two HTML pages look similar. A previous study (Hunt and McIlroy, 1976) determined the similarity between two text files by computing the Longest Common Subsequence (LCS) (Maier, 1978) using dynamic programming. This mechanism has been widely used in "Diff" programs since it was introduced in 1978 but it is not appropriate for evaluating the similarity of the layout of two HTML pages, because the layout of an HTML page is determined by the sequence of HTML tags. Therefore, we need to focus on the tag sequence instead of lines. In addition, two HTML pages may be similar in appearance but have different values in some HTML tags or different HTML tags that are invisible when we open the HTML page in a browser. For example, when we check the detailed information for a book on the website of a library, the listed information is different for each book, but all the pages for listing book information are structurally similar to each other. Based on such observations, we develop the Longest Common Tag Subsequence (LCTS) solution to assess the similarity of HTML pages. LCTS treats two HTML pages as two sequences of HTML tags, which are defined as a series of tag names along with their attributes (key-value pairs that do not contain actual text contents). LCTS finds the longest common subsequence of two HTML files in the following way. We first prune the DOM trees of HTML pages by removing pure text node and context-related tag attributes, such as href, src, alt, and title, and then compute the LCTS for two HTML pages under comparison. For the compared two pages, they are considered as similar pages, if they have similar tag sequences, based on that max (1-(LCTS amount)/ (Tags amount)) is less than our predefined threshold (30%). In this paper, the predefined threshold is set based on the experimental result shown in Section 6.3. Different web applications have different web page styles and the threshold may need to adjust. We suggest applying LCTS on a small group of web pages in the web application under test to set the threshold before performing PIUIVT.

5. Implementation

Our approach is implemented in Java with related web technologies. PIUIVT consists of three components: input-field identifier, test-input generator, and test-result assessor (shown in Fig. 3).

The input-field identifier crawls HTML pages by their URLs and uses an HTML analyzer (HTML Parser, 2006) to generate a DOM tree for each HTML page. The input-field identifier extracts input fields based on the type of nodes in the DOM tree. To get descriptive text of input fields, PIUIVT filters input-field nodes and the nodes surrounding input-field nodes from the DOM tree. Then, the input-field identifier uses the pseudo-code shown in Fig. 7 to identify descriptive text of input fields. Fig. 7 shows how to get the descriptive text before an input field (the process of getting the descriptive text after an input field is symmetric to the pseudo-code shown in Fig. 7). Since there may be some font information inserted in the

```
getPreText(Input node, String lastString):
    ∀ node_i before n, node_i is the brother node of n,
        String text= "";
        if node_i ∈ uselessTagSet then
            jump over it
        else if node_i ∈ endTagSet then
            getPreText(node_i, text)
            else text=getTagText(node_i)
                if text is sensible then
                    return text.concat(lastString)

getTagText(Node n):
    Case TextTag or LinkTag:
        return filtrateString(n.getText())
    Case SelectTag:
        return getOptionTags()[option.size/2].getOptionText()
    Case InputTag:
        if type of this input tag equals submit or button then
            return filtrateString(n.getValue())
    Case LableTag:
        if text of this tag is sensible
            return filtrateString(n.getText())
        else getTagText(n.children)
```

Fig. 7. Pseudo code of the input description abstractor.

text around an input field, the process of text identification is recursive. When the descriptive text is identified, the identified text is compared with the predefined key words in a topic base to identify the type of input fields. The descriptive text of an input field may match several types. In this case, a type is selected on the following order: first the type matching the longest string in the descriptive text and then the type appearing the most times in the descriptive text. If the descriptive text still matches several types after such a process, we suggest the possible types to users and allow users to decide what the type is. Furthermore, if the type cannot be obtained with descriptive text or PIUIVT fails to get the descriptive text, PIUIVT tries to identify the type based on the value of some other attributes of the input field, including the value of its name, id, and value. PIUIVT treats these values as the descriptive text of the input field and then identifies the type.

Given the type or acceptable input domain of an input field (i.e., the values listed in the options of a selection list, or the values of a check box or a radio box), the test-input generator automatically maps the type to a regular expression that defines valid inputs for this type, or constructs a regular expression to define the acceptable input domain of this input field. We collect predefined regular expressions of valid test inputs from RegExLib (Regular Expression Library, 2007). RegExLib provides various types of regular expression, such as Email, Uri, numbers, strings, dates, times, address, phone, markup, and code. These regular expression types cover the 68 different kinds of key words we identified for the 437 input fields. If the input-field identifier fails in identifying the type of an input field, PIUIVT assigns a default regular expression for the input field, and allows user interaction in modifying the regular expression.

