Applying higher strength combinatorial criteria to test case prioritization: a case study

Schuyler Manchester
Utah State University, Logan, UT 84322
schuyler.manchester@gmail.com

Renée Bryce
University of North Texas, Denton, TX 76203
renee.bryce@gmail.com

Sreedevi Sampath, Nishant Samant
University of Maryland Baltimore County
Baltimore, MD 21250
sampath@umbc.edu,samnish1@umbc.edu

D. Richard Kuhn and Raghu Kacker
National Institute of Standards & Technology
Gaithersburg, MD 20877
kuhn@nist.gov,raghu.kacker@nist.gov

Abstract

Faults in software systems often occur due to interactions between parameters. Several studies show that faults are caused by 2-way through 6-way interactions of parameters. In the context of test suite prioritization, we have studied prioritization by 2-way inter-window interaction coverage and found that this criteria is effective at finding faults quickly in the test execution cycle. However, since faults may be caused by interactions between more than 2 parameters, in this paper, we provide a greedy algorithm for test suite prioritization by n-way combinatorial coverage of inter-window interactions. While greedy algorithms that generate Combinatorial Interaction Test suites enumerate and track the coverage of all possible

JCMCC **86** (2013), pp. 51-72

t-tuples and constraints, we have noticed that our user-session-based test suites often do not contain every possible t-tuple and we can take advantage of this in our algorithm by only storing t-tuples that appear in the test suite. Our empirical study shows both time and memory usage associated with our algorithm for 3-way inter-window parameter-value interaction coverage. Further, we conduct an empirical study where we compare 2-way and 3-way combinatorial coverage of inter-window parameter interactions in terms of the rate of fault detection for a web application called Schoolmate and a user-session-based test suite. Our results show that the rate of fault detection for 2-way and 3-way prioritization are within 1% of each other, but 2-way provides a slightly better result. A closer look at the characteristics of the web application, test cases, and faults reveal that most faults are triggered by 2-way interactions. We motivate the need for future work to examine a larger set of empirical studies to identify characteristics of web applications that benefit from prioritization with higher strength inter-window event interaction coverage.

1 Introduction

Billions of dollars in costs can be incurred due to software defects caused by inadequate software testing [29]. At the same time, software testing is often performed under limited time and budget constraints. Algorithms, tools, and techniques are needed for efficient and effective testing, so that testing may be completed within a limited time budget.

Studies have found that some faults occur due to interactions between parameters, and in particular due to interactions between 6 or fewer parameters [20]. Wallace and Kuhn [18] reviewed 15 years of recall data for medical devices gathered by the U.S. Food and Drug Administration (FDA) and studied failure data on 109 medical devices. Of these 109 cases, 97% of the reported failures could have been detected by testing all pairs of parameter settings. Pairwise testing, or 2-way testing which aims to cover interactions between two parameters, is an effective approach, but pairwise testing may also miss some of the software bugs [11, 19, 20, 22, 36]. For example, in the web browser software studied by Kuhn et al. [20] only 76% of the faults were triggered by 2-way interactions between parameters. The remaining faults were triggered by higher strength interactions (e.g., 3-way through 6-way) between parameters. Thus, it is often important to test higher strength interactions between parameters during software testing.

Generating higher strength interaction test suites in the smallest number of test cases is NP-hard [10]. However, several approaches generate higher strength interaction test suites with trade-offs in effectiveness of time to

generate test suites, sizes of test suites for fixed-level or mixed-level inputs, and ability to accommodate constraints [10]. Tools such as ACTS (formerly, FireEye) [23] generate test suites that provide higher strength coverage, accommodate constraints and have been downloaded by hundreds of users.

More recently, we prioritized existing test suites by 2-way inter-window parameter-value coverage for event-driven systems, i.e., web and GUI systems [8, 9, 31]. Test suite prioritization is a test suite management technique where test cases are ordered for execution based on certain criteria such that faults may be found early in the test execution cycle. Our previous work applies test prioritization to the domain of web applications and prioritizes user-session-based test cases, i.e., test cases created from usage logs of the web system [8, 31]. In our empirical studies, we compare 2-way inter-window parameter-value interaction coverage with several other prioritization criteria. 2-way is among the best criterion in several of our subject applications. However, since existing research notes that in traditional software some faults are missed by 2-way test suites, we decided to investigate higher strength prioritization strategies, such as 3-way.

In this work, we examine a greedy algorithm to prioritize by t-way combinatorial coverage. Bryce et al.'s previous work generates covering arrays that represent t-way interaction test suites [4, 5, 6, 7]. The inputs that they examined to generate test suites were significantly smaller and they focused on obtaining 100% coverage of all t-way interactions. However, in our application of test suite prioritization, particularly for user-session-based test suites for web applications, we found that the test cases are much larger in terms of the length of test cases, and they often have an incomplete coverage of t-tuples. For example, a web application may allow users to select a set of dates, but users may not try selecting every possible combination of months, days, and years available. Consider that the parameters are continuous, but also can be considered discrete (e.g., although dates are continuous, the dates entered will generally be the current date +/- two years). If we use this estimate it gives approximately 1460 different levels or values for any date entered given the options for month, day, and year).

