Test Case Multiplication for Web Applications

Farn Wang, Jun-Wei Lin, Ming-Hong Weng
Graduate Institute of Electrical Engineering
National Taiwan University, Taipei, Taiwan 10617, ROC
{farn, d01921014, r01921080}@ntu.edu.tw

ABSTRACT
This paper presents a novel approach of test case regeneration for web applications. The proposed technique automatically multiplies original test cases with 2 types of *multiplication operators: input value* and *behavior*, to improve the fault detection effectiveness of the original test suite. Different from previous works, our approach is black-box and needs neither information of server-side source code nor transition diagrams of the application's GUI states, and thus easily applicable. We implemented our approach and conducted experiments using 4 real world web applications. 22 faults in 4 applications were detected, and all of the faults were not revealed by the original test suites. Experimental results also show that the proposed technique achieves better effectiveness in terms of mutation score.

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

General Terms
Algorithms, Experimentation

Keywords
Test case multiplication, test case regeneration, web application testing

1. INTRODUCTION
Web applications are ubiquitous nowadays and close to our financial, healthcare and other critical infrastructure. Due to a crucial role they play and sensitive data they handle, bugs in web applications may result in broad-reaching and severe consequences. Testing is an essential and practical engineering activity to evaluate and improve the quality of software by revealing its faults. However, web applications may heavily interact with users, databases, and possibly with other web applications. To test the functional correctness of web applications in all such interactions can easily incur combinatorial explosion in the test suite. In addition, the web contents may also be dynamically generated based on users' local systems that bear severe vulnerabilities to attacks, such as SQL injection and cross-site scripting [25]. Consequently, testing web applications can be more challenging than testing traditional software. At the moment, testing web programs still mostly rely on the experience of test engineers in writing efficient and effective test cases that check some key execution scenarios. Unfortunately, test suites written by human engineers are costly and labor-intensive [3].

The work is partially supported by National Science Council (NSC), Taiwan, ROC and Industry Technology Research Institute (ITRI), Taiwan, ROC.

To reduce human effort in designing test cases, techniques of test case regeneration or augmentation [5, 22, 30] try to generate additional test data from pre-existing test data. For web applications, previous studies [2, 24] indicate that recombination and reordering of test cases can improve the effectiveness and coverage of the original test suite. However, most of previous techniques need server-side information such as session variables or database states of the application to perform the regeneration processes, which may limit their application. For example, in acceptance testing [18], testers or users may not have the application's source code. All that they have may be the documents capturing requirements, use cases or business rules, etc., to help them conduct testing to determine if the requirements are met. Therefore, regenerating new test data without server-side information for a more effective test suite is an interesting concern.

Table 1. A simplified test suite containing two test cases. A test case consists of a sequence of test steps, and each test step is composed of an action (operation or verification) and one or many arguments (elements' locators and require input data.)

Test Case	Test Step	Action	Argument 1	Argument 2	…
T1: Add Group	a_1	Go To	path='/admin.php'		
	a_2	Click Link	link='Groups'		
	a_3	Click Button	value='Add Group'		
	a_4	Input Text	id='groupname'	"group1"	
	a_5	Click Button	value='Add'		
	a_6	Verify	text='group1 is added'		
T2: Add User	a_7	Go To	path='/admin.php'		
	a_8	Click Link	link='Users'		
	a_9	Click Button	value='Add User'		
	a_{10}	Input Text	id='username'	"user1"	
	a_{11}	Input Text	id='password'	"abc123"	
	a_{12}	Input Text	id='confirmpw'	"abc123"	
	a_{13}	Select From List	id='group'	value="group1"	
	a_{14}	Click Button	value='Add'		
	a_{15}	Verify	text='user1 is added'		

In web application development, functional testing or acceptance testing is usually conducted to make sure that user requirements are satisfied. The test cases are usually recorded or written as runnable test cases supported by test automation frameworks such as Selenium [23] or Robotframework [21]. Table 1 illustrates part of a simplified executable test case used in our experiment. The test suite in Table 1 contains two test cases. Generally, each test case consists of a sequence of test

steps. Each test step is an action with one or many arguments. An action is an operation on, or verification of the application's Graphical User Interface (GUI). Arguments are GUI elements' locators and required input data. For example, step a_1 in test case T1 in Table 1 goes to a web page named '*admin.php*', and step a_2 clicks a link with text '*Groups.*' Step a_3 clicks a button with the value '*Add Group*', and step a_4 types '*group1*' in the input field identified by '*groupname.*' Finally, step t_6 makes sure that the current page contains text '*group1 is added*', otherwise fails. Similarly, the second test case, *Add User*, in Table 1, adds a new user *user1* belonging to *group1*, and confirms that the user is added by examining the existence of text '*user1 is added*.'

This paper presents a novel approach called test case multiplication to regenerate effective test cases from pre-existing test cases. The proposed method takes only original test cases as input, and modifies one or more test steps of them to generate new test cases. We introduce 2 types of modification. The first one is called *multiplication of input value*, which removes or alters input values from test steps. This multiplication is partially inspired by traditional black-box testing techniques such as boundary value analysis [18] and previous work in web-service testing [19], but applied in a different context.

We also introduce a novel type of modification: *multiplication of behavior*. Actions of one or more test steps will be added or replaced to simulate user behaviors different from ones in the original test cases, such as double click, operating delay, script repetition, and other divergent behaviors. To our knowledge, the proposed approach is the first approach performing test case regeneration for web applications without server-side information, which makes the approach labor-saving and easily applicable.

Test cases supported by test automation frameworks can be used not only in automatic regression testing, but also in stress testing [10]. Stress testing is usually conducted to evaluate a critical system at or beyond the limits of its specified requirements. A common way of performing stress testing to a web site is simultaneously executing the same test cases from multiple clients to evaluate an application's robustness, availability, and reliability under extreme conditions. However, if we can introduce more variances into test cases used in different clients when performing stress testing, more unpredictable usage patterns and sequences of requests to a web application can be generated. It would be beneficial to stress testing which focuses on how an application reacts to huge workload, random events, chaos and unpredictability. In Section 5.2 we will show that the technique proposed in this paper can also be used to generate test cases for stress testing from pre-existing test cases.

To evaluate the effectiveness of the proposed approach, we implemented our technique and conducted experiments with 4 open source applications. 22 faults in these 4 applications were found, and all of the faults were not revealed by the original test suites. More amazingly, to our knowledge, all these 22 faults are newly discovered and have never been reported before. (Please check section 5.3 for description of the faults.) Considering that some of the applications have huge download counts, the revealing of those faults show the promise of our approach in the industry projects.

We also conducted another experiment by seeding faults into one of the above applications, and investigated how many extra mutants can be killed by the multiplied test cases. The results show that the multiplied test cases totally killed 3 times extra mutants comparing with the original test cases.