The test-input generator next perturbs regular expressions automatically and generates test inputs based on the perturbed regular expressions. To implement our perturbation operators, we first use an open source library dk.brics.automaton (dk.brics.automaton, 2007) to parse a regular expression into sets and regular expression operators defined in Section 4.2. Next, we mutate the parsed regular expression by removing mandatory sets, disordering sets, changing regular expression operators, replacing or inserting invalid characters into a set, and inserting SQL injections into a set. We have implemented all six types of perturbation operators (MO1–MO6).

Then, we use a method provided by dk.brics.automaton to generate a set of acceptable strings for the perturbed regular expressions.

The test-result assessor produces a test result based on retrieving result pages, checking whether the test result is crashing, and doing similarity comparison. We use regular expressions for checking whether there exist specific patterns of the combination of HTTP status code (Fielding et al., 2007) in the text of the result page. If none matches, following the steps described in Fig. 6, we assess the similarity among the result page of an invalid input, the original page, and the result page of a valid input.

6. Evaluation of the approach

6.1. Comparison with scanners

In this section, we analyze popular web-application scanners to show their limitations on testing UIV of web applications, and present the comparison of PIUIVT and vulnerability scanners of web applications. Scanners of web applications are tools that automatically crawl through web applications and parse the HTML pages of web applications to identify vulnerabilities by launching various attacks. Both PIUIVT and scanners are applied on testing the MvnForum. The results show that, comparing with scanners, PIUIVT detects more semantic-related UIV vulnerabilities.

The scanners that we choose are among the 10 most popular scanners summarized by Fyodor (Top 10 Web Vulnerability Scanners, 2006). Since not all of them can be freely used for any web application, we first analyze these scanners based on their specifications and demonstration sites.

The mechanism of scanners is different from ours. The existing scanners fetch HTML pages of a web application from the client side, and are primarily based on pre-defined defects or rules to detect whether there exist security vulnerabilities in the web application. Most of these scanners focus on security-vulnerability detection using known defects recorded in a database, such as the Open Source Vulnerability Database (OSVDB, 2008). Both Nikto2 (2008) and Wikto (2008), two open-source scanners, use OSVDB to scan for possible existence of directories and files that hackers usually try to find and treat as an entry point such as the / admin directory and *.*property* file. Furthermore, Google can be used as a tool to search for signatures of online websites (Google Hacking Database, 2008). Wikto and Acunetix Web Vulnerability Scanner (2008) use such signatures to populate to their vulnerability database. A database-based approach is good for defect detection of a server, network, and popular software, but those recorded defects cannot be applied to detect new defect types in various applications under test. It is just like we cannot use known virus signatures to detect new-born viruses.

Besides defect-database-based detection, some of the top 10 scanners, such as Paros (Paros - for Web Application Security Assessment, 2008) and IBM Rational AppScan (Appscan Suite for Web Application Security Testing, 2008), provide rule-based SQL injection and XSS injection detection capabilities. XSS injection testing, which is relatively easy to implement, adds script text to an input and checks whether the added script text exists on the response page. SQL injection, especially its assessment of test results, is somewhat complex. One of the approaches that IBM Rational AppScan uses for test-result assessment is capturing error messages such as "`OleDbException`" to determine whether an SQL error happens. Similarly, from source code, Paros also uses some naive detection such as searching for "SQL, ODBC, JDBC" in defect detection. Such an approach can cause high false alarms. Paros reports 48 high-risk alerts for MvnForum, but all of them are false alarms because MvnForum uses search engine optimization technologies such as adding popular keywords (including "JDBC" and "MySQL") in meta tags of every page.

The limitation of the preceding approaches lies in that they aim at universal solutions for any web application. Actually, web-application testers can often understand a web application well based on its descriptive text in HTML pages, and the testers need flexibility for designing test inputs but are reluctant to generate test inputs directly by themselves. Our approach focuses on filling the gap of this kind of requirement. We automatically generate inputs based on testers' design reflected by RegExs.

