Automated Web Application Testing Using Search Based Software Engineering

Nadia Alshahwan and Mark Harman
CREST Centre
University College London
London, UK
{nadia.alshahwan.10,mark.harman}@ucl.ac.uk

***Abstract*—This paper introduces three related algorithms and a tool, SWAT, for automated web application testing using Search Based Software Testing (SBST). The algorithms significantly enhance the efficiency and effectiveness of traditional search based techniques exploiting both static and dynamic analysis. The combined approach yields a 54% increase in branch coverage and a 30% reduction in test effort. Each improvement is separately evaluated in an empirical study on 6 real world web applications.**

***Index Terms*—SBSE; Automated Test data generation; Web applications**

I. Introduction

The importance of automated web application testing derives from the increasing reliance on these systems for business, social, organizational and governmental functions. Over the past ten years, internet user numbers grew by approximately 400% [17]. In 2009, online retail sales grew by 11% compared to 2.5% for all retail sales [27]. Amazon, the leading online retailer, increased its sales by 29.5% [16].

One of the advantages of web applications is their continual availability. The service provided by a web application is not limited by location or time, since geographically separated users may have concurrent access. However, these advantages impose a demand for high availability.

Web time is considered to be 5 to 7 times faster than normal time [11]: Web technologies change more frequently and their adopters seek early acquisition of market share. This pressure on development time squeezes the testing phase, especially when it is unautomated, labour intensive and therefore slow.

However, inadequate testing poses significant risks: Studies showed that trust and convenience are major factors affecting customer loyalty using web applications [4]. Both recent and historical studies have shown that online shoppers exhibit impulsive purchasing habits [7], [9], indicating that downtime can prove costly. For example, downtime was estimated to cost Amazon $25k per minute even as early as 2001 [25].

Search based testing has been used widely as a way to automate test data generation for traditional, stand alone applications, thereby making testing less reliant on slow laborious processes. Search based test data generation has also proved to be effective and complementary to other techniques [19], [21]. However, of 399 research papers on SBST,[1] only one [20] mentions web application testing issues and none applies search based test data generation to automate web application testing.

Popular web development languages such as PHP and Python have characteristics that pose a challenge when applying search based techniques such as dynamic typing and identifying the input vector. Moreover, the unique and rich nature of a web application's output can be exploited to aid the test generation process and potentially improve effectiveness and efficiency. This was the motivation for our work: We seek to develop a search based approach to automated web application testing that overcomes challenges and takes advantage of opportunities that web applications offer.

In this paper we introduce an automated search based algorithm and apply it to 6 web applications. We also introduce enhancements that seed the search process with constants collected statically and values collected dynamically and mined from the web pages provided by the application.
The primary contributions of the paper are as follows:

1) We introduce the first automated search based approach to web application testing, and a tool that implements the approach.
2) We introduce the use of Dynamically Mined Value seeding (**DMV**) into the search process. Our empirical study shows that this approach statistically significantly increases coverage in all applications studied and also significantly reduces effort in all but one. These findings may prove useful for other SBST paradigms.
3) We report the results of an empirical study of effectiveness and efficiency of our algorithms in terms of branch coverage of server-side code, fitness evaluations, execution times and fault finding ability.

The rest of this paper is organized as follows: Section II provides a brief background on search based test data generation. Section III introduces the proposed approach, whilst Section IV describes the implementation of the approach. Section V presents the evaluation together with a discussion of the results. Section VI presents related work and Section VII concludes.

[1]Source: SBSE Repository at
`http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/`

II. Background

Search Based Software Engineering (SBSE) is an approach that reformulates software engineering problems into optimization problems [21]. First, the possible solutions need to be encoded in a way that makes similar solutions (preferably) proximate in the search space. Then, a fitness function needs to be defined and used to compare solutions. Finally, operators that alter unsuccessful solutions need to be selected in a way that directs the search to a 'better' solution.

Hill Climbing is a local search algorithm often used in SBSE and found to be effective for testing [15]. A random solution is first chosen from the search space and evaluated. The neighbouring solutions of that random solution are then evaluated to find a better solution. If a better solution exists, that solution is selected to replace the previous solution. The process is repeated until a solution is found for which no further improvements can be made. The approach has the advantage of being simple and fast. However, its success depends on the randomly chosen starting solution.

