Automated Testing of Web Applications with Text Input*

Chi-Yun Wu[1] Farn Wang[1,2] Ming-Hong Weng[3] Jun-Wei Lin[1]

1: Department of Electrical Engineering, National Taiwan University
2: Graduate Institute of Electronic Engineering, National Taiwan University
3: Institute for Information Industry
farn@ntu.edu.tw

Abstract

This paper presents a highly automated procedure for testing web applications that require text input from users. Such a text input page usually queries for personal profile data, user accounts, and passwords. The proposed technique explores the pages of a web system. When a page that requires text input is encountered, our testing procedure then checks the test data bank for appropriate example personal profile data to feed into the text input fields of the page. Perturbation to the example profile data can also be injected to test how an application under test can defend itself against illegal input data. Experimental results show that our technique can automatically pass through web pages that query registration data and user account identification.

Keywords: testing, black-box, web, apps, table, mutation

1 Introduction

Web applications are ubiquitous nowadays in every aspect of our life, including financial, healthcare, entertainment, etc. Due to the 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 [1]. 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. The test cases are usually recorded or written as runnable test cases supported by test automation frameworks such as Selenium[1] or Robotframework[2]. Unfortunately, test suites written by human engineers are costly and labor-intensive[3].

To reduce human effort in designing test cases that create diverse workload, we propose a highly automated software testing procedure that involves repetitive iteration of the following steps.

(1) Our procedure first scans the pages of a web program and builds a state-transition diagram of the web application. In the diagram, the states are the pages while the edges are the transition relation between pages. For each edge, we also label it with the action that transits the source page to the destination page. However, the step may fail to scan some pages due to inappropriate input data. For example, we may fail to pass the log-in page due to invalid user account password. We may also fail to register to a web program due to invalid user contact address.

(2) After the first step, the users may check the state-transition diagram and spot actions that were deflected due to inappropriate input data. Our testing procedure then finds appropriate test input from our test data bank for this action. However, to increase the diversity of workload to the web program, we also employ test case regeneration or augmentation[4, 8, 10] to inject perturbation to the test data. For example, we may

*Our work is partially supported by Ministry of Science and Technology (MOST), Taiwan, ROC and Innovative DigiTech-Enabled Applications & Services Institute (IDEAS), Institute for Information Industry (III), Taiwan, ROC.

[1] Selenium - Web Browser Automation: http://docs.seleniumhq.org/
[2] Robotframework: http://robotframework.org/

change an email address to illegal strings, a birthday string to date expressions in other format. Such mutation should help in testing the defense of the web program against bad input data. After appropriate input data are generated for those deflected actions, we then go back to step (1) to rescan the pages.

Our testing procedure may repeat these two steps until either the testing budget runs out or certain testing adequacy criterion is fulfilled.

Previous studies [2, 9] introduce perturbation to the test input by recombining and reordering test cases to improve the effectiveness and coverage of an original test suite. However, most 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[7], 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. In contrast, our techniques of introducing test data perturbation is purely black-box and may have general applicability.

The remainder of this paper is organized as follows. Section 2 explains our testing procedure. Section 3 reports our implementation and experiments. Section 4 is the conclusion.

2 Testing procedure design

Our goal is to relieve the test engineers of the burden of writing test cases to web programs. Our testing procedure can almost automatically explore webpages and deduce input test data to either test the syntax protection of those test input fields or pass webpages that need complicate user data. In table 1, we show our testing procedure. In the while-loop starting at line 3, we iteratively explore the webpages deeper and deeper from the root page. The while-loop continues until either a webpage crash is detected or the users feel that enough testing has been done.

Line 4 scans all the pages that can be reached under button coverage. In our implementation, we use the popular web-crawler Crawljax[3] [5, 6] to scan all pages. Specifically, Crawljax tries to action every actionables in every visited page to a certain depth of exploration within a time budget. If an action brings about program crashes, the testing procedure exits with the failure report. (See line 5.)

Line 6 builds a finite-state transition diagram to represent the page transition relation of the web application. The edges of the diagram are labelled with the actions for the page transition. Line 7 then identifies those actions (possibly with the help of the users) that did not bring about the expected response page. For example, a failed login action or a failed submission of registration data may lead us to a warning page. This is the only step that our testing procedure needs dynamic human intervention. Here the users need let us know which actions do not result in the expected response pages. Then if the source page of an action needs several lines of text input, the testing procedure turns to our test data bank for the new input data when the page is encountered again in the next round of the while-loop.

[3] http://crawljax.com/

Table 1. Testing procedure in the abstract level

```
1:  Query for the url of the web program under test.
2:  Set the exploration action depth d = 1.
3:  while more tests are needed do
4:      Scan the web program with depth d.
5:      if some page carshes then report the action sequence
        that leads to the crash and exit. end if
6:      Construct the state-transition diagram out of the ac-
        tions to all pages scanned in this iteration.
7:      for every action that did not end up in an expected
        page due to inappropriate input data do
8:          Randomly choose a user profile example in our
            test data bank.
9:          Add action rules that use appropriate test input
            data from the chosen example.
10:     end for
11:     Increment d by 1.
12: end while
```

2.1 Brief review of Crawljax

Crawljax [5, 6] is a novel crawler that can handle web applications with JavaScript and dynamic DOM structure in webpages. Such web programs are also called AJAX applications. Crawljax automatically and dynamically analyzes page state changes of dynamic web applications in web browsers. It triggers page transitions by scanning the DOM trees, recognizing candidate actionable elements that are capable of changing the state, and then firing the actionables. With successive actions, Crawljax incrementally infers a state-transition diagram of an AJAX application. In such a diagram, the states are the pages with DOM trees while the edges are actions that transit from one page to another.