To generate a combinatorial interaction test suite for t-way coverage, previously proposed greedy algorithms require that we enumerate every possible t-way interaction among the parameters, store these in memory, and track the coverage of these tuples as tests are ordered. However, this is prohibitive for large test suites. In our application of user-session-based testing, we have found sparse coverage of t-tuples as users do not systematically cover every possible t-way combination when using the web application. Thus, our application of test suite prioritization by t-way interaction coverage poses the challenge that test cases are often much longer

than inputs that we have considered in combinatorial interaction test suite generation in the past. On the other hand this application offers the opportunity that exhaustive enumeration of every possible t-tuple is not necessary and the algorithm can simply store the t-tuples available in the test suite.

Our contribution in this paper is two-fold: (1) an algorithm that only uses memory for valid t-tuples in a test suite, and (2) an empirical study that examines 3-way inter-window parameter-value interaction coverage for test suite prioritization. The algorithm that we introduce in this paper takes advantage of the incomplete coverage of t-way interactions in order to use less memory and accommodate larger inputs. We evaluate the time and memory requirements of the algorithm and the fault finding effectiveness of the prioritized test orders with a case study on an open source web application, Schoolmate. We gather user sessions for Schoolmate and prioritize them using our 3-way algorithm. We contrast the effectiveness of 3-way with the other prominent prioritization criteria (including 2-way) and measure the rate of fault detection, i.e., how quickly the test order locates faults. The rate of fault detection is the most commonly used measure of effectiveness of a prioritization criterion [28]. Though we evaluate the algorithm with a case study of web applications, the algorithm can be applied when working with test suites of any software systems that have sparse t-tuple coverage.

In the remainder of this paper, Section 2 presents background on combinatorial test suite prioritization, Section 3 describes our algorithm, Section 4 presents our experiments, Section 5 summarizes our findings, and Section 6 provides conclusions and areas of future work.

2 Background and Related Work

In this section, we discuss related work in two areas (1) web applications and user-session-based testing, and (2) test suite prioritization and combinatorial strategies for prioritization.

2.1 Web application testing

Several approaches exist to generate test cases for web applications. Tools, such as HTTPUnit [16] and RationalRobot [26] allow testers to record and play back test sequences and to measure performance. Other tools check for broken links, validate HTML code, and measure performance. More semi automated approaches generate test cases for web applications. For instance, Veriweb offers a simple solution that starts at a given URL

User 1
index.php
showbooks.php?book_name="java for beginners"&book_type="programming"
buybooks.php?book_id="7"
shippingMethod.php?carrier="ups"&type="ground"

Table 1: Example User-session-based Test Case

and non-deterministically traverses links in a web application [3]. Other approaches exist to generate test cases for web applications from models of the web system, such as finite state machine, UML, etc. [15, 37, 24, 2, 21, 27]. Offut et al. use HTTPUnit and HtmlUnit to run tests that bypass client-side checks [25]. Additional work to test Rich Internet Applications exists but is outside of the scope of our work here. We focus on a particular type of web testing that occurs during the maintenance phase of the system, user-session-based testing.

User-session-based testing. In user-session-based testing, test cases are automatically constructed from web logs for use in regression testing. Since web applications are accessible through the Internet, each HTTP POST and GET request that a user makes is written to a log file. The logs are then parsed into test cases by using the IP addresses, cookies, and time stamp for each POST and GET request in order to identify the steps of each user and to create the user-session-based test cases [12, 34, 35]. A user session is, thus, a sequence of base requests and parameter name-value pairs associated with the requests. Table 1 shows an example user session for a bookstore application.

Existing work on user-session-based testing falls under the categories of test case generation, test suite prioritization, test case reduction, and test suite repair. Elbaum et al. conduct empirical studies and show that user-session-based testing is a good option to augment white box testing techniques as they found different faults [12]. Sampath et al. [32] and Sprenkle et al. [35] present a framework for user-session-based testing of web systems. The test case creation heuristics presented in their work is leveraged in this paper but extended as we parse Apache web server logs and provide an XML format for test cases [30]. While there are advantages to user-session-based testing, two major problems arise over time: (1) user-sessions may become invalid during regression testing (i.e., the structure of the web application changes, including page names, links, options on a page, etc.) and (2) a large number of user-sessions build up, making it unrealistic to run all tests in practice. Alshahwan et al. work on the first issue of repairing user-session-based test cases for use in regression testing [1]. Two approaches have been taken to address the second issue of managing large

test suites, that of test suite prioritization [8, 31] and reduction [34, 33]. In this work, we focus on test case prioritization.

2.2 Test case prioritization

Test case prioritization is formally defined by Rothermel et al. [28] as: Given T, a test suite, Π, the set of all test suites obtained by permuting the tests of T, and f, a function from Π to the set of real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi, f(\pi) \geq f(\pi')$. In this definition, Π refers to the possible prioritizations of T and f is a function that is applied to evaluate the orderings.

Several criteria have been applied for test case prioritization of traditional systems [13, 28]. Bryce et al. examine 2-way and 3-way inter-window event coverage for test case prioritization on GUI applications [9]. For all the subject applications with more than two windows, they find that 2-way and 3-way alternate in providing the best overall rate of fault detection. This work provides some motivation for us to explore the application of 3-way inter-window parameter-value interaction coverage in the domain of web applications.

