ADVANCED SOFTWARE TESTING TECHNIQUES BASED ON COMBINATORIAL DESIGN

by

WENHUA WANG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2010

Copyright © by Wenhua Wang 2010

All Rights Reserved

ACKNOWLEDGEMENTS

I am sincerely grateful to Prof. Lei for his tutoring in my PhD program. Through these years, he has always been offering motivation and encouragement to me. Without his mentoring and support, it would not have been possible to make it to this point.

I also owe my gratitude to Dr. Che, Dr. Fegaras, Dr. Khalili, and Dr. Kung for their serving in the committee. They instructed me in computer science knowledge and research methodology. Their valuable advice guided me to gain an insight into my research work.

In my security testing project, Dr. Csallner and Dr. Liu also instructed me in security knowledge and problem solving skills. I appreciate their tutoring too.

During my PhD program, many people in UTA also gave me help, especially Ms. McBride and Ms. Costabile. Thanks for their help!

Finally, I would not be in this position without my parents' continuous support in the last thirty years. I would like to dedicate this work to them.

July 15, 2010

ABSTRACT

ADVANCED SOFTWARE TESTING TECHNIQUES BASED ON COMBINATORIAL DESIGN

Wenhua Wang, PhD

The University of Texas at Arlington, 2010

Supervising Professor: Yu Lei

Combinatorial testing refers to a testing strategy that applies the principles of combinatorial design to the domain of software test generation. Given a system with *k* parameters, combinatorial testing requires all the combinations involving *t* out of *k* parameters be covered at least once, where *t* is typically much smaller than *k*. The key insight behind combinatorial testing is that while the behavior of a system may be affected by many parameters, most faults are caused by interactions involving only a small number of parameters. Empirical studies have shown that combinatorial testing can dramatically reduce the number of tests while remaining effective for fault detection.

Existing work on combinatorial testing has mainly focused on functional requirements, and has only considered non-interactive systems, i.e., systems that take all inputs up front without interacting with the user in the middle of a computation. In this dissertation, we propose three new combinatorial testing techniques, two of which deal with interactive web applications, and the third one deals with non-functional security requirements: (1) *Combinatorial construction of web navigation graphs*: A navigation graph captures the navigation structure of a

web application. The main challenge of constructing navigation graphs is handling dynamic web pages that are only generated at runtime and in response to user requests. We develop a combinatorial approach that generates user requests to discover these dynamic web pages. We report a software tool called *Tansuo*, and demonstrate the effectiveness of our approach using several real-life open-source web applications. (2) *Combinatorial test sequence generation for web applications*: One important aspect of web applications is that they often consist of dynamic web pages that interact with each other by accessing shared objects. It is nearly always impossible to test all possible interactions that may occur in a web application of practical scale. We develop a combinatorial approach to systematically exercising these interactions. Our experimental results show that our approach can effectively detect subtle interaction faults that may exist in a web application. (3) *Detection of buffer overflow vulnerabilities:* Buffer overflow vulnerabilities are program defects that can cause a buffer to overflow at runtime. Many security attacks exploit buffer overflow vulnerabilities to compromise critical data structures. We develop a combinatorial approach to detecting buffer overflow vulnerabilities. Our approach exploits the fact that combinatorial testing often achieves a high level of code coverage. Experimental results show that our approach can effectively detect buffer overflow vulnerabilities.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Software systems have become ubiquitous in modern society. As more and more businesses, e.g., enterprise management, healthcare services, and financial services, are conducted using software systems, our lives are being made more convenient. However, we are also becoming more dependent on the proper functioning of these software systems. Consequently, software quality has become a major concern in the software engineering industry.

Software testing is an effective way to ensure the quality of a software system. It is nearly always impossible to exhaustively test all the input combinations. Figure 1.1 presents an options configuration screen in Microsoft Word 2003. In this screen, testers would have to test 24576 combinations, even for a small set of configuration options that includes the 14 options in the "*Show*" category of the "*View*" tab, in order to perform exhaustive testing. The number of combinations will be astronomical if all the configuration options are considered. Therefore, it is necessary to select a subset of these combinations to test.

To test software effectively with limited budgets and time, the notion of combinatorial design has been introduced for test selection [13][14][27]. The key insight behind combinatorial testing is the following. While the behavior of a software system could be affected by many parameters, empirical study results [14] show that most faults are caused by individual parameters or interactions involving only a small number of parameters. Thus, for the purpose of fault detection, we only need to test the combinations that involve a small number t of parameters for a software system with n parameters, where t is typically much smaller than n.

We will refer to a combination involving *t* parameters as a *t*-way combination, and covering all the *t*-way combinations as achieving the *t*-way combinatorial coverage [13][14][27].

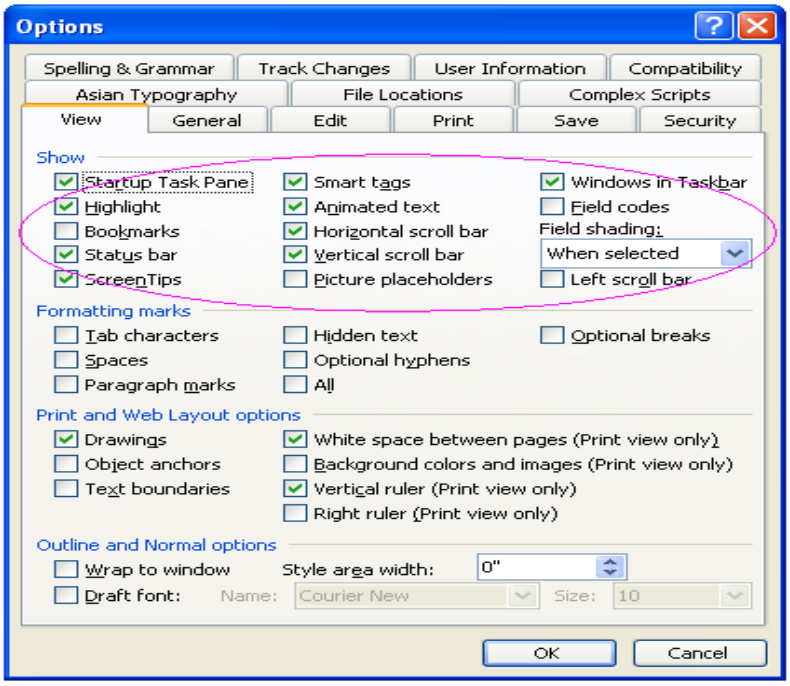

Figure 1.1 An Options Configuration Example from Microsoft Word 2003.

To illustrate the notion of *t*-way combinatorial coverage, consider the following example. In this example, a form has three parameters *P1*, *P2*, and *P3*, each parameter having two values "0" and "1". Figure 1.2 shows a set of 2-way tests for this form. Each row represents a test, and each column represents a parameter (in the sense that each entry in a column is a value of the parameter represented by the column). An important property of this test set is that each of the three possible pairs of columns, i.e., columns *P1* and *P2*, columns *P1* and *P3*, and columns *P2* and *P3*, contains all the four possible pairs of values of any two (out of three) parameters, i.e., {00, 01, 10, 11}. Thus, this test set achieves 2-way combinatorial coverage for this form. Note that an exhaustive test set would consist of 2^3 = 8 tests. As another example, only 10 tests are needed to achieve 2-way combinatorial coverage for the example presented in Figure 1.1, if *IPOG* algorithm is used [35].

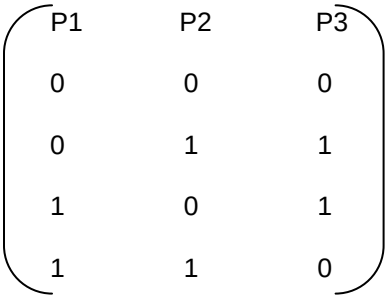

Figure 1.2 A 2-Way Test Set

In the last decades, many empirical results [32][33][72] have shown the effectiveness of combinatorial testing in practical applications. Many approaches [27] have also been reported to generate test sets that are as small as possible but still achieve a combinatorial coverage criterion. However, existing research work mainly focuses on functional requirements, i.e., ensuring that a system implements all the functions as designed, and has only considered non-interactive systems, i.e., systems that take all inputs up front without interacting with the user in the middle of a computation. There has been little work on how to adapt combinatorial design to dealing with interactive web applications and non-functional security requirements. Tests generated by existing approaches may not work in web application testing and security testing. Consider the options configuration example presented in Figure 1.1. The order in which the different options appear in a combination is insignificant, i.e., it does not affect the testing result. But, in web application testing, the order in which two web pages are visited is significant. In particular, two web pages may be visited in one order but not the other. For example, consider a book review web application that allows a user to vote for a book only once. After the user has voted for a book, the web application shows the up-to-date voting statistics to the user. In this example, the "*Voting*" web page can be visited after the "*Result*" web page, but not in the reverse order. Therefore, it is important to consider the order in which the different web pages appear when we try to test the possible interactions among the different web pages.

To investigate the potential value of combinatorial testing in web application testing and security testing, this dissertation presents three projects in adapting combinatorial testing to these areas. Two of the three projects deal with interactive web applications, and the third one deals with security requirements. In the three projects, we develop three combinatorial testing techniques, including a combinatorial approach to building navigation graphs for dynamic web applications, an interaction-based test sequence generation approach for testing web applications, and a combinatorial approach to detecting buffer overflow vulnerabilities.

- The combinatorial approach to building navigation graphs for dynamic web applications is proposed to capture the navigation structure of a web application and represent it with a web navigation graph. The generated web navigation graph can facilitate testers in web sequence testing [75] and regression testing. However, in dynamic web applications, web pages are only generated at runtime and in response to user requests, which is a big challenge for web navigation graph generation. To generate practical web navigation graphs, we develop a combinatorial approach that generates user requests to discover these dynamic web pages. In addition, a dynamic web application may produce an infinite number of web pages, which results in an infinite web navigation graph. To control the web navigation graph size, a URL abstraction mechanism is also proposed in this approach.
- The interaction-based test sequence generation approach for testing web applications generates test sequences from web navigation graphs. One important aspect of web applications is that they often consist of dynamic web pages that interact with each other by accessing shared objects, e.g., session objects and databases. As for the faults caused by the interactions between web pages, test sequences are needed to detect them. However, it is nearly always impossible to test all the possible interactions that may occur in a web application of practical

scale. To effectively detect subtle interaction faults, we develop a combinatorial approach to systematically exercise these interactions with a small set of test sequences. When applied to a web application, the generated test sequences systematically detect the interaction faults that may exist in a web application.

- The combinatorial approach to detecting buffer overflow vulnerabilities adapts combinatorial testing to detecting buffer overflow vulnerabilities in the software under test. Buffer overflow vulnerabilities are program defects that can cause a buffer to overflow at runtime. Many security attacks exploit buffer overflow vulnerabilities to compromise critical data structures. To systematically detect buffer overflow vulnerabilities, we develop a combinatorial approach to generate tests by combining parameter values with combinatorial testing techniques. Our approach exploits the fact that combinatorial testing often achieves a high level of code coverage, which helps our approach reach potentially vulnerable statements in the software under test.

To evaluate the effectiveness of the three approaches, experiments have been conducted on open source applications. Our empirical results show that combinatorial testing can be effective in capturing web navigation structures and detecting faults in web applications as well as security-related faults, which indicates the value of combinatorial testing in web application testing and security testing.

The remainder of this paper is organized as follows. Chapter 2 discusses research work related to the three approaches proposed in this dissertation, including research work on combinatorial testing, web navigation graph generation, web test sequence generation, and buffer overflow vulnerability detection. Chapter 3 introduces the combinatorial approach to building navigation graphs for dynamic web applications. Chapter 4 presents the interaction-based test sequence generation approach for testing web applications. The combinatorial

approach to detecting buffer overflow vulnerabilities in the software under test will be introduced in Chapter 5. Chapter 6 concludes this dissertation and discusses future work.

CHAPTER 2

RELATED WORK

2.1 Overview

This dissertation proposes three domain-specific combinatorial testing approaches: (1) a combinatorial approach to web navigation graph generation; (2) a combinatorial approach to web test sequence generation; and (3) a combinatorial approach to buffer overflow vulnerability detection. The first two approaches deal with interactive web applications, and the last one deals with security testing. The remaining of this chapter is organized as follows. Section 2.2 discusses related work on combinatorial testing in general. Section 2.3 discusses related work on web navigation graph generation. Related work on web test sequence generation will be discussed in Section 2.4. Section 2.5 discusses related work on buffer overflow vulnerability detection.

2.2 Combinatorial Testing

Combinatorial testing refers to a testing strategy that applies the principles of combinatorial design to the domain of software test generation [13]. It creates tests by combining parameter values with combinatorial test generation strategies that will be introduced in Section 2.2.1. Section 2.2.2 discusses empirical studies in combinatorial testing.

2.2.1. Combinatorial Test Generation Strategies

Many combinatorial test generation strategies have been proposed to generate test sets that are as small as possible but still satisfy a given coverage criterion, e.g., 2-way combinatorial coverage and 3-way combinatorial coverage. Grindal et al. [27] surveyed fifteen important strategies that have been reported in the literature. Two representative strategies, i.e.,

AETG strategy [13] and *IPO* strategy [35], are described as follows. D. M. Cohen et al. [13] proposed a greedy algorithm for combinatorial testing, called Automatic Efficient Test Generation (*AETG*). In this algorithm, a test is created in a way such that it covers as many uncovered *t*-way combinations as possible. First, *AETG* strategy puts a value *v* of a parameter *p* into a new candidate test *ct*, if *v* exists in the greatest number of uncovered *t*-way combinations. Second, *AETG* strategy randomly selects *p'* out of the rest parameters, and puts its value *v'* into *ct* so that {*v*, *v'*} exists in the greatest number of uncovered *t*-way combinations. Third, completes *ct* by including values of the rest parameters one by one, with the policy used at the second step. Fourth, *AETG* strategy repeats the above three steps to generate 50 candidate tests, and picks one out of the 50 candidate tests as a final test if this candidate test covers the greatest number of *t*-way combinations. *AETG* strategy repeats the four steps until all *t*-way combinations have been covered. Lei et al. [35] proposed another *t*-way testing strategy called In-Parameter-Order (*IPO*). The *IPO* strategy generates a *t*-way test set to cover all the *t*-way combinations between the first two parameters and then extends the test set to cover the all the *t*-way combinations of the first three parameters. This process is repeated until the test set covers all the *t*-way combinations of the parameters.

Existing work on combinatorial test generation significantly differs from the work described in this dissertation. In particular, tests are generated to test functional requirements, where all the parameter values are considered equal during test generation. This is different from our work in detecting buffer overflow vulnerabilities, where special values that may trigger a buffer overflow are treated differently than other values. Moreover, existing work has only considered non-interactive systems, and they cannot be applied to testing interactive systems, such as web applications.

2.2.2. Empirical Studies

Cohen et al. [13] applied *ATEG* to testing ten UNIX commands. Experiment results show that 2-way combinatorial coverage helped to achieve more than 80% branch coverage.

Burr et al. [7] further evaluated the code coverage of *AETG* on Nortel internal email system. Experimental results show that *AETG* achieved 84% branch coverage and 93% code coverage with 47 tests generated for 2-way combinatorial coverage. In addition to code coverage, empirical studies have also been conducted on the effectiveness of fault detection, i.e., how many faults can be detected by *t*-way combinations. Cohen et al. [13] applied *ATEG* to testing the interface modules of two released *Telcordia* version: *Telcordia-1994* and *Telcordia-1995*. With the tests that achieved 2-way combinatorial coverage, 67 new faults were detected in *Telcordia-1994* and 57 new faults were detected in *Telcordia-1995*. Wallace et al. [74] analyzed 15 years of medical device recall data from US Food and Drug Administration (*FDA*). In the 109 faults with clear description, 97% reported faults can be detected by tests that achieve 2-way combinatorial coverage.

Kuhn et al. [32][33] investigated the fault detection effectiveness of *t*-way combinatorial coverage (*t*=1,2,3,4,5,6). In [32], Kuhn et al. examined 194 reported faults of *Mozilla* web browser and 171 reported faults of *Apache* web server in bug tracking databases. Empirical results show that more than 70% of these faults can be detected by individual parameter values or 2-way combinations. Moreover, covering all 6-way combinations can detect all the reported faults. In [33], Kuhn et al. investigated 329 faults reported for a large distributed data management system from NASA Goddard Space Flight Center. Empirical results show that tests that achieve 2-way combinatorial coverage can detect 93% reported faults and all reported faults can be detected by the tests that achieve 4-way combinatorial coverage.

The work described in this dissertation adds additional results to the literature. In particular, our results indicate that combinatorial testing can be effective for testing interactive web applications and for testing non-functional requirements such as security.

2.3 Related Work on Web Navigation Graph Generation

2.3.1. Web Crawling

Web crawling has been an active research area in recent years [10][29][36][42][50]. Web crawling is related to our research work on web navigation graph generation because it navigates through a collection of web pages, and needs to deal with dynamic web pages. (There are two types of web crawling: surface/regular web crawling, which does not interact with web forms, and deep web crawling, which interacts with web forms. We focus our attention on deep web crawling.) However, web crawling is about "content discovery", i.e., it aims to discover as much information as possible from different web pages, while our research work is about "structure discovery", i.e., it aims to capture the navigation relationship among different web pages. Consequently, web crawling employs techniques that are very different from ours. Specifically, web crawlers often deal with the web page explosion problem by picking web pages that are information-rich and by discarding the others [10][40]. For example, web crawling discards web pages for the login functionality because these web pages contain little information. But the associated navigation behavior of this kind of web pages is very important in the navigation structure, especially for the security-critical navigation structure. Therefore, our approach keeps these web pages for the login functionality in generated web navigation graphs. In our approach, web pages are abstracted based on their URLs, instead of their contents.

Web crawling is also different from our approach in capturing input combination dependent navigation structure. In web applications, some navigation structure can only be detected after users provide correct input combinations. For example, the web pages for the administration functionality can be accessed only after users input the correct administrator username/password combination. Otherwise, a user will be navigated to an "*error*" web page, if the user submits a wrong username/password combination. To capture the navigation structure behind web forms, most web crawlers have focused on the problem of how to select values for individual parameters. One common approach used by those crawlers is to build a pre-defined

list of values for the parameters that are frequently encountered. This approach is also useful in our work. However, the problem of how to effectively combine those parameter values has been largely left open. The very recent work by Madhavan et al. [40] is an exception. Their approach uses a bottom-up search strategy to identify parameter combinations that could lead to the largest number of distinct response pages. This differs from our work, where we generate combinations to achieve a well-defined coverage criterion.

2.3.2. Web Application Testing

Our research work on web navigation graph generation is related to web application testing techniques in which test sequences are generated on the fly and by navigating through an web application [4][38]. In particular, our work is closely related to *VeriWeb* [4]. *VeriWeb* applies a general software model checking technique called *VeriSoft* [22] to web applications. To test a web application, *VeriWeb* tries to explore all possible navigation paths in a systematic manner. *VeriWeb* uses an exploration algorithm that is very similar to ours. That is, both *VeriWeb* and our approach explore web applications in a depth-first manner, and restore states after backtrack to a previous web page. However, the two approaches significantly differ in the way they handle the web page explosion problem and capture the input combination dependent navigation structure behind web forms. Specifically, *VeriWeb* addresses the web page explosion problem by allowing a limit to be set on the length of navigation paths it explores. This is different from our use of abstract URLs. *VeriWeb* addresses the input combination dependent navigation structure capture problem by allowing the user to supply and reuse pre-defined parameter values. It is unclear which strategy is used to combine parameter values during exploration.

Several models have been developed and used in model-based web application testing [1][37][51]. In particular, the UML model proposed by Ricca and Tonella [51] is close to our web navigation graph generation approach. A fundamental difference between their model and ours is that they do not abstract dynamic web pages. That is, each dynamic web page is

represented as a separate node in their model. In addition, their model did not discuss how solve the input combination dependent navigation structure capture problem.