This paper makes the following contributions:

1. The first black-box approach performing automatic test case regeneration for web applications, which makes it easily applicable and useful especially when server-side information is unknown to testers.
2. The introduction of a novel test case multiplication approach: *multiplication of behavior*, which adds or replaces actions in one or many test cases to simulate variances in user behaviors.
3. The implementation and an empirical evaluation of the proposed approach. Experiments with 4 real world web applications confirm the effectiveness of our approach. The evaluation shows that our approach can help detect faults which are not revealed by the original test suites.

The remainder of this paper is organized as follows. In Section 2, we use examples to explain the ideas of our approach. The algorithm and multiplication operators of the proposed approach are described in Section 3. Section 4 discusses our implementation of the approach. Section 5 presents our experimental evaluation on open source web applications. Related work and conclusions are given in Section 6 and Section 7, respectively.

2. RUNNING EXAMPLE

Our goal is to relieve the test engineers of the burden of writing test cases to web programs. We propose to use algorithms to automatically regenerate test cases out of pre-existing test cases written by test engineers. The natural innovation of test case regeneration [5, 22, 30] introduces perturbation to the input test cases and constructs test cases that are a little different from those input ones. There are many such proposals that modify data input in test cases for new test cases. In the following, we explain how we are inspired from various sources in designing our multiplication operators. We cluster our multiplication operators in two groups. The first group is called input-value multiplication and involves subsection 2.1, 2.2, and 2.3. The second group is called behavior multiplication and is discussed in subsection 2.4, 2.5, and 2.6.

2.1 Simple Changes to Input Values

In the literature [3], there are suggestions for input values that are likely to reveal typical program bugs. Empirically, values near the boundary of the input value ranges are likely to expose bugs. For example, NULL strings for text input may trigger bugs when a program interface module does not carefully check for simple anomalies in the user input. Similarly, very long strings to text input, random punctuations to text input, negative numbers to integer input, extreme large numbers to integer input, and etc., may also trigger bugs when the program interface module is not carefully designed. In this work, we have adopted several multiplication operators that change data input in test cases to such boundary values. For instance, for the test case T1

in Table 1, we have a multiplication operator that replaces the input value in step a_4 with a blank.

2.2 Input Value Changes for Breaking HTML Documents

Another source of inspiration is the special operating domain of web programs. The user-interface of web programs are in HTML format and runs on a stateless protocol (i.e. HTTP). Data input to web applications may be processed and used for the construction of the next web-page. As a result, text input with specific patterns may later be mistakenly interpreted as HTML tags in the following web page. One suggestions from the literature is to insert string "><" to text input to check whether the web application would interpret the string as the boundary of two tags since HTML documents use "<" and ">" to respectively start and end tags. In this work, we also adopt multiplication operators of this type. For instance, we have implemented a multiplication method that can add "><" (special HTML characters) into input value in step a_{12}. The varied test case then serves to examine if the application somewhat implements input validation.

2.3 Input Value Changes as Security Attacks

Security attacks to web applications are also very common these days. In fact, security testing is itself a hot research area, and also raises specific concerns for testing web applications. The common patterns adopted in security test cases can also be useful in introducing perturbations to input test cases in this work. We adopted one such technique, i.e., *SQL* injection. SQL is the acronym for *Structured Query Language* and is a standard language for processing databases underlying the servers-side of many web programs. For example, if the text input in an original test case involves the following SQL query:

```
SELECT username FROM adminusers where
username='admin' AND password='admin'
```

We may design a multiplication operator that changes the query to the following:

```
SELECT username FROM adminusers where
username='admin' AND password='' OR
'1'='1'
```

The original SQL only selects the record with correct combination of username and password. The varied SQL selects all usernames from the table *adminusers*. Such a change may then create a test case that tests with unauthorized database accesses.

2.4 Action Changes as Environment Perturbations

Web applications interact with human users, server computation via network, and database responses via query languages. Such interaction can vary due to users' spontaneous behavior, network delay, communication failures, and database processing workload. For instance, unintentional delay from a user (e.g. looking for a credit card while purchasing online) may cause session timeout which drops all the information stored in the session. Another example happens when part of the application's GUI (e.g. a button) ought to be idempotent, that is, pressing the button multiple times should have the same effect as pressing it once. It would be difficult for test engineers to systematically enumerate all such variances in their construction of test suites. In comparison, by writing methods that mimic the variances exemplified in the above, we can mechanically regenerate test cases with those variances.

In this work, we introduce two multiplication operators in this aspect. The first is random insertion of sleep action to a test case. Such a variance may then mimic the prolonged response time due to users' absence, network delays, and etc.

The second is a change of single-click action to double-click action. Such a change may test for difference in test execution result when double-click is actioned. Specifically, we have implemented a method that duplicates the click action with value '*Add*' once more in step a_5 in Table 1 for a new test case that checks how the application deals with the two consecutive and identical requests.

2.5 Scenario Change as Cycle Repetition

The techniques mentioned in subsection 2.1 to 2.4 works on single test steps and may not reveal tricky bugs that can only take effects when specific sequential patterns are observed in test execution. When a web application has certain memory or can count event occurrences, repetitive execution of certain test case segments are then necessary for bugs related to the memory or counting mechanism. For example, the response from a web application when a user account is first added to its database may not be the same when the same account is later added to the database again. But in general, it can be difficult to know how the application should respond without knowledge of the functional requirement or server-side information of the application. As we have said, we are interested at developing black-box testing techniques without prior server-side information. We thus propose the technique to repeat cycles in test cases. Consider the graphical representation of a part of a test case in Figure 1.

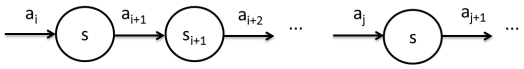

Figure 1. The same webpage *s* is reached twice after test step a_i and a_j are executed, respectively

Here the test case actions are labeled on the arcs in sequence. The ovals to and from arcs represent the webpages between actions. The webpage after step a_i is identical to the page after a_j. Conceptually, we may want to check whether test steps from a_{i+1} to a_j can be repeated several times. This can be visualized as the following cycle structure at page *s*.

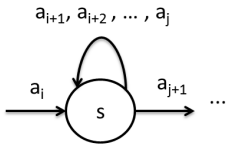

Figure 2. The sequence of test steps a_{i+1} to a_j in Figure 1 can be regarded as a loop at webpage *s*.

For instance, if we find that the webpages after step a_2 and step a_5 are similar for test case T1 in Table 1, we may repeat steps a_3, a_4, and a_5 once more to see what would happen. We

have implemented such a multiplication operator that automatically identifies repeatable parts of a test case and repeats that part in the test case once more. Indeed such a multiplication operator helped us in revealing a severe fault of a web application, Gallery 3, in our experiments. As Figure 3 illustrates, the fault directly throws database exception to the administrator. The bug may imply that our techniques can really be useful in filing valuable bug reports.