To assess the effectiveness of PIUIVT, we tested MvnForum's member registration page (described in Section 3). Since this registration page contains one CAPTCHA text box, and inputs to two input fields need to be the same as those to two other corresponding retype input fields, it makes all top-ten tools not usable: a crawler cannot fetch the dynamically generated pages because of CAPTCHA and the comparison between input fields while a proxy needs human's diligent typing. PIUIVT provides interactive means to let users manually overcome these challenges, automatically generate RegExs for input fields with common types, and allow users to choose perturbation operators. Based on these inputs, PIUIVT can automatically generate a set of test inputs.

Another fact is that without knowing the meaning of an input field, scanners cannot deliberately generate out-of-scope inputs for testing related faults. We test the demonstration website of IBM Rational AppScan with PIUIVT, and we detect that the demonstration website has a seeded fault that allows a user to transfer money from one account to his or her own account. IBM Rational AppScan does not detect this defect, although this fault was intentionally injected by the tool vendor. For MvnForum, many input fields are given by selection or scoped digits such as year, month, day, and age. For thorough testing, we need to check what shall happen if an input is out of scope. After our empirical study, we found two faults of MvnForum 1.0.2, which are fixed in the latest version. We detect both of these two faults by submitting invalid inputs, which induces SQL exceptions in MvnForum. MvnForum displays its internal function names when some generated unexpected inputs were entered.

6.2. A case study

To demonstrate the effectiveness of perturbation-based test input generation, we apply perturbation operators on a benchmark web application with 20 seeded UIV vulnerabilities, compared to testing it with web-application scanners. Test results show that our approach complements web-application scanners by detecting more types of vulnerabilities. Furthermore, we analyze the test inputs generated by perturbation operators for MvnForum, and explain why UIV vulnerabilities can be detected by these test inputs.

We create a web application including 20 UIV vulnerabilities (eight of the vulnerabilities are from the NIST SAMATE Reference Dataset Project, 2007). We use PIUIVT and scanners (Wikto and Paros) to test the application. These vulnerabilities include a tainted output that allows cross-site scripting or SQL injection attack, an exception that leaks internal path information to the user, a tainted input that allows arbitrary files to be read and written, etc. We classify these 20 vulnerabilities into four types based on the type of test inputs (shown in Table 3). Based on the classification of UIV for web applications proposed by Offutt et al. (2004), we classify invalid inputs into four types and list them in Column 1 of Table 3. Column 2 of Table 3 lists probable exceptions caused by each type of invalid inputs. Column 3 shows the types of perturbation operators used to generate test inputs that cause exceptions. The last two columns of Table 3 list the ratios of vulnerabilities detected by the two approaches. The ratios are "the number of detected vulner-

Table 3
UIV fault detection results by three approaches.

Inputs	Probable exception	Corresponding MOs	Detection ratio	
			PIUIVT	Scanners
Invalid length	1. Substring exception (false assumption) 2. Null pointer exception 3. Buffer/memory overflow 4. Out of boundary access	MO1, MO5 MO1 MO1 MO1	2/2	0/2
Invalid type	5. ParseInt, parseFloat, etc exception 6. Date, Address parse exception 7. Parse URL exception 8. Send mail error (invalid mail address)	MO1 MO1–MO5 MO1–MO5 MO1–MO5	4/4	0/4
Invalid value	9. Buffer/memory overflow 10. Visit file error (invalid file name) 11. Foreign key constraint violation caused database exception 12. Network connection error (invalid host or port)	MO5 MO1–MO5 MO5 MO5	7/9	0/9
SQL injection and XSS	13. Unexpected script execution 14. SQL error 15. Mal-operation on database (such as illegal insert, delete, and query)	MO6 MO5, MO6 MO6	3/5	5/5

abilities in a type/the total number of vulnerabilities in the type". We manually counted the number of detected vulnerabilities and checked whether the test results are correct.

Similar to the analysis in the previous section, scanners focus on detecting dangerous configurations on a server, and try to conduct XSS and SQL injection based on a predefined database. As a result, scanners cannot detect the lack of UIV on this website, while PIUIVT can easily detect the lack of UIV. In contrast, the scanners work better for SQL injections and XSS attacks than PIUIVT, because the scanners generate such attacks based on a large database of defects. If we incorporate such a database with PIUIVT, PIUIVT would achieve similar capability as the scanners in testing SQL injections and XSS attacks. On the other hand, PIUIVT's detection ratio of "Invalid Value" is 7/9. The two "Invalid Value" vulnerabilities that PIUIVT cannot detect are wrong restrictions that reject some valid inputs. PIUIVT cannot detect such a wrong restriction, as the detection requires specific valid test inputs instead of invalid test inputs (the detailed discussion is provided in Section 7).