2.4 Related Work on Web Test Sequence Generation

2.4.1. Model-Based Web Application Testing

Model-based testing typically builds an abstract model of the web application under test, and then generates test sequences from the model to satisfy some coverage goals. Existing model-based testing techniques for web applications extend traditional testing techniques, e.g., those based on control flow and/or data flow, to the web application domain. Liu et al. [37] proposed the Web Application Test Model (*WATM*) to model data flow between structural artifacts at the following five levels: function, function clusters, object, object clusters and web application. They later proposed an agent-based approach to manage the web application testing process [49]. Ricca and Tonella [51] proposed a UML model to represent the dynamic components of a web application. This model was extended later with control flow information to facilitate testing boundary conditions associated with the branches in the code [70]. Lucca et al. [38] proposed an object-oriented model for functional testing at both levels of unit testing and integration testing. Lucca et al. [39] also presented a state-chart model to model and test the browser interactions in web applications. They proposed to apply several coverage criteria presented by Binder [6] to the state chart model. These criteria include all-states, all-transitions, all-transition-k-tuples, and all-round-trip-paths. The all-states criterion is similar to a criterion designed to cover all nodes in a web navigation graph. The all-transitions criterion is equivalent to the *AllEdges* criterion that we propose to use as a contrast in our experiments. The all-transition-k-tuples criterion aims to cover all possible sequences of transitions of size *k* consecutively. This criterion is different from our *AllOrderedPairs* criterion that covers all the web page interaction pairs in a web application (The formal “ordered pair” definition in Chapter 4). Our *AllOrderedPairs* criterion does not require that the two web pages in an ordered pair be connected by a direct edge in the navigation graph of a web application.

Similarly, the test sequences do not have to contain web pages in an ordered pair consecutively one after another. In [1], hierarchical FSM-based models for large web applications are proposed to perform structural testing based on traditional test coverage criteria, including all-pages, all-actions, all-links and all-forwards.

Note that our interaction-based testing approach is also a model-based testing approach. The novelty of our work lies in the fact that we generate test sequences to achieve 2-way combinatorial coverage, which is a concept unexplored in the current web application testing domain.

2.4.2. Session-Based Web Application Testing

Several techniques have been developed that record real usage data and use the usage data to generate test sequences. Elbaum et al. [20] proposed three user session-based testing strategies: 1) replaying captured user sessions directly; 2) combining interactions from different user sessions; and 3) inserting user session values into tests generated from the model proposed by Ricca and Tonella [51]. Sprenkle et al. [61][64] addressed the problem of testing multi-user interactions by taking the underlying application state into account when executing tests. They also proposed several oracles for web application testing. To address the challenge of working with large numbers of user sessions, Sampath et al. [55] proposed a technique to reduce the size of a user-session-based test suite by clustering user sessions based on concept analysis. In [62], they compared their concept analysis-based reduction to requirement-based reduction techniques, in terms of cost, program coverage and fault detection effectiveness. Sampath et al. [54] also investigated the effectiveness of reducing test suites with a criterion designed to cover all web page sequences of size 2 that occur in the original suite of logged user sessions. This study revealed that certain faults are detected only by the occurrence of a certain sequence of web pages in a test. Note that the approach of covering sequences of size 2 is different from the approach presented in this dissertation. In the

AllOrderedPairs strategy, we do not require that two web pages in an ordered pair appear consecutively one after another in one test sequence.

Compared to model-based testing, user session-based testing does not need model construction that can be difficult for large and/or complex web applications. However, the fault detection ability of user-session-based techniques depends to a large degree on the quality of collected user sessions [19]. In addition, user-session-based techniques require field deployment and extensive user participation, which significantly limits the applicability of those techniques.

2.5 Related Work on Buffer Overflow Vulnerability Detection

2.5.1. Buffer Overflow Detection

Some existing research work [18][53] focuses on detecting whether a buffer overflow has occurred at runtime. Their approaches keep track of boundaries of each buffer in a software system. After each operation on a pointer of a buffer, their approaches check whether this pointer is still within boundaries of this buffer. If not, their approaches will alarm that a buffer overflow occurs. In other words, these approaches just detect whether a buffer overflow occurs in a specific runtime scenario. They do not consider how to develop scenarios to trigger buffer overflow vulnerabilities in a software system.

In contrast with the above existing research work, our approach on buffer overflow vulnerability detection focuses on developing tests to detect existing buffer overflow vulnerabilities in a software system. It means our approach generates a set of tests with a well-defined criterion. Note that, in Chapter 5, our approach uses the techniques in [18][53] to detect triggered buffer overflows in runtime scenarios, when testing a software system. From what has been discussed above, we can conclude that our approach and the existing approaches are complementary to each other in detecting buffer overflow vulnerabilities in a software system.

2.5.2. Buffer Overflow Prevention

Some existing research work focuses on preventing buffer overflows, or more precisely, preventing the potential damage that may be caused by buffer overflows at runtime [17][65]. For example, *StackGuard* [16] may terminate a process after it detects that a return address on the stack has been overwritten due to the overflow of a buffer that is near this return address. Existing approaches to runtime prevention of buffer overflows can incur significant runtime overhead [53]. In addition, these approaches are mainly designed for damage control for the potentially vulnerable software. This is in contrast with our research work that aims at developing and releasing the software that are free from buffer overflow vulnerabilities.

In this dissertation, our research work on buffer overflow vulnerabilities focuses on detecting buffer overflow vulnerabilities during the development stage. In particular, we discuss testing based approaches, i.e., approaches that involve actual program executions. We will not discuss approaches that are based on pure static analysis [11], as they involve quite different techniques. Note that static analysis suffers from the problem of false positives and/or negatives.

2.5.3. Fuzz Testing

Fuzz testing [67][69] is a most widely used black-box testing approach in the security testing. It is the most related to our approach. Fuzz testing typically starts from one or several normal tests, and then randomly mutates parameter values in normal tests with special inputs to derive new tests. These new tests will be used to trigger buffer overflow vulnerabilities in a software system. Advanced fuzz testing techniques [24][25][67][69] can also incorporate domain knowledge and/or employ heuristics, e.g., assigning different weights to different components.

However, poor code coverage is a major limitation of fuzzing [25]. For instance, Figure 2.1 presents a normal HTTP request. *JBroFuzz* (a web application fuzzer for HTTP/HTTPS requests) [31] generates its first set of tests by replacing "*POST*" with its special data for triggering buffer overflow vulnerabilities. Then, *JBroFuzz* replaces "*cgi-bin/test.php*" with its

special data to generate its second set of tests. This process will be repeated until all the fields in this request have been fuzzed. From this instance, we can see fuzz testing does not consider how to generate tests by combining input values effectively, which causes its poor code coverage. In contrast, our approach samples the input space and combines input values in a systematic manner to achieve a *t*-way combinatorial coverage. Empirical study results on combinatorial testing have indicated that there is a correlation between input combinatorial coverage and branch/code coverage. Thus, our approach solves the poor code coverage problem by introducing combinatorial testing to manipulate input values.

```
POST /cgi-bin/test.php HTTP/1.1
Content-Length:  20

aaaaaaaaaaaaaaaaaaaa
```

Figure 2.1 A HTTP Request Example

2.5.4. Symbolic Execution

Recently there has been a growing amount of interest in the approaches [9][23][58] that combine symbolic execution and testing to detect the buffer overflow vulnerabilities in the software under test. In these approaches, symbolic execution is used to collect path conditions consisting of a sequence of branching decisions. These branching decisions are then negated systematically to derive test inputs that will explore different paths when executed. In order to detect buffer overflow vulnerabilities, memory safety constraints are also formulated and solved together with these path conditions. A potential problem with these approaches is the path explosion problem. Techniques based on functional summaries[24], generational search [25][24], and length abstraction [71], have been developed to alleviate this problem. These approaches generate tests in a fully automatic manner. However, symbolic execution

often involves extensive instrumentation, either at the source cod or binary code level. Thus, the resulting solutions are usually specific to a particular language, build environment, or platform. In addition, for large and/or complex software system, numerous and complicated constraints that have to be solved present significant challenges to the capacity of existing constraint solvers. It is worth noting that symbolic executions are often much slower than actual program executions.

CHAPTER 3

WEB NAVIGATION GRAPH GENERATION

3.1 Background

A web application navigation graph, or simply a navigation graph, is a representation of the navigation structure of a web application. In a navigation graph, a node represents a web page and an edge represents a direct transition between two web pages. Particularly, a web page is identified by a URL [1]. As is shown in Figure 3.1, a standard URL includes five parts, i.e., *scheme*, *authority*, *path*, *query* and *fragment*. An edge is identified by the two URLs of the nodes it connects.

In a dynamic web application, a transition from one web page to the other web page can be triggered in two ways: 1) clicking a link in the current web page, and 2) submitting a form in the current web page. When a user submits a form, a web application may dynamically navigate the user to different web pages according the user's inputs for the form. In other words, a transition caused by a form submission may be dynamically determined at run time, according to the specific input combinations. Therefore, it is more difficult to capture the transitions triggered by form submissions than capture the transitions associated with links. In dynamic web applications, most navigation structures of web applications are hidden behind web forms, which has been illustrated by the experiment results in Section 3.5.4. As the prevalence of dynamic web applications, it is necessary to capture navigation structures behind web forms during navigation graph generation. However, this also presents a big challenge to web navigation graph generation, which will be discussed in Section 3.2.

The navigation graph can be used as an aid in web sequence testing [75] and regression testing for a web application. For example, after a web application has been implemented, testers have to test the conformance between the designed navigation structure and the implemented navigation structure. It means testers have to check: 1) whether all the designed (expected) navigation paths have been implemented; and 2) whether an undesigned (unexpected) navigation path has been introduced. If an unexpected navigation path has been introduced, it may cause a serious security problem, e.g., the violation of access control. Therefore, it is valuable to generate navigation graphs to validate the implemented navigation structure in a web application. As another example, navigation graphs can be used to facilitate impact analysis, i.e., how to identify web pages that could be potentially affected by a modified page.

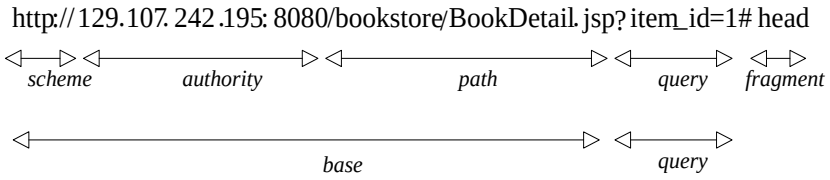

Figure 3.1 A URL Example.

3.2 Challenges and Contributions

The main challenge of generating a practical navigation graph is dealing with dynamic web pages. (If a web application only consists of static web pages, its navigation graph can be built using a classical graph traversal algorithm, e.g., a depth-first search algorithm.) Unlike a static web page, whose content is prescribed and stored on a web server, a dynamic web page does not physically exist until a request for this web page is submitted, typically through an HTML form. The existence of dynamic web pages creates two problems for generating a navigation graph:

- A potentially infinite number of dynamic web pages may be generated in a web application. If a dynamic web page is directly modeled as a node, the size of a

navigation graph may be infinite. For example, after a user logs in, a web application may dynamically generate a personalized web page to greet the user. Since the number of user accounts can be infinite, the number of personalized web pages generated by the web application can be infinite. As another example, Google searching web application dynamically generates different web pages according to potentially infinite user requests.

- Some dynamic web pages may not be reached unless appropriate requests (or correct input combinations) are supplied. In other words, in order to ensure coverage, user requests must be generated carefully during the construction of a navigation graph. For example, consider a web application where a user can log in as a regular user or an administrator. Web pages that can only be visited by an administrator would be missed if a user do not log in with the correct administrator username/password combination.

The first problem is referred as the web page explosion problem that is popular in the implementation of dynamic web applications. In dynamic web applications, many web pages may be generated from the same template, as the dynamically generated greeting web page example discussed above. Therefore, these web pages share the same navigation structure. If a navigation graph includes every web page, it may incur a lot of redundancy, which causes two problems: 1) enormous navigation paths to test, which is not practical, and 2) wasting effort in testing lots of redundant web pages.

The second problem is referred as the request generation problem (or input combination dependent navigation structure capture problem). Considering that a request often consists of multiple parameter values, the request generation problem can be further divided into two smaller problems: (1) How to select appropriate values for individual parameters? (2) How to effectively combine individual parameter values to generate requests? In this chapter, we focus on the second aspect of the request generation problem, i.e., how to combine parameter values to generate requests, assuming that these values are from default values and

option values in web pages or manually generated by users with techniques, e.g., domain analysis and equivalence partitioning.

To the best of our knowledge, little work has been reported on the effective construction of navigation graphs. However, the above two problems have been encountered and addressed in a similar context, i.e., web crawling. Web crawling refers to discovering useful information by navigating through web applications. Many web applications store information in a database, and provide the user with an HTML form, through which a query can be submitted to retrieve information of interest. Therefore, like navigation graph construction, web crawling also has to deal with the challenge of how to interact with forms. However, unlike the navigation graph construction, which is interested in "structure discovery", i.e., how different web pages interconnect with each other, web crawling is interested in "content discovery", i.e., how to discover useful information that is contained in those web pages. This difference has a profound impact on techniques that are developed in the two different contexts.

In this chapter, a combinatorial approach to building navigation graphs is proposed to solve the above two problems. To address the web page explosion problem, we use the notion of an abstract URL. Conceptually, a (concrete) URL can be broken into two components, *base* and *query*, as is shown in Figure 3.1. The *query* component is optional, and typically consists of a set of parameter-value pairs. Given a (concrete) URL *u*, the abstract URL for *u* is obtained by removing the values, but retaining the parameter names, in the *query* component. For example, given a URL *u* = "*http://test.com/foo.jsp?x=1&y=2*", the abstract URL is "*http://test.com/foo.jsp?x&y*". In this approach, web pages that have the same abstract URL are represented as a single node in a navigation graph. The rationale behind this abstraction is that these web pages are likely to be generated from the same template, and are thus similar in their structures and associated navigation behavior (i.e., they have the same set of predecessor/successor web pages). For real-life web applications, this abstraction allows us to

bound the number of nodes, and in turn the size of the navigation graph, while largely preserving the navigation structure.

To address the request generation problem, we assume that individual parameter values are given, and use a combinatorial testing technique to combine the parameter values. Assume that a form has *k* parameters, each with *d* possible values. To reach every possible web page that could be generated by submissions of this form, we could try to submit the form with every possible combination of values of those parameters. Doing so, however, can be prohibitively expensive, due to a potentially large number of combinations. In this approach, we submit the form with a subset of parameter value combinations that achieve a well-defined combinatorial coverage, namely *t*-way (*t*=1,2,3) combinatorial coverage [13]. That is, given any *t* out of the *k* parameters, we ensure that every combination of the *t* parameters is covered in at least one submission test. (In the remainder of this chapter, we will refer to a combination of values of all *k* parameters as a submission test, and a combination of values of any *t* parameters as a combination, unless otherwise specified.)

The 2-way combinatorial coverage has been shown to be very effective for general software testing, while dramatically reducing the number of tests that need to be executed [13][14][33]. In particular, empirical studies indicate that 2-way combinatorial coverage leads to 84% branch coverage and 93% code coverage in general software testing [7]. Based on the empirical study results, we assume achieving *t*-way (*t*=1,2,3) combinatorial coverage on form parameters can also help to achieve good web application navigation structure coverage. The argument for this assumption is that parameter interactions may just exist among a few parameters in a form. Therefore, submitting tests that achieve *t*-way combinatorial coverage on form parameters is enough to trigger most transitions behind web forms. As a consequence, navigation structures representing these transitions will be captured. Figure 3.2 presents a concrete example from the *Bookstore* application [45]. To create a new account, a user has to input "*password*" and "*confirmation password*" in the "*Reg*" form of the "*Registration*" page. After

the user submits the form, statements {1, 2, 3, 4, 5} will be executed to check the interactions between these two parameters. If the "*password*" is the same to the "*confirmation password*", statements {8, 9} will be executed. As a result, the user will be navigated to the "*Default*" web page by following the transition "*Registration -> Default*". Otherwise, statements {6, 7} will be executed and the user will go to the "*Registration*" web page again by following the transition "*Registration -> Registration*". This example shows the relationship between code coverage and navigation structure coverage. Empirical study results have already presented the correlation between the *t*-way input combination coverage and good code coverage [7]. Therefore, we assume that proper manipulation on form parameter values can also improve the navigation structure coverage in web applications.

```
1.  ……
2.  String fldmember_password = getParam(request, "member_password");
3.  String fldmember_password2 = getParam(request, "member_password2");;
4.  ……
5.  if ( ! fldmember_password.equals(fldmember_password2)) {
6.  sRegErr += "\nPassword and Confirm Password fields don't match";
7.      …… }
8.  else{
9.    response.sendRedict("Default.jsp")}
```

Figure 3.2 A Simplified Example from Bookstore Application.

Table 3.1 summarizes another four real-life examples that need specific input combinations to trigger input combination dependent transitions behind web forms. If the input combinations meet the constraints in Table 3.1, the user will be navigated to a normal web page. Otherwise, the user will be navigated to a web page for error handling. Thus, from what has

been discussed above, we believe that it is valuable to apply *t*-way coverage on the form input parameters, which can help capture web application navigation structure effectively with reasonable effort.

Table 3.1 Real-life Examples for Parameter Interactions

Subject	Constraints from subject web applications
On-line registration system (etrade.com)	*email* and *confirm email* have to be the same.
Flight ticket booking system (studentuniverse.com)	1) *depart date* should not be late than *return date*. 2) *depart city* should not be the same to *arrival city*.
On-line payment system (citicards.com)	*payment amount* should be no more than the *current balance amount*.
On-line transfer System (chase.com)	*payment account number* should be the same to *confirm payment account number*.

3.3 A Combinatorial Approach for Navigation Graph Generation

In this section, the combinatorial approach to building navigation graphs will be introduced in details. Section 3.3.1 gives a formal definition of a navigation graph. Section 3.3.2 presents an algorithm that implements our approach. Section 3.3.3 provides additional discussions.

3.3.1. Basic Concepts

First, we define a navigation graph. Intuitively, a node in a navigation graph represents a group of web pages that have the same abstract URL. Recall from Section 3.2 that we abstract a URL by removing the parameter values, while retaining the parameter names, in the *query* component, if this component exists. (If a URL does not have a *query* component, its abstraction is the same as the URL itself.) Abstracting URLs helps to control the web page explosion problem while preserving the navigation structure of a web application. There exists an edge from one node *n* to another node *n'* if there is a direct transition from a web page *P* represented by node *n* to a web page *P'* represented by node *n'*, i.e., web page *P'* can be immediately visited after web page *P*.

In the following, we formalize the definition of a navigation graph. Let *abs*(*P*) denote the abstract URL of a web page *P*. Let *pages*(*n*) denote the group of web pages represented by a node *n*. Let $P \rightarrow P'$ denote a direct transition from a web page *P* to a web page *P'*. Then, a navigation graph *G* can be formally defined as follows: $G = (V, E)$, where (1) *V* is a set of nodes such that for each node $n \in V$, $\forall P, P' \in pages(n)$, $abs(P) = abs(P')$; and (2) $E \subseteq V \times V$ is a set of edges such that for each edge (*n*, *n'*), there exists at least one direct transition $P \rightarrow P'$, where $P \in pages(n)$, $P' \in pages(n')$.

3.3.2. Algorithm BuildNavGraph

Figure 3.3 shows algorithm *BuildNavGraph*, which implements our approach. Algorithm *BuildNavGraph* incorporates another two sub algorithms, i.e., algorithm *Traverse* in Figure 3.4 and algorithm *ProcessForm* in Figure 3.5. It takes as input the URL of the *home* page of a web application, and produces as output the navigation graph of the web application. Algorithm *BuildNavGraph* explores a web application in a depth-first manner, so it has a framework that is similar to that of a classic depth-first search algorithm. Therefore, we will not explain the algorithm line by line. Instead, we will focus on how algorithm *BuildNavGraph* differs from a classic depth-first search algorithm.

First, algorithm *BuildNavGraph* uses a different approach to decide whether to explore a newly encountered URL (line 5 in Figure 3.4 and line 19 in Figure 3.5). Specifically, a newly encountered URL is explored only if its abstraction does not yet exist in the navigation graph. In other words, we will not explore a newly encountered URL *u'*, with *abs*(*u'*) = *abs*(*u*), has been explored before. This is necessary to ensure that the exploration process comes to an end. However, it also introduces a risk of missing some web pages that may be reached only if *u* is actually explored. More discussions on this risk, as well as an optimization that can reduce this risk, will be provided in Section 3.3.3.