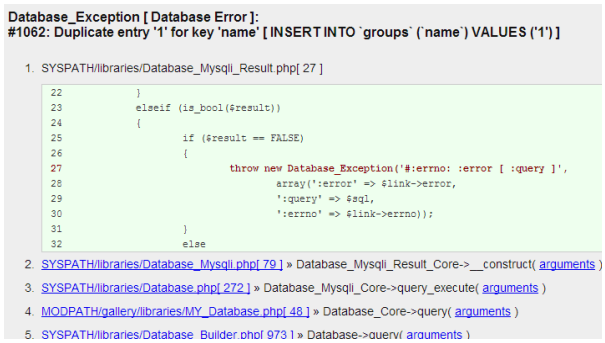

Figure 3. A fault found by the proposed technique which repeating part of a test case in one of our experiments. The fault directly throws database exception to the administrator.

To implement the cycle repetition operator, one technical challenge is the identification of identical (or similar) webpages that bound a repeatable segment of a test case. For this purpose, we execute all input test cases and record the intermediate webpages (as HTML documents) in the test execution. Then we compare the HTML documents of the intermediate webpages. In our implementation, we only check for equivalence of webpages in the syntax level. Specifically, we first `sanitize' all webpage documents by removing their HTML tags which are of minor importance for recognizing their contents, such as the "<head>" tag and all tags enclosed by it. For the rest, we then remove all attributes of them and leave the tags themselves and their contents. For example, a pair of tags " paragraph1 " will become " paragraph1 " after sanitizing, and a webpage and its sanitized text string are shown in Figure 4.

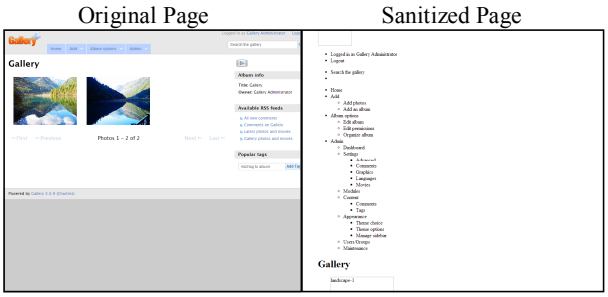

Figure 4. An example of a web page before and after sanitizing.

Two webpages are considered equivalent if their sanitized documents are the same text string. Such an implementation allows for quick webpage equivalence checking. In the future, other implementation with higher precision for webpage identification can be experimented.

2.6 Crossover from Genetic Programming

In fact, our multiplication operators can be discussed in the broad framework of genetic programming [12, 15] that mimics the evolution principle to find good solutions to difficult problems. In finding solutions for a problem, a genetic program maintains a population of solutions. Then it evaluates the gain (or performance) of the solutions with a given object function. Then in a generation, the program keeps a certain percentage of the solutions with the best object values. The kept solutions are survivors of the generation. Then the program applies multiplication operators to the survivors to generate the next and new population. In this way, after several generations, usually a good solution will emerge in the final population.

There are two types of multiplication operators related to genetic programming. The first is called *mutation* which resembles the ideas of our multiplication operators discussed in subsection 2.1 through 2.4.

The second is called *crossover* that views the solutions as sequences, identifies similar elements in the respective sequences of two solutions, and switches the respective tails of two sequences for the sequences of two new solutions. To illustrate the crossover operator, we have two input test cases on the top of Figure 5 with the same webpage s respectively after test step a_i and b_j. Then we can switch the two tails after step a_i and b_j in the two test cases and obtain the two new test cases in the bottom of Figure 5. The operator also needs collecting all the intermediate webpages in the execution of all input test cases. Then the identification of intermediate webpages is carried out before we switch the tails for generating new test cases.

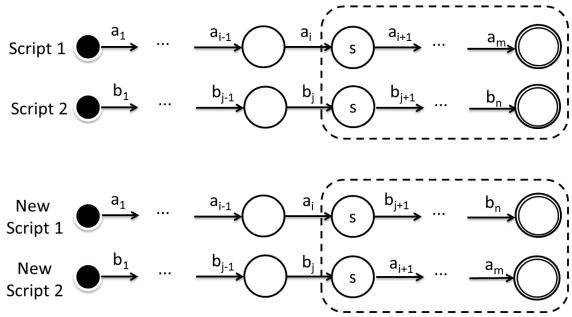

Figure 5. How the *CROSSOVER* operator works: Create new test cases by exchanging the *tails* of two cases.

Genetic programming has been also used to invent the important technology of *mutation testing* [9, 14] which is a fault-based test case generation technique which modifies the program to create multiple faulty versions called *mutants*. The test cases in mutation testing is designed to fail the mutants (i.e., to kill the mutants). A test suite killing more mutants is considered a more effective suite because it is likely to reveal more real faults. We also used mutation testing in one of our experiments.

In mutation testing, the rules used for modifications are known as mutation operators. In this research, the concept of modifying syntactic objects is adopted and applied to test cases instead of the source code which in genetic programming, traditional mutation operators are applied to. Therefore, new types of

operators called *multiplication operators* have been defined in this work.

We can also view our cycle repetition operator as a special case of the crossover operator. That is, our cycle repetition operator can be viewed as the special crossover operator applied to the same test case. For instance, in a test case $a_1a_2...a_n$, if there exist $i<j$ such that the webpage after action a_i and a_j are identical, then we can apply the crossover operator to two copies of the test case on position i and j. Then the outcome of the operation is two new test cases, one with segment $a_i...a_{j-1}$ deleted and one with $a_i...a_{j-1}$ repeated once more. However, in our implementation, we decide to keep both the crossover operator and cycle-repetition operator. The reason is that cycle-repetition has special meaning in software design and needs special testing effort to check that the cycle repetition is implemented correctly. Moreover, in the future, when an input test suite is large, we may have to select some of the multiplied test cases for test execution. Then separating cycle-repetition test cases from crossover test cases can help us in allocating appropriate testing budget for revealing bugs involving loop execution and counting mechanism.

3. MULTIPLICATION ALGORITHMS

In this section, we present the algorithms for all our multiplication operators. Without loss of generality, a test suite is considered to be a set of test cases, and a test case can be regarded as a sequence of test steps. Specifically, the length of a test case

$$T = a_1\ a_2\ ...\ a_n$$

is denoted $|T| = n$. T is mathematically a mapping from $[1, |T|]$ to actions. For example, $T(1) = a_1$ and $T(3) = a_3$.

For convenience, given j and k with $1 \le j < k \le |T|$, we let $T[j, k]$ be $T(j)\ldots T(k)$. Also given two sequences T_1 and T_2, we use T_1T_2 to denote the concatenation of T_1 and T_2.