Korel [18] introduced the Alternating Variable Method (AVM) into the search process. This method makes changes to one input variable while fixing all other variables. Branch distance is used to measure how close an input comes to covering the traversal of a desired branch. If the changes to a variable affect branch distance, AVM applies a larger change in the same direction at the next iteration. This 'acceleration' could cause the approach to 'over shoot' the nearest local optimum. In this case, AVM restarts its search at the previous best solution so far. The approach will then cycle through variables, repeating the same process, until the branch is covered or no further improvement is possible.

A variety of scripting languages can be used to implement web applications including PHP, Perl, Java, ASP and JSP. In this paper we shall focus on PHP; one of the most popular web scripting languages in current use [28]. We focus on PHP in order to provide a concrete web application testing tool to implement and evaluate our approach. However, many aspects of our approach may also apply to other web application languages.

III. Approach

Our approach aims to produce a test suite that maximizes branch coverage of the application under test. The algorithm starts with a static analysis phase that collects static information to aid the subsequent search based phase. The search based phase uses an algorithm that is derived from Korel's Alternating Variable Method (AVM) but which additionally incorporates constant seeding and Dynamically Mined Values (**DMV**) from the execution and web pages constructed by the application as it executes.

The rest of this section describes our approach in more detail. Section III-A discusses issues in applying search based techniques to web applications and the solutions we adopt. Section III-B describes the fitness functions we used, while Section III-C introduces our algorithms.

A. Issues and Solutions in Web Application Testing

Static and dynamic analysis phases are used to address the issues raised by web application testing and which are either absent or less pernicious in the traditional Search Based Software Testing paradigm.

Issue: Interface Determination

Description: In various web scripting languages, such as PHP, ASP and JSP, the interface is not explicitly specified. There is no 'program header' that specifies how many inputs a program expects nor what their types are. A number of global arrays (e.g. `GET`, `POST`, `REQUEST`) are usually set on the client-side before a request is submitted. These global arrays use the input name as an array index and the input's value as the corresponding array element. These arrays can be accessed by the server-side code at any point in the program.

Solution: In order to determine the 'input interface' automatically, we perform static analysis on the source code to determine the required inputs. We collect each call to the global arrays (e.g. `GET`, `POST`, `REQUEST`) and then extract the names of the inputs and the associated submit method. We also note the location where these inputs are accessed. For every branch we seek to cover, all input variables that are accessed before that branch are selected to form the input interface. To determine input types, we perform static analysis that determines the type of inputs based on the type of constants to which they are compared or from which they are assigned. Our approach is similar to that of Halfond et al. [12]. However, our analysis does not, as yet, infer types for all inputs and needs to be augmented manually.

Issue: Dynamic Typing

Description: Web development languages such as PHP, Python and Ruby are dynamically typed. All variables are initially treated as strings. If used in an arithmetic expression, they are treated as numeric at that operation. However, the same input can be treated as numeric in one expression and as a string in a different expression within the same script. This makes it hard to decide the type of variables involved in a predicate, posing a problem when deciding which fitness function to use.

Solution: To solve this problem, types of variables are checked dynamically at run-time using built-in PHP functions and then directed to the appropriate fitness function.

Issue: User Simulation

Description: In dynamic web applications, the user's interactions with the application's dynamic content need to be simulated to test the application as a whole. Web applications usually have a top level entry page that the user accesses first. User choices on the entry page are passed to the server-side code for processing. A client-side page is then generated and displayed to the user. Some applications have other top level pages that can be accessed only through these client-side pages. Identifying these top level pages raises issues when trying to generate test data automatically for an application as a whole.

Solution: Our static analysis identifies top level pages that expose new parts of the application accessible only through

client pages during the static analysis phase. A file that is not included by any other file is treated as a top level file. The test data generation process is performed for each top level file.

Issue: Dynamic Includes

Description: PHP supports dynamic includes, where the name of the included file is computed at run-time. An example of this is when the user's choice determines the natural language to be used in the text of the application.

Solution: To deal with dynamic file includes, we used a similar approach to the one proposed by Wassermann and Su [30]; for include statements that contain variables as part of the included filename, we use a safe approximation that includes any file available to the application that matches the include expression.