Second, a classic depth-first search algorithm is designed to traverse all the nodes in a navigation graph. As a result, it usually does not keep track of all the edges that are visited

during the search process. In other words, a classic depth-first search is usually used to build a spanning tree of the original graph. This is different from our algorithm, which tries to capture the entire navigation graph structure. Therefore, it is important to add the corresponding edges into the resulting navigation graph (line 4 in Figure 3.4 and line 18 in Figure 3.5) even if a newly encountered URL will not be explored (because its abstraction already exists in the graph).

Third, the algorithm *ProcessFrom*, as is shown in Figure 3.5, deals with each form, which represents the key contribution of our approach. A web page may contain multiple forms, each of which is dealt with by one call of *ProcessFrom*. Suppose that we are dealing with form *f*. We first obtain the values of individual parameters in form *f* (line 2 in Figure 3.5). This can be done either interactively, i.e., asking the user to provide the parameter values as each form is encountered, or up front, i.e., asking the user to predefine the possible values for each parameter that may appear in a form. Note that the latter can be extremely useful for test automation, but requires a priori knowledge about what parameters may appear in a web application, as well as what values those parameters can take.

In Figure 3.5, the first loop (lines 4 to 21) deals with each action in the form. An action may or may not require parameter values to be submitted to the server side. If an action does require parameter values to be submitted, we will generate a 2-way submission test set for those parameters. Each submission test is then used once to perform the action. If an action does not require any parameter value to be submitted, then it can simply be performed, after which the URL of the succeeding web page is added to list *l*. Note that list *l* is used to hold all the URLs of the succeeding web pages that can be reached from the current web page either through a link or a form submission.

Finally, after we finish exploring a node, we need to backtrack to its parent node *p*, in Figure 3.4. Before we explore another child node of node *p*, it is important to restore the state of the application, e.g., the session state and database state, back to the state when *p* was encountered but none of its children had been explored (line 10 in Figure 3.4). This state

restoration ensures the exploration process to be semantically correct, as the exploration of different children of a particular node should be independent from each other.

In general, there are two approaches to state restoration. The first approach is referred to as the *snapshot-based* approach. In this approach, a state is explicitly represented that contains all the information that may affect the future behavior of the subject web application, e.g., the values of program variables. When a state is first encountered, we save all the information contained in the state. This information is later used to restore the state as needed. This approach typically requires access to the source code.

The second approach is referred to as the *execution-based* approach. In this approach, a state is not explicitly represented. Instead, a state is associated with a sequence of transitions that can be executed to restore this state. This approach can be further classified into two approaches. Let *S* be a state to be restored. The *forward execution-based* approach saves a sequence *Q* of transitions that were executed to reach state *S* from the web application's initial state, and restores state *S* by restarting the web application (thus bringing the web application back to the initial state) and then re-executing *Q*. *The backward execution-based* approach saves a sequence *Q'* of transitions that were executed from state *S* to the current state *S'*, and restores state *S* by rolling back the execution of *Q'* from *S'*. Assume that $Q' = t_1 t_2 \ldots t_n$. To roll back the execution of *Q'*, we execute a sequence $Q'' = t_n' t_{n-1}' \ldots t_1'$ of transitions from *S'*, where t_i' is a reverse transition of t_i, i.e., t_i's performs an operation that undoes the effect of t_i. These two approaches can be implemented without access to source code, but it may be time-consuming to re-execute or roll back transitions.

As what has been discussed in the first paragraph of this sub section, we conduct a depth-first exploration of the navigation structure of the subject web application. During the exploration, we only need to back up to a state that was visited earlier in the current path, i.e., not an arbitrary state. Thus, we only need to store transitions executed on the current path in the *forward execution-based* approach or their reverse transitions in the *backward execution-*

based approach. Note that all the transitions executed on the current path are stored on the search stack. While algorithm BuildNavGraph is presented as a recursive procedure in Figure 3.3, Figure 3.4 and Figure 3.5, it can be easily implemented as an iterative procedure such that the search stack can be accessed.

Both the *forward execution-based* approach, and a *hybrid* approach that combines the *snapshot-based* approach and the *backward execution-based* approach have been implemented in our tool, named *Tansuo*. Their implementation details will be introduced in Section 3.4.

Now, we consider the time and space complexity of algorithm *BuildNavGraph*. The time complexity is similar to that of a classic depth-first search, except that we need to take into account the time for generating *t*-way submission test sets and for performing those tests. Assume that a form has at most *k* parameters, each of which takes at most *d* values. The size of a *t*-way submission test set is $O(d^t \log k)$. The time for generating those tests is $O(d^{t+1}k^{t-1}\log k)$, if the *IPOG* algorithm [35] is used. The time for performing all those tests is $O(\tau d^t \log k)$, where τ is the longest time required to perform a submission test. Therefore, the total time complexity of algorithm *BuildNavGraph* is $O(|G|+d^{t+1}k^2\log k + \tau d^t \log k)$. The space complexity of algorithm *BuildNavGraph* is the same as that of a classic depth-first search algorithm, i.e., $O(|G|)$.

Algorithm BuildNavGraph

Input: The URL of the *home* page of a web application

Output: The navigation graph $G = (V, E)$ of the application

```
1. BuildNavGraph(URL home) {
2.    let G = <V, E> be an empty graph
3.    Traverse (home, G)
4.    return G
5. }
```

Figure 3.3 Algorithm BuildNavGraph.

Algorithm Traverse

Input: The URL of a page and the navigation graph *G* of an application

```
1. Traverse (URL u, Graph G) {
2.    for each static link u' in u {
3.       add a node labeled with abs(u') into V,  if it does not exist
4.       add an edge labeled with (abs(u), abs(u')) into E, if it does not exist in E
5.       Traverse (u', G) if abs(u') is encountered for the first time
6.    }
7.    for each form f in u {
8.       ProcessForm(f, G);
9.    }
10. restore the application to the state reached right after u is encountered
      (but not traversed yet)
11.}
```

Figure 3.4 Algorithm Traverse.

Algorithm ProcessForm

Input: The form *f* and URL of the navigation graph *G* of an application

```
1. ProcessForm (Form f, Graph G) {
2.    obtain the values of individual parameters in f
3.    let l be an empty list
4.    for each action a in f {
5.       if (action a requires submission of param values) {
6.          generate a t-way submission test set s for action a
7.          for each submission test t in s {
8.             perform action a with test t
9.             add the URL of the succeeding page to list l
10.         }
11.      }
12.      else {
13.         perform action a
14.         add the URL of the succeeding page to list l
15.      }
16.      for each URL u' in list l {
17.         add a node labeled with abs(u') into V, if it does not exist in V
18.         add an edge labeled with (abs(u), abs(u')) into E, if it does not exist in E
19.         Traverse (u', G) if abs(u') is encountered for the first time
20.      }
21.   }
22. }
```

Figure 3.5 Algorithm ProcessForm.

3.3.3. Discussions

There are several cases in which our approach may not fully capture the navigation structure of a web application. First, in our approach, web pages having the same abstract URL are represented as a single node in a navigation graph. As an abstract URL drops all the parameter values in the *query* component of a (concrete) URL, we assume that the navigation behavior of a web page, in terms of the set of web pages that could be reached from this web page, does not depend on specific parameter values. This may not be true for some web applications. In addition, an abstract URL does not contain information about web application states, e.g., the values of session variables. The same web page may have different navigation behavior depending on different web application states and such navigation behavior may not be captured by our approach. Adding more information to the abstract URL, i.e., making the abstraction finer-grained, would help capture more navigation behavior. However, doing so may prolong the exploration process, and may unnecessarily increase the size of a navigation graph. This is because many nodes can have the same navigation behavior even if they have different parameter values and/or are visited at different application states.

There is an optimization that can be done to make a navigation graph more complete without adding more information to an abstract URL. In algorithm *BuildNavGraph*, a newly encountered URL is explored only if its abstract URL does not yet exist. We can change this decision so that a newly encountered URL is explored only if the abstract edge leading to the URL does not yet exist. The rationale for this change is that a web page that is reached by a different edge may likely be a different web page, even if another web page with the same abstract URL has been visited before. The reason has been discussed in the first paragraph in this sub section. This optimization has been used in the experiments in the Section 3.5, and has been shown to be very effective.

There is a second case in which the navigation structure of a web application may not be fully captured. In order to explore all web pages that could be generated by a form, we perform a set of submission tests that achieve *t*-way combinatorial coverage. Obviously, *t*-way combinatorial coverage does not cover all the combinations. A web page would not be explored if it could only be generated by submitting one or more specific combinations that do not appear in the *t*-way submission test set. Theoretically, achieving a higher degree of combinatorial coverage will help to make the resulting navigation graph more complete. To investigate the value of different degrees of combinatorial coverage, we implemented the *BuildNavGraph* algorithm with *t*-way combinatorial coverage (*t*=1,2,3). Comparison experiments on *t*-way coverage have been conducted in Section 3.5. Experiment results show that 2-way combinatorial coverage has already captured the most navigation structure. It means 3-way combinatorial coverage does not contribute to capturing any more navigation structure.

In spite of the cases just described, our experimental results, as presented in the Section 3.5, suggest that our approach produces close to complete navigation graphs for the web applications we studied and the costs are affordable.

3.4 Tansuo: A Prototype Tool

To facilitate exploring navigation structures of web applications, a tool, named *Tansuo*, has been developed to explore web applications automatically. *Tansuo* is programmed in Java, and integrated with *HTTPUnit* to explore navigation structures (*HTTPUnit* provides convenient interfaces for users to manipulate components in web pages.). As what has been discussed in Section 3.3, two different state restoration approaches, i.e., the *forward execution-based* approach and the *hybrid* approach, have been implemented for different situations. In the following, they will be discussed separately.

3.4.1. Tansuo's Architecture

Figure 3.6 presents the architecture of *Tansuo*. *Tansuo* includes seven components: *Builder*, *Fetcher*, *Parser*, *Form Handler*, *ACTS*, *State Manager* and *Viewer*.

Builder. This component is the core of *Tansuo*. It coordinates the other six components to explore the navigation structure of a web application. First, *Builder* calls *Fetcher* to obtain a designated web page from a web server. Second, *Builder* calls *Parser* to extract transitions from this web page, and *Builder* will follow those transitions to explore the web application further. Third, after exploring a navigation path, *Builder* calls *State Manager* to restore application states before it explores a new navigation path.

Fetcher. This component is responsible for fetching a web page from the server side, upon *Builder's* request. In practice, the *Fetcher* can adapt to various web applications, e.g., JSP, ASP and PHP web applications.

Parser. This component is responsible for parsing a web page. Components in the web page are wrapped into objects, which facilitates handling these components. *Parser* extracts all components related to transitions, i.e., links and forms, existing in the current web page and returns them to *Builder*.

Form Handler. This component is responsible for interacting with forms. Specifically, *Form Handler* is responsible for three tasks: (1) Obtaining parameter values either from a user interactively or by reading them from a set of XML files; (2) Generating parameter combinations by using a combinatorial test generation tool, called *ACTS* [34][35]; (3) Submitting a form with those combinations.

Note that in the interactive mode, *Tansuo* pops up a GUI for users to input form parameter values when a web form is encountered. *Form Handler* also stores user-provided values into XML files, so that those values can be reused later. To help users to generate parameter values, an auxiliary component, named *Collector*, has been developed to gather

default values, option values in the current web page. The collected parameter values will be presented to users as input candidates.

ACTS. This component is the engine for generating combinations that achieve *t*-way combinatorial coverage on form parameters. *ACTS* [35] gets form parameter values from *Form Handler* and gets the degree value of combinatorial coverage from *Builder*. Then it generates submission tests that achieve *t*-way combinatorial coverage and returns them to *Form Handler*.

State Manager. This component is responsible for restoring the application state. When *Builder* backs up to a previous web page, it calls *State Manager* to restore the application state before exploring a new navigation path. This state restoration is necessary because exploring the previous navigation path may have already changed the application state. Two state restoration approaches have been implemented for different situations (in terms of the availability of source code). Their details will be discussed in Section 3.4.2.

Viewer. This component is responsible for displaying a web page that is currently being explored by *Builder*. When a user is asked to provide parameter values for a form, displaying the current web page helps the user to understand the context better.

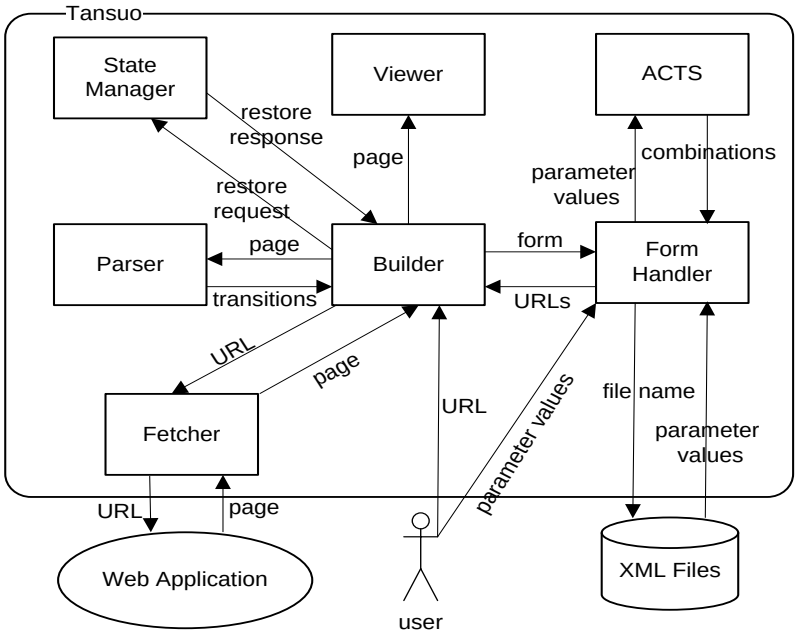

Figure 3.6 Tansuo's Architecture.

3.4.2. State Restoration Approaches

Tansuo implements two different state restoration approaches, i.e., the *forward execution-based* approach and the *hybrid* approach, for different situations. The *forward execution-based* approach can work without access to the source code. But its performance may not be good when working for large web applications. The *hybrid* approach takes advantage of the available source code to improve the performance of *Tansuo*. The implementation details of these two approaches will be introduced in the following. An experimental comparison of these two approaches is presented in Section 3.5.

Forward Execution-Based State Restoration: In this approach, we represent a state by the sequence of concrete URLs visited to reach the state. Each time we visit a new web page, we save the concrete URL of this page into the search stack. When we back up to a page *p* from a page *p'*, we first pop out the URL of page *p'*. Then, we restart the web application on

the server side. If the web application uses a database in the backend, the database is also reset.

Next we re-submit the remaining URLs in the search stack from bottom to top, i.e., in the order as they were pushed into the stack. Doing so re-exercises the navigation path from the initial page to page *p*. This effectively restores the state reached when *p* was first visited. At this point, we are ready to explore the next child of page *p*. Re-submitting the URLs on the search stack may be time consuming. However, this approach is the only choice if the source code of a web application is not available.

Hybrid State Restoration: This approach combines the *snapshot-based* approach and the *backward execution-based* approach. We target web applications written using JSP and Java servlet technologies. We consider that a state consists of two components: (1) the values of all session variables; and (2) the state of the database, i.e., all the records in the database, if a database is used for persistent storage in the backend. It is possible that a state may contain other information. For example, if the web application reads and/or writes a file, this file should be included in the state of the web application. We believe it is common for practical web applications to use a database, instead of files, in the backend. This is the case for all the web applications used in our case studies.

In order to achieve maximal efficiency, we treat the two state components differently, as is shown in Figure 3.7. Specifically, we use the *snapshot-based* approach to restore the values of session variables, and we use the *backward execution-based* approach to restore the state of the database. The details of our implementation are described below.

Instrumentation: We use source code instrumentation to insert additional code into the subject web application. At runtime, it is this additional code that is executed to save and restore states as requested by the *Builder*. During instrumentation, we first identify all the session variables and database operations. A variable is identified to be a session variable if it is

declared to be of type HttpSession. We assume that JDBC is used to communicate with the database. Thus, all the JDBC calls are identified to be database operations.

Next, we generate and insert three types of code into the source code of the subject application:

(1) SessionStateManager: Code for saving and restoring the values of session variables. This code saves the values of session variables into a log file and restores these values at a later point as requested. This code is inserted at the beginning of each web page.

(2) JDBCWrapper: Code for recording JDBC calls. For each JDBC call encountered in a web page, we replace it with a call to a wrapper. At runtime, this wrapper intercepts three JDBC calls, including *insert*, *update* and *delete*. Whenever this wrapper receives a request for one of the three JDBC calls, it records the corresponding reverse call, as well as information needed to execute this reverse call, into a log file, before it delegates to the actual JDBC call.

Table 3.2 shows the reverse calls of the three JDBC calls we intercept. It also shows the information we must save for each reverse call. For update and delete, we have to retrieve the record(s) before they are actually updated and deleted. In practice, these three calls often deal with a single record. JDBC calls that deal with multiple records are likely to be query calls, i.e., they do not change the database. Thus, the information we must save for a reverse call is typically limited.

Table 3.2 Information About Reverse JDBC Calls

Original JDBC call	Reverse JDBC call	Information to be saved
insert	delete	primary key(s) of the record(s) to be inserted
update	update	all the fields of the record(s) to be updated
delete	insert	all the fields of the record(s) to be deleted

(3) DBStateManager: Code for restoring the state of the database. This code executes a sequence of reverse JDBC calls to restore the database to a previous state. This code is inserted at the beginning of each web page.

State Management Protocol: The *Builder* collaborates with the instrumented subject web application to manage the state restoration process. Specifically, the *Builder* defines three new tags, including *RestoreState*, *OldStateID*, and *NewStateID*. The *RestoreState* tag is always added into each HTTP request, indicating whether a state needs to be restored. The *OldStateID* tag is added into a request only if *RestoreState* is "true", and it identifies which state to be restored. The *NewStateID* tag is always added into a request to identify the new state that will be saved as a result of executing a new web page. The *Builder* is responsible for the ID management, i.e., assigning proper values to the two tags.

At runtime, the protocol proceeds as follows. The *Builder* submits a URL request to the web server. The web server locates the web page identified by the URL. Before the web page is executed, *SessionStateManager* and *DBStateManager* are executed. Recall that the two managers are added to the beginning of each web page. *SessionStateManager* first checks whether *RestoreState* is set to "true". If so, it restores the session state identified by *OldStateID*. Regardless of the values of *RestoreState*, *SessionStateManager* always saves the current session state into a log entry identified by *NewStateID*. If *RestoreState* is set to "true", then the session state that was just restored will be saved again, but with the *NewStateID*. Like *SessionStateManager*, *DBStateManager* restores the database state if *RestoreState* is set to "true". However, it does not save the database state, which is left to JDBCWrapper. Only after the two managers are executed is the actual requested web page executed. During the execution of the web page, JDBCWrapper saves, for each JDBC call, the corresponding reverse call with the necessary information into a log entry identified by *NewStateID*.

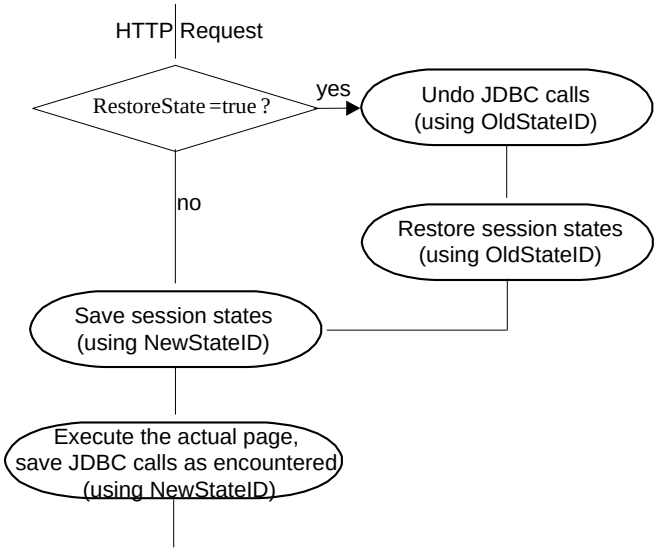

Figure 3.7 The workflow of hybrid state restoration

3.5 Experiments

We used *Tansuo* to generate navigation graphs for five web applications [45] and measured its runtime performance and costs. In Section 3.5.4, we first studied the effectiveness and costs of *t*-way (*t*=1,2,3) combinatorial coverage. Then, we evaluated the completeness of the generated web navigation graphs on the five web applications. We also compared *Tansuo* with *WebSphinx* [42] and *Link Checker Pro* [36] on all five web applications..