Bryce et al. [8] and Sampath et al. [31] examine several prioritization criteria, including the combinatorial criterion, pairwise inter-window parameter-value interaction coverage (2-way), applied to user-session-based test suites and empirically evaluate them on three web applications, including an online bookstore, a course project manager, and a conference management system. All three applications were seeded with faults. They found that prioritization criteria based on the *longest tests with respect to the number of POST/GET requests*, *longest tests with respect to the number of parameters that users assigned values* and *2-way combinatorial coverage of inter-window interactions* are efficient techniques as compared to the original order in which test cases were logged or ordered at random.

However, since existing literature recognizes that certain faults are detected by interactions between parameters that are stronger than pairwise interactions, we are interested in investigating this hypothesis for web applications. In the remainder of this paper, we propose an algorithm for n-way combinatorial interaction coverage and present an empirical evaluation of the efficiency and effectiveness of the algorithm for t=3.

3 t-way prioritization for test suites with sparse t-tuple coverage

Our previous work focuses on generating covering arrays for Combinatorial Interaction Testing (CIT) [4, 5, 6, 7]. In such test suite generation, we generate a covering array in which all t-tuples, with the exception of constraints, are covered at least once. However, in our work on test suite prioritization, we noticed that the user-session based test suites did not contain all possible t-tuples [8, 31]. For instance, we examined three web applications and found that the associated user-session-based test suites did not contain an exhaustive collection of all t-tuples of parameter-values between windows. This makes sense because a web application has many fields in which different users could manually enter personal data such as user ids, passwords, mailing addresses, and other textual data. This observation has led us to design an algorithm that does not identify all parameter-values in a system and enumerate the possible t-tuples, but rather only stores the t-tuples that appear in the test suite.

In this section, we illustrate t-way test suite prioritization for $t = 2$ and $t = 3$ and then present the prioritization algorithm.

3.1 Example of t-way prioritization

Consider an e-commerce application where users purchase and ship items. Table 2 defines four pages of the web application. The user may select one of three options for the *shipment time* parameter on the first page, one of three options for the *postal carrier* parameter on the second page, one of three options for *tracking* parameter on the third page, and one of three options for *insurance* parameter on the fourth page. Selecting different options will execute different lines of code in the system. For instance, if the user selects any *tracking* option other than "None", the system generates a unique tracking identifier and directs the user to a separate page that describes the conditions of the desired tracking. Thus, having test coverage for the different values for tracking could potentially uncover a fault that might have been overlooked by a different test.

Table 3 contains an example test suite that accesses the pages. This test suite contains four different test cases and covers twenty unique 2-tuples and fifteen unique 3-tuples as shown in Table 4. The tuples represent the *inter-window parameter-value interactions* in the test case.

When prioritizing by 2-way, we select the first test case such that it covers the largest number of 2-tuples. The second column of Table 4 shows

Page 1, Shipment Time	Page 2, Carrier	Page 3, Tracking	Page 4, Insurance
5 - 10 days	USPS	Status tracking (Stat)	up to $100
1 - 3 days	UPS	Signature confirmation (Sig)	up to $1000
overnight	Fedex	None	None

Table 2: Web application example with 4 parameters.

Test case	Ship Time	Carrier	Track	Insur	Parameter-Values (P-V) Covered
t_1	5-10 days	USPS	None	$100	Ship Time:5-10 days, Carrier:USPS, Track:None, Insur:$100
t_2	1-3 days	USPS	None	$100	Ship Time:1-3 days, Carrier:USPS, Track:None, Insur:$100
t_3	overnight	UPS	Sig.	$1000	Ship Time:overnight, Carrier:UPS, Track:Sig., Insur:$1000
t_4	1-3 days	Fedex	Stat.	$100	Ship Time:1-3 days, Carrier:Fedex, Track:Stat., Insur:$100

Table 3: Test suite example of Table 2 web application.

that all four test cases cover six 2-tuples. We then break the tie at random, select t_2, and mark the pairs in this test as covered. We then examine which of the remaining tests cover the most remaining uncovered pairs. Test case t_1 covers three uncovered pairs, t_3 covers six new uncovered pairs, and t_4 covers five uncovered pairs, so we choose t_3. We mark the pairs in t_3 as covered and examine the last two remaining tests to select the test that covers the most uncovered pairs. We select test t_4 as it covers the most uncovered pairs. The ordering for 2-way prioritization is then $\{t_2,t_3,t_4,t_1\}$.

To prioritize by 3-way, we select the first test case that covers the most 3-tuples. All four test cases cover four 3-tuples, so we break the tie at random and select t_3. We then mark the 3-tuples covered in test t_3 as covered. In the next iteration, there is a tie among all of the tests as they cover the same number of 3-tuples, so we break the tie at random and select t_2. In the next iteration, t_4 covers more uncovered 3-tuples than t_1, so we select t_4. Finally, we add the last test case, t_1 to the test suite. The final ordering by 3-way is $\{t_3,t_2,t_4,t_1\}$.