In addition, when we use PIUIVT to test MvnForum, because MvnForum does not check whether a submitted date value is meaningful, we can submit an invalid date value, and the registration process accepts this invalid input. As the date is stored in a MySQL *datetime* field, an erroneous date (0000-00-00) is stored in its corresponding database. After inserting an invalid birthday into the database, we next query the information of registered users, and MvnForum throws an exception with its internal function name ("*Error executing SQL in MemberDAOImplJDBC.getMember_forPublic(pk)*"). We analyze the error log and find that this error is caused by "*java.sql.SQLException: Value '0000-00-00' cannot be represented as java.sql.Date*".

Besides the perturbation-based test input generation, following the classification of UIV given by Offutt et al. (2004), there should be two other types of UIV tests that consider inter-value constraints or control flow restrictions. For example, MvnForum has a form that allows users to post a thread (a thread is a set of messages grouped visually in a hierarchy by topic) in an existing forum. MvnForum inserts the text in the thread into a database table that has the same forum ID as the forum where the thread is posted. To test whether MvnForum checks the existence of forum ID for a posted thread, we can post a thread to a non-existent forum in MvnForum with an out-of-scope ID. Such test inputs are strongly correlated with the business logic of a web application, and it is quite difficult to generate such test inputs without domain knowledge. In our approach, PIUIVT allows testers to design such test inputs and generates an "out-of-scope id" by perturbing a RegEx for the valid input domain.

6.3. Evaluation of LCTS

We use LCTS and four traditional approaches of similarity comparison to evaluate the similarity between different HTML pages with similar layouts but different contents, or similar contents but different layouts. The experimental results show that LCTS works better than those four traditional approaches when evaluating the similarity of HTML pages' layout.

Three of the compared approaches are based on an LCS algorithm, *2*LCS/(LEN1+LEN2)*100%, LCS=the longest common subsequence of PAGE1 and PAGE 2, LEN1=length of PAGE1, LEN2=length of PAGE2, PAGE1 and PAGE 2 are two web pages*, on different levels (character, line, and word). The fourth one counts the same characters of two files from the beginning to the first different character from the two files. Table 4 presents our comparison results on HTML pages in MvnForum. The higher percentage stands for the larger similarity. In Column 1, Line Diff, Word Diff, and Char Diff stand for the algorithms of longest common line, word, and character sequence, respectively, and Char Comp stands for character comparison. Column 2 in Table 4 presents the similarity between different threads' post pages, and each percentage in this column is an average value measuring on ten pairs of pages. Different threads' post pages have different contents but similar layouts. We expect the values in Column 2 to be high. Column 3 in Table 4 presents the similarity between successful login and failed login pages. As there are two types of failed login pages: failed because of a wrong user

Table 4
Applying different similarity comparison approaches to HTML pages in MvnForum.

Approach	Different thread's posts (%)	Admin login page: success vs failed (%)	Admin login page: different failed pages (%)
LCTS	81.01	23.72	67.69
Line Diff	79.99	34.30	55.23
Word Diff	71.13	34.23	60.70
Char Diff	77.59	40.23	60.16
Char Comp	0.40	0.08	0.14

name and failed because of a wrong password, each percentage in Column 3 is an average of similarities among successful login page and each type of failed login pages. As a successful login page and failed login pages have different layouts, we expect that the values in Column 3 are low. The last column of Table 4 presents the similarity between these two kinds of failed login pages, and we expect that the values in Column 4 are high.

From Table 4, the character-based comparison reports low similarity for pages no matter whether these pages are similar or different due to the variance of values in template slots. Since MvnForum's HTML code contains appropriate line breaks, Line Diff performs well, but LCTS is better to distinguish different kinds of pages' similarity (the difference between a successful login page and a failed login page should be smaller than the difference between two failed login pages, and LCTS detects such a difference more precisely than Line Diff). In addition, not every web application generates HTML pages with appropriate line breaks. For example, if we use Google to search "software" and "testing", the similarity calculated by LCTS is 78.76%, and by Line Diff is 25.00%. This result is because no line ending exists in Google-generated HTML pages. Char Diff tends to report high similarity, because the character set is limited. As a result, Char Diff may consider two totally different pages as similar pages. Furthermore, computing Char Diff on two HTML pages is expensive in terms of computation and memory cost. After all, algorithms of LCS are in quadratic complexity. For approximately 10K-size pages, Char Diff requires minutes to finish while others finish in milliseconds using a single-core computer. If we do not use Hirschberg's algorithm (Hirschberg, 1975) to improve the algorithm of Char Diff, the memory requirement cannot be met in our study platform (about 600M memory needed in a Java implementation using a typical LCS algorithm for Char Diff (Maier, 1978)). In summary, Char Diff is not practical. Word Diff is also expensive and less effective; its runtime cost is greater than one tenth of the cost of Char Diff, because the average character length of an English word is less than 10.