3.5.1. Research Questions

RQ1. How effective is *Tansuo* with *t*-way (*t*=1,2,3) combinatorial coverage on navigation graph generation?

RQ2. How efficient of *Tansuo* in terms of time and memory costs?

RQ3. How complete is a navigation graph built by *Tansuo*?

RQ4. How do the two state restoration approaches compare with each other?

RQ5. How does *Tansuo* compare with existing tools?

3.5.2. Metrics

For RQ1, we compare the effectiveness of *t*-way (t=1,2,3) coverage by measuring detected nodes and edges in web navigation graphs. For RQ2, we compare time costs and memory costs of different combinatorial coverage criteria and total combinations tried in a whole exploration process. For RQ3, the completeness of a generated navigation graph is measured by the number of missed nodes and edges when compared with a complete navigation graph generated by a manually performed static analysis on the source code. For RQ4, we run *Tansuo* on each web application twice. At the first time, *Tansuo* uses the *forward execution-based* approach to restore the application state. At the second time, *Tansuo* uses the *hybrid* approach to restore the application state. At last, we compare their generated navigation graphs and their time and memory costs during the navigation graph generation. For RQ5, we compare *Tansuo* with two other tools - *WebSphinx* and *Link Checker Pro* - by recording the number of nodes and the number of edges generated by each tool to capture navigation structures of web applications.

3.5.3. Experimental Setup

Subject Applications: We used five web applications [45]: *Bookstore*, *Bug Tracking System* (*Bugtrack*), *Classifieds*, *Links* and *Portal* in our experiments. Table 3.3 shows some server-side characteristics of the subject applications, including the number of non-commented lines of code (NLOC), classes, methods, and branches in the five web applications. All these statistics were collected with *Clover* [12]. Table 3.4 shows some client-side characteristics of the subject applications, including the number of forms, actions, parameters (*params*), the average number of parameters per action (*APA*), and the average number of values per parameter (*AVP*). Note that these client-side factors affect the size of navigation graphs, as well as the time for building navigation graphs of these web applications. We note that all of the applications are implemented in JSP.

Test data generation: As what has been introduced in Section 3.2, this approach focuses on how to generate requests by effectively combining form parameter values. As for the form parameter value generation, we used existing techniques, e.g., domain analysis and equivalence partitioning. The parameter generation includes two major steps. First, we prefer using data from web pages, default values from text fields, option values from menus, option values from radio buttons and check-box buttons. In reality, some parameters have dozens of values, e.g., state names, city names, and employee names. In this case, we picked a few values out of these option values for efficiency. Second, as for parameters without default values and option values, we generated input data for them manually with existing techniques. All the input values used in experiments have been verified manually. They can be processed properly by web applications.

In experiments, we generated only normal input values for form parameters. We did not use malicious input values, even though navigation graphs generated with both normal input values and malicious input values will likely be more complete than the navigation graphs generated with only normal input data. The reason for using only normal input data in experiments is that there is no good way to systematically generate malicious input data. Generating test data ad hoc may cause the unfairness in comparison experiments. Fuzz testing [67][69] may help to generate anomalous input data automatically. But there are two major disadvantages with the input data generated with fuzz testing techniques. First, the quality of generated input data is poor. Input data generated from fuzz testing may be easily rejected for syntax problems, which may lead to capturing only shallow the navigation structure. However, the goal of our approach is to capture the deep navigation structure behind web forms. Second, the input data generated from fuzz testing may achieve different effectiveness in capturing the navigation structure because of the different implementations of syntax checking. This fact may cause unfair comparisons between different combinatorial coverage criteria.

Submission test generation: This approach generates *t*-way (*t*=1,2,3) combinatorial coverage tests with *IPOG* algorithm [35]. In Chapter 1, a 2-way combinatorial coverage test generation example has been demonstrated. A 3-way combinatorial coverage test generation is similar to the 2-way combinatorial coverage test generation, except for the stronger degree of the combinatorial coverage. In the 3-way combinatorial coverage test generation, we have to generate submission tests that cover all the combinations of any three parameters.

The 1-way combinatorial coverage requires that every value of a parameter be covered at least once by the generated tests. Each choice strategy [27] is the simplest 1-way combinatorial coverage test generation strategy. However, in reality, base choice strategy [27] is more popular in software testing. Thus, in experiments, we used base choice strategy to generate 1-way combinatorial coverage tests.

Base choice strategy generates tests with four major steps. First, identify parameters and their corresponding values. Second, designate a base value for each parameter. Usually, the base value is the most common value of a parameter. In our experiments, we used a default value from a web page as the base value for a parameter. As for the parameter that has no default value, we designated one value as the base value, based on our domain knowledge. Third, generate a base test with the base value of each parameter. Fourth, derive new tests by changing the base value of one parameter to one of its non-base values. The fourth step will be repeated until all the values of every parameter have been covered.

To illustrate base choice strategy clearly, we use an example to show how the base choice strategy works in the following. First, in this example, each parameter, i.e., *P1*, *P2*, and *P3*, in this example has two values “0” and “1”. Second, designate “0” as the base value as for each parameter. Third, generate a base test, i.e., {0,0,0}. Fourth, derive new tests from the base test, i.e., {1,0,0}, {0,1,0} and {0,0,1}. As is shown in Figure 3.9, every value of each parameter has been included in at least one submission test.

Machine Configuration: The experiments were carried out on a computer with the following configuration: CPU: 1.66GHz, RAM: 2G, Hard disk: 80G. The machine was running Windows XP SP2, the Resin 2.1.8 web server, Apache 2.0.48, and the MySQL Server 4.1.

Table 3.3 Server-side Characteristics of Subject Applications

Subject	Characteristics			
	NLOC	Classes	Methods	Branches
Bookstore	18385	27	925	4392
Bugtrack	8094	13	438	1946
Classifieds	11599	18	618	2730
Links	8849	13	499	2074
Portal	17621	27	915	4084

Table 3.4 Client-side Characteristics of Subject Applications

Subject	Characteristics				
	Forms	Actions	Params	APA	AVP
Bookstore	18	63	66	1.05	3.35
Bugtrack	8	19	27	1.42	6.15
Classifieds	11	29	27	0.93	5.07
Links	11	24	26	1.08	5.77
Portal	19	39	95	2.44	3.40

p1	p2	p3
0	0	0
1	0	0
0	1	0
0	0	1

Figure 3.8 A 1-Way Submission Test Set.

3.5.4 Results and Discussions

RQ1: Effectiveness of *Tansuo* with *t*-way combinatorial coverage

We evaluated the effectiveness of *Tansuo* with *t*-way (*t* =1,2,3) combinatorial coverage on five web applications with the same test data set. As you can see in Table 3.5 (all the results have been verified by manually checking the source code of subject web applications), 1-way combinatorial coverage causes missing lots of navigation structures. The navigation graphs generated with 1-way combinatorial coverage cover 32.88%-80.49% nodes and 27.91%-72.49% edges, compared with the navigation graphs generated with 2-way combinatorial coverage. After compared with 2-way combinatorial coverage and 3-way combinatorial coverage, we can see that 3-way combinatorial coverage did not help us to capture more navigation structure. We verified the results by manually checking the source code of subject web applications. We did not find the navigation structure that requires 3-way input combinations to capture from their implementation.

Table 3.5 Comparisons on Navigation Graphs Generated with T-Way Coverage

Subject	One-way		Two-way		Three-way	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
Bookstore	24	127	73	455	73	455
Bugtrack	12	45	33	135	33	135
Classifieds	33	195	41	269	41	269
Links	25	95	44	232	44	232
Portal	32	286	66	642	66	642

RQ2: Efficiency *Tansuo* in terms of time and memory costs

Table 3.6 shows the time memory costs (in terms of hours) during generating navigation graphs with *t*-way (*t* =1,2,3) combinatorial coverage criteria. From the empirical results, we can see the time costs of 1-way combinatorial coverage are 28.62%-55.89% of the time costs of 2-way combinatorial coverage. Moreover, Table 3.7 shows that the memory costs (in terms of MB) of 1-way combinatorial coverage are 65.99%-91.84% of the memory costs of 2-way combinatorial coverage.

Recall that *Tansuo* explores a web application in a depth-first manner. The maximum memory usage occurs at one of those back-up points, i.e., after *Tansuo* finishes exploring the current navigation path, and right before it backtracks to explore the next navigation path. We recorded the memory usage at each of these back-up points and reported the highest of these values in Table 3.7.

Table 3.6 Comparisons on Time Cost between T-Way Coverage

Subject	One-way	Two-way	Three-way
Bookstore	0.0469	0.1639	0.1741
Bugtrack	0.0102	0.0235	0.0427
Classifieds	0.0403	0.0721	0.0944
Links	0.0123	0.0344	0.0400
Portal	0.0654	0.3106	0.3542

Table 3.7 Comparisons on Memory Cost between T-Way Coverage

Subject	One-way	Two-way	Three-way
Bookstore	30.6562	46.4585	49.3367
Bugtrack	19.9174	21.6875	21.8633
Classifieds	24.2109	39.9023	39.9844
Links	14.6800	21.2383	22.2852
Portal	64.8672	84.3750	86.3984

Table 3.8 Comparisons on Total Combination between T-Way Coverage

Subject	One-way	Two-way	Three-way
Bookstore	417	1455	2327
Bugtrack	306	978	2130
Classifieds	550	896	1465
Links	242	684	990
Portal	941	2216	3732

In Table 3.6 and Table 3.7, the costs of 3-way combinatorial coverage are similar to the costs of 2-way combinatorial coverage. The reason is that the time cost is mainly determined by analyzing/processing web pages and the memory cost is mainly determined by holding parsed elements, like a form. In Table 3.5, we can see there is no extra navigation structure captured by 3-way combinatorial coverage, compared with 2-way combinatorial coverage. Therefore, it is

reasonable that they have similar time and memory costs. Comparisons between 1-way combinatorial coverage and 2-way combinatorial coverage also demonstrate that time and memory costs increases with the navigation graph size.

Table 3.8 shows that 3-way combinatorial coverage tried much more submission tests than 2-way combinatorial coverage. But that just incurred slightly increases in time and memory costs, which also illustrates that the major factor for the time and memory costs is the generated navigation graph size.

RQ3: Completeness

We evaluated the completeness of the navigation graphs on five web applications, as is shown in Table 3.9. To perform this evaluation, we manually generated complete web navigation graphs, in terms of abstract nodes and edges, from the source code of the five web applications. Each JSP file in the source code was studied and abstract URLs were extracted. Abstract URLs were generated from either form actions or from the value of the attribute *href* of the anchor (<*a*>) tag. Each such abstract URL became a node in the resulting navigation graph. For each abstract URL, the corresponding web page was studied to find transitions to other web pages. These transitions were modeled as edges in the resulting navigation graph.

Note that, 2-way combinatorial coverage has achieved the best trade-off between test coverage and test effort during the navigation graph generation. Therefore, in RQ3, RQ4 and RQ5, we chose the navigation graphs generated with 2-way combinatorial coverage to compare with navigation graphs generated by other approaches.

Table 3.9 shows the numbers of nodes and edges of the manually generated navigation graphs for five web applications. For example, for *Bookstore* application, the navigation graph generated by *Tansuo* missed 12.0% of the nodes and 15.9% of the edges. After carefully studying the navigation graphs generated by the manual exploration and *Tansuo's* exploration, we found that the reason for missing nodes and edges is that *Tansuo* missed some scenarios during its exploration, e.g., the navigation structure for page-flipping. For example, the

"*OrderGrid*" web page, in *Bookstore*, lists orders placed by a user. If total orders are no more than 20, all of them will be listed in one web page. But, if a user placed 25 orders, the first 20 orders will be listed in the current web page and, the last 5 orders will be listed in the second web page. In this case, page-flipping is needed for users to browse all these orders. *Bookstore* places a link in the current web page, so that the user can navigate to the second web page by clicking this link. Initially, there is no order listed in the "*OrderGrid*" web page. During exploration, *Tansuo* only placed one order to drive the exploration for the reason of efficiency. As a result, there was only one order listed in the "*OrderGrid*" web page, and *Tansuo* failed to capture the navigation structure related to page-flipping.

As another example, the "*OrderRecord*" web page, in *Bookstore*, shows detailed information of an order. If a user accesses this web page, the user's access right will be checked. If the user does not have this access right, she/he will be navigated to the "*Login*" web page by following the transition "*OrderRecord -> Login*". However, this navigation path is missed by *Tansuo*. The reason is that before accessing the "*OrderRecord*" web page, *Tansuo* has already logged in as the user that has this access right.

Table 3.9 Completeness Experiment Results

Subject	Manual		Tansuo			
	Nodes	Edges	Nodes	%	Edges	%
Bookstore	83	541	73	88.0	455	84.1
Bugtrack	37	170	33	89.2	135	79.4
Classifieds	51	327	41	80.4	269	82.3
Links	47	257	44	93.7	232	90.2
Portal	75	788	66	88.0	642	81.5

RQ4: Comparisons between the *forward execution-based* approach and the *hybrid* approach

In [76], we have pointed out that the application state restoration process was the most time consuming activity in the *forward execution-based* approach. The time taken to restore the application state includes the time taken to reset the database and the time to re-exercise the path from the *home* page to the current web page. A large number of submission tests for a

form can significantly increase the state restoration time, since the application state has to be restored before each submission test is performed.

The *hybrid* approach is proposed to solve the performance problem. As is shown in Table 3.10, the *forward execution-based* approach and the *hybrid* approach generated the same navigation graphs. But, in Table 3.11, we can see the time cost of the *hybrid* approach has been reduced dramatically. Meanwhile, the memory overhead of the *hybrid* approach has increased 2.29%-10.86%, which is affordable. There are two reasons for the extra memory overhead: 1) running extra instrumentation code requires extra memory; 2) keeping state restoration data also requires extra memory.

Table 3.10 Comparisons on Graph Statistics between Two State Restoration Approaches

Subject	Forward execution		Hybrid	
	Nodes	Edges	Nodes	Edges
Bookstore	73	455	73	455
Bugtrack	33	135	33	135
Classifieds	41	269	41	269
Links	44	232	44	232
Portal	66	642	66	642

Table 3.11 Comparisons on Costs between Two State Restoration Approaches

Subject	Forward execution		Hybrid	
	Time(Hour)	Memory(M)	Time(Hour)	Memory(M)
Bookstore	33.4415	42.6318	0.1639	46.4585
Bugtrack	0.1321	19.5625	0.0235	21.6875
Classifieds	0.2999	39.0078	0.0721	39.9023
Links	0.1275	19.4570	0.0344	21.2383
Portal	1.2218	80.3544	0.3106	84.3750

RQ5: Comparisons with other tools

We compared *Tansuo* with *WebSphinx* [42] and *Link Checker Pro* (*LCP*) [36]. Note that *WebSphinx* and *LCP* do not handle forms. In addition, they do not make any web page URL abstraction. For example, *"http://test.com/BookDetail.jsp?item_id=1*" and "*http://test.com/BookDetail.jsp?item_id=2*" are identified as two different web pages in their

navigation graphs. If there are thousands of such web pages in a web application, the navigation graphs generated by *WebSphinx* and *LCP* will be very huge, while contributing little to represent the navigation structure of a web application. Table 3.12 shows the results of our comparisons. Both *WebSphinx* and *LCP* generated similar numbers of nodes and edges, while *Tansuo* generated significantly more nodes and edges than *WebSphinx* and *LCP*. This suggests that the ability to interact with forms is vital to build high-coverage navigation graphs.

It would be ideal, if comparisons were made with tools that handle web forms, like *VeriWeb*. Unfortunately, we were not able to obtain access to such tools. Alternatively, we can discuss the value of the input combination generation in capturing the navigation structure behind web forms in another way. To answer RQ1, we ran *Tansuo* to generate web navigation graphs with 1-way combinatorial coverage and 2-way combinatorial coverage separately. In these experiments, *Tansuo* handled web forms in the same way. However, as is shown in Table 3.5, the 1-way combinatorial coverage missed 19.51%-67.12% nodes and 27.51%-72.09% edges, compared with 2-way combinatorial coverage. The comparison results show that poor input combination coverage may cause losing lots of navigation structures during the web navigation graph generation, which indicates the value of the *t*-way input combination generation in our approach.

Table 3.12 Comparisons to WebSphinx and LCP

Subject	WebSphinx		LCP		Tansuo	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
Bookstore	11	11	11	11	73	455
Bugtrack	7	7	7	7	33	135
Classifieds	15	16	9	9	41	269
Links	11	12	11	11	44	232
Portal	17	22	17	22	66	642

3.5.5. Threats to Validity

Although our studies investigated *Tansuo* with 5 medium to large web applications, the number, the size (in terms of NLOC), and the specific technologies (HTML, JSP, MySQL) of the

subject web applications prevent a generalization of our results to the entire domain of web applications. All the conclusions are drawn from the experiments we have conducted. Some of them may not be true for other real-life web applications. For example, in answering RQ1, 3-way combinatorial coverage did not help to capture more navigation structures. This conclusion may not be true, if we worked on a more complex web application that requires 3-way or 4-way input combinations. In fact, this kind of web applications is not rare. For example, when a user resets her/his password, the user may be required to input values for "*old password*", "*new password*" and "*confirm password*". If the value of "*old password*" is different from the value of "*new password*", and the value of "*new password*" is the same to the value of "*confirm password*", the user will be navigated to a "*success*" web page. Otherwise, any other input combination of these three parameters will lead to an "*error*" web page. Within this example, 3-way combinations are necessary to capture more navigation structures.

We manually explored the source code to generate web navigation graphs for answering RQ3. Although extreme care was taken to accurately model navigation graphs, the human involved in the exploration process could have made errors when analyzing the source code, which may affect the completeness of the web navigation graphs generated by *Tansuo*.

CHAPTER 4

WEB TEST SEQUENCE GENERATION

4.1 Background

Along with the proliferation of web applications comes a growing amount of concern about the reliability of those web applications. A failure, even partial functionality loss, may cause an entire business to standstill and cost millions of dollars [78]. Also, users' confidence in web applications depends to a large degree on whether their business transactions are handled correctly by web applications. Reliability is considered to be the biggest challenge in the further promotion of web applications [66].

One important aspect of web applications is that they often consist of dynamic web pages that interact with each other by accessing shared objects, e.g., the session objects used to track a user in a sequence of requests and the persistent data storage like a database used to exchange data between different web pages users may access. Interactions between dynamic web pages need to be carefully tested as they may give rise to subtle faults that can not be detected by testing individual web pages in isolation. For example, a session object may be spoiled in web page *P* and this session object would be used later in web page *P'*. Testing web page *P* and web page *P'* individually would not detect this fault, because this fault would not cause any failure when web page *P* does not interact with web page *P'*. To detect this fault, a test sequence that accesses web page *P* first and then accesses page *P'* is required. Therefore, it is valuable to generate test sequences to detect such faults existing in the interactions between web pages in a web application.

4.2 Challenges and Contributions

A straightforward approach to testing interactions between dynamic web pages is to test all possible sequences in a web application. The rationale is that each sequence represents a specific way in which those web pages interact with each other, i.e., one possible interaction among those web pages. There are, however, two problems with this approach. First, because the number of test sequences grows exponentially as the size of a web application increases, it is nearly always impractical to test all possible sequences due to the limited resources and tough deadlines. Second, faults only manifest in a small number of test sequences. Thus, it is inefficient to test all possible sequences, many of which do not contribute to fault detection. Existing research work, as is discussed in Chapter 2, has proposed strategies, like covering all the edges, to detect interaction faults existing in web applications. The existing strategies can not guarantee that all the interactions in a web application would be tested systematically.