3.2 Algorithm for t-tuple prioritization

Phase 1: Preprocessing. We iterate through each test cases t_i in the test suite TS. For each URL, we identify each parameter, p, that has been assigned a value, v, on each page. We refer to the assignment of a value to a parameter as a "parameter-value". We then create a list of all t-way inter-window parameter-value interactions (t-tuples) in the test suite and

Test case	2-way P-V Covered	3-way P-V Covered
t_1	(Ship Time:5-10 days, Carrier:USPS), (Ship Time:5-10 days, Track.:None), (Ship Time:5-10 days, Insur:$100), (Carrier:USPS, Track.:None), (Carrier:USPS, Insur:$100), (Track.:None, Insur:$100)	(Ship Time:5-10 days, Carrier:USPS, Track.:None), (Ship Time:5-10 days, Carrier:USPS, Insur:$100), (Ship Time:5-10 days, Track.:None, Insur:$100), (Carrier:USPS, Track.:None, Insur:$100)
t_2	(Ship Time:1-3 days, Carrier:USPS), (Ship Time:1-3 days,Track.:None), (Ship Time:1-3 days, Insur:$100) (Carrier:USPS, Track.:None) (Carrier:USPS, Insur:$100) (Track.:None, Insur:$100)	(Ship Time:1-3 days, Carrier:USPS, Track.:None), (Ship Time:1-3 days, Carrier:USPS, Insur:$100), (Ship Time:1-3 days, Track.:None, Insur.:$100), (Carrier:USPS, Track.:None, Insur:$100)
t_3	(Ship Time:overnight,Carrier:UPS), (Ship Time:overnight, Track.:Sig.), (Ship Time:overnight, Insur:$1000) (Carrier:UPS, Track.:Sig.) (Carrier:UPS, Insur:$1000) (Track.:Sig., Insur:$1000)	(Ship Time:overnight, Carrier:UPS, Track.:Sig.), (Ship Time:overnight, Carrier:UPS, Insur:$1000), (Ship Time:overnight, Track.:Sig., Insur:$1000), (Carrier:UPS, Track.:Sig., Insur:$1000)
t_4	(Ship Time:1-3 days, Carrier:Fedex), (Ship Time:1-3 days,Track.:Stat.), (Ship Time:1-3 days, Insur:$100) (Carrier:Fedex, Track.:Stat.) (Carrier:Fedex, Insur:$100) (Track.:Stat., Insur:$100)	(Ship Time:1-3 days, Carrier:Fedex, Track.:Stat.), (Ship Time:1-3 days, Carrier:Fedex, Insur:$100), (Ship Time:1-3 days, Track.:Stat., Insur:$100), (Carrier:Fedex, Track.:Stat., Insur:$100)

Table 4: 2-way and 3-way P-V covered in test suite described in Table 3.

store them in our *tuplesList*. This preprocessing stage allows us to only store tuples in memory that are contained in the test suite.

Phase 2: t-way Prioritization The greedy algorithm selects the test case that has the largest count of uncovered t-tuples from the *tuplesList*, marks those t-tuples as covered (i.e., removes them from the *tuplesList*), and then repeats this process until the entire test suite has been prioritized. In each iteration, for each test case, the $tCountMax$ is computed as the number of uncovered t-tuples that are in the test case. The test case with the highest $tCountMax$ is added to the test suite and then the t-tuples that are covered in this test case are removed from the *tuplesList* that stores the "uncovered t-tuples". Figure 1 provides pseudocode for this algorithm.

In the next section, we present our empirical study that shows the scalability of the algorithm for $t = 3$ and the effectiveness of the test orders.

```
// Preprocessing of test suite
sizeOfTestSuite = 0
foreach test case t_i in the test suite TS
  foreach URL url in t_i
    foreach param p assigned a value v in t_i
      tuple tu = url + p + v
      if tuplesList does not contain tu
        tuplesList add tu
    end foreach
  end foreach
  sizeOfTestSuite++
end foreach

// Test Suite Prioritization
bestTest = select a test that covers the most unique t-tuples from tuplesList
mark test_bestTest as used
selectedTestCount = 1
while(selectedTestCount < sizeOfTestSuite)
  tCountMax = -1
  for j=1 to (sizeOfTestSuite-selectedTestCount)
    if test_j is not used
      compute tCount as the number of newly covered t-tuples from tuplesList in test_j
      if(tCount > tCountMax)
        tCountMax = tCount
        bestTest = j
      else if(tCount == tCountMax)
        break the tie at random
  end for
  add test_bestTest to T_pi
  mark test_bestTest as used
  remove the tuples that appear in test_bestTest from tuplesList
  selectedTestCount++
end while
```

Figure 1: Algorithm for test suite prioritization by t-way combinatorial coverage.

4 Empirical Study

The main research questions in our study are

1. How does the t-way prioritization algorithm scale, for t=3, for user-session-based test suites?
2. How effective are the prioritized test orders generated by 2-way and 3-way at detecting faults?