To further study the capability of LCTS, we apply PIUIVT on an open-source-code search engine, called Koders (2008). Koders has three fields of search conditions, the search terms, programming language, and type of licenses. Programming languages and types of licenses are selection lists. PIUIVT submits "email validator" as the search term, "Java" as the programming language, and "X' OR 'A' = 'A" as the type of licenses, to Koders. Because Koders is a well-developed web application, it is robust enough to deal with invalid inputs, and treats the invalid search condition as predefined default input. The test-result page has a similar layout of a normal search-result page, but different contents. We evaluate the similarity between this test-result page and a normal search-result page by LCTS, Line Diff, Char Comp, Char Diff, and Word Diff, and similarity ratios are 87.03%, 70.90%, 0.46%, 67.77%, and 56.88%. The similarity ratio computed by LCTS is much larger than the others.

The results show that our LCTS solution is more practical for measuring of the similarity of HTML pages than existing approaches.

6.4. Threats to validity

The threats to external validity primarily include the degree to which the subject programs are representative of true practice. Our subjects are from various sources. These threats could be further reduced by experiments on more subjects and third-party tools. We currently use a third-part library dk.brics.automaton (2007). The main threats to internal validity include faults in our tool implementation and faults in the third party library that we use to parse regular expressions and generate test inputs. To reduce these threats, we have manually inspected the generated test inputs for several program subjects. These threats can be further reduced by conducting experiments using more regular expression parsers and regular-expression-based string generators.

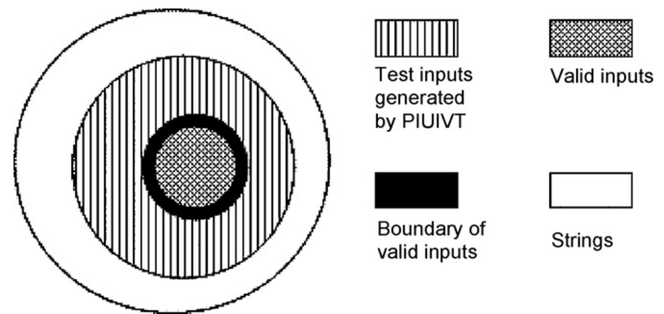

Fig. 8. Different test-input domain.

7. Discussion

PIUIVT provides a perturbation-based approach to generate UIV test inputs to improve the effectiveness of UIV testing. The perturbed object is a regular expression describing valid inputs for an input field. It is not difficult to get such regular expressions on Internet. In regexlib.com, there are 777 regular expressions, which cover most of common types of inputs for web applications. In addition, our approach intends to reduce the manual efforts in testing UIV but not to replace all manual efforts. Due to the diversity of web applications, some manual efforts may be required for defining input type or regular expressions.

A challenge for PIUIVT is that it cannot detect UIV vulnerabilities when there is a wrong constraint of inputs. For example, a web application may expect that the input year is between 1999 and 2008, but present 1998–2008 in its selection list. If a user enters 1998, the web application may throw an exception. However, based on the information in an HTML page, PIUIVT supposes that a valid input should be a year between 1998 and 2008, and generates test inputs out of this period. As a result, the exception induced by 1998 cannot be caused by test inputs generated by PIUIVT. As shown in Fig. 8, test inputs generated by PIUIVT belong to the complement of valid inputs, and near the boundary of valid inputs. If the information in an HTML page cannot reflect the real boundary of valid inputs, the effectiveness of test inputs generated by PIUIVT would be compromised.