In this chapter, we propose a novel web test sequence generation approach to address the above problems. We define the research problem as follows: *Given a web application containing dynamic web pages, how could we generate, in a systematic manner, a small number of test sequences that are effective for detecting interaction faults?* The key idea of this approach is generating test sequences to cover all 2-way interactions, i.e., interactions between any two web pages. In other words, if a web page *P* could reach another web page *P'*, there must exist one test sequence in which both *P* and *P'* are visited in the given order (but not necessarily in a row). Our approach involves three major steps: First, a graph model is built to capture the navigation structure of a web application under test, where each node represents a web page (or a portion of it), and each edge represents a direct transition from one node to another. This navigation graph generation problem has been solved in Chapter 3. Second, all the 2-way interactions that may occur in a web application are computed from the navigation graph. Finally, a set of paths are selected from the navigation graph to cover all the 2-way interactions. These paths are then used as test sequences to test the web application.

For the purpose of evaluation, we built a prototype tool and applied our approach to two web applications, namely, *Bookstore* [45] and *CPM* [55]. We compared our approach with an approach that generated test sequences to cover all the edges in a navigation graph. We refer to our approach as the *AllOrderedPairs* approach and the latter approach as the *AllEdges* approach. The empirical comparison shows that *AllOrderedPairs* approach can effectively detect some interaction faults that cannot be detected by the *AllEdges* approach.

Our approach is inspired by the research work on 2-way combinatorial testing for detecting functionality faults in the software [7][13][35]. The 2-way combinatorial testing has been shown very effective for detecting general software faults. We believe that the notion of achieving 2-way combinatorial coverage will also be effective for detecting faults in web applications, for which our case studies have provided some initial evidence.

To the best of our knowledge, our work is the first attempt to apply combinatorial testing to generating test sequences for web applications. More importantly, our work deals with two unique challenges in the context of testing web applications. First, for all the existing combinatorial testing approaches, the order in which different components appear in a combination is insignificant. This is in contrast to our approach, where the order of the web pages being visited in an interaction is important. In particular, web page *P* may reach web page *P'*, but web page *P'* may not reach web page *P*, which has been explained with a concrete example in Chapter 1. Second, existing work provides limited support for handling constraints. Constraints are used to exclude invalid combinations, based on the domain semantics, from the resulting test set. In our approach, the possible constraints among different web pages are implicitly captured in the navigation structure represented by a navigation graph. The notion of using a navigation graph to represent interaction constraints, as well as the required algorithmic support, is novel.

4.3 An Interaction-Based Test Sequence Generation Approach

4.3.1 Basic Concepts

We first briefly review the notion of a navigation graph. As what has been introduced in Chapter 3, a navigation graph is used to represent the navigation structure of a web application. A node in a navigation graph can be a static node, which represents a static web page, or a dynamic node, which represents a dynamic web page generated by a form submission. We distinguish the *home* page of a web application as a special node called *home* node. There exists an edge from one node *m* to another node *n* if node *n* can be visited immediately after node *m* through a transition. Note that a transition can be represented by a hyperlink or a form submission in a web page. Formally, a navigation graph *G* can be denoted as $G = (V, E, n_0)$, where $V = V^s \cup V^d$ with V^s being a set of static nodes, and V^d being a set of dynamic nodes, and $E \subseteq V \times V$ is a set of edges, and n_0 is the *home* node.

Note that a dynamic web application could potentially generate an infinite number of web pages. This is because many web pages may be dynamically generated from the same template, according to different requests from users. Theoretically, users' requests may be infinite, which may cause infinite web pages. If a web page was directly represented as an individual node, the size of a navigation graph would be unbounded. This problem has already been solved in Chapter 3.

Next, we introduce the notion of 2-way interaction coverage. The term "2-way interaction" refers to the interaction between two dynamic nodes. Let $G = (V, E, n_0)$ be a navigation graph. Formally, a 2-way interaction in *G* is an ordered pair (*m*, *n*), where *m* and *n* are two dynamic nodes, and there exists a path from *m* to *n* in *G*. Note that static nodes do not access shared objects and thus have no interaction with other nodes. (Static nodes are included in a navigation graph to capture the navigation structure, which is needed to generate executable test sequences.) Also note that the order of nodes in a 2-way interaction is

significant, as a node *m* may reach a node *n*, but the reverse may not be true. (We only consider navigations through links and form submissions within a web application. That is, we do not consider navigations due to actions that are performed on the web browser.)

The 2-way interaction coverage requires that a set of paths be selected from a navigation graph as test sequences so that every ordered pair is covered in at least one of those test sequences. Let $P = n_1 n_2 \ldots n_l$ be a path in a navigation graph. Let $p = (m, n)$ be an order pair. Then, p is covered in P if there exists $1 \leq i < j \leq l$ such that $n_i = m$, and $n_j = n$. Note that in *P*, nodes *m* and *n* must appear in the given order, but they do not need to appear consecutively.

To help better understand the notion of the 2-way interaction coverage, let us compare it with the *AllEdges* coverage. The latter requires that every edge in a navigation graph be covered by at least one test sequence. Figure 4.1 shows an example navigation graph. Two test sequences, *ABDEG* and *ACDFG*, are sufficient to cover all the edges in the graph. But these two test sequences fail to cover two 2-way interactions, namely, (*B*, *F*) and (*C*, *E*). If a fault is only triggered by these two interactions, then this fault would be detected by a test set satisfying the 2-way interaction coverage, but may not be detected by a test set satisfying the *AllEdges* coverage.

It is interesting to note that 2-way interaction coverage does not subsume edge coverage. Figure 4.2 shows a navigation graph that consists of three nodes *A*, *B*, and *C*, and three edges (*A*, *B*), (*A*, *C*), and (*B*, *C*). In this graph, path *P* = *ABC* satisfies the 2-way interaction coverage but does not satisfy the *AllEdges* coverage. The reason is that (*A*, *C*) as a 2-way interaction is covered by path *P*, since *A* and *C* appear in *P* in the given order, but (*A*, *C*) as an edge is not covered by this path, since *A* and *C* do not appear in *P* in a row.

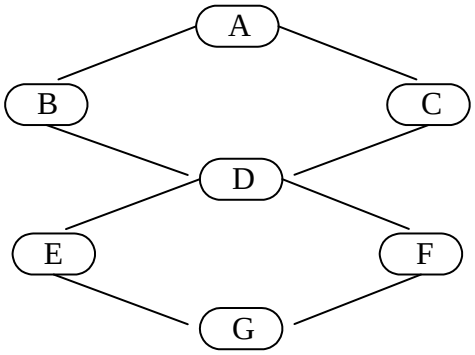

Figure 4.1 Navigation Graph Example 1.

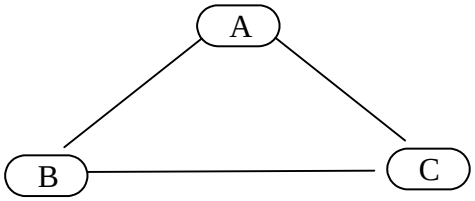

Figure 4.2 Navigation Graph Example 2.

4.3.2. Algorithm Generate-Sequences

Figure 4.3 shows an algorithm called *Generate-Sequences*, which implements the interaction-based test sequence generation approach. Algorithm *Generate-Sequences* takes as input a navigation graph *G* of a web application under test, and produces as output a set *seqs* of sequences that covers all the ordered pairs in *G*. The algorithm begins by computing all the set *pairs* of ordered pairs in the navigation graph (line 1). Note that we only consider ordered pairs involving two dynamic nodes. We point out that this computation basically requires determining reachability from one node to another, which is a classical problem in the graph theory and can be solved using the algorithms that have been reported in the literature [15].

Next, the algorithm tries to generate a set of test sequences to cover all the ordered pairs computed earlier. This is accomplished by a *while* loop (lines 3 to 13) in which each iteration generates one test sequence to cover a set of pairs that have not been covered before until all the pairs are covered. Each iteration of the *while* loop works as follows: First, a list *L* of

nodes is built in which every two adjacent nodes is an ordered pair in set *pairs*, i.e., an ordered pair that has not been covered yet (line 4). The purpose of building this list is to guide the creation of a test sequence *S* so that *S* will cover a good number of ordered pairs that have not been covered yet. An optimal approach would build *L* in a way such that the size of the resulting set of test sequences is minimal. (The size of the resulting test sequence set can be measured in different ways, e.g., in terms of the total number of requests if we ignore the cost of test setup and teardown.) It is easy to see that finding an optimal solution is a NP-hard problem, due to the combinatorial nature of the problem. Here we describe a heuristic approach to building *L*. In this approach, we begin by picking an arbitrary pair (*m*, *n*) from *pairs*, and add *m* and *n* into *L* in the given order, i.e., *L* = (*m*, *n*). Next, we try to extend *L* using the following three rules: (1) if there is a pair (*m'*, *m*) in *pairs*, then we add *m'* into *L* at its the beginning, i.e., *L* = (*m'*, *m*, *n*); (2) if there exists a pair (*n*, *n'*) in *pairs*, we add *n'* into *L* at its end, i.e., *L* = (*m*, *n*, *n'*); (3) if there are two pairs (*m*, *o*) and (*o*, *n*), we add *o* into the middle of *L*, i.e., *L* = (*m*, *o*, *n*). We will refer to the three rules as the *front-end*, *back-end*, and *middle extension*, respectively. These three rules can be easily generalized to keep extending *L* until *L* can no longer be extended, i.e., no more nodes can be added into *L*. We note that this approach has been implemented in our prototype tool to conduct our empirical studies.

Now we are ready to discuss how to actually create a test sequence *S* out of *L*. This is done by first initializing *S* to be an empty sequence (line 5) and then appending to *S* a shortest path from the first node to the second node of *L*, and a shortest path from the second node to the third node of *L*, and so on (lines 6 to 9). In other words, *S* is created by adding into *L* a shortest path *P* between every two adjacent nodes to connect them. Note that node list *L* itself is not necessarily a path in the navigation graph *G*, and thus can not be directly used as a test sequence. This is because there may not exist an edge connecting every two adjacent nodes in *L*. Also note that *P* can always be found since *L* is built in a way that every two adjacent nodes,

say n_i and n_{i+1}, where $1 \leq i < k$, is an ordered pair, implying that there must exist at least one path from n_i to n_{i+1}.

After the sequence *S* is created, the set *covered* of pairs that are covered by sequence *S* is computed (line 10) and then removed from set *pairs* (line 11). Note that the indices *i* and *j* in the computation of the set *covered* do not have to be adjacent. This is because an ordered pair (*m*, *n*) is covered in a path if *m* and *n* appear in the path in the given order (not necessarily in a row).

We comment that the test sequences generated by algorithm *Generate-Sequences* do not necessarily start from the *home* node n_0. In practice, some web applications may require that every test sequence start from the *home* node. For example, a web application may require the user to log in before any other web page is visited. In this case, if a sequence does not start from the *home* node, it is necessary to add at the beginning of the sequence a shortest path from the *home* node n_0 to the first node of the sequence.

Algorithm Generate-Sequences

Input: A navigation graph $G = (V, E, n_0)$ of the web application under test

Output: a set *seqs* of paths covering all the ordered pairs in G

1. *pairs* = { (m, n) | m and n are dynamic nodes in G, and there exists a path from m to n in G }
2. let *seqs* be an empty set (of test sequences)
3. while (*pairs* is not empty) {
4. build a list $L = (n_1, n_2, ..., n_k)$} of nodes such that $k \leq |V|$ and for $1 \leq i < k$, $(n_i, n_{i+1}) \in$ pairs
5. let S be an empty sequence (of nodes)
6. for ($1 \leq i < k$) {
7. let P be a shortest path from n_i to n_{i+1}
8. $S = S \bullet P$
9. }
10. *covered* = $\{(n_i, n_j) \mid 1 \leq i < j \leq k, n_i, n_j \in L\}$
11. *pairs* = *pairs* − *covered*
12. add S into *seqs*
13. }
14. return *seqs*

Figure 4.3 Algorithm Generate-Sequences.

4.3.3. An Example Scenario

We demonstrate how algorithm *Generate-Sequences* works by using an example scenario from *Bookstore*, one of the two web applications used in our experiments. Figure 4.4 shows a portion of the navigation graph for *Bookstore* application, where the *Default* node is the *home* node. For the ease of reference, each web page is identified by a name, instead of its URL. We first generate all the ordered pairs in the navigation graph (line 1 in Figure 4.3). Those pairs are shown in Table 4.1. Note that in the navigation graph, every node can reach itself (through other nodes). Therefore, there exists an ordered pair from each node to itself, e.g., *D7* = (*Default*, *Default*), *A7*= (*AdvSearch*, *AdvSearch*), and so on.

Next, we try to generate test sequences to cover all the ordered pairs in Table 4.1 (lines 3 to 13 in Figure 4.3). We first try to build a node list *L* (line 4), using the heuristic approach described in Section 4.3.2. Assume that we first pick *D1* = (*Default*, *AdvSearch*), and add *Default* and *AdvSearch* into *L* (and remove *D1* from Table 4.1):

***L* = {*Default*, *AdvSearch*}**

Now we try to extend *L* using the three extension rules, i.e., the *front-end*, *back-end*, and *middle extension*. Without loss of generality, assume that we first apply *back-end extension*, where we try to find an ordered pair whose first node is the last node of *L*, i.e., *AdvSearch*. Note that *A1* = (*AdvSearch*, *Books*) is one such pair. Thus, we add *Books* into the end of *L* (and remove *A1*, as well as *D2*, which is also covered by *L*, from Table 4.1):

***L* = {*Default*, *AdvSearch*, *Books*}**

Similarly, as *B1* = (*Books*, *BookDetail*) is an ordered pair, we add *BookDetail* into the end of *L* (and remove *B1*, as well as *D3* and *A2*, which are also covered by *L*, from Table 4.1):

***L* = {*Default*, *AdvSearch*, *Books*, *BookDetail*}**

We keep applying *back-end extension* to *L* until we get the following sequence:

***L* = {*Default*, *AdvSearch*, *Books*, *BookDetail*, *ShoppingCart*, *ShoppingCartRecord*, *Default*, *MyInfo*, *ShoppingCart*, *AdvSearch*}**

At this point, we cannot find any ordered pair whose first node is *AdvSearch*. Next we apply *front-end extension*, where we try to find an ordered pair whose second node is the first node of *L*. We find *M3* = (*MyInfo*, *Default*) to be one such pair. Thus, we add *MyInfo* to the beginning of *L* (and remove *M3*, as well as *M4*, *M5*, *M6* and *M7*, which are also covered by *L*, from Table 4.1):

L = {MyInfo, Default, AdvSearch, Books, BookDetail, ShoppingCart, ShoppingCartRecord, Default, MyInfo, ShoppingCart, AdvSearch}

Note that all the ordered pairs whose second node is *MyInfo*, namely, *D6*, *A6*, *B5*, *BD4*, *S4*, *R2*, and *M7*, are already covered in *L*, implying that those pairs have been removed from Table 4.1. Thus, at this point, we cannot find any ordered pair to extend *L* from the front end.

Next we try to apply *middle extension*. However, none of the remaining ordered pairs, i.e., *B7*, *BD6*, *BD7*, *S6*, *S7*, *R5*, and *R6*, satisfies the condition for *middle extension*. At this point, we finish building *L*.

Now we generate a test sequence out of *L* such that every two adjacent nodes in *L* are connected via a shortest path between the two nodes (lines 5 to 9 in Figure 4.3). It turns out that most adjacent nodes in *L* have a direct edge between them, except for adjacent nodes *Default* and *MyInfo*, which can be connected by a shortest path (*Default*, *ShoppingCart*, *MyInfo*), and adjacent nodes *ShoppingCart* and *AdvSearch*, which can be connected by path (*ShoppingCart*, *Default*, *AdvSearch*). Thus, we generate the following test sequence:

***S* = {*MyInfo*, *Default*, *AdvSearch*, *Books*, *BookDetail*, *ShoppingCart*, *ShoppingCartRecord*, *Default*, *ShoppingCart*, *MyInfo*, *ShoppingCart*, *Default*, *AdvSearch*}**

Note that sequence *S* does not cover all the ordered pairs. For example, *BD7* = (*BookDetail*, *Books*) is not yet covered. The same process can be repeated to generate additional test sequences until all the ordered pairs are covered, which is not explained for the purpose of brevity.

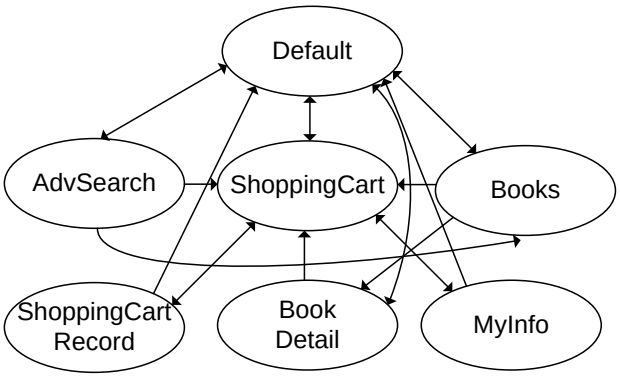

Figure 4.4 Navigation Graph for Bookstore.

Table 4.1 The Pairs Set for Bookstore

Pair	Default	Pair	AdvSearch Node
D1	Default, AdvSearch	A1	AdvSearch, Books
D2	Default, Books	A2	AdvSearch, BookDetail
D3	Default, BookDetail	A3	AdvSearch, ShoppingCart
D4	Default, ShoppingCart		(AdvSearch,
D5	Default, ShoppingCartRecord	A4	ShoppingCartRecord)
D6	Default, MyInfo	A5	AdvSearch, Default
D7	Default, Default	A6	AdvSearch, MyInfo
		A7	AdvSearch, AdvSearch
Pair	Books		BookDetail
B1	Books, BookDetail	BD1	BookDetail, ShoppingCart
B2	Books, ShoppingCart	BD2	(BookDetail,
B3	Books, ShoppingCartRecord		ShoppingCartRecord)
B4	Books, Default	BD3	BookDetail, Default
B5	Books, MyInfo	BD4	BookDetail, MyInfo
B6	Books, AdvSearch	BD5	BookDetail, AdvSearch
B7	Books, Books	BD6	BookDetail, BookDetail
		BD7	BookDetail, Books
Pair	ShoppingCart	Pair	ShoppingCartRecord
S1	(ShoppingCart,	R1	ShoppingCartRecord, Default
	ShoppingCartRecord)	R2	ShoppingCartRecord, MyInfo
S2	ShoppingCart, Default	R3	(ShoppingCartRecord,
S3	ShoppingCart, MyInfo		ShoppingCart)
S4	ShoppingCart, ShoppingCart	R4	(ShoppingCartRecord,
S5	ShoppingCart, AdvSearch		AdvSearch)
S6	ShoppingCart, Books	R5	ShoppingCartRecord, Books
S7	ShoppingCart, BookDetail	R6	ShoppingCartRecord, Books
		R7	(ShoppingCartRecord,
			ShoppingCartRecord)
Pair	MyInfo		
M1	MyInfo, ShoppingCart		
M2	MyInfo, AdvSearch		
M3	MyInfo, Default		
M4	MyInfo, Books		
M5	MyInfo, BookDetail		
M6	MyInfo, ShoppingCartRecord		
M7	MyInfo, MyInfo		

4.4 Experiments

4.4.1. Research Questions

Our experiments are designed to answer the following research questions.

1. How does the *AllOrderedPairs* test sequence generation approach compare with *AllEdges* test generation approach with respect to program coverage?
2. How does the *AllOrderedPairs* test sequence generation approach compare with *AllEdges* test generation approach with respect to fault detection effectiveness?

4.4.2. Metrics

We measure the effectiveness of the two test generation strategies by measuring statement coverage and number of faults detected.

4.4.3. Experimental Setup

Subject Applications: We used two web applications, *Bookstore* [45] and *CPM* [55], in our experiments. *Bookstore* is an online e-commerce application that users can use to browse, search and buy books [45]. *CPM* [55] is a course project manager developed at Duke University. *CPM* allows course instructors to create grader accounts for teaching assistants. Instructors and teaching assistants can create student accounts, post student grades and post available time slots for students to demonstrate their course projects. Students can view their grades and sign up for specific demo time slots with a grader. New grader/student/course accounts can be created and deleted as necessary. More details on the applications are presented in previous work by Sampath et al. [55].