4.1 Subject application

Our subject application is an open-source web-based application called Schoolmate. It is written in PHP with a MySQL backend. Schoolmate is designed as a solution for elementary, middle, and high schools to manage classes, registration, assignments, and grades. There are four different

Description	Schoolmate
Files	63
Unique parameter-values in test suite	2,611
LOC	6652
PHP methods	15
JavaScript methods	70
Branches	618
Total number of tests	125
Total number of tests with Administrator users	59
Total number of tests with Student users	44
Total number of tests with Teacher users	55
Largest count of gets/posts in a test case	193
Average count of gets/posts in a test case	36.5
Largest count of parameters in a test case	874
Average count of parameters in a test case	163.04
2-way parameter-value interactions covered in test suite	278,109
3-way parameter-value interactions covered in test suite	477,450
Number of seeded faults	66

Table 5: Characteristic of subject application and test suite.

types of users that can log into this application: 1) Admin 2) Teacher 3) Parent and 4) Student. The parent and student have identical web pages that they can access, except a parent will have a list of their assigned children. Therefore we refer to this functionality as Parent/Student in the remainder of the paper. Table 5 describes the characteristics of the application, test suite and seeded faults.

4.2 Test Suites

The test suites for this study were gathered by an undergraduate software testing class. The class was instructed to login to the web application and test out as many web pages as possible. The test cases are constructed using the IP addresses that are associated with each GET/POST request. If there is more than a 45 minute break in between a GET/POST request from the same user, we begin a new test case. We initially collected a large test suite with 125 test cases, but then broke it down into three smaller test suites where each test suite contains tests for either an Administrator, Teacher, or Parent/Student user. While there is overlap in the code that some of the user types access, we split the test suites based on user types because different user types are required in order to access certain parts of the code and seeded faults. Table 5 shows that the three test suites have between 44 to 59 test cases.

4.3 Faults

A total of 66 faults were seeded into Schoolmate by a graduate student. Each seeded fault version belongs to two categories, user type and fault classification. We seeded faults for each type of user of the system, i.e., Admin, Teacher, Parent/Student, and Any. We use the fault classification proposed by Sampath et al. [34] and Guo et al. [14] to seed faults in Schoolmate.

4.4 Prioritization criteria

We evaluate five prioritization criteria in this work.

1. **2-way.** We select the test cases in non-ascending order of the number of inter-window parameter-value combinations between two separate pages. Ties are broken at random.
2. **3-way.** We select the test cases in non-ascending order of the number of inter-window parameter-value combinations between three separate pages. Ties are broken at random.
3. **Length (Gets/Posts).** We select the test cases in non-ascending order of the number of GET/POST requests. Ties are broken at random.
4. **Number of parameter-values.** We select the test cases in non-ascending order of the number of parameter-values. Ties are broken at random. Criteria 3 and 4 are chosen because they perform well in our previous work [8].
5. **Random.** The random ordering use the `random` function that is available in Java to randomly swap the ordering of the test cases. The tool will produce a different random ordering each time that the user chooses to prioritize at random. This ordering is used as a control in our experiments.

4.5 Experimental Framework

The usage logs for Schoolmate are converted into test cases and then prioritized within our tool, CPUT [30]. The t-way prioritization algorithm is implemented in CPUT for $t = 2$ and $t = 3$, in addition to other criteria, such as length, random and frequency-based.

We then execute the test cases using a replay tool. We created a new replay tool that could execute the XML format test cases. We also conducted the fault detection experiments using the framework presented by

Sprenkle et al. [35]. Initially, we execute the test cases on a clean version of the application and save the returned files. This is the expected output, since we consider the non-fault-seeded version of the application as our gold standard. Then, one fault is seeded in the application at a time, and all the test cases are executed. The returned HTML files are saved (this is the actual output). The test oracle is then executed on the returned files to determine if the test case detects the fault. We present the results from the *struct* oracle here. The *struct* oracle [35] compares the expected and actual output in terms of the HTML tags in the files, to identify differences. A fault matrix is generated that shows how many faults and which faults are detected by each test case.

5 Results and Discussion

We first review the scalability of our algorithm on several web logs for the Schoolmate application and then present our findings on the fault-finding effectiveness of prioritization by 3-way inter-window parameter-value interaction coverage for our application and test suites.

5.1 Scalability

To study the scalability potential of our algorithm, we record each component's execution time and space requirements of the output. The experiment was run on a MacBook Pro (2.53 GHz Intel dual core processor, 8 GB 1067 MHz DDR3 RAM) running OSX Lion. In this study, we split the log into 1 day usage, 5 days usage, 10 days usage, 15 days usage (which is the entire log file), and doubled the size of the log file. To double the size of the log file, we modified the log file by changing the year from 2010 to 2011 for the date. We then replaced digits in the IP addresses and cookies to make them different. For instance, we swapped occurrences of the number '4' with the number '3'.

Table 6 summarizes the results. We present the results for each log file separately. For each log file, we present the time taken by the test case creation engine and the different prioritization criteria (column 4). We also present the space occupied by the output of the different components of the framework (column 5). We note that the test creation engine takes from a few seconds to slightly over one minute for the double log file because it parses the web log into test cases and also stores the data from test suite using the pre-processing part of our algorithm that is run before the test suite prioritization options are available to the user. Once the web log pars-