The observation on possible behaviors of an attacked web application is based on our experience. We do not conduct a manual analysis on the behaviors of attacked web applications as the manual analysis for input fields, as it is very time consuming or infeasible for us to collect the source code of 50 websites with UIV vulnerabilities, set them up, and observe their behaviors for invalid inputs. In future work, we plan to collect and analyze statistical data about attacked web applications' behaviors as empirical evidence. In addition, we currently compare the effects of LCTS and other similarity measures based on three different kinds of web pages. In future work, we plan to perform the comparison based on more web pages to evaluate how often each of the similarity measures could correctly report two web pages that should be classified as different (or as the same) with respect to the PIUIVT methodology.

8. Related work

Lucca and Fasolino (2006) classify strategies for web-application testing into white-box testing, black-box testing, and gray-box testing.

White-box testing generates test inputs based on an abstract structure of source code, which can be generated based on client-

side source code (Liu et al., 2000a) or server-side source code (Liu and Tan, 2006). Liu et al. (2000a,b) analyze HTML documents to create data-flow models for a web application, and generate test inputs based on the data-flow models. Ricca and Tonella (2001) propose a UML model of web applications as a high-level representation. Benedikt et al. (2002) use a model checker to explore web-site execution paths that can be followed by a user in a web application. Compared with the model-based approaches, our approach focuses on the UIV testing of web applications and does not require any model. Liu and Tan (2006) abstract a control flow diagram from server-side source code. Based on the control flow diagram, they verify and generate UIV test inputs. Halfond and Orso (2007) use static analysis of the server-side source code to extract input fields, and then generate test inputs based on the input fields. Benedikt et al. (2002) generate test inputs based on the analysis of PHP applications, monitors the application for crashes, and validates that the output conforms to the HTML specification. On one hand, since server-side code of a web application may be written in different languages, it is not trivial to automatically generate control flow diagrams based on static analysis. On the other hand, without test oracles, it is difficult to automatically determine test results for the server-side testing.

Black-box testing does not require the knowledge of the implementation of the software artifacts under test but generate test inputs based on the specified or expected functionality of the artifacts (Lucca and Fasolino, 2006). Lucca et al. (2002) exploit an object-oriented model of a web application, and propose approaches of unit testing and integration testing based on this model. Andrews et al. (2005) build hierarchies of Finite State Machines (FSMs) that model subsystems of web applications, and then generate test requirements as subsequences of states in the FSMs. Similar to the approach of Andrews et al., we use regular expressions, which are translated into FSMs during test-input generation, to define valid inputs, but we perturb regular expressions to generate UIV test inputs. Offutt et al. (2004) develop a strategy to create client-side tests that intentionally violate explicit and implicit checks on users' inputs, but they just describe their testing strategy and define specific rules and adequacy criteria for tests without providing an approach to generate UIV tests. Huang et al. (2003) combine dynamic analysis, fault injection, and behavior monitoring techniques to assess the security of a web application. Different from our approach, their approach focuses on SQL injection and XSS vulnerabilities. Wang et al. (2004) emphasize the importance of on-line testing of web applications, and UIV testing targeted by our approach can be a component of on-line testing.

A representative technique of gray-box testing of web applications is user-session-based testing (Lucca and Fasolino, 2006). Elbaum et al. (2003, 2005) present several techniques for using gathered user sessions to help test web applications. Their techniques can be applied either in a system's beta testing phase or during subsequent maintenance, but such techniques are not directly for UIV testing.

Our approach for identifying descriptive text for input fields is similar to approaches used by web macro recording tools such as Chickenfoot (Bolin et al., 2005) and RoboFox (Koesnandar et al., 2008). Different from these approaches, our approach for identifying descriptive text for input fields is based on ranking priorities of HTML tags around an input field. We analyze HTML tag sequences to identify which HTML tag contains the description text of an input field.

The basic idea of our LCTS algorithm is similar to the approaches of identifying duplicated web pages (Lucia et al., 2006; Sprenkle et al., 2005; Di Lucca et al., 2002), but our approach extracts the template from an HTML page, instead of just comparing their tag trees, i.e., our approach includes only presentation structures by removing context-related attributes and values in tags. In addition, we compared LCTS with traditional Diff algorithms to show the effectiveness of LCTS on evaluating structural similarity among HTML pages. On one hand, our approach is more simplified than the approaches based on edit distance (Di Lucca et al., 2002). On the other hand, our approach is more effective than traditional Diff algorithms.