Navigation Graph and Test Case Characteristics: Table 4.2 also shows the characteristics of navigation graphs (number of nodes and number of edges of the navigation graph) and Table 4.3 shows characteristics of test cases for each application. From the last row of Table 4.3, we see that on average, the length of test cases generated by both *AllEdges* and *AllOrderedPairs* test generation strategies is the same (around 8 requests), except for

Bookstore's AllOrderedPairs test cases. The *AllOrderedPairs* test generation algorithm is designed to (a) cover as many ordered pairs as possible in a test sequence, and (b) find the shortest paths between two consecutive nodes in the test sequence. Thus, the length of a generated test case depends on these two factors above. Since *Bookstore's* navigation graph has high connectivity—each node is connected to several other nodes, and since the algorithm is designed to find the shortest path between every two consecutive nodes, long test sequences are created, however, the number of test cases is small—7 *AllOrderedPairs* test sequences. Also, since *AllOrderedPairs* does not subsume *AllEdges* (as described in Section 4.3.1), we find that the *AllOrderedPairs* test sequences for our subject applications do not necessarily cover all the edges in the graph (*AllOrderedPairs* test sequences cover 60.7% of edges in *Bookstore*, and 78.4% of edges in *CPM*).

Experimental Framework: We used the framework presented in Sprenkle et al. [61] for capturing program coverage and fault detection information. The framework has three main components: a customized tool for replaying the test cases, *Clover* [12] for instrumenting and measuring program coverage, and a fault detection component that allows the insertion of hand-seeded faults into the web application, the application of oracles to determine if a test case detects a fault or not, and the creation of fault detection reports based on the faults detected by a test suite [55]. We augmented faults seeded by Sampath et al. [55] with faults that are likely to occur when two web pages interact with each other. Table 4.4 presents the number of seeded faults in each web application.

For the fault detection study, we use both the *diff* and the *struct* oracle, presented by Sprenkle et al. [61][63]. The *diff* oracle applies the Unix utility '*diff*' on the HTML responses returned on executing the test sequences on the clean and faulty versions of the application and reports any difference between the HTML responses as a failure. Since the *diff* oracle considers any difference in the HTML as a failure, differences in real-time content, e.g., the current date, are flagged as a failure by the oracle, thus leading to false positives. The *struct* oracle is more

conservative—it filters the HTML responses and reports only differences in the HTML tags. The obvious disadvantage of the *struct* oracle lies in its inability to capture faults that arise from differences in the content of the HTML page. Sprenkle et al. [63] discuss more about the oracles and the tradeoffs. In this chapter, we present results from both *diff* and *struct* oracles.

We implemented the *Generate-Sequences* algorithm in our prototype tool to generate test sequences. This tool generates test sequences that cover all the 2-way interactions (*AllOrderedPairs*) and all the edges (*AllEdges*). After generating the test sequences, our tool also verifies whether all the 2-way interactions are covered by test sequences for covering all the 2-way interactions and whether all the edges are covered by test sequences for covering all edges. It also shows us statistics of the comparisons between the *AllOrderedPairs* approach and *AllEdges* approach. Our web test sequence generation algorithm generates only the base requests for the test sequences and ensures that 2-way interactions are covered by the *AllOrderedPairs* test sequences. To execute the test cases correctly, we manually augment the requests with name-value pairs. This is similar to how testers provide test inputs to ensure correct execution of test sequences in traditional programs. We also use an initial data store state that is reset before each test case is executed, to avoid cascading faults.

Table 4.2 Characteristics of Subject Applications

Statistics	Bookstore	CPM
Technologies	JSP, MySQL	Java servlets, File-based data store, HTML
Non-commented Source LOC	7615	9401
Number of classes	11	75
Number of Methods	319	173
Number of Seeded Faults	72	197
Number of Nodes	41	64
Number of Edges	63	125

Table 4.3 Characteristics of Test Cases

Statistics	AllEdges		AllOrderedPairs	
	Bookstore	CPM	Bookstore	CPM
Number of test cases	15	41	7	261
Percent of ordered pairs covered	53.12%	15.03%	100%	100%
Percent of edges covered	100%	100%	60.7%	78.4%
Total number of requests	133	330	154	2273
Longest test case length	20	25	40	105
Shortest test case length	4	3	12	3
Average Test Case length	8.87	8.05	22	8.71

4.4.4. Results and Discussions

From Table 4.4, for *Bookstore* application, we observe that both *AllEdges* approach and *AllOrderedPairs* approach have the same code coverage, 85.32%, but *AllOrderedPairs* detects 6 to 8 more faults than the *AllEdges* approach. By design, in *Bookstore*, certain methods are included in every web page of the web application (through an include JSP statement), even though these methods are never called by the other methods in the web page—these methods are designed to be called by a user with different privileges (an admin user), instead of an end-user. In this chapter, since our tool focuses on covering 2-way interactions of regular user accessible web pages and functions, we report program coverage results for *Bookstore* application after removing such repeated code from the coverage report generated by *Clover*. The program coverage is same for *AllOrderedPairs* and *AllEdges* because the same web pages are accessed with the same parameters and values, thus resulting in same code coverage. That means the test sequences for *AllEdges* approach and *AllOrderedPairs* approach should have the same detection ability for faults in unit testing. The primary advantage of the

AllOrderedPairs test sequences is that they can guarantee the 2-way interaction coverage, while the *AllEdges* approach cannot.

From our experiments, we observe that all the faults detected by the *AllEdges* sequences are also detected by the *AllOrderedPairs* test sequences. One example of a fault that is caught by *AllOrderedPairs* but missed by *AllEdges* is presented here: we found that a fault is exposed when the "*Login*" web page is accessed the second time in a test sequence. Since (*Login*, *Login*) was an ordered pair for *Bookstore* application, the ordered pair (*Login*, *Login*) appeared in one of the *AllOrderedPairs* test sequences, and the fault was detected by an *AllOrderedPairs* test sequence. However, since there was no direct edge from "*Login*" web page to "*Login*" web page, the *AllEdges* test sequences were not required to generate a test sequence with two occurrences of the "*Login*" web page in them, thus failing to detect the fault.

Table 4.5 presents the program coverage and fault detection results for *CPM*. Test sequences from the *AllOrderedPairs* approach have higher program coverage than sequences from *AllEdges* approach. However, we see a large difference between the numbers of faults detected by each approach. From our experiments, we observe that all the faults detected by the *AllEdges* test sequences are also detected by the *AllOrderedPairs* test sequences. Fault detection by the *AllOrderedPairs* test sequences improved by a factor of 2.57 over the *AllEdges* test sequences. From Table 4.3, we see that the *AllEdges* test sequences for *CPM* cover only 15.03% of the 2-way interactions, whereas the *AllOrderedPairs* test sequences cover 100% of the interactions. We also found that test sequences that cover the most ordered pairs (2273 and 1523 ordered pairs), detect the most faults in the web application (35 and 33 faults, respectively). Thus, we observe that there is a relation between the number of ordered pairs covered in a test sequence and the number of faults it detects.

Also, *CPM* application is more complex in logic than *Bookstore* application. There are many 2-way interactions through data storage. For example, the 2-way interaction (*CreateCourseServlet*, *CatchGroupSignupServlet*) is not covered by the *AllEdges* test

sequences. If a fault exists in storing the course name of a new course in the "*CreateCourseServlet*" web page, the *AllEdges* test sequences will fail to detect it. But, since the interaction is present in the *AllOrderedPairs* test sequences, such a fault can be detected by test sequences that contain the 2-way interaction (*CreateCourseServlet*, *CatchGroupSignupServlet*).

Another reason for the difference is the complex logic in sequences for *AllOrderedPairs* test sequences and the name-value pairs supplied to the test sequences. For example, the request for creating a grader may just occur once in a test sequence generated by the *AllEdges* approach. But in a test sequence created by the *AllOrderedPairs* approach, the same request may occur multiple times because we are trying to cover all the 2-way interactions. Thus, when an existing grader is created again by a request that appears later in the test sequence, the error tolerance code will be covered, because that grader has already been created. This is also another reason for the higher code coverage for *AllOrderedPairs* approach when compared to the *AllEdges* approach.

A disadvantage of the *AllOrderedPairs* test cases is that there are more *AllOrderedPairs* test sequences (261) than *AllEdges* (41) test sequences—thus, the *AllOrderedPairs* test cases take longer to execute and require more resources. However, we believe the tradeoff in improved fault detection effectiveness is worth the increased test execution time. Also, it is important to note that the fault detection of the *AllOrderedPairs* test sequences is still only 36.5% of the total seeded faults (with the *struct* oracle). But, this is expected. There are three reasons for this fault detection ratio. First, our *AllOrderedPairs* test generation algorithm only generates the base requests—the name-value pairs to the request are still manually supplied. The particular name-value pairs used in test sequences have a significant impact on code coverage and fault detection. In the future, we plan to implement strategies to systematically generate name-value pairs for web application requests, which can improve the code coverage and the fault detection ratio. Second, some faults can be triggered

only in multi-user scenarios, e.g., two users log in with the same account. But, in our experiments, our test sequences are executed in single user scenarios. Thus, it is reasonable for missing these faults seeded for multi-user scenarios. Third, the initial database data may also cause failing to detect faults. When verify the experiment results, we figure out that some faults should be detected by our approach but they are missed indeed. For example, the price calculation fault in the "*ShoppingCart*" web page of *Bookstore* application is missed. The reason is that there is no order listed in the "*ShoppingCart*" web page when it is accessed. As a result, the statements for calculating prices are not executed at all, which makes our approach miss this fault. If the initial data in the database contain one or several orders, this fault would be detected by our approach.

Table 4.4 Program Coverage and Fault Detection: Bookstore

Statistics	All Edges	All Ordered Paris
Total Faults	72	72
Detected Faults	diff oracle: 61 struct oracle:45	diff oracle: 69 struct oracle:51
Statement Coverage	85.32%	85.32%

Table 4.5 Program Coverage and Fault Detection: CPM

Statistics	All Edges	All Ordered Paris
Total Faults	197	197
Detected Faults	diff oracle: 37 struct oracle: 28	diff oracle: 124 struct oracle: 49
Statement Coverage	62.8%	67.5%

4.4.5. Threats to Validity

One important threat to validity of our results is that we conducted our experiments on only two subject programs. Though the subject applications are fairly large-sized applications, we cannot generalize our results to all web applications. We also manually generated parameter-values to the requests generated by our tool—the effectiveness of the test

sequences largely depends on parameter-values used in the test sequences. In the future, we will closely investigate the problem of test input generation for web applications. Also, we do not report on the time to generate test sequences and the time to execute test sequences for fault detection effectiveness—these are measures that we plan to evaluate in the future.

CHAPTER 5

BUFFER OVERFLOW VULNERABILITY DETECTION

5.1 Background

As software is at the core of a computerized system, software security is a priority concern in many security assurance efforts [41][69]. Moreover, most computerized systems are inter-connected through an intra- or inter-network or both. As a result, software security breaches can often be accomplished remotely in the cyberspace, i.e., without direct physical access to a victim system. This makes software a favorite target for security attacks and partly contributes to the increase of software security breaches in recent years [68].

Computer security, in general, is a very active research area. Significant progress has been made in areas such as cryptographic algorithms, access control, intrusion detection, privacy management, and protocol design [26]. However, most research work in these areas has focused on the algorithmic aspects and/or at the design level. But, unfortunately, secure algorithms or designs do not necessarily lead to secure implementations, as mistakes are often made during the implementation. In this chapter, we distinguish *software security* from general computer security. Specifically, we use the term *software security* to refer to security assurance at the implementation level, i.e., how to ensure that actual software implementations are secure.

Existing approaches to software security can be largely classified into two categories. The first category is based on static analysis [11][48], which checks security properties by analyzing the source code of a subject program, i.e., without executing the program. The second category is based on dynamic analysis or testing [3][52], which executes the subject program and checks whether the program's runtime behavior satisfies some expected security properties. Static analysis can be very fast, but suffers from false positives or false negatives or

both. Testing, on the other hand, only reports problems that have actually been detected at runtime. However, testing requires test input selection and program execution, which can be difficult and time consuming. Software security properties can also be classified into two types: *application-specific* and *application-independent* properties. The former refers to properties that are specific to an application domain. For example, a software component responsible for the access control should ensure that an object under control is accessed only by a subject with sufficient privilege. The latter refers to properties that are independent from an application domain. For example, a software implementation, no matter which domain it is in, should be free from buffer overflow vulnerabilities. Buffer overflow is a typical topic in security testing. In this chapter, a new idea about buffer overflow vulnerability detection will be introduced.

A buffer overflow occurs when data is written beyond the boundary of an array-like data structure. Buffer overflow vulnerabilities are program defects that can cause a buffer overflow to occur at runtime. Many security attacks exploit buffer overflow vulnerabilities to compromise critical data structures [73], so that they can influence or even take control over the behavior of a victim system. Therefore, it is valuable to detect buffer overflow vulnerabilities before software is released.

5.2 Challenges and Contributions

Theoretically, symbolic execution-based buffer overflow vulnerability detection approaches [9][23][58], as what has been introduced in Chapter 2, are excellent ideas in the research work. They derive test inputs by analyzing the data and/or control structure of the source code, which systematically detects buffer overflow vulnerabilities in the software like human reasoning. But, in reality, they are not practical for two limitations of symbolic execution-based buffer overflow vulnerability detection: 1) involving instrumentation at the source code or binary code level, which makes these approaches specific to a particular programming language, building environment or platform; and 2) depending on the capability of constraint

solvers that, currently, are not practical to solve numerous and complicated constraints in real-life applications, like *Gzip* [28] and *Pine* [47].

Black-box testing does not depend on the capability of constraint solvers and it is easy to be applied to various implementations. Thus, black-box testing is widely used in practice. However, it often suffers from poor code coverage. Code coverage is considered to be an important indicator of testing effectiveness [2]. A black-box testing approach that has recently gained recognition in practice is called fuzz testing or fuzzing [67][69], which generates test data randomly. A concrete example for fuzz testing has been introduced in Chapter 2. Fuzz testing has been shown effective for detecting software vulnerabilities. However, like other black-box testing techniques, poor code coverage is considered to be a major limitation of fuzz testing [25].

A major design objective of our approach is to achieve good code coverage while retain the advantages of black-box testing. Our approach was motivated by a reflection on how buffer overflow vulnerabilities are exploited by an attacker in practice. In most cases an attacker can influence the behavior of a victim system only by controlling the values of its external parameters. External parameters are factors that could potentially affect the system behavior. Examples of external parameters include input parameters, configuration options, and environment variables. Therefore, launching a successful attack often amounts to a clever way of exploring the input space, typically by tweaking external parameter values. (In this research project, we do not consider interactive systems, i.e., systems that require a sequence of user interactions in the course of a computation, where each interaction may depend on the outcome of previous interactions.) Security testing essentially needs to do the same thing, but in a more systematic manner and with a good intent.

In a typical exploit attempt, the tweaking of external parameter values consists of the following two steps. First, the attacker identifies one external parameter *P* to carry attack data. We will refer to *P* as an *extreme parameter*. (The name of "*extreme parameter*" indicates that

the parameter will take an "*extreme value*" in a test, as discussed in Section 5.3.) *P* is often chosen such that, during program execution, its value is likely be copied into a buffer *B* that is vulnerable to overflow and that is located close to a critical data structure *C* that the attacker intends to compromise. A common example of such a data structure is the return address of a function call on the stack. The attack data is intended to overwrite *C* in a specific way that allows the attacker to gain control over the program execution. Second, the attacker tries to assign proper values to the rest of the external parameters. Some of these parameters can take arbitrary values, i.e., the exploit attempt will succeed or fail, regardless of the values of these parameters. However, other parameters are important for steering the program execution to reach the vulnerable point, i.e., a statement that actually copies the value of *P* into buffer *B* and whose execution may give rise to a buffer overflow. We will refer to these parameters as control parameters. Note that these two steps can be repeated by taking different parameters as the extreme parameter.

In this chapter, we present a dynamic approach to detecting buffer overflow vulnerabilities. It is a specification-based or black-box testing approach. That is, we generate test data based on the specification of a subject program. Our approach is to systematize the tweaking of external parameter values in a typical exploit attempt as described above. Specifically, we identify two conditions that must be met in order to expose buffer overflow vulnerabilities. Our approach is centered on how to generate tests such that these two conditions are likely to be satisfied for potentially vulnerable statements. We provide guidelines on how to identify extreme and control parameters and their values. Moreover, for each value of an extreme parameter, we adapt combinatorial testing [7][13] to generating a group of tests such that one of these tests is likely to steer program execution to reach a vulnerable point. Software testing based on combinatorial design has been shown very effective for functionality faults in the software under test [33]. In particular, empirical results suggest that there exists a high correlation between combinatorial coverage and code coverage [7][13]. It is this correlation

that is exploited in our approach to increase the likelihood for our tests to reach a vulnerable point, and thus the likelihood to detect buffer overflow vulnerabilities.

Note that attack data often need to be carefully crafted in order to carry out a real attack. However, for testing, our goal is to demonstrate the possibility of detecting a buffer overflow vulnerability, i.e., not to acquire specific control to do any real harm. As discussed in Section 5.3, this simplifies the selection of parameter values, especially for the extreme parameters.

For the purpose of evaluation, we implemented our approach in a prototype tool called *Tance*. We conducted experiments on five open-source programs: *Ghttpd* [21], *Nullhttpd* [44], *Gzip* [28], *Hypermail* [30] and *Pine* [47]. The experiment results show that our approach can effectively detect buffer overflow vulnerabilities in these applications. In particular, we examined vulnerabilities reports in three public vulnerability databases. This examination showed that our approach detected all the known vulnerabilities except for one for the first four programs. For *Pine*, insufficient information was available to determine whether the reported vulnerabilities were the same as the ones we detected. In addition, our approach detected a number of new vulnerabilities that have not been reported in the three databases.

5.3 A Combinatorial Approach for Buffer Overflow Vulnerability Detection

5.3.1. Main Idea

We use an example to illustrate the main idea of our approach. Assume that a statement *L* copies a string variable *S* into a buffer *B*, without checking whether *B* has enough space to hold *S*. Thus, *L* is potentially vulnerable to buffer overflow. In order for a test *T* to detect this potential vulnerability, two conditions must be met:

- *C1*: *L* must be executed during the execution of test *T*. In other words, when *T* is executed, the control flow must be able to reach the point where *L* is located.

- *C2*: When *L* is executed, either *S* is a string that is unexpectedly long, or *B* is a buffer that is unexpectedly small, or both. In other words, either *S*, or *B*, or both, have to take an extreme value.

Note that extreme values are often syntactically legal but not semantically meaningful. As a result, extreme values are often unexpected and not tested during normal functional testing.

Our approach is centered on how to generate tests such that conditions *C1* and *C2* are likely to be satisfied for potentially vulnerable statements like *L*. In our approach, we first identify a group of extreme parameters and a set of extreme values for each of these parameters. An external parameter *p* is identified to be an extreme parameter if *p* taking an extreme value may cause variables like *S* or *B* to take an extreme value. Next, we identify a set of control parameters for each extreme parameter, and a set of values for each of these control parameters. Let *p* be an extreme parameter. Intuitively, an external parameter *p'* is identified to be a control parameter of *p* if business logic suggests that the value of *p'* may affect how the value of *p* is processed. The values of a control parameter, which we will refer to as control values, are identified such that they could potentially lead to different business scenarios. In our approach, the extreme and control parameters and their values are identified manually based on specifications and/or domain knowledge. Some general guidelines on how to perform this identification are provided in Section 5.3.2.

Now we discuss how tests are actually generated in our approach. Consider the example again. If we knew that *L* was a vulnerable statement, ideally we would want to generate a single test to satisfy both *C1* and *C2*. Unfortunately, vulnerable statements like *L* are not known a priori. Our approach takes a different perspective. Instead of trying to generate an ideal test for a specific vulnerable statement, we try to generate a group of tests for each extreme value (of each extreme parameter) such that each extreme value can reach as many vulnerable statements as possible. Specifically, for each extreme value *v* of each extreme

parameter *p*, we generate a combinatorial test set *T* such that (1) *p* takes value *v* in each test in *T*; and (2) *T* covers all the *t*-way combinations of all the control parameters of *p*, where *t* is a small integer number that is expected to be no greater than 6 in practice [33]. The reason why we want to achieve *t*-way coverage for all the control parameters of *p* is to exploit the correlation between combinatorial coverage and code coverage such that, if there is a vulnerable statement like *L* where *v* could cause a variable like *S* or *B* to take an extreme value, then one of the tests in *T* will be likely to reach this statement.