We seek to regenerate a test suite out of an existing one. The multiplication operators are then systematically applied to the existing test cases for new test cases to improve fault detection. The existing test cases either can be manually generated or can also come from any existing automatic approach [1, 6, 13, 20, 27].

As said previously, we define two types of multiplication: input value and behavior. Multiplication of input value changes the input values of test steps in a test case to generate new test cases. Multiplication of behavior changes or repeats test steps in a test case, or swap steps in two test cases.

3.1 Multiplication of Input Value

Empirical study indicates that test cases examining boundaries have a higher payoff than test cases not [18]. To perform boundary value analysis, the following two multiplication operators are defined:

EMPTY_INPUT (test case T, int k)

Here k is an integer in $[1, |T|]$ for a test step index of T. Specifically, the action of $T(k)$ must be "Input Text." Then the output of the operator is a test case T' that is identical to T except that $T'(k)$ is basically $T(k)$ with input text argument changed to the blank string.

LARGE_INPUT (test case T, int k)

Here k is an integer in $[1, |T|]$ for a test step index of T. Specifically, the action of $T(k)$ must be "Input Text." The output of the operator is a test case T' identical to T except that $T'(k)$ is basically $T(k)$ with input text argument changed to a random string with a size of 1,024 characters.

Cross-site scripting and SQL Injection are two of the most prominent vulnerabilities for web applications [25]. Cross-site scripting is exploited through malicious inputs containing HTML contents with client-side scripts, and these inputs are interpreted by browsers while rendering web pages. SQL injection is the execution of malicious database statements from the user interface to the database. Many test input generation techniques for exposing these two classes of vulnerabilities have been proposed in the literature [16, 27, 28]. In this paper, we define the following two multiplication operators for basic security testing of the application:

MALFORMED_STRING (test case T, int k)

Here k is an integer in $[1, |T|]$ for a test step index of T. Specifically, the action of $T(k)$ must be "Input Text." The output of the operator is a test case T' identical to T except that $T'(k)$ is basically $T(k)$ with "><" randomly inserted to the input text argument. Since angle brackets are parts of HTML tags, if an application does not properly handle such them, the application may generate invalid HTML documents based on user input texts.

SQL_INJECTION (test case T, int k)

Here k is an integer in $[1, |T|]$ for a test step index of T. Specifically, the action of $T(k)$ must be "Input Text." The output of the operator is a test case T' identical to T except that $T'(k)$ is basically $T(k)$ with the string "' OR '1' = '1" by which the input value is replaced. If the web application uses this input text to construct queries in SQL, unauthorized database accesses may happen. Thus the *SQL_INJECTION* operator can help us in testing the protection of the application against attacks.

3.2 Multiplication of Behavior

Different from multiplication of input value, multiplication of behavior changes actions in test cases. We have implemented the following behavior multiplication operators.

DOUBLE_CLICK (test case T, int k)

Here k is an integer in $[1, |T|]$ for a test step index of T. Specifically, the action of $T(k)$ must be "Click Button." Then the output of the operator is a test case T' that is identical to T except that the action of $T'(k)$ is changed to a "Double Click Button." This operator can test if elements clicked in a test case are idempotent.

SLEEP (test case T, int k)

Here k is an integer in $[1, |T|-1]$ for a test step index of T. The output of the operator is the following test case:

$$T(1)\ldots T(k)\ sleep(d)\ T(k+1)\ldots T(|T|)$$

Here *sleep(d)* is an action that waits for d time units. Currently, we set d a random number between 1 and 10 seconds. This operator inserts random waiting time after test step k.

CYCLE_REPETITION (test case T, int j, int k, pages S)

Here j and k are two integers in [1, $|T|$] with $j < k$. S is a sequence of webpages: s_1, ..., $s_{|T|}$ in HTML format. For each h in [1, $|T|$], $S(h)=s_h$ is the page after test step $T(h)$. Similarly, we can use $S(h)$ to denote s_h.

The operator generates a test case identical to T but with $T(j+1)…T(k)$ repeated one more time if $S(j)$ and $S(k)$ are the same. Specifically, its algorithm consists of the following steps:

1. **If** *sanitized(S(j)) ≠ sanitized(S(k))*, return nothing.
2. **Else return** $T(1)…T(k)\ T(j+1)…T(k)\ T(k+1)…T(|T|)$.

Intuitively, this operator repeats the sequence of test steps between two equivalent webpages in a test case. However, an adjustment to its implementation is necessary to avoid generating a large number of test cases. Each generated test case repeats steps between two webpages in the group. By definition, if there is a group of m equivalent webpages after sanitizing, the *CYCLE_REPETITION* operator should generate $C_2^m = m(m-1)/2$ new test cases. As we can see, the combinatorial numbers can blow up the test suite sizes. For example, webpages $S(1)$, $S(2)$, $S(4)$, and $S(5)$ are identical. Then we may use the operator to generate the following six test cases:

- $T(1)\ T(2)\ T(2)\ T(3)…T(|T|)$ (for $j=1$, $k=2$)
- $T(1)…T(4)\ T(2)…T(4)\ T(5)…T(|T|)$ (for $j=1$, $k=4$)
- $T(1)…T(5)\ T(2)…T(5)\ T(6)…T(|T|)$ (for $j=1$, $k=5$)
- $T(1)…T(4)\ T(3)\ T(4)\ T(5)…T(|T|)$ (for $j=2$, $k=4$)
- $T(1)…T(5)\ T(3)…T(5)\ T(6)…T(|T|)$ (for $j=2$, $k=5$)
- $T(1)…T(5)\ T(5)\ T(6)…T(|T|)$ (for $j=4$, $k=5$)

We can find the test case for $j=1$ and $k=4$ repeats a shorter sequence of $T(2)…T(4)$ and that for $j=1$ and $k=5$ repeats a longer sequence of $T(2)…T(5)$. In our work, we eliminate the former test case because we think the faults revealed by a shorter sequence are probably revealed by a longer one. Therefore, we regard consecutive webpage segments in a test case as an abstract state when applying the repetition operator. Specifically, we inductively define the concept of identity segment as follows. Given a test case T and webpage sequence S such that $|T|=|S|$. Two position indices j and k in [1,$|T|$] with $j<k$ are in the same identity segment with T and S, in symbols idseg(T,S,j,k), if one of the following two cases hold.

- $j+1=k$, and $S(j)=S(k)$.
- $j+1<k$, idseg($T,S,j,j+1$) is true, and idseg($T,S,j+1,k$) is true.

For the just-mentioned example, there are two identity segments {1,2} and {4,5}. For convenience, we let idseg(T,S,j,j) also be true. The operator will then only be applied to parameter j and k chosen from one representative of each identity segment. As a result, in our implementation, the following reduced test suite will be generated from this operator.