There are some commercial or open source tools for web application security testing, such as the top-10 scanners (Top 10 Web Vulnerability Scanners, 2006). All of these vulnerability scanners have the following steps. First, scanners fetch pages by using a crawler with seed URLs, or fetch pages according to user actions by using a proxy. Second, scanners detect database-based security holes and common paths (such as where the admin page is). Third, scanners analyze a fetched page and conduct predefined injections. Fourth, scanners analyze response pages for test-result assessment. Except the second step, which requires a specific attack signature database, the others rely on the right inputs and predefined injection for vulnerability scanning. However, since there is no option for a user's control, these scanners can find limited types of defects, and cannot report those defects related to semantic invalid input value.

9. Conclusion

We have proposed an approach, called PIUIVT, to improve the effectiveness of UIV testing for web applications. Our approach combines the automation of vulnerability scanners of web applications and the flexibility of proxy-based UIV testing tools. We have proposed and studied six types of perturbation operators for UIV test-input generation. The empirical study shows that PIUIVT is more effective in terms of UIV vulnerability detection related to input types and values of web applications than existing approaches. We also proposed the LCTS solution to assess the similarity of HTML pages to detect problems in testing. The experimental results show that our LCTS solution is more practical for measuring of the similarity of HTML pages than existing approaches. On the other hand, our experimental results are limited to be from the single case study that we carried out and we plan to add more subjects in the future to enhance the validity of our experiment.

Acknowledgments

The work of the authors from Beihang University is sponsored by the National Natural Science Foundation of China (NSFC) (Major Research Plan) No. 90718018, "Research on the test-based software trusty growing models and its evaluation methods". The work of the authors from North Carolina State University is supported in part by NSF grants CNS-0720641, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, and an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

Acunetix Web Vulnerability Scanner, http://www.acunetix.com/ (2008).
Andrews, A., Offutt, J., Alexander, R., 2005. Testing web applications by modeling with fsms. Software Syst. Model. 4 (3), 326–345.
Appscan Suite for Web Application Security Testing, http://www.watchfire.com/products/appscan/default.aspx (2008).
K. Beaver, The importance of input validation, http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1214373,00.html (2006).
Benedikt, M., Freire, J., Godefroid, P., 2002. Veriweb: Automatically testing dynamic web sites. In: Proc. WWW.
Bolin, M., Webber, M., Rha, P., Wilson, T., Miller, R.C., 2005. Automation and customization of rendered web pages. In: Proc. UIST, pp. 163–172.
Burp proxy, http://www.portswigger.net/proxy/ (2009).
Di Lucca, G., Di Penta, M., Fasolino, A., 2002. An approach to identify duplicated web pages. In: Proc. CMPSAC, pp. 481–486.