Intuitively, our approach uses extreme parameters and their values to satisfy *C2* and control parameters and their values to satisfy *C1*. Observe that in each test we generate, there is a single extreme parameter. This implicitly assumes that variables like *S* or *B* in our example derive their extreme values from a single external parameter. That is the following hypothesis, made in our approach:

Hypothesis Single-Extreme-Parameter (SEP): It is often the case that a buffer is overrun by an extreme value (of an internal variable) that is derived from a single external parameter.

This hypothesis is consistent with the fact the attacker typically picks one external parameter to carry attack data, as discussed in Section 5.1. As an effort to validate this hypothesis, we inspected buffer overflow vulnerability reports in three public databases. The results of our inspection, as presented in Section 5.5, provide strong evidence for the validity of this hypothesis in practice.

5.3.2. Algorithm BOVTest

Figure 5.1 presents an algorithm called *BOVTest* (short for Buffer Overflow Vulnerability Test) that implements our approach. This algorithm takes as input a program specification *M* and an integer *t*. *M* is used as the basis for identifying external parameters and their values, and *t* is used as the strength for combinatorial test generation. The output of algorithm *BOVTest* is a test set *T* for detecting buffer overflow vulnerabilities in an implementation of *M*.

Algorithm BOVTest

Input: A program specification M , and an integer t

Output: A test set T for detecting buffer overflow vulnerabilities in an implementation of M

1. let P be the set of all the external parameters of M
2. identify a set $Px \subseteq P$ of extreme parameters and a set of extreme values for each parameter p in Px
3. initialize T to be an empty test set
4. for each extreme parameter px in Px {
5. identify a set $Pc \subseteq P$ of control parameters of px and a set of control values for each parameter p in Pc
6. let $Pd = P - Pc$, and identify a default value $d(p)$ for each parameter p in Pd
7. for each extreme value v of px {
8. build a t-way test set T' for parameters in Pc using their control values
9. for each test τ' in T' {
10. create a complete test τ such that for each parameter p, $\tau(p) = v$ if $p = px$, $\tau(p) = \tau'(p)$ if $p \in Pc$, and $\tau(p) = d(p)$ otherwise, where $\tau(p)$ (or $\tau'(p)$) is the value of parameter p in test τ (or τ')
11. $T = T \cup \tau$
12. }
13. }
14. }
15. return T;

Figure 5.1 Algorithm BOVTest

Algorithm *BOVTest* begins by identifying all the external parameters in *M* (line 1). Generally speaking, an external parameter is a factor that could potentially affect the program behavior. This includes not only input parameters, but also configuration options and environment variables and other factors that could potentially affect the program behavior.

Next, it identifies a set *Px* of extreme parameters as well as their extreme values (line 2). To identify extreme parameters and extreme values, we observe that extreme parameters often have variable lengths, or indicate the sizes of some other parameters (and thus are likely to be used as the capacity of a buffer). In the former case, an extreme value is often a string value of an excessive length; in the latter case, an extreme value is often an excessively small value. Note that in both cases, the specific values of an extreme parameter are often not significant for the purpose of testing, i.e., in terms of triggering a buffer overflow. A key insight behind this observation is that buffer overflow vulnerabilities are in essence a mishandling of certain length/size requirements. This observation can be used as a guideline for identifying extreme parameters and values. For example, in a network protocol, *user payload* is likely to be an extreme parameter, as it is of a variable length. Furthermore, we can identify a payload that is longer than the typically expected as one of its extreme values. On the other hand, this observation can also be used to exclude certain parameters from *Px*. For example, string parameters of fixed length are typically not extreme parameters.

After we identify extreme parameters and their values, we generate a *t*-way combinatorial test set for each extreme value of each extreme parameter (lines 4 to 14). For each extreme parameter *px*, we first identify two groups of external parameters, namely, *Pc*, and *Pd*, as well as their values (lines 5 to 6):

- *Pc* consists of all the control parameters of an extreme parameter *px*. An external parameter *p'* is a control parameter of *px*, if business logic suggests that the value of *p'* could affect how the value of *px* is processed, based on specification and/or domain

knowledge. For each parameter in *Pc*, we identify a set of control values. Control values can be identified using traditional techniques such as domain analysis and equivalence partitioning. Oftentimes, different control values signal different business scenarios, leading to different branches in a program. We point out that security testing is often performed after normal functional testing. Therefore, it is often possible for us to take advantage of the knowledge and experience accumulated during functional testing. In particular, we expect that most control values have been identified and tested in functional testing.

Again, consider a network protocol. Assume that we have identified *user payload* as an extreme parameter. Then, *message type* is likely to be a parameter that could affect how *user payload* is processed in the implementation. This is because the payload often needs to be interpreted differently depending on the type of a message. The values of this *message type* parameter, used as a control parameter, would be the different types that are specified in the protocol specification.

- *Pd* is the complement set of *Pc* (line 6). In other words, *Pd* consists of all the external parameters that are not control parameters of *px*. For each parameter in *Pd*, we simply identify a single default value, which can be any valid value in the domain of the parameter. (Note that a value is valid if it is allowed by the specification. Otherwise, it is invalid.) Note that these default values do not directly contribute to the detection of buffer overflow vulnerabilities in our approach. Instead, these values are only needed to construct complete, thus executable, tests.

In general, the fewer the parameters in *Pc* (and the more the parameters in *Pd*), the fewer the tests generated for each extreme value of *px*. Note that an imperfect identification of *Pc* and *Pd* may increase the number of tests and/or miss some vulnerabilities, but it does not invalidate our test results. That is, any vulnerability detected by our approach is a real vulnerability. More discussions on this are provided in Section 5.3.3.

The actual generation of a *t*-way test set for each extreme value *v* of each extreme parameter in *Px* proceeds as follows. We first generate a *t*-way test set *T'* for all the parameters in *Pc*, using their control values (line 8). Each test in *T'* is then used a base test to create a complete test by (1) adding *v* as the value of *px*, and (2) adding the default value of each parameter in *Pd* (and thus not in *Pc*). Consider the example shown in Figure 5.2. Assume that a system has five parameters, *p1*, *p2*, *p3*, *p4*, and *p5*. Assume that *p4* is an extreme parameter, and has an extreme value *LS*, indicating a very long string. Also assume that *p1*, *p2*, and *p3*, with control values "0" and "1", are in *Pc*, and *p5* is in *Pd*, with a default value "0". To generate a 2-way test set for the extreme value *LS* of *p4*, we first generate a 2-way test set for *p1*, *p2*, and *p3*, which is the test set *T'* shown in Figure 5.2. Next, we add into each test of *T* the value *LS* as the value of *p4*, and "0" as the default value of *p5*. Thus, we obtain the following complete test set *T*:

p1	p2	p3	p4	p5
0	0	0	LS	0
0	1	1	LS	0
1	0	1	LS	0

Figure 5.2 An Example Test Set.

Finally, we analyze the time complexity of algorithm *BOVTest*, which is dominated by the three *for*-loops (lines 4 to 14). Assume that the *IPOG* [35] algorithm is used to generate a *t*-way test set (line 8). The time complexity of the *IPOG* algorithm for generating a *t*-way test set for *k* parameters, each having at most *d* values, is $O(d^{t+1} \times k^t \times \log k)$, and the size of the test set is $O(d^t \times \log k)$ [35]. Let *dc* be the maximal number of control values a control parameter can take. The time complexity for generating *T'* (line 8) is $O(d_c{}^{t+1} \times |P_c|^t \times \log |P_c|)$, and the

size of T' is $O(d_c{}^t \times \log|P_c|)$. It takes $O(|P|)$ to create a new test (line 10). The time complexity for the innermost *for*-loop (lines 9 to 12) is $O(d_c{}^t \times \log|P_c| \times |P|)$. Note that $P_c \leq P$. The time complexity for the *for*-loop in the middle (lines 7-13) is $O(d_c{}^{t+1} \times |P|^t \times \log|P|)$. Let *dx* be the maximum number of extreme values an external parameter can take. The time complexity of the outermost *for*-loops is $O(d_x \times d_c{}^{t+1} \times |P|^{t+1} \times \log|P|)$. Let *d* be the maximum of *dc* and *dx*. Then, the time complexity of the entire algorithm is $O(d^{t+2} \times |P|^{t+1} \times \log|P|)$. Note that the number of tests generated by algorithm *BOVTest* is $O(d^{t+1} \times |P| \times \log|P|)$.

5.3.3 Discussions

Recall that a buffer overflow occurs when data is written beyond the boundary of an array-like structure. Let *D* be the data to be written. Let *B* be an array-like structure. As discussed earlier, if the size of *D* is larger than the capacity of *B* (either *D* is excessively long, or *B* is excessively small, or both), then *D* will be written beyond the boundary of *B*, i.e., *B* will overflow. There is a more subtle case to consider. In languages like *C*, an array-like structure is often accessed using a pointer and such a pointer can be moved forward and/or backward using explicit arithmetic operations. For example, an array variable in *C* is in fact a pointer that points to the beginning of the array. It is possible that the pointer may be moved beyond the upper or even lower boundary of a buffer due to explicit pointer arithmetic operations. If data of any size is written at this point, a buffer overflow will occur.

The above scenario suggests that attention should be paid to external parameters whose values may be used as an offset in a pointer arithmetic operation (in addition to external parameters of variable length, and external parameters that indicate the length of other parameters). For example, in a record keeping application, the record number may be used as an offset to locate a record. If proper checks are not performed, a negative value, or an

unexpectedly large positive value, of the record number could move the base pointer beyond the lower or upper boundary of the structure that keeps all the records.

It is worth noting that extreme values matter typically because of their extreme properties, instead of their specific values. For example, what matters for an extreme string is typically its length, instead of the specific characters in the string. This observation simplifies the selection of extreme values.

As mentioned, while the extreme and control parameters and their values are identified manually in our approach, this identification does not have to be perfect. In practice, we can exploit this flexibility to scale up or down our test effort. On the one hand, when adequate resource is available, more parameters and values can be identified to acquire more confidence at the cost of creating more tests. For example, if we are unsure about whether an external parameter should be considered to be a control parameter, it is safe to do so. This will likely create more tests, but will also help to detect vulnerabilities that otherwise would not be detected. On the other hand, when the resource is constrained, we can focus our effort only on a subset of parameters and values that we believe are the most important ones to test. Doing so will reduce the number of tests, but may miss some vulnerabilities. Nonetheless, any vulnerability that is detected by our approach is guaranteed to be a real vulnerability.

5.4 Tance: A Prototype Tool

We implemented our approach in a prototype tool called *Tance*. Figure 5.3 shows the architecture of *Tance*, which consists of the following major components:

- **Controller**. This is the core component of *Tance*. It is responsible for driving the entire testing process. In a typical scenario, after *Controller* receives from the user the external parameter model of a subject application, it calls *Test Generator* to generate combinatorial tests based on the given parameter model. For each test, *Controller* uses *Test Transformer* to transform it into an executable test format, i.e., a format that is

accepted by the subject application. Then, *Controller* calls *Test Executor* to execute each test automatically.

- **Test Generator**. This component is responsible for the actual test generation. In other words, this component implements algorithm *BOVTest*. This component first uses a combinatorial test generation tool, called *ACTS* (formerly known as *Fireeye*) [35] to generate a base combinatorial test set, which is used later to derive a set of complete tests as discussed in Section 5.3.
- **Test Transformer**. This component is responsible for transforming each combinatorial test into a format that is accepted by subject applications. This step is necessary because a combinatorial test only consists of parameter values, but applications often require a test to be presented in a particular format. For example, a web server requires each test to be presented as a HTTP request. This component needs to be customized for different applications. *Tance* provides a programming interface that allows the user to hook a third party component into its testing framework.
- **Test Executor**. This component is responsible for carrying out the actual test execution process. For example, this tool will send test requests automatically to HTTP servers. In addition, *Test Executor* will restart HTTP servers before running the next test so that there is no interference between different tests. This component also needs to be customized for different programs. *Tance* also provides a programming interface for integration with an existing test execution environment.
- **Bounds Checker**. This component is used to detect the actual occurrence of a buffer overflow. Before running an application, the user has to instrument it with a bounds checking tool [18].

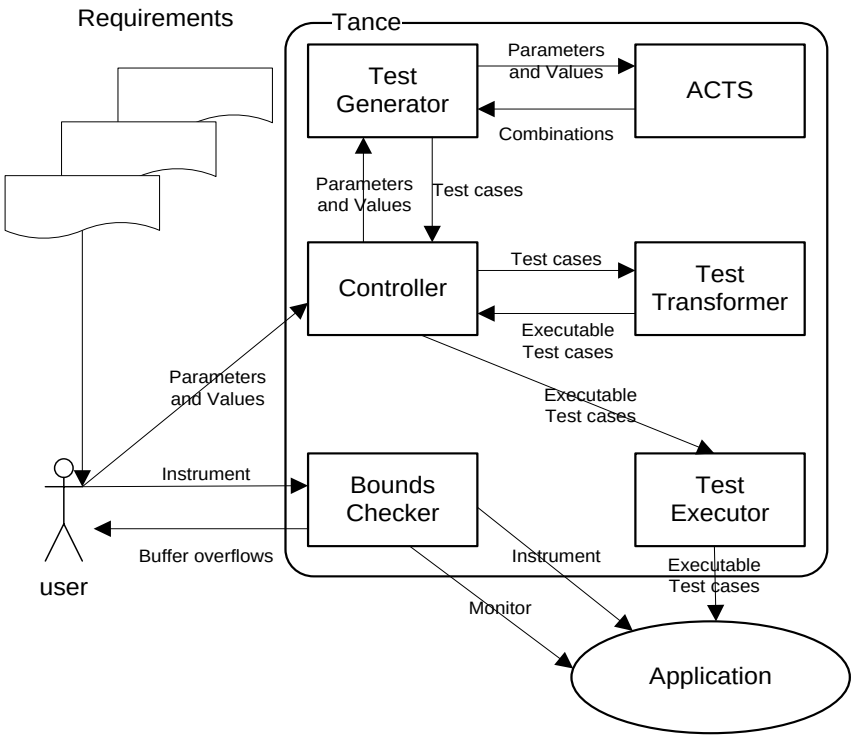

Figure 5.3 Tance's Architecture.

5.5 Experiments

In Section 5.5.1, we present the results of our inspection on three public vulnerability databases, as an effort to validate the *SEP* hypothesis. In Section 5.5.2, we describe subject applications as well as the computing environment used in our case studies. In Section 5.5.3, we present the vulnerability detection results of applying our approach to five open-source applications and compare our approach with fuzz testing on two HTTP server applications.

5.5.1. Validation of the SEP Hypothesis

To validate the *SEP* hypothesis, we checked three public vulnerability databases: *SecurityFocus* [56], *SecurityTracker* [57], and the *National Vulnerability Database* (*NVD*) [43]. For each database, we conducted a search using the keyword "buffer overflow" to retrieve reports of buffer overflow vulnerabilities. For *SecurityFocus*, the reports were sorted based on

"ranking", and we inspected the top 100 reports. For *SecurityTracker* and *NVD*, the reports cannot be sorted, and we inspected the first 100 reports returned by our search. Among the reports we inspected, there are 15 reports cross-referenced between *SecurityTracker* and *NVD*.

Table 5.1 shows the results of our inspection. We classify the reports into three categories. The first category (*Explicitly Stated*) includes reports that contain an explicit statement confirming the satisfaction of the *SEP* hypothesis. For example, the report *CVE-2010-0361* in *NVD* contains the following statement: "*Stack based buffer overflow … via a long URI in HTTP OPTIONS request.*". This statement explicitly states that a single parameter "*URI*" takes an extreme value. The second category (*Reasonably Inferred*) includes reports that are written in a way that reasonably suggests the satisfaction of the *SEP* hypothesis, despite the lack of an explicit statement. For example, the report *CVE-2007-1997* in *NVD* contains the following statement: "*… via a crafted CHM file that contains a negative integer … leads to a stack-based buffer overflow.*". This statement does not explicitly identify a parameter, but it is reasonable to believe that one of the fields in a *CHM* file can be modeled as an extreme parameter taking the negative value. The third category (*Not Clear*) includes reports that do not fall into the first two categories. For these reports, we do not have adequate information to make a reasonable judgment. We did not find any report that explicitly states that two or more parameters must take extreme values at the same time to trigger a buffer overflow. In other words, none of the examined reports explicitly disproves the *SEP* hypothesis.

Table 5.1 Validation of the SEP Hypothesis

Database	Explicitly Stated	Reasonably Inferred	Not Clear
SecurityFocus	61	9	30
SecurityTracker	62	29	9
NVD	49	25	26

5.5.2. Experimental Setup

Subject Programs: Our case studies use the following five applications:

(1) *Ghttpd* (version 1.4.4) is a fast and efficient web server that supports a reduced set of HTTP requests [21].

(2) *Nullhttpd* (version 0.5.0) is a web server that handles HTTP requests [44].

(3) *Gzip* (version 1.2.4) is a widely used GNU compression utility tool [28].

(4) *Hypermail* (version 2.1.3) is a tool that facilitates the browsing of an email archive. It compiles an email archive in the Unix mailbox format to a set of cross-referenced HTML documents [30].

(5) *Pine* (version 3.96) is a widely used tool for reading, sending, and managing emails [47].

All five applications are programmed in *C*. Note that *Ghttpd* and *Nullhttpd* have been used in other empirical studies for buffer overflow detection methods [59]. Table 5.2 (LOC = lines of uncomment code) shows some statistics about the size of these five applications.

Platform Configuration: The five case studies were conducted on a 3GHz machine that has 2GB RAM, running Red Hat Enterprise Linux WS release 4, gcc-3.4.6 and bgcc-3.4.6.

Table 5.2 Statistics of Subject Applications

Subject	Files	Functions	LOC
Ghttpd	4	16	609
Nullhttpd	11	38	2245
Gzip	34	108	5809
Hypermail	57	401	23057
Pine	449	4883	154301

5.5.3. Results and Discussions

In our studies, we identified the extreme and control parameters and their values in a fairly straightforward manner. In particular, we intentionally avoided the use of any advanced domain knowledge. On the one hand, this illustrates that while advanced domain knowledge helps to make our approach more effective, it is not required. On the other hand, this is part of our effort to reduce the threats to validity, as discussed in Section 5.6.

Specifically, all the string parameters of variable length were identified to be extreme parameters. For each of these parameters, we identified a single extreme value, which is a string typically much longer than normally expected. An integer parameter is identified to be an extreme parameter only if it obviously indicates the length of another parameter such as "*Content-Length*". For each integer extreme parameter, we identified three extreme values, including a positive number that is smaller than the actual length of the other parameter, zero, and a negative number whose absolute value is small. For each extreme parameter *p*, we identified a parameter *p'* to be a control parameter of *p* only if a very strong connection existed between *p* and *p'*. For example, in *Ghttpd* and *Nullhttpd*, a parameter named "*Request-Method*" indicates whether the request is a *GET* or *POST* request. This parameter was identified to be a control parameter of an extreme parameter "*Message-Body*", which represents the payload carried in an HTTP request.

Table 5.3 (NP=# of Parameters, NXP=# of Extreme Parameters, ANXV=Average # of Extreme Values per Extreme Parameter, ANCP=Average # of Control Parameters per Extreme Parameter, ANCV=Average # of Control Values per Control Parameter) shows statistics on the number of different types of parameters and values for each subject application. Note that, for *Ghttpd* and *Nullhttpd*, we used the same set of extreme and control parameters and values. The reason is that both applications are HTTP servers and we identified the parameters and their values from the same HTTP specification. Also note that *Pine* has two operational modes, read

and write. These two modes are tested separately in our experiments, because these two modes have very different interfaces and use different sets of parameters.