Log file	Component name	Component output	Execution time (in seconds)	Output Space requirements
1 day	Test case creation engine	XML format tests	1.503	101 KB (10 test cases)
	Test prioritization (Length)	Order file	0.001	1 KB
	Test prioritization (No. of params)	Order file	0.001	1 KB
	Test prioritization (2-way)	Order file	0.022	1 KB
	Test prioritization (3-way)	Order file	0.494	1 KB
5 days	Test case creation engine	XML format tests	4.958	499 KB (50 test cases)
	Test prioritization (Length)	Order file	0.003	1 KB
	Test prioritization (No. of params)	Order file	0.002	1 KB
	Test prioritization (2-way)	Order file	0.265	1 KB
	Test prioritization (3-way)	Order file	7.868	1 KB
10 days	Test case creation engine	XML format tests	14.03	1326 KB (110 test cases)
	Test prioritization (Length)	Order file	0.007	2 KB
	Test prioritization (No. of params)	Order file	0.007	2 KB
	Test prioritization (2-way)	Order file	2.14	2 KB
	Test prioritization (3-way)	Order file	109.25	2 KB
15 days	Test case creation engine	XML format tests	23.63	1888 KB (173 test cases)
	Test prioritization (Length)	Order file	0.01	3 KB
	Test prioritization (No. of params)	Order file	0.009	3 KB
	Test prioritization (2-way)	Order file	5.449	3 KB
	Test prioritization (3-way)	Order file	287.138	3 KB
double	Test case creation engine	XML format tests	72.23	3775 KB (348 test cases)
	Test prioritization (Length)	Order file	0.022	5 KB
	Test prioritization (No. of params)	Order file	0.019	5 KB
	Test prioritization (2-way)	Order file	15.625	5 KB
	Test prioritization (3-way)	Order file	1981.072	5 KB

Table 6: Execution time and size of test suites for Schoolmate logs.

ing and pre-processing are complete, the user may choose the prioritization technique to run. Prioritization by the length and number of POST/GET requests generally takes negligible time, reported as 0.001 seconds, up to 0.022 seconds. Prioritization by the number of parameters-values in a test is also negligible, taking 0.001 to .019 seconds. The 2-way prioritization criteria is also in the order of seconds—starting at 0.022 seconds for the 1 day log and going up to 15.625 seconds for the double log file. The 3-way prioritization criteria takes from 0.494 seconds, up to 1,981 seconds.

The output of the test case creation engine is the test suite. We see that tests created vary from 10 to 348 tests, occupying from a few bytes to 3,775 kilobytes. The output of the different prioritization criteria is stored in the order file which contains the test case names printed to file. The size of this output file also ranges from a 1 to 5 kilobytes.

From these results, we note that the time taken by the t-tuple prioritization algorithm is in the order of a few seconds. Therefore, our algorithm has the potential to scale to larger usage logs and test cases on which the test prioritization criteria need to be applied.

5.2 Rate of fault detection

We prioritized our three test suites by the prioritization criteria described above and measure the rate of fault detection with the metric, Average Percentage of Fault Detection (APFD) [28]. APFD measures the area under the curve that plots test suite fraction and the number of faults detected by the test ordering. Rothermel et al. [28] define APFD as follows: For a test suite, T with n test cases, if F is a set of m faults detected by T, then let TF_i be the position of the first test case t in T', where T' is an ordering of T, that detects fault i. Then, the APFD metric for T' is given as

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_m}{mn} + \frac{1}{2n} \quad (1)$$

Intuitively, APFD measures the area under the curve that plots test case size on the x-axis and percent of unique faults detected on the y-axis. APFD is a measure of how quickly faults are detected by the test suite, which is essential in regression testing scenarios.

Out of the total 66 faults seeded, the Student test suite with 44 test cases detects 33 faults, the Admin test suite with 59 test cases detects 50 faults, and the Teacher test suite with 44 test cases detects 39 faults. The full test suite detects 51 of the 66 faults, but in this section, we present our results with the test suites divided by the user-type. We divide the test suite into three smaller test suites based on the user type as the three different user types could access different parts of code and we did not want one user type to dominate over others.

We ran our algorithm 5 times for each combinatorial prioritization criterion and report the average of these runs in Table 7. For all three test suites, 2-way and 3-way provide the best APFD and are within 1% of each other with 2-way performing slightly better. Prioritization by length of GET/POST requests and the No. of parameter-values in a test case alternate in providing the 3rd and 4th best APFD for the Admin, Teacher, and Student test suites. The random ordering is the least effective for the Admin and Student test suites. The results among 2-way, length by GET/POST requests, No. of parameter-values, and random ordering are consistent with previous literature [8], but the results for 3-way are the first within the domain of user-session-based testing for web applications.

We examined the faults detected by the 2-way and 3-way test suites to understand our results. We noticed that we seeded faults in our system based on the existing fault classification [14, 34] without regard to whether they were faults caused by pairwise of 3-way interactions between parameters.

An example of a web application fault that could be triggered by 3-way inter-window interaction between parameters is as follows: First add a new teacher, then navigate to a page where classes are managed, and finally edit an existing class to assign the newly added teacher to the class. The fault could be introduced when assigning the teacher to an existing class where it would actually assign them as a substitute teacher instead of the main teacher. This fault thus occurs due to interactions on parameters from three different windows. An example of a web application fault triggered by 2-way inter-window parameter-value interactions is as follows: A user creates a new semester, the application sets a hidden parameter "addsemester" to the value "4" (new semester id). After this new semester is successfully saved, a fault is introduced in the web page, by replacing all of the other semester's hidden ids to "4" (new semester id). The user then attempts to delete a previously existing semester, which is when the fault actually appears. The attempted delete should have parameter "deletesemester" set to the value "1" (the previously existing semester's id), however the fault has every semester id value set to "4". Thus the interaction of these two parameter-values on different pages causes a fault when attempting a deletion of a semester immediately after adding a new semester.