Crawljax allows its users to control navigation through the webpages by specifying exploration depth, time budget, and action rules. For example, a user may specify what action to take when certain buttons are recognized. A user

may also specify what data to fill in when certain data fields appear. In this work, we use the Crawljax-inferred diagrams to identify actions that do not end up with expected responses due to inappropriate test input data. Then we use our test data bank to prepare appropriate test input data so that an expected response may occur.

2.2 Preparation of test data bank

The actionables in a webpage may not be simple clickable buttons. Options and text input may also very well appear. Especially, many web applications require the users to register and then latter login with a valid account and a matching password. In this work, we focus on these two types of text input. We prepare a test data bank. Each element of the test data bank is a valid example of user profile data, including the first name, the last name, the birthday, the mailing address, the email addresses, the phone numbers, account name, password, and etc. When we need appropriate text input data to pass a page for expected response, we select a user profile data example in the test data bank and add action rules to Crawljax so that when this page is encountered again in the next iteration of the while-loop from line 3 in table 1, appropriate text input data can be selected from this example user profile data. For example, if we see some text input field with keywords like phone numbers or mobile numbers, we will use the phone number items from the chosen example to fill in.

In case, the testing procedure cannot decide the appropriate input data for certain text input fields, it simply uses random strings.

2.3 Injection of perturbation to test input data

Traditional load testing projects use the same test scripts to create huge workload to bombard an app under test. Recently there are papers for techniques to diversify the test scripts in the hope of revealing more functional deficiencies of the applications[2, 7, 9]. For example, Ammann and Offutt suggested many mutation operators to input values that are likely to reveal typical program bugs [3]. In this work, we adopted the following mutation operations to test input data from the test data bank so as to increase the probability to reveal bugs.

(1) *Mutation to boundary values*: 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.

(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 suggestion 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.

(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 de facto 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 mutation 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 adminuser. Such a change may then create a test case that test with unauthorized database accesses.

3 Implementation and experiment

We have endeavored to implement our testing procedure on top of Crawljax and carried out experiments. The implementation is on a VMWare running Ubuntu 14.04.1 and Linux version 3.13.0. The CPU of the server is Intel(R) Core(TM) i7/2.80GHz with 3G memory. The browser is Mozilla Firefox 31.0.

We have passed the testing of several web programs with account control, user registration data pages, and tens of

pages, and more than 30 actionable items in a page.

3.1 An example web app

In the following, we use a small example web program to explain our experiment. The example is a web program called "*Dine Out Together*" which allows users to register for free, find restaurants, and people to dine out together and make friends with. The main page snapshot is shown in figure 1(a). The four images in the lower half are respectively for registration, finding people to dine out with, finding restaurants, and exit. Figure 1(b) are the page for filling in registration data. After a successful registration, the page in figure 1(c) notifies the users of the completion of a successful registration. Figure 1(d) and (e) respectively show the pages for the members that can be invited to dine out and the registered restaurants.

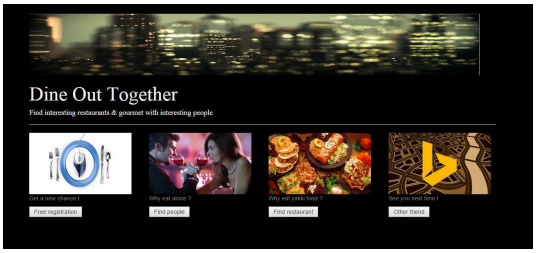
3.2 Experiment

Figure 2(a) shows the page-transition diagram of the app after exploration with depth of one action. This diagram was generated by Crawljax by setting the exploration depth to 1 and time budget to 20 minutes. Apparently at this stage, we did not see a page that really need text input data. But neither have we actioned on every actionable in all visited pages. Thus we need to explore the pages with action depth = 2. That is, we need to execute the while-loop at line 3 in table 1 one more time.

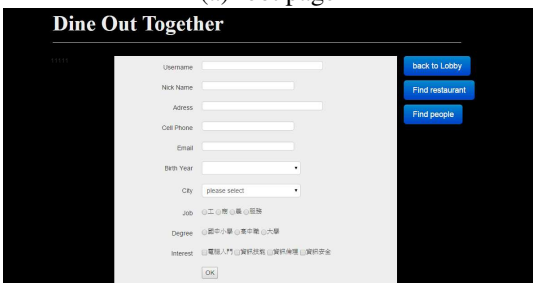
Figure 2(b) shows the page transition diagram after exploration with depth = 2. In this exploration, after state 1, we can go to state 2 which is the page for registration data input. Since there is no specific action rule to guide the test input, it is very likely that the test input action may result in registration failure if the web program implements a strict syntax checking of the member names, birthdays, email addresses, phone numbers, etc. Luckily our testing procedure stops exploration from state 2 which is at action depth 2. Then line 7 in table 1 will add action rules to Crawljax so that in the 3rd round, our testing procedure knows what texts to fill in for the text input fields of state 2 and can successfully pass state 2 and gets to state 3 in the page-transition diagram of figure 2(c).