- $T(1)\ T(2)\ T(2)\ T(3)…T(|T|)$ (for $j=1$, $k=2$)
- $T(1)…T(5)\ T(2)…T(5)\ T(6)…T(|T|)$ (for $j=1$, $k=5$)
- $T(1)…T(5)\ T(5)\ T(6)…T(|T|)$ (for $j=4$, $k=5$)

Also, the algorithm is revised as follows:

1. **If** *sanitized(S(j)) ≠ sanitized(S(k))*, return nothing.
2. **Elseif** *sanitized*($S(j)$) ≠ *sanitized*($S(j$-1))
   ∧ *sanitized*(S(k)) ≠ *sanitized*($S(k$+1)),
   then return $T(1)…T(k)\ T(j+1)…T(k)\ T(k+1)…T(|T|)$.

CROSSOVER (test case T1, test case T2 int j, int k, pages S1, pages S2)

Here j and k are two integers in [1, $|T1|$] and [1, $|T2|$], respectively. $S1$ is a sequence of webpages in HTML format. For each h in [1, $|T1|$], s_h is the page after test step $T1(h)$. We also use $S1(h)$ to denote s_h. $S2$ is a sequence of webpages with respect to $T2$, and $S2(h)$ is used to denote the page after test step $T2(h)$.

The operator generates two test cases in which the tails of each other are switched. Similar to the *CYCLE_REPETITION* operator, we also regard consecutive webpages belonging to a test case as an abstract state when the *CROSSOVER* operator performs. Specifically, its algorithm is as follows:

1. **If** *sanitized*($S1(j)$) ≠ *sanitized*($S2(k)$), return nothing.
2. **Elseif** *sanitized*($S1(j)$) ≠ *sanitized*($S1(j$-1))
   ∧ *sanitized*($S2$(k)) ≠ *sanitized*($S2(k$-1)),
   then return $T1(1)…T1(j)\ T2(k+1)…T2(|T2|)$, and
   $T2(1)…T2(k)\ T1(j+1)…T1(|T1|)$

3.3 The Algorithm

```
Algorithm multiplyTestSuite(test suite K)
// K is a set of test cases.
1.  Let K' = ∅.
2.  Foreach T ∈ K and multiply operator OP, do {
3.    Switch (OP) {
4.    Case CYCLE_REPETITION:
5.      Foreach test case T in K, {
6.        Let S = getWebpages(T).
7.        Foreach j ∈ [1,|T|-1] and k ∈ [j+1,|T|],
8.          Let K' = K' ∪ {CYCLE_REPETITION(T,j,k,S)}
9.      }
10.   Case CROSSOVER:
11.     Foreach test case T1 and T2 in K, do {
12.       Let S1=getWebpages(T1) and S2=getWebpages(T2).
13.       Foreach j ∈ [1,|T1|] and k ∈ [1,|T2|],
14.         Let K' = K' ∪ {CROSSOVER(T1,T2,j,k,S1,S2)}
15.     }
16.   Default:
17.     Foreach k ∈ [1,|T|], let K'=K' ∪ {OP(T,k)}
18.   }
19.   Return K'
20. }
```

Figure 6. The algorithm for generating new test cases with the proposed multiplication operators

Algorithm in Figure 6 describes how we use the multiplication operators in subsection 3.1 and 3.2 to construct a test suite out of an original test suite. At line 6 and 12, we use a procedure getWebpages() to get the page sequence from the execution of a test case. Due to page-limit, we shall skip its algorithm.

The main loop at line 2 multiplies test cases for every input test case and multiplication operator. Inside the main loop, at line 17, for a multiplication operator that does not need page equivalence information (i.e. all operators except for the *CYCLE_REPETITION* and *CROSSOVER*), we apply the operator to each test step in the test case and then the action or the corresponding input value will be replaced by the operator to generate new test cases.

For the *CYCLE_REPETITION* operator, we first get the list of webpages at line 6 and the test steps between every two equivalent pages will be repeated to generate new test cases at line 7-8.

Finally, for the *CROSSOVER* operator, we first call getWebpage() twice to get the page lists respectively for the two test cases *T1* and *T2* at line 12. Then at line 13-14, we regenerate two test cases for every pair of a test step of *T1* and a test step of *T2* by switching the tail step sequences of *T1* and *T2*, if possible.

The number of regenerated test cases by the operators except for the *CYCLE_REPETITION* and the *CROSSOVER* depends on corresponding number of actions in the original test cases. For example, with the test cases in Table 1, four new test cases are generated by *EMPTY_INPUT*. This also holds for other operators of type multiplication of input value. In the first regenerated test case, value of argument *group1* at test step a_4 is removed. Value of *user1* at step a_{10}, value of *abc123* at step a_{11}, and value of *abc123* at step a_{12} are removed in the second, third and the fourth regenerated test cases, respectively.

For the *CYCLE_REPETITION* operator, if there are *m* identity segments of a group of equivalent webpages, then there are

$$C_2^m + m = \frac{m(m-1)}{2} + m$$

regenerated test cases.

For the *CROSSOVER* operator, suppose there are *m* identity segments in *T1* and *n* identity segments in *T2*. Then the operator regenerate at most 2*mn* test cases, because each crossover exchanges the tails of two test cases with respect to one identity segment from T1 and one from T2 and regenerates two new test cases.

3.4 Test Oracles

A *test oracle* gives feedback, either *pass* or *fail*, during test execution by comparing the expected results with the actual results [7]. In testing web applications, the verification steps included in test cases are used as oracles, like test steps a_6 and a_{15} in Table 1. Although the verification steps may be duplicated or exchanged after multiplication (e.g., by the *CYCLE_REPETITION* or *CROSSOVER* operators), each of the oracles remains unchanged. That is, the expected test results written in the verification steps are the same after multiplication. For test cases multiplied by the *DOUBLE CLICK* and *SLEEP* operators, the included oracles work well, so the test reports of these test cases can be automatically interpreted. However, for test cases multiplied by other operators, the oracles may become inappropriate. For example, emptying the input value of step a_4 in the test case T1 of Table 1 is expected to cause a fail of adding a group after clicking the '*Add*' button at step a_5. So a new oracle that verifies if input validation is performed would be more appropriate than the original oracle in step a_6. In this paper, we do not deal with the test oracle problem, and manually examine all test reports after executing the multiplied test cases to determine if faults are exposed. The investigation of handling oracles after multiplication is left to future work.

4. IMPLEMENTATION

Our work is partially sponsored by a project for *Testing as a Service* (*TaaS*) on *Cloud OS* developed by *Industrial Technology Research Institute* (*ITRI*), Taiwan [8]. The ITRI Cloud OS is an end-to-end cloud data center management solution designed for Amazon Web Service-like *Iaa*S (*Infrastructure-as-a-Service*). Based on the Cloud OS, servers for deploying the software under test (SUT) and running the test cases can be requested on demand. Test cases can be run separately and simultaneously towards servers on which the copies of the SUT are deployed.