B. Fitness Function

The fitness function we use in our approach is similar to that used by Tracy et al. [29]. That is, for a predicate $a\ op\ b$ where op is a relational operator, fitness is zero when the condition is true and $|a - b|$ when the condition is false. A fitness of zero denotes the situation where the test vector assessed by the fitness function covers the desired branch. That is, our approach seeks to minimize fitness values throughout the search process. Like Tracey et al., a value k (in our case 1) is added to penalize incorrect test data but we add that value only in case of $<$, $>$ and $\neq$. For strings we use Levenshtein distance [23], following Alshraideh and Bottaci [3]. The Levenshtein distance is the minimum number of insert, delete and substitute operations needed to convert one string to another string. The Levenshtein distance is suitable for $=$ and $\neq$ operators. For other operators we convert the ASCII code of a string to a decimal representation and use the same fitness used for normal numeric types following Zhao et al. [32].

As a pre-processing step, compound predicates involving logical operators are expanded, using a pre-transformation, to simple relational predicates.

C. Test Data Generation Algorithms

Our algorithms for test data generation are all based on Hill Climbing using Korel's AVM [18]. When a target branch is selected, AVM is used to mutate each input in turn while all other inputs remain fixed. When the selected mutation is found to improve fitness, the change in the same direction is accelerated. To avoid 'over shoot', when fitness is close to zero we decelerate.

We note branches that we reached but failed to cover and target them on subsequent iterations. That is, a branch is reached if its immediately controlling predicate is executed, while a branch is covered if the branch itself is traversed. This 'exploration' approach eliminates the need for calculating the so-called approach level [21]. This is because we attempt to cover a branch only when it is reached i.e., all transitively controlling predicates on some path have been satisfied. We call this an 'exploration' approach because the technique maintains a 'current frontier' of reached but as yet uncovered branches, seeking to push back this 'frontier' at each top level iteration. A similar approach was used by Michael et al. [22] for evolutionary testing.

Algorithm 1 NMS: Overall Test Data Generation Algorithm: Top level units are extracted from the File Tree Analyser results. Each unit is called with no parameters to get an initial 'work list' of reached branches. For each work list branch, the input vector is mutated iteratively until the branch is covered or the stopping criterion is satisfied. Near misses and collateral coverage are recorded for later use.

Require: Application Name *AppName*
Require: Static Analysis results *AnalysisDB*
$\mathcal{U}$: queue of top level file units to be processed. Retrieved from the File Tree Analyser results.
$\mathcal{B}$: queue of branches reached but not covered.
$\mathcal{C}$: Coverage table of all branches with the best achieved distance.
$\mathcal{T}$: Test cases that achieved best distance for each reached or covered branch.
$\mathcal{F}$: set of branch and fitness values achieved for the executed test case.
Input : setOf(inputname, value)
IV: Input Vector consisting of setOf(*Input*)
Distance: holds fitness value for a certain branch.

1: $\mathcal{U}$:= getTestUnits(*AppName*, *AnalysisDB*)
2: $\mathcal{T} := \phi$
3: **for all** U in $\mathcal{U}$ **do**
4: *IV*:= ϕ
5: $\mathcal{F}$:=executeTestcase(U, *IV*)
6: $\mathcal{T}$:=updateTestdata($\mathcal{T}$,U,*IV*,$\mathcal{F}$)
7: $\mathcal{C}$:=updateCoveragedata($\mathcal{C}$, $\mathcal{F}$)
8: **while** first run or coverage improved **do**
9: $\mathcal{B}$:= getReachedBranches($\mathcal{C}$)
10: **for all** B in $\mathcal{B}$ **do**
11: initState()
12: *IV* := setInputVector(B, *AnalysisDB*,$\mathcal{T}$)
13: *Input* := NULL
14: *CurrentDistance* := getBranchDist(B, $\mathcal{C}$)
15: **while** *CurrentDistance* > 0 and not no improvements for 200 tries **do**
16: initilaizeDB()
17: *Input*:= mutateInputs(*IV*,*Input*, *CurrentDistance*, *PreviousDistance*)
18: *IV*:= replaceInputValue(*IV*,*Input*)
19: $\mathcal{F}$:=executeTestcase(U, *IV*)
20: $\mathcal{T}$:=updateTestdata($\mathcal{T}$,*IV*,$\mathcal{F}$)
21: $\mathcal{C}$:=updateCoveragedata($\mathcal{C}$, $\mathcal{F}$)
22: *PreviousDistance* = *CurrentDistance*
23: *CurrentDistance* = getBranchDist(B, $\mathcal{C}$)
24: **end while**
25: **end for**
26: **end while**
27: **end for**
28: **return** $\mathcal{T}$

At each iteration we also keep track of input values that cause any 'near misses'. A near miss is an input vector that causes fitness improvement for a branch other than the targeted branch. Near misses are used in place of random values when initializing a search to cover that branch. We call this approach 'Near Miss Seeding' (**NMS**).