3.3 Discussion

In our experiment, we did not modify the source code of Crawljax. Then it is clear that for web programs with a lot of actionable items in each page, the exploration of all dynamic pages can easily become combinatorially explosive and hence out of reach. In the future, it may also be important to use coverage techniques that can help us in avoiding taking the same actions repeatedly.

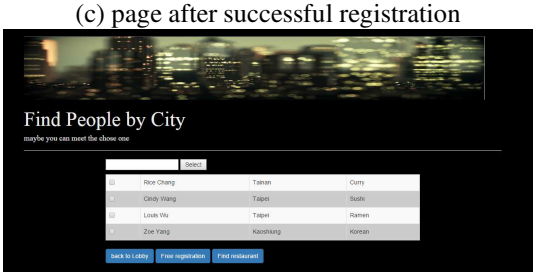


(a) root page

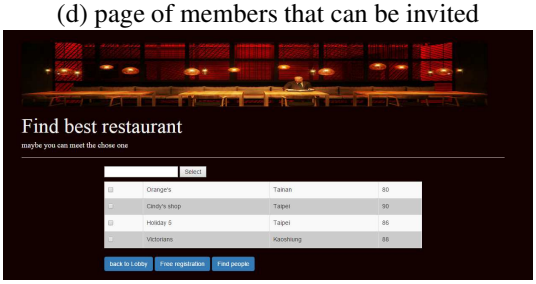


(b) registration page

(c) page after successful registration

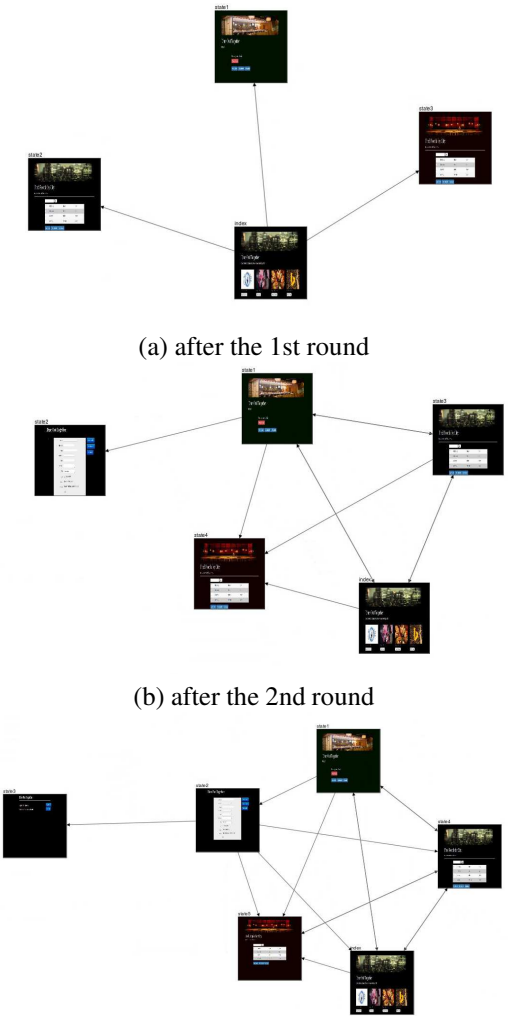


(d) page of members that can be invited



(c) page of restaurants

Figure 1. Snapshots of the wepbages of the Dine Out Together



(a) after the 1st round

(b) after the 2nd round

(c) after the 3rd round

Figure 2. Page-transition diagrams

We also found that many text input pages are related to personal, either legal or natural, data profile. Thus our design of test data bank seems quite promising. In the long run, it would be interesting to see what other types of test data can be useful enough in practice so that we can save it to our test data bank.

4 Conclusion

We present an automatic testing procedure for web applications with complex table pages to fill in. The experiment also show promise that our techniques may work in testing real-world web apps. Our techniques also can be easily modified to accommodate different test case generation algorithms. In the future, it is also possible to incorporate safety assertions and responsiveness requirements for the evaluation of function failure.

References

[1] The open web application security project (owasp). https://www.owasp.org/.

[2] N. Alshahwan and M. Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, page 45âĂŞ55, 2012.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] A. Arcuri. Longer is better: On the role of test sequence length in software testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, 2010.

[5] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[6] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, January/February 2012.

[7] G. J. Myers. *The Art of Software Testing (Second Edition)*.

[8] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 218âĂŞ227, 2008.

[9] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 253âĂŞ262, 2005.

[10] S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Journal of Software Testing, Verification, and Reliability*, 22(3):171âĂŞ201, May 2012.