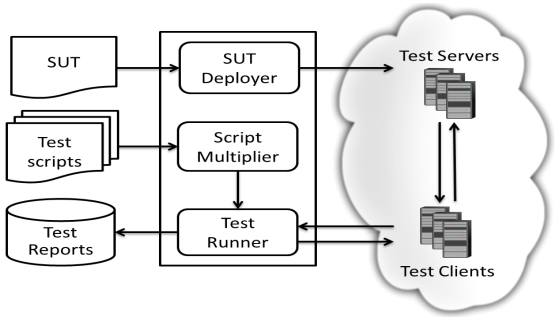

Figure 7. The architecture of the tool which implements our approach. The SUT and multiplied test cases are deployed on test servers and test clients, respectively. Test reports are collected from test clients by the test runner.

Figure 7 describes the architecture of our implementation in this project. Although the proposed approach completely works without the information of the SUT, our implementation includes SUT deployment for two reasons. First, in the experiments, frequently redeploying the application to restore its initial configuration or database is necessary. Second, execution of test cases is sometimes time-consuming, but it can be accelerated by deploying copies of the SUT and running the test cases individually towards each copy.

With the original test cases and the compressed file of the SUT provided by the user, the Script Multiplier implements the algorithm and the multiplication operators described in Section 3 to regenerate new test cases. The multiplied test cases are assigned to Test Clients by the Test Runner, and then Test Clients run the assigned test cases on Test Servers on which the SUT are deployed by the SUT Deployer. Before execution of each test case, the SUT Deployer resets configurations of the SUT. Finally the test reports are retrieved from Test Clients by the Test Runner.

The SUT Deployer, Script Multiplier and Test Runner are implemented in Java. In our implementation, we focus on SUT running under *LAMP* (acronyms referring to Linux, Apache, MySQL and PHP) environment and test cases supported by Robotframework [21], which, like LAMP, is open-source and has

active user communities. However, our approach can be easily applied to other types of web applications and test cases written in other formats such as Selenium [23].

5. EVALUATION

To investigate the fault detection effectiveness of the proposed approach, we conducted two experiments using four real world PHP web applications to answer the following research questions:

Q1. How much can our approach improve the fault finding ability of an original test suite?

Q2. Are some multiplication operators more effective in finding faults than the others?

In the first experiment, we generated test cases which covered and tested the applications' main functions without revealing faults. The test cases were then multiplied with the proposed approach and executed for detecting new real faults. In the second experiment, we seeded faults in one of the applications, and executed the corresponding original and multiplied test cases used in the first experiment to compare the numbers of mutants killed by them.

5.1 Subject Programs

We selected the following four open source PHP applications to perform the evaluation:

- **Complaint Management System**: a web-based program used to manage customers' complaints. There are 48 PHP files and 2538 lines of executable code in total. The application has been downloaded more than 551 times in Sourceforge.
- **Ultra Light Forum (1.1):** a web-based program used to write and reply topics. There are 48 PHP files and 3259 lines of executable code. It has been downloaded more than 963 times in Sourceforge.
- **SchooleMate (1.5.4)**: a PHP/MySQL solution for administration of elementary, middle and high schools. There are 63 PHP files and 4927 lines of executable code. It has been downloaded more than 16171 times in Sourceforge.
- **Gallery (3.0.9)**: a web-based photo album organizer. This is a popular application with 505 PHP files and 15320 lines of executable code. It has enjoyed more than 12844736 downloads in Sourceforge.

Among them, the *SchoolMate* program has been used by other research on web application testing [2, 6], and the *Gallery* program is mature and active in Sourceforge as witnessed by more than twelve million downloads.

5.2 Experimental Setup

The original test cases used in the experiments were prepared based on the user's manual of the applications. For each main function and scenario in the manual, we prepared test cases and organized them into test scripts as required by ITRI. We made sure that all main functionality or scenarios in the manual are covered so that the test cases are representative enough for functional or acceptance testing. All the original test cases were executed successfully and did not reveal faults. The original test cases are described in Table 2.

For each application, we multiplied the original two test scripts with the proposed approach. The number of test cases multiplied by each multiplication operator is reported in Table 3. Table 3 shows that for each application, hundreds of test cases are regenerated from the original two test scripts. For example, for *Gallery*, 17 new test cases are respectively regenerated by the operators concerning input values, because there are totally 17 "Input Text" actions in the original two test cases. Similarly, the number of test cases generated by the *SLEEP* operator is the number of test steps in the original test cases.

Table 2. The original test scripts for each web application used in this study. The #Test Cases column lists the number of test cases contained in the script. The Covered Functionality column lists the main functions the script covers.

App Name		#Test Cases	Covered Functionality
Complaint Management System	Script1	4	Submit complaints
	Script2	7	Close complaints
Ultra Light Forum	Script1	4	Create topics
	Script2	4	Replay topics
SchoolMate	Script1	13	Administration
	Script2	6	Teachers and students
Gallery	Script1	10	Administration
	Script2	6	Albums and photos

Table 3. The number of multiplied test cases for each application. CMS: Complaint Management System. ULF: Ultra Light Forum. SM: SchoolMate. GAL: Gallery.

Multiplication Operator	# Multiplied Test Cases			
	CMS	ULF	SM	GAL
EMPTY_INPUT	18	18	55	17
LARGE_INPUT	18	18	55	17
MALFORMED_STRING	18	18	55	17
SQL_INJECTION	18	18	55	17
DOUBLE_CLICK	9	8	25	12
SLEEP	65	63	202	135
CYCLE_REPETITION	17	16	40	28
CROSSOVER	18	20	32	28
Total	181	179	519	271

Although the total size of regenerated test cases seems large, it would not be a problem if the test evaluation can be automated. It should be noticed that the test oracles in the test cases multiplied by the *DOUBLE_CLICK* and *SLEEP* operators are fully qualified. That is, regardless of the number of these test cases, the test reports which return *pass* can be automatically filtered because they do not reveal faults. The advantage of using multiplied test cases in stress testing will be investigated in other research.

To conduct the first experiment, we tested each application with its corresponding multiplied test cases, and evaluated if there are faults revealed by the test cases. We filtered out all the test reports returning *pass* to reduce the effort of test evaluation. Indeed, we may omit false negatives, i.e., tests are marked as passed even there are bugs in reality. But at present we focus on test reports returning *fail* and leave the false negatives problem to future work.

In the second experiment, we created 15 faulty versions (mutants) of the *Gallery* program by applying the mutation

operators shown in Table 4 to six files of the program. The 15 mutants were then tested by the original two test scripts and the multiplied test cases used in the first experiment to determine how many mutants can be killed by these two groups of test cases.

Table 4. The mutation operators used to create faulty versions of the Gallery program. The #Applied column lists the number of faulty versions created by applying the operator.