We also noticed that some failures were observable only when the user session contained a certain sequence of URLs, where sequences of size 2 or size 3 caused the failure to be observed in the test case. For example the fault is introduced by a typo in the database insertion statement when a new username is created, that causes the username to save with an extra character appended to it. The fault is observed only when the user navigates to the page where they can view all users of the system. However, the fault itself is not caused by interactions of parameters across multiple pages.

Since there were no faults seeded in the system that would have been expressly identified by the 3-way test suites, it could explain why 2-way and 3-way test orders performed equivalently. In Schoolmate, there are few opportunities for 3-way interaction faults, but, in the future, we will examine other applications and seed faults that capture 3-way interactions between parameters and then evaluate the test orders to see if the 3-way orderings are better at detecting these faults than the 2-way orderings.

5.3 Threats to validity

In our prioritization algorithm, ties are broken at random. To address this, we execute the prioritization algorithm 5 times and report the average APFD of the 5 test orders. Threats to external validity are factors that may impact our ability to generalize our results to other situations. The

Admin					
% of test suite	2-way	3-way	Length-GET/POST	P-Vs	Random
10%	70.44	70.42	70.13	67.33	60.72
20%	87.02	86.76	71.48	72.65	78.08
30%	87.02	86.76	76.99	77.07	79.44
40%	88.94	88.59	83.73	85.13	80.61
50%	89.83	89.42	85.3	86.85	81.93
60%	89.83	89.42	86.03	86.85	82.33
70%	89.83	89.42	86.03	86.85	83.31
80%	89.83	89.42	86.03	86.85	84.25
90%	89.83	89.42	86.03	86.85	84.25
100%	90.01	89.6	86.06	86.88	84.37
Student					
% of test suite	2-way	3-way	Length-GET/POST	P-Vs	Random
10%	64.6	64.6	63.57	56.35	50.49
20%	64.6	64.6	63.57	63.15	56.05
30%	64.6	64.6	63.57	63.15	60.06
40%	64.6	64.6	65.39	65.12	61.22
50%	67.03	67.03	65.39	65.12	62.96
60%	67.03	67.03	65.39	65.86	63.39
70%	67.03	67.03	65.86	65.86	63.62
80%	67.03	67.03	65.86	65.86	64.11
90%	67.34	67.03	65.86	66.17	64.23
100%	67.34	67.27	66.1	66.17	64.3
Teacher					
% of test suite	2-way	3-way	Length-GET/POST	P-Vs	Random
10%	68.91	68.91	64.24	64.24	53.6
20%	68.91	68.91	66.91	68.14	58.78
30%	68.91	68.91	68.06	68.14	63.74
40%	68.91	68.91	68.06	68.14	63.92
50%	68.91	68.91	68.06	68.14	67.18
60%	68.91	68.91	68.06	68.14	68.67
70%	68.91	69.82	69.08	69.21	69
80%	70.61	69.82	69.08	69.21	69.08
90%	71.23	71.31	70.56	70.37	69.22
100%	71.23	71.31	70.56	70.37	69.36

Table 7: Average APFD of the different test orders.

first threat to validity is that we use only one web application that may not be representative of all web applications. Moreover, the characteristics of original test suites impact our results in how they were constructed and their fault detecting ability. The seeded faults also impact the generalization of our results. We use random ordering and two criteria from previous studies as a control to compare to our prioritization techniques in order to minimize this threat. Future work may examine both a larger set of web applications and real systems that have real faults that were not seeded. Threats to construct validity are factors in the experiment design that may cause us to inadequately measure concepts of interest. We use the most commonly used evaluation metric for measuring the effectiveness of prioritized test suites, APFD. However, we consider each test case to be of equal cost and each fault of equal severity. Further, we measure the scalability of

the algorithm with respect to time and space requirements. Future work may include more extensive costs and severities.

6 Conclusions and Future Work

Algorithms for Combinatorial Interaction Testing provide systematic coverage of t-way interactions in a system. Our application of t-way combinatorial coverage for test suite prioritization of user-session-based testing differs in that the test suite already exists and may not contain all possible t-way interactions in a system since test cases are generated by users that visit a website. It is unlikely for users of many systems to exhaustively cover all t-way interactions during their visits, particularly when users have unique user ids, passwords, and personal information that they enter into a system. This raises the need for an algorithm that does not enumerate all possible t-tuples to track and instead only stores the valid t-tuples in the test suite in order to save memory. Our experiments show that our approach scales well for a medium-sized web application, Schoolmate, and user base in which we capture test cases for 15 days and then double the log file. In the fault detection experiments with Schoolmate we find that prioritization by 2-way and 3-way criteria were most effective, both performing within 1% of each other. However, 2-way prioritization provided a slightly better rate of fault detection. A closer look at the data revealed that the system contained more faults triggered by 2-way than by 3-way inter-window parameter-value interactions. These results are similar to previous work by Kuhn et al. that report that systems typically have more faults triggered by lower strength interaction coverage [17].