dk.brics.automaton, http://www.brics.dk/automaton/index.html (2007).
Elbaum, S., Karre, S., Rothermel, G., 2003. Improving web application testing with user session data. In: Proc. ICSE, pp. 49–59.
Elbaum, S., Rothermel, G., Karre, S., Fisher II, M., 2005. Leveraging user-session data to support web application testing. IEEE Trans. Softw. Eng. 31 (3), 187–202.
Fiddler, http://www.fiddlertool.com/fiddler/ (2009).
Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T., Hypertext transfer protocol – http/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html (2007).
Google Hacking Database, http://johnny.ihackstuff.com/ghdb.php (2008).
HTML DOM Tutorial, http://www.w3schools.com/HTMLDOM/default.asp (2009).
Halfond, W.G.J., Orso, A., 2007. Improving test case generation for web applications using automated interface discovery. In: Proc. ESEC-FSE, pp. 145–154.
Hirschberg, D., 1975. A linear space algorithm for computing maximal common subsequences. Commun. ACM 18 (6), 341–343.
Howard, M., LeBlanc, D., 2003. Writing Secure Code. Microsoft Press, Redmond, Wash.
HTML Parser, http://htmlparser.sourceforge.net/ (2006).
Huang, Y., Huang, S., Lin, T., Tsai, C., 2003. Web application security assessment by fault injection and behavior monitoring. In: Proc. WWW, pp. 148–159.
Huang, Y.W., Yu, F., Hang, C., Tsai, C., Lee, D.T., Kuo, S., 2004. Securing web application code by static analysis and runtime protection. In: Proc. WWW, pp. 40–52.
Hunt, J.W., McIlroy, M.D., An Algorithm for Differential File Comparison, Technical Report SECLAB-05–04, Bell Laboratories (1976).
Koders, http://www.koders.com/ (2008).
Koesnandar, A., Elbaum, S., Rothermel, G., Hochstein, L., Scaffidi, C., Stolee, K.T., 2008. Using assertions to help end-user programmers create dependable web macros. In: Proc. SIGSOFT/FSE, pp. 124–134.
Liu, H., Tan, H.B.K., 2006. Automated verification and test case generation for input validation. In: Proc. AST, pp. 9–14.
Liu, C.-H., Kung, D.C., Hsia, P., Hsu, C.-T., 2000a. Object-based data flow testing of web applications. In: Proc. APAQS, pp. 7–16.
Liu, C.-H., Kung, D.C., Hsia, P., Hsu, C.-T., 2000b. Structural testing of web applications. In: Proc. ISSRE, pp. 84–96.
Lucca, G.A.D., Fasolino, A.R., 2006. Testing web-based applications: the state of the art and future trends. Inf. Softw. Technol. 48 (12), 1172–1186.
Lucca, G.D., Fasolino, A., Faralli, F., 2002. Testing web applications. In: Proc. ICSM, pp. 310–319.
Lucia, A.D., Scanniello, G., Tortora, G., 2006. Using a competitive clustering algorithm to comprehend web applications. In: Proc. WSE, pp. 33–40.
Maier, D., 1978. The complexity of some problems on subsequences and supersequences. J. ACM 25 (2), 322–336.
Mvnforum, http://www.mvnforum.com/mvnforumweb/index.jsp (2006).
Nikto2 release 2.02, http://www.cirt.net/code/nikto.shtml (2008).
NIST SAMATE Reference Dataset Project, http://samate.nist.gov/SRD/index.php (2007).
Offutt, J., Wu, Y., Du, X., Huang, H., 2004. Bypass testing of web applications. In: Proc. ISSRE, pp. 187–197.
Open Source Vulnerability Database, http://osvdb.org/ (2008).
Open Web Application Security Project, Top 10 2007. http://www.owasp.org/index.php/Top_10_2007.
Paros - for Web Application Security Assessment, http://www.parosproxy.org/index.shtml (2008).
Perturbation-based User-Input-Validation Testing of Web Applications, https://sites.google.com/site/asergrp/projects/PIUIVT (2010).
Regular Expression Library, http://regexlib.com/ (2007).
Ricca, F., Tonella, P., 2001. Analysis and testing of web applications. In: Proc. ICSE, pp. 25–34.
Sprenkle, S., Gibson, E., Sampath, S., Pollock, L., 2005. Automated replay and failure detection for web applications. In: Proc. ASE, pp. 253–262.
Tamperie, http://www.bayden.com/TamperIE/ (2009).
Tipton, H.F., Krause, M., 2006. Information Security Management Handbook, 6th ed. Auerbach Publications, New York.
Top 10 Web Vulnerability Scanners, http://sectools.org/web-scanners.html (2006).
Wang, Q., Quan, L., Ying, F., 2004. Online testing of web-based applications. In: Proc. COMPSAC, pp. 166–169.
C. Weber, Testing Your Web Applications for Cross Site Scripting Vulnerabilities, http://www.microsoft.com/technet/community/columns/secmvp/sv0505.mspx (2005).
Wikto: Web Server Assessment Tool, http://www.sensepost.com/research/wikto/ (2008).

Nuo Li is a researcher at ABB Corporate Research Center, P.R. China. She received her Ph.D. degree from Beihang University, Beijing, P.R. China, in 2008. She worked as a research assistant in the Department of Computer Science at North Carolina State University during 2007-2009. Her research interests include automated software engineering with emphasis on software testing and engineering process improvement.

Tao Xie is an Associate Professor in the Department of Computer Science at North Carolina State University. He received his Ph.D. in Computer Science from the University of Washington in 2005. His research interests are in software engineering, focusing on automated software testing and mining software engineering data.

Maozhong Jin is a Professor in Beihang University, Beijing, P.R. China. His research interest includes programming language processing and software engineering.

Chao Liu is a Professor, Associate Dean of School of Computer Science and Director of Software Engineering, Beihang University, Beijing, P.R. China. He is also vice director of Software Engineering Specialty Group of China Computer Federation, and managing director of Beijing Software Industry Association. He received his Ph.D. degree and M.S. degree in Computer Software and Theory at Beihang University, and his B.S. Degree in Mathmatics at Beijing University of Posts and Telecommunication. His research interests include software quality engineering, software testing, and software process improvement.