Operator	Description	#Applied
BooleanTrue	Replace boolean TRUE with FALSE	2
BooleanFalse	Replace boolean FALSE with TRUE	2
LogicalIf	Wrap a conditional with a !	4
BooleanOr	Replace logical operator OR with AND	2
ScalarInteger	Replace a integer with a random integer	2
ScalarString	Replace a string with a random string	3
	Total	15

5.3 Results

Table 5 shows the number of faults revealed by the multiplied test cases for each application. Totally, 22 faults in four applications were detected. For *Complaint Management System*, a fault was detected by test cases generated by the *CYCLE_REPETITION* operator, and the failure occurred when adding the same account twice. For *Ultra Light Forum*, a fault related to input validation was revealed by test cases multiplied by the *LARGE_INPUT*, *MALFORMED_STRING,* and *SQL_INJECTION* operators, respectively. Test cases generated by the *MALFORMED_STRING* operator detected 18 faults in *SchoolMate* which did not properly handle special characters in input fields and resulted in invalid and incomplete web pages. Finally, there are two faults in *Gallery*, the first one is an improper input validation message, which was revealed by test cases generated by the *LARGE_INPUT* operator. The second fault appears when adding a group with an existing name, and the *CYCLE_REPETITION* operator can generate the fault-finding test cases from the original test cases. Clearly, the fault-finding ability of the original test cases has been improved by the proposed approach.

Table 5 also indicates that four of the multiplication operators (*LARGE_INPUT*, *MALFORMED_STRING*, *SQL_INJECTION* and *CYCLE_REPETITION)* are more effective in fault finding than the others, because test cases multiplied by these operators found new faults and other test cases did not. However, the results may be biased due to the types of faults revealed in this experiment.

Table 5. The number of faults revealed by the multiplied test cases for each application. CMS: Complaint Management System. ULF: Ultra Light Forum. SM: SchoolMate. GAL: Gallery.

Multiplication Operator	#Faults				
	CMS	ULF	SM	GAL	All Apps
EMPTY_INPUT	0	0	0	0	0
LARGE_INPUT	0	1	0	1	2
MALFORMED_STRING	0	1	18	0	19
SQL_INJECTION	0	1	0	0	1
DOUBLE_CLICK	0	0	0	0	0
SLEEP	0	0	0	0	0
CYCLE_REPETITION	1	0	0	1	3
CROSSOVER	0	0	0	0	0
Total	1	1*	18	2	22*

* the same fault revealed by test cases multiplied by different operators is only counted once

Results of the second experiment are presented in Table 6. We notice that the test cases multiplied by the *EMPTY_INPUT*, *LARGE_INPUT*, *MALFORMED_STRING*, *SQL_INJECTION* and *CYCLE_REPETITION* operators killed more mutants than the original test cases. In general, the multiplied test cases found 12 of the 15 faults, while the original test case found only 4 of them. The results may indicate that improvement in fault-finding ability can be achieved with the proposed approach. Three seeded faults were not found because they were in statements not covered by both the original test cases and multiplied test cases.

We also notice that some faults can only be revealed by test cases multiplied by specific multiplication operators. For example, the second fault was only detected by test cases generated by the *CYCLE_REPETITION* operator. Similar situations also happened on the fifth and sixth faults, and each of them was only revealed by test cases generated by the *LARGE_INPUT* and *EMPTY_INPUT* operator, respectively. The findings also show that the proposed multiplication operators are valuable in web application testing.

Among the proposed operators, the *DOUBLE_CLICK*, *SLEEP* and *CROSSOVER* operators did not improve the fault detection effectiveness of the original test cases in both experiments. However, these operators may be useful for revealing other types of faults. The *DOUBLE_CLICK* operator, like we mentioned in Section 2.4, can be used to test if a GUI element, e.g., a button confirming a purchase, is idempotent. In addition, the random waiting time ranged from 1 to 10 seconds we set in the *SLEE*P operator may be too short to trigger bugs caused by delay, and a few adjustments may be necessary. Finally, the *CROSSOVER* operator has the merit of creating new test paths from the original test cases. The value of these operators could be demonstrated if more experiments are conducted. Moreover, at the moment, we only experimented one multiplication to one test case. In the future, we plan to see whether fault-finding capability can be further improved if multiple multiplication operations can be applied to a test case.

5.4 Threats to Validity

Internal threats: The implementation of the proposed approach and the set-up of the evaluation could affect the validity of results. To ensure the correctness of test execution, we adopted Robotframework [21], a mature and active test automation framework to test execution and report generation. We also restored the initial states of the applications and corresponding databases before each test execution to make sure that the results are not affected by previous execution. In the first experiment, we did not search for seeded faults, but found real faults in real programs. In the second experiment, we applied mutation testing, a widely studied technique for determining the fault-finding ability of the test cases, and created faulty versions of the application with common mutation operators.

External threats: The applications selected and the starting test cases used might affect generality of results. One of the subject programs, *SchoolMate*, is also used by other research on

Table 6. For Gallery, if each mutant is killed by the test cases multiplied by different multiplication operators, respectively. For the 1st to 15th mutant, if it is killed by the test cases generated by the multiplication operator, the IsMutantKilled column is put 1, otherwise 0. The Total column lists the total number of mutants killed.

Multiplication Operator	IsMutantKilled															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
None (The Original Test Cases)	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	4
EMPTY_INPUT	1	0	1	1	0	1	1	0	1	1	1	1	0	0	0	9
LARGE_INPUT	1	0	1	1	1	0	1	0	1	1	1	1	0	0	0	9
MALFORMED_STRING	1	0	0	1	0	0	1	1	1	1	1	1	0	0	0	8
SQL_INJECTION	1	0	0	1	0	0	1	1	1	1	1	1	0	0	0	8
DOUBLE_CLICK	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	4
SLEEP	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	4
CYCLE_REPETITION	1	1	0	0	0	0	1	0	1	1	1	1	0	0	0	7
CROSSOVER	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	4

web application testing [2, 6]. Also, we chose the *Gallery* program whose size is much larger than other subjects. *Gallery* is mature and widely used with more than twelve million downloads. Nevertheless, our approach found faults in Gallery. Regarding the original test cases used for the experiments, we ensured that all main functionality or scenarios in the manuals are covered by the cases to make them representative for functional or acceptance testing.

5.5 Limitations

Support for other actions and data types: There are some actions often performed in web applications but not supported in current strategy. For example, "*select items from a list*" is a common action when using web applications, and "*drag and drop*" within or outside a browser is often implemented in Ajax-based applications. Automatically identifying and modifying these actions can help create more test paths in testing. Moreover, the proposed approach does not deal with the data types of input values. The regenerated test cases would be more effective if the types are considered while replacing the input values.