More formally, our top level approach is described in Algorithm 1, which starts by calling the application with empty inputs for every top level file (Line 5). Every execution of a test case returns a list ($\mathcal{F}$) of all branches in that execution together with the distance achieved for them. This list is used to update a coverage table ($\mathcal{C}$) and the test suite ($\mathcal{T}$) for every branch that recorded an improvement in distance. A 'work list' of reached branches is extracted from the coverage table.

Every branch in the work list is then processed in an attempt to cover it. First, the state and database are initialized and the user (in this case our test tool) is logged into the application (Line 11). The input vector is then constructed using the analysis data. Values are initialized using the input values that caused the branch to be reached and random values for any additional inputs (Line 12). One input is mutated at a time and the new test case executed until the branch is covered or no improvements are possible.

Algorithm 2 describes the mutation process. If no input was selected for mutation or the last mutation did not affect distance, a new input is selected. If the last mutation caused distance to increase, a new operator is selected. If the last mutation caused distance to decrease, the operation is accelerated. Finally, the selected input is mutated (Line 12). Algorithms 1 and 2 describe our unaugmented search based approach.

We make a few modifications to use constants collected from the source code of the application in the search process. Constants are used to initialize inputs in Line 12 of Algorithm 1 instead of random values and assigned to the input in Line 12 of Algorithm 2 when the input type is string. This approach was first proposed by Alshraideh and Bottaci [3] in the context of testing traditional applications. We call this process 'Static Constant Seeding' (**SCS**).

We also modify the algorithm to seed values dynamically mined during execution into the search space in a similar way to static constants. We call this process 'Dynamically Mined Value' (**DMV**) seeding. More details about **DMV** are given in the next section.

D. Dynamically Mined Value Seeding

Constants collected statically are specific to the application and therefore can aid in covering branches that depend on these constants. In a similar way, collecting values dynamically from predicates can also prove beneficial. These collected values are not only specific to the application but also specific to the predicates from which they were collected. We seed these values into the search space when targeting their associated branches.

Web applications offer a wealth of valid input values in their dynamically generated HTML. The output HTML is returned to the user (in our case the tool) in a structured form that makes it possible to extract these values and to associate them with their respective input fields. The source of these values is `Form` data and embedded URLs. `Form` definitions can contain valid values in drop-down menus, check boxes, radio buttons and hidden values. Embedded URLs can also contain valid values in their query strings. These fields are populated from different sources that can include databases, configuration files and/or external data sources. The input fields associated with these values can affect coverage indirectly or through hard to cover branches. An example that illustrates the potential of **DMV** is the following predicate taken from one of the applications (PHPSysInfo) we used in our study:

```
if (file_exists($lng.'.php')) {
..}
```

Generating a value for input `$lng` that would cover the true branch of the control statement might be hard since the condition is a flag condition that would not provide much guidance to the search process. However, a `Form` in the dynamically generated HTML (Figure 1) has a drop-down menu that contains a list of language options available for the application (i.e. the language file exists in the application's file system). Using one of the values in the drop down menu for this input field would cover the branch.

Algorithm 2 Mutation Algorithm (mutateInputs): The algorithm determines the next input to which the search moves, based on distance achieved by the last mutation. It also decides when to change the mutation operator and when to accelerate the selected operation

Require: *IV*, *Input*, *CurrentDistance*, *PreviousDistance*, *AnalysisDB*
1: **if** *Input* is NULL or *CurrentDistance* = *PreviousDistance* **then**
2: *Input* := selectNewInput(*Input*,*IV*)
3: **else**
4: **if** *CurrentDistance* > *PreviousDistance* **then**
5: changeMutationOperator()
6: **else**
7: **if** *CurrentDistance* < *PreviousDistance* **then**
8: accelerateOperation()
9: **end if**
10: **end if**
11: **end if**
12: *Input*:= mutate(*Input*,*AnalysisDB*)
13: **return** *Input*

In our approach we mine the HTML returned when executing test cases (Line 19 of Algorithm 1) to collect such values and subsequently seed them into the search when mutating the inputs associated with them.