Future work may examine a larger set of empirical studies with applications in which faults may potentially be triggered by higher strength interactions, and considering intra-window parameter interactions. Another area would be to have a slight variation on the way the t-way scores are calculated. For instance weights may be applied for preference to specific pages, parameters, or values.

7 Acknowledgements

This work is supported by the National Institute of Standards and Technology, Information and Technology Lab Award number 70NANB10H048. Any opinions, findings, and conclusions expressed herein are the authors' and do not reflect those of the sponsors. We thank student researchers,

Chelynn Day, Nilesh Chaturvedi, Sachin Jain and Devin Minson for their contributions to CPUT.

References

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *International Conference on Software Testing, Verification and Validation*, pages 298–307, Apr. 2008.

[2] A. Andrews, Jeff Offutt, and Roger Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, Jul. 2005.

[3] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic web sites. In *the Eleventh International Conference on World Wide Web*, May 2002.

[4] Renée Bryce and Charles Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification, and Reliability*, 17(3):159–182, Aug. 2007.

[5] Renee Bryce and Charles Colbourn. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal*, 48(10):960–970, Oct. 2007.

[6] Renée Bryce and Charles Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification, and Reliability*, 19(1):37–53, Mar. 2009.

[7] Renée Bryce, Charles Colbourn, and Myra Cohen. A framework of greedy methods for constructing interaction tests. In *International Conference on Software Engineering*, pages 146–155, May 2005.

[8] Renée Bryce, Sreedevi Sampath, and Atif Memon. Developing a single model and test prioritization strategies for event-driven software. *Transactions on Software Engineering*, 37(1):48–64, Jan. 2011.

[9] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *the Workshop on Domain-Specific Approaches to Software Test Automation*, pages 1–7, Sep. 2007.

[10] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58:121–167, 2004.

[11] D.R.Kuhn and V.Okun. Psuedo-exhaustive testing for software. In *30th NASA/IEEE Software Engineering Workshop*, pages 153–158, 2006.

[12] S. Elbaum, S. Karre, and Gregg Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, pages 49–59, Sep. 2003.

[13] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, Sep. 2004.

[14] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. *Empirical Software Engineering and Measurement*, pages 303–305, 2008.

[15] W. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *ESEC / SIGSOFT Foundations of Software Engineering*, pages 145–154, Sep. 2007.

[16] HttpUnit. http://httpunit.sourceforge.net/, accessed on Dec. 19, 2011.

[17] D. Kuhn, R. Kacker, and Y. Lei. Practical combinatorial testing. In *NIST Tech Report 800-142*, pages 1–70, Oct. 2010.

[18] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *Transactions on Software Engineering*, 30(6):418–421, Oct. 2004.

[19] D.R. Kuhn and M.J. Reilly. An investigation of the applicability of design of experiments to software testing. In *27th NASA/IEEE Software Engineering Workshop*, Dec. 2006.

[20] D.R. Kuhn, D.R. Wallace, and A. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[21] David C. Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *the Asia-Pacific Conference on Quality Software*, pages 111–120. IEEE Computer Society, Oct. 2000.

[22] K.Z.Bell. *Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach.* PhD thesis, North Carolina State University, 2006.

[23] Y. Lei, R. Kacker, D. Kuhn, V. Okun, and J. Lawrence. Ipog/ipod: Efficient test generation for multi-way software testing. *Journal of Software Testing, Verification, and Reliability*, 18(3):125–148, 2008.

[24] C.H. Liu, D. Kung, P. Hsia, and C.T. Hsu. Structural testing of web applications. In *International Symposium on Software Reliability Engineering*, pages 84–96, Oct. 2000.

[25] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *International Symposium on Software Reliability and Engineering*, pages 187–197. IEEE Computer Society, Nov. 2004.

[26] Rational Robot. http://www.ibm.com/software/awdtools/tester/robot/, accessed on Dec. 19, 2011.

[27] Flippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *the International Conference on Software Engineering*, pages 25–34. IEEE Computer Society, May 2001.

[28] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[29] RTI. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standarads and Technology, May 2002.

[30] Sreedevi Sampath, Renee Bryce, Sachin Jain, and Schuyler Manchester. A tool for combinatorial-based prioritization and reduction of user-session-based test suites. In *International Conference on Software Maintenance: Tool Demo Track*, pages 574–577, Sep. 2011.

[31] Sreedevi Sampath, Renee Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web application testing. In *International Conference on Software Testing, Verification and Validation*, pages 141–150, Apr. 2008.

[32] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. Composing a framework to automate testing of operational web-based software. In *International Conference on Software Maintenance*, pages 104–113. IEEE Computer Society, Sep. 2004.

[33] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, and Lori Pollock. Web application testing with customized test requirements—an experimental comparison study. In *International Symposium on Software Reliability Engineering*, pages 266–278, Nov. 2006.

[34] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web applications. *Transactions on Software Engineering*, 33(10):643–658, Oct. 2007.

[35] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *International Conference of Automated Software Engineering*, pages 253–262, Nov. 2005.

[36] 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.

[37] Wenhua Wang, Sreedevi Sampath, Yu Lei, and Raghu Kacker. An interaction-based test sequence generation approach for testing web applications. In *IEEE International Conference on High Assurance Systems Engineering*, pages 209–218, Nanjing, China, 2008. IEEE Computer Society.