Test oracle handling: Like we mentioned in Section 3.4, Test oracles in some of the multiplied test cases may become slightly inappropriate due to modified input values or behaviors, and thus human evaluation is needed. In our experiments, we only performed evaluation on test reports returning *fail* and may omit possible false negatives. We are currently exploring a more intelligent way to handle the test oracle problem.

6. RELATED WORK

In web application testing, previous studies [2, 24] tried to add supplementary test values to an existing test suite. Alshahwan and Harman [2] applied dataflow testing to test case regeneration, and generated test sequences from existing HTTP requests to cover definition-use pairs for each server-side session variable and database state. Sprenkle et al. [24] suggested that replaying test cases without maintaining consistent database states helped detecting extra faults and contributed to additional statement coverage. Unlike these techniques, our approach is black-box, and no server-side information is needed.

Another work related to our approach is proposed by Offutt et al. [19, 29]. Their method called data perturbation modifies values of existing XML messages to testing web services. There are three main differences between their technique and ours. First, their technique is presented for web services and XML and executable test cases. Second, our approach modifies input values directly, whereas the data perturbation approach needs XML schemas to perform modification. Finally, we proposed a novel test case multiplication approach, multiplication of behavior, which adds or replaces actions in one or many test cases to simulate different user behaviors. On the other hand, traditional regeneration and augmentation techniques [5, 22, 30] have not taken characteristics of web applications into consideration.

There is a large body of work discussing test case generation for web applications [1, 4, 6, 11, 13, 17, 20, 27]. Many of them are model-based [4, 13, 17, 20] and try to create test cases to satisfy particular testing coverage based on the models derived from the interface or source code of applications. Other works adopt dynamic symbolic execution [6, 27] and generate inputs to solve symbolic constraints derived from control flow paths. Also, a search-based method [1] and a technique leveraging user session data [11] are also proposed for test generation. All of these techniques can be candidates to produce the starting test suites required by our approach.

7. CONCLUSION

In this paper, we presented the test case multiplication technique which regenerates test suites from the original test suites in form of executable test cases by modifying the input values or behaviors. The proposed approach has two categories. Multiplication of input value removes or replaces input data in a test case. Multiplication of behavior changes or repeats behaviors in one or more test cases. The novelty of our work is in several respects. First, our technique is black-box and needs neither server-side information nor transition diagrams of the application's GUI states. Second, many of the proposed multiplication operators consider characteristics of web applications. Third, our multiplication of behavior approach is the first work on test case regeneration for web applications.

We evaluated the improvement to fault detection effectiveness by our approach on four open-source web applications. Test cases multiplied by our approach revealed 22 faults over four applications which were not detected by the original test cases. Also, more mutants were killed by test cases multiplied by our approach.

8. REFERENCES

[1] Alshahwan, N. and Harman, M. 2011. Automated Web Application Testing Using Search Based Software Engineering. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2011), 3–12.

[2] Alshahwan, N. and Harman, M. 2012. State Aware Test Case Regeneration for Improving Web Application Test Suite Coverage and Fault Detection. *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), 45–55.

[3] Ammann, P. and Offutt, J. 2008. Introduction to software testing. Cambridge University Press.

[4] Andrews, A.A., Offutt, J. and Alexander, R.T. 2005. Testing Web applications by modeling with FSMs. *Software & Systems Modeling*. 4, 3 (Jul. 2005), 326–345.

[5] Arcuri, A. 2010. Longer is Better: On the Role of Test Sequence Length in Software Testing. *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2010), 469–478.

[6] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A. and Ernst, M.D. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*. 36, 4 (Jul. 2010), 474–494.

[7] Binder, R.V. 2000. Testing object-oriented systems: models, patterns, and tools. Addison-Wesley.

[8] Cloud Computing System and Application Services Technologies: https://www.itri.org.tw/eng/econtent/research/research02_02.aspx?sid=26. Accessed: 2014-04-05.

[9] DeMillo, R.A., Lipton, R.J. and Sayward, F.G. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*. 11, 4 (1978), 34–41.

[10] Di Lucca, G.A. and Fasolino, A.R. 2006. Testing Web-based applications: The state of the art and future trends. *Information and Software Technology*. 48, 12 (Dec. 2006), 1172–1186.

[11] Elbaum, S., Rothermel, G., Karre, S. and Fisher II, M. 2005. Leveraging User-Session Data to Support Web Application Testing. *IEEE Trans. Softw. Eng.* 31, 3 (Mar. 2005), 187–202.

[12] Goldberg, D.E. 1988. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley.

[13] Halfond, W.G.J. and Orso, A. 2007. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2007), 145–154.

[14] Hamlet, R.G. 1977. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*. SE-3, 4 (1977), 279–290.

[15] Holland, J.H. 1975. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press.

[16] Kieyzun, A., Guo, P.J., Jayaraman, K. and Ernst, M.D. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009* (May 2009), 199–209.

[17] Mesbah, A., van Deursen, A. and Roest, D. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering*. 38, 1 (Jan. 2012), 35–53.

[18] Myers, G.J., Sandler and Badgett, T. 2011. *The art of software testing*. John Wiley & Sons.

[19] Offutt, J. and Xu, W. 2004. Generating Test Cases for Web Services Using Data Perturbation. SIGSOFT Softw. Eng. Notes. 29, 5 (Sep. 2004), 1–10.

[20] Ricca, F. and Tonella, P. 2001. Analysis and Testing of Web Applications. *Proceedings of the 23rd International Conference on Software Engineering* (Washington, DC, USA, 2001), 25–34.

[21] Robotframework: http://robotframework.org/. Accessed: 2014-04-05.

[22] Santelices, R., Chittimalli, P.K., Apiwattanapong, T., Orso, A. and Harrold, M.J. 2008. Test-Suite Augmentation for Evolving Software. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2008), 218–227.

[23] Selenium - Web Browser Automation: http://docs.seleniumhq.org/. Accessed: 2014-04-05.

[24] Sprenkle, S., Gibson, E., Sampath, S. and Pollock, L. 2005. Automated Replay and Failure Detection for Web Applications. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2005), 253–262.

[25] The Open Web Application Security Project (OWASP): https://www.owasp.org/. Accessed: 2014-04-05.

[26] Thummalapenta, S., Lakshmi, K.V., Sinha, S., Sinha, N. and Chandra, S. 2013. Guided Test Generation for Web Applications. *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), 162–171.

[27] Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H. and Su, Z. 2008. Dynamic Test Input Generation for Web Applications. *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), 249–260.

[28] XSS Filter Evasion Cheat Sheet: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accessed: 2014-04-05

[29] Xu, W., Offutt, J. and Luo, J. 2005. Testing Web Services by XML Perturbation. *16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005* (2005), 257–266.

[30] Yoo, S. and Harman, M. 2012. Test Data Regeneration: Generating New Test Data from Existing Test Data. *Softw. Test. Verif. Reliab.* 22, 3 (May 2012), 171–201.