Table 5.3 External Parameter Models

Subject	NP	NXP	ANXV	ANCP	ANCV
Ghttpd	42	5	1.4	2.7	4.8
Nullhttpd	42	5	1.4	2.7	4.8
Gzip	16	3	3	13.3	12.3
Hypermail	28	16	2	15.5	12.8
Pine (read)	10	10	1.4	2.4	1.9
Pine (write)	7	7	1.3	2	1.7

Next we present some statistics on the number of tests we generated for the five subject applications. Recall that we generate a group of tests for each extreme value of each extreme parameter. Table 5.4 shows the total number of tests generated for each application (*Total*), and the minimum (*Min*), maximum (*Max*), and average (*Avg*) number of tests for each extreme value. Note that algorithm *BOVTest* generates the same number of tests for every extreme value of the same extreme parameter.

Table 5.4 Statistics on Number of Tests

Subject	Total	Min	Max	Avg
Ghttpd	191	3	36	27.3
Nullhttpd	191	3	36	27.3
Gzip	32	10	12	10.7
Hypermail	200	10	10	10
Pine (read)	89	3	8	6.4
Pine (write)	49	3	8	5.4

Table 5.5 presents information about the buffer overflow vulnerabilities detected by our approach. Note that we obtained the number of reported vulnerabilities from two databases, i.e., *SecurityFocus* and *SecurityTracker*. The results show that our approach detected a total of 9 new vulnerabilities for the five applications. In addition, our approach detected all the reported vulnerabilities for *Ghttpd*, *Nullhttpd*, and *Gzip*. For *Hypermail*, a buffer overflow vulnerability was reported but was not detected by our approach. An inspection revealed that this buffer overflow vulnerability involved a long string returned by a *DNS* server, which was not modeled as an

external parameter in our experiments. For *Pine*, there are 7 reported vulnerabilities, and our approach also detected 7 vulnerabilities. However, no adequate information is available for us to determine whether the 7 reported vulnerabilities are the same as those detected by our approach. Therefore, we put *LOI*, the short for Lack of Information, in the table.

To compare our approach with fuzz testing, we applied both *Tance* and *JBroFuzz* [31] to *Nullhttpd* and *Ghttpd*. As what has been discussed in Chapter 2, fuzz testing derives new tests from normal tests by mutating parameter values one by one. Fuzz testing does not generate the normal tests by itself. The normal tests have to be provided by users. To simulate this testing process, we applied a typical test generation methodology, i.e., the default testing [27], to generate a set of normal tests for fuzz testing to mutate. In experiments, *JBroFuzz* generated 1152 tests to trigger buffer overflow vulnerabilities in *Ghttpd* and *Nullhttpd*. Table 5.6 shows that *JBroFuzz* has detected one buffer overflow vulnerability in *Ghttpd* and three buffer overflow vulnerabilities in *Nullhttpd*. We verified these four buffer overflow vulnerabilities manually. All of them have been detected by *Tance* in experiments too. One common characteristic of the buffer overflow vulnerabilities detected by *JBroFuzz* is that their positions are shallow and hence are easy to reach. For example, in *Nullhttpd*, the three buffer overflow vulnerabilities can be reached easily without passing any checking condition. However, as for the other two buffer overflow vulnerabilities missed by *JBroFuzz*, both of them exist in deep braches, at least two checking conditions have to be passed before reaching vulnerable points. This comparison also provides extra evidence for the poor code coverage limitation of fuzz testing.

Note that we just compared our approach with fuzz testing on the two HTTP server applications. The reason is that HTTP sever applications are popular in use and also have more security concerns on them. Thus, some testing tools have already been developed for them, including the fuzz testing tools e.g., *JBroFuzz*. However, as for the rest three applications, we

have not found existing fuzz testing tools for them. Therefore, we did not compare our approach with fuzz testing on them.

Finally, we point out that care should be taken for counting the number of vulnerabilities. This is because the same vulnerability could cause multiple buffer overflows to occur at runtime. Therefore, it would be less meaningful if we simply counted the number of buffer overflows reported by the bounds checking tool. In our experiments, we consider buffer overflows reported on the same buffer and reached along the same execution path to be caused by the same vulnerability. In other words, we only count one buffer overflow vulnerability for all the buffer overflows that occur on the same buffer and along the same execution path.

Table 5.5 Vulnerability Detection Results

Subject	Detected	Reported	Missed	New
Ghttpd	1	1	0	0
Nullhttpd	5	1	0	4
Gzip	2	1	0	1
Hypermail	5	2	1	4
Pine	7	7	LOI	LOI

Table 5.6 Comparisons on Detected Buffer Overflow Vulnerabilities between Tance and JBroFuzz

Subject	Tance	JBroFuzz
Ghttpd	1	1
Nullhttpd	5	3

5.5.4. Threats to Validity

As discussed in Section 5.3, the effectiveness of our approach depends on the proper identification of the extreme and control parameters and their values. Since our experiments use programs that have known vulnerabilities, the validity of our results would be in jeopardy if knowledge of the known vulnerabilities were used to identify these parameters and their values in our experiments. To alleviate this potential threat, we tried to only use explicit information that was available in the specification. In addition, each time we identified a particular type of parameter or value, we provided an explicit explanation about how our decision was made in a

way that only used information available in the specification, rather than other sources. These explanations were cross-checked by two of the testers and are available for review [77].

The validity of our results also depends on the correctness of two tools, namely *ACTS* [35] and the bounds checker [18], used in our experiments. We note that both of these two tools have been available for public access for a significant amount of time, and have been used to conduct experiments for other research projects.

The main external threat to validity is the fact that the five open source applications used to conduct our experiments may not be representatives of true practice. We note that these applications are real-life applications themselves, and they have been used in other studies.

CHAPTER 6

CONLUSIONS AND FUTURE WORK

6.1 Conclusions

This dissertation applies the notion of combinatorial design to web application testing and security testing. Specifically, this dissertation presents three combinatorial testing techniques to deal with interactive web applications and security requirements, including a combinatorial approach to building navigation graphs for dynamic web applications, an interaction-based test sequence generation approach for testing web applications, and a combinatorial approach to detecting buffer overflow vulnerabilities.

The combinatorial approach to building navigation graphs for dynamic web applications is developed to capture the navigation structure of a dynamic web application. The generated navigation graph can facilitate testers in web sequence testing and regression testing. There are three contributions of this approach: 1) solving the web page explosion problem in dynamic web applications by proposing a URL abstraction mechanism; 2) solving the request generation problem with *t*-way input combination generation for web forms; 3) developing a tool, name *Tansuo*, to generate web navigation graphs automatically.

The interaction-based test sequence generation approach to testing web applications is developed to generate test sequences from web navigation graphs. The generated test sequences cover all the interactions between any two web pages in a web application. When applied to testing web applications, these test sequences systematically detect faults that may be caused by interactions between any two web pages. There are two contributions of this approach: 1) proposing a new algorithm based on combinatorial design to generate test

sequences that achieve 2-way interaction coverage; 2) developing a prototype tool to automatically generate 2-way interaction coverage test sequences.

The combinatorial approach to detecting buffer overflow vulnerabilities adapts combinatorial testing techniques to detect buffer overflow vulnerabilities in the software under test. There are two contributions of this approach: 1) proposing a combinatorial design based algorithm to generate advanced tests for triggering buffer overflow vulnerabilities; 2) developing a tool, named *Tance*, to automatically generate and execute tests and monitor triggered buffer overflows.

Experiments have been conducted on open source applications to evaluate the effectiveness of these three approaches. Empirical results show that these approaches are effective in capturing web navigation structures, detecting faults that are caused by the interactions between web pages, and detecting buffer overflow vulnerabilities, which indicates the effectiveness of combinatorial design in web application testing and security testing.

6.2 Future Work

There are three venues to continue our research work:

- Experiments on large and/or complex real-life applications. Although we have conducted experiments on several applications of significant size, it is still valuable to evaluate the effectiveness of our approaches on large and/or complex real-life applications. For example, experiments conducted on *amazon.com* and *chase.com* will further evaluate the values of our web navigation graph generation approach and web test sequence generation approach.
- Automatic identification of testing values for individual parameters. All the three approaches, proposed in this dissertation, focus on how to automatically combine individual parameter values that must be provided by users. In the

future, we will investigate how to automatically identify individual parameter values so that we can fully automate the entire test generation process.

- Finally, we plan to improve the user interfaces of our tools developed for the three approaches so that they are accessible to public users.

REFERENCES

[1] A. Andrews, J. Offutt, and R. Alexander, “Testing Web Applications by Modeling with FSMs”, Software and Systems Modeling, 4(3): 326-345, 2005.

[2] J.H. Andrews, L.C. Briand, Y. Labiche and A.S. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”, IEEE Transactions on Software Engineering, 32(8): 608-624, 2006.

[3] D. Aitel, “The Advantages of Block-based Protocol Analysis for Security Testing”, Immunity Inc, 2002. DOI= http://www.net-security.org/article.php?id=378.

[4] M. Benedikt, J. Freire, and P. Godefroid, “VeriWeb: Automatically Testing Dynamic Web Sites”, Proceedings of the 11th International World Wide Web Conference, 2002.

[5] T. Berners-Lee, R.T. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): General Syntax”. DOI = http://labs.apache.org/webarch/uri/rfc/rfc3986.html.

[6] R. Binder, "Testing Object-Oriented Systems" (Addison Wesley. 2000).

[7] K. Burr, and W. Young, “Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage”, Proceedings of the International Conference on Software Testing Analysis and Review, pp. 503-513, 1998.

[8] K. Burroughs, A. Jain, and R.L. Erickson, “Improved Quality of Protocol Testing Through Techniques of Experimental Design”, Proceedings of the IEEE International Conference on Communications, pp. 745-752, 1994.

[9] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, “EXE: Automatically Generating Inputs of Death”, Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 322-335, 2006.

[10] R. Cai, J. Yang, W. Lai, Y. Wang and L. Zhang “iRobot: An Intelligent Crawler for Web Forums”, Proceedings of the 17th International World Wide Web Conference, pp. 447-456, 2008.

[11] B. Chess, and G. McGraw, “Static Analysis for Security”, IEEE Security and Privacy, 2(6):76-79, 2004.

[12] Clover. DOI = http://www.atlassian.com/software/clover/.

[13] D.M. Cohen, S.R. Dalal, J. Parelius and G.C. Patton, “The Combinatorial Design Approach to Automatic Test Generation”, IEEE Software, 13(5): 83-88, 1996.

[14] D.M. Cohen, S.R. Dalal, M. L. Fredman and G.C. Patton, “The AETG System: An Approach to Testing Based on Combinatorial Design”, IEEE Transitions on Software Engineering, 23(7): 437-44, 1997.

[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms" (The MIT Press, 2001, Second edn.), pp. 629-632.

[16] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, and Q. Zhang, “StackGuard: automatic adaptive detection and prevention of bufferoverflow attacks”, Proceedings of the 7th conference on USENIX Security Symposium, pp. 5-5, 1998.

[17] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, “Buffer Overflows: Attacks and Defenses for the Vulnerabilities of the Decade”, Proceedings of Foundations of Intrusion Tolerant Systems, pp. 227-237, 2003.

[18] R. Dhurjati and V. Adve, “Backwards-compatible Array Bounds Checking for C with Very Low Overhead”, Proceedings of the 28th IEEE International Conference on Software Engineering, pp. 162-171, 2006.

[19] S. Elbaum, S. Karre, and G. Rothermel, "Improving Web Application Testing with User Session Data", Proceedings of the 25th International Conference on Software Engineering, pp. 49-59, 2003.
[20] S. Elbaum, G. Rothermel, S. Karre, and M.F. II, "Leveraging User Session Data to Support Web Application Testing", IEEE Transactions on Software Engineering, 31(3), pp. 187-202, 2005.
[21] Ghttpd-1.4.4. DOI= http://gaztek.sourceforge.net/ghttpd/.
[22] P. Godefroid, “Model Checking for Programming Languages using VeriSoft”, Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 174-186, 1997.
[23] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing”, Proceedings of the 2005 ACM SIGPAN Conference on Programming Language Design and Implementation, pp. 213-233, 2005.
[24] P. Godefroid, “Compositional Dynamic Test Generation”, Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 47-54, 2007.
[25] P. Godefroid, M. Levin, and D. Monlnar, “Automated Whitebox Fuzz Testing”, Proceedings of the Network and Distributed Security Symposium, 2008.
[26] D. Gollmann, “Computer Security”, Wiley, 2006.
[27] M. Grindal, J. Offutt, and S.F. Andler, “Combination Testing Strategies: A Survey”, Software Testing, Verification, and Reliability, 15 (3): 167-199, 2005.
[28] Gzip-1.2.4. DOI= http://www.gzip.org/.
[29] B. He, M. Patel, Z. Zhang, and K.C. Chang, “Accessing the Deep Web”, Communications of the ACM, 50(5):94-101, 2007.
[30] Hypermail-2.1.3. DOI= http://www.hypermail.org/.
[31] JBroFuzz. DOI= http://sourceforge.net/projects/jbrofuzz.
[32] D.R. Kuhn, and M.J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing", Proceedings of the 27th NASA Goddard Software Engineering Workshop, pp. 91-95, 2002.
[33] D.R. Kuhn, D.R. Wallace, and A.M. Gallo, "Software Fault Interactions and Implications for Software Testing", IEEE Transactions on Software Engineering, 30(6): 418-421, 2004.
[34] Y. Lei, R. Carver, R. Kacker, D. Kung, A Combinatorial Strategy for Testing Concurrent Programs, Software Testing, Verification & Reliability, 17(4):207-225, 2007.
[35] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence, “IPOG/IPO-D: Efficient Test Generation for Multi-way Combinatorial Testing”, Software Testing, Verification & Reliability, 18(3): 125-148, 2007.
[36] Link Checker Pro. DOI = http://www.link-checker-pro.com Feb. 3, 2009.
[37] C. Liu, D.C. Kung, P. Hsia, and C Hsu, "Structural Testing of Web Applications", Proceedings of 11th International Symposium on Software Reliability Engineering, pp. 84-96, 2000.
[38] G.D. Lucca, A. Fasolino, and F. Faralli, "Testing Web Applications", Proceedings the 18th International Conference on Software Maintenance, pp. 310-319, 2002.
[39] G.D. Lucca, and M.D. Penta, "Considering Browser Interaction in Web Application Testing", Proceedings of the 5th International Workshop on Web Site Evolution, pp. 74-81, 2003.
[40] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, “Google’s Deep-Web Crawl”, Proceedings of the 34th International Conference on Very Large Data Bases, PP. 1241-1252, 2008.
[41] G. McGraw, “Software Security”, IEEE Security & Privacy, 2(2): 80-83, 2004.
[42] R.C. Miller, and K. Bharat, “SPHINX: A Framework for Creating Personal, Site-specific Web Crawlers”, Proceeding of the 7th International World Wide Web Conference, pp. 119-130, 1998.

[43] National Vulnerability Database. DOI= http://nvd.nist.gov/.
[44] Nullhttpd-0.5.0. DOI= http://www.nulllogic.ca/httpd/.
[45] Open Source Web Applications with Source Code. DOI= http://www.gotocode.com, May. 3, 2010.
[46] S. Pertet, and P. Narsimhan, "Causes of Failures in Web Applications", CMU-PDL-05-109, Carnegie Mellon University, 2005.
[47] Pine-3.96. DOI= http://www.washington.edu/pine/.
[48] M. Pistoia, S. Chandra, S.J. Fink, and E. Yahav, “A Survey of Static Analysis Methods for Identifying Security Vulnerabilities in Software Systems”, IBM Systems Journal, 46(2):265-288, 2007.
[49] Y. Qi, D. Kung, and E. Wong, "An Agent-based Testing Approach for Web Applications", Proceedings of the 29th Annual International Computer Software and Applications Conference, pp. 45-50, 2005.
[50] S. Raghavan, and H. Garcia-Molina, “Crawling the Hidden Web”, Proceedings of the 27th International Conference on Very Large Data Bases, pp. 129-138, 2001.
[51] F. Ricca, and P. Tonella, "Analysis and Testing of Web Applications", Proceedings of the 23rd International Conference on Software Engineering, pp. 25-34, 2001.
[52] J. Rőning, M. Laakso, A. Takanen and R. Kaksonen, “PROTOS –Systematic Approach to Eliminate Software Vulnerabilities”. DOI= http://www.ee.oulu.fi/research/ouspg/.
[53] O. Ruwase, and M.S. Lam, “A Practical Dynamic Buffer Overflow Detector”, Proceedings of the 11th Annual Network and Distributed System Security Symposium, pp. 159-169, 2004.
[54] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock," Web Application Testing with Customized Test Requirements—An Experimental Comparison Study", Proceedings of the 17th International Symposium on Software Reliability Engineering, pp. 266-278, 2006.
[55] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald, "Applying Concept Analysis to User-Ssession-Based Testing of Web Applications", IEEE Transactions on Software Engineering, 33(10): 643-658, 2007.
[56] SecurityFocus. DOI= http://www.securityfocus.com/.
[57] SecurityTracker. DOI= http://www.securitytracker.com/.
[58] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C”, Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 263-272, 2005.
[59] E.C. Sezer, P. Ning, C. Kil and J. Xu, “Memsherlock: An Automated Debugger for Unknown Memory Corruption Vulnerabilities”, Proceedings of the 14th ACM Conference
[60] on Computer and Communications Security, pp. 562-572, 2007.
[61] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated Replay and Failure Detection for Web Applications", Proceedings of the 20th International Conference of Automated Software Engineering, pp. 253-262, 2005.
[62] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, A. Souter, "An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications", Proceedings of the 21st International Conference on Software Maintenance, pp. 587-596, 2005.
[63] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated Oracle Comparators for Testing Web Applications", Proceedings of the 8th IEEE International Symposium on Software Reliability, pp. 117-126, 2007..
[64] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "A Case Study of Automatically Creating Test Suites from Web Application Field Data", Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 1-9, 2006.

[65] Stack Shield. DOI=http://www.angelfire.com/sk/stackshield.
[66] G.A. Stout, "Testing a Website: Best Practices", DOI= http://home.comcast.net/~glennastout/papers/TestWebsite_Stout.pdf.
[67] M. Sutton, A. Greene, and P. Amini, “Fuzzing: Brute Force vulnerability Discovery”, Addison-Wesley, 2007.
[68] Symantec Internet Security Threat Report. DOI=http://www.symantec.com/about/news/release/article.jsp?prid=20090413_01.
[69] A. Takanen, J.D. Demott and C. Miller, “Fuzzing for Software Security Testing and Quality Assurance“, Artech House, 2008.
[70] P. Tonella, and F. Ricca, “A 2-layer Model for the White-Box Testing of Web applications”, Proceedings of the 6th IEEE International Workshop on Web Site Evolution, pp. 11-19, 2004
[71] R. Xu, P. Godefroid, and R. Majumdar, “Testing for Buffer overflows with length Abstraction”, Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 27-38, 2008.
[72] X. Yuan, M. Cohen, and A.M. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 405-408, 2007.
[73] M. Zhivich, T. Leek, and R. Lippmann, “Dynamic buffer Overflow Detection”, Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools, 2005.
[74] D.R. Wallace, and D.R. Kuhn, “Failure Modes in Medical Device Software: An Analysis of 15 years of Recall Data”, International Journal of Reliability, Quality, and Safety Engineering, 8(4), 2001.
[75] W. Wang, S. Sampath, Y. Lei and R. Kacker, “An Interaction-based Test Sequence Generation Approach for Testing Web Applications”, Proceedings of the 11th International IEEE High Assurance Systems Engineering Symposium, pp. 209-218, 2008.
[76] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence, “A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications”, Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pp. 211-220, 2009.
[77] W. Wang, and D. Zhang, “External Parameter Identification Report”. University of Texas at Arlington, DOI= https://wiki.uta.edu/pages/viewpageattachments.action?pageId=35291531.
[78] Web Application Development–Bridging Gap between QA and Development. DOI=http://www.stickyminds.com.

BIOGRAPHICAL INFORMATION

Wenhua Wang received his B.S. Degree in Telecommunication Engineering from Huazhong University of Science and Technology, Wuhan, China, in June 2002, M.S. Degree in Software Engineering from Tsinghua University, Beijing, China, in June 2005, and PhD degree in Computer Science and Engineering from the University of Texas at Arlington in August 2010. His research interests include combinatorial testing, web application testing and security testing.