IV. The SWAT Tool

We developed a tool called the 'Search based Web Application Tester' (SWAT) to implement our approach and

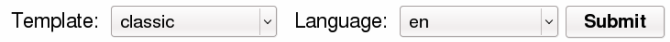


Fig. 1. Form taken from PHPSysInfo

embed it within an end-to-end testing infrastructure. SWAT's architecture is illustrated in Figure 2. The tool is composed of a pre-processing component, the Search Based Tester and the Test Harness.

The original source code is passed through the Predicate Expander and Instrumenter. This produces a transformed version of the code where predicates with logical operators are expanded and control statements are instrumented to calculate fitness in addition to the predicates' original behaviour. The code is also instrumented to collect run-time values to be used in subsequent Dynamically Mined Value seeding.

The Static Analyser performs the analysis needed to resolve the issues mentioned in Section III-A. The results are stored in the Analysis Data repository and used later by the Search Based Tester. The Constant Extractor mines the code for constants to be used in subsequent Static Constant Seeding. The Input Format Extractor analyses the code to extract the input vector. The File Tree Analyser generates a tree in which nodes denote files and edges denote include relationships. This information is used to determine the top level test units to be processed.

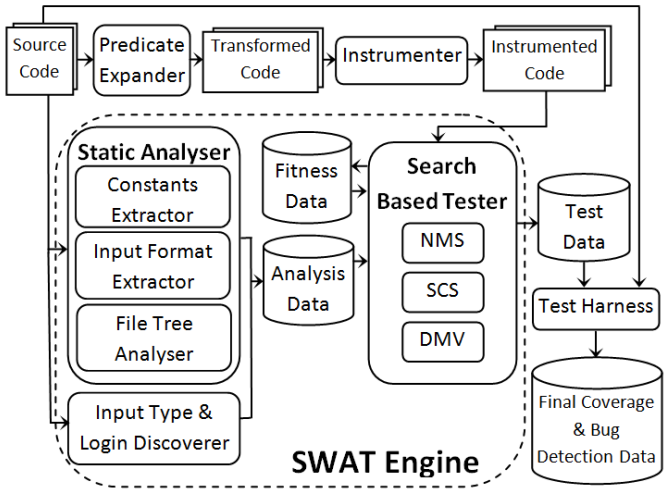


Fig. 2. SWAT tool architecture

The 'Input Type and Login Discoverer' component performs a simple combination of static and dynamic analysis to infer input types and to identify the login process. This is the only component for which results need to be augmented manually; this is because the technique for type inference is unable to infer types for all inputs. The Login Discoverer is used to dynamically extract the variables used to store the username, password, login URL and any other inputs that need to be set for login. The concrete values for username and password are provided to the tool.

Stratego/xt [8] and PHP-Front [6] were used to develop the Predicate Expander, the Instrumenter, the Static Analyser and the static analysis part of the Input Type and Login Discoverer. Stratego/xt is a program transformation language and PHP-Front provides libraries for Stratego/xt supporting PHP. The Input Format Extractor was taken from the PHP-Front project with minor alterations. All other transformation tools have been developed from scratch. The dynamic part of the Input Type and Login Discoverer was developed using Perl and Java.

The Search Based Tester uses the transformed source code and the analysis data to implement the input generation described by Algorithms 1 and the augmentations needed for **SCS** and **DMV**. The Test Harness uses the generated test data to run the tests on the original source code and to produce coverage and bug data. When a test case is executed, the generated HTML together with the web Server's error logs are parsed for PHP execution errors. The Search Based Tester and Test Harness are implemented in Perl and use the HTTP, HTML and LWP libraries.

V. Evaluation

For the evaluation, we implemented three versions of the tool. Each version adds one of the enhancements described in Section III-C in the following way:

- **NMS** implements the Near Miss Seeding unaugmented approach described in Algorithm 1 in Section III-C.
- **SCS** is **NMS** with Static Constant Seeding.
- **DMV** is **SCS** with Dynamically Mined Value seeding.

Each branch was allocated the same budget of fitness evaluations for each version of the tool. In this way we can evaluate the effects of each of our enhancements on the unaugmented traditional search based approach.

We designed our experiment to answer the following research questions:

RQ1: How does each of our enhancements affect branch coverage?

To answer this question we compare branch coverage for each of the algorithms. Coverage was measured on the original untransformed application. The original application was instrumented to record coverage without the transformations to expand predicates and calculate fitness.

RQ2: How does each of our enhancements affect efficiency of the approach?

To answer this question we calculate the number of fitness evaluations needed per application and per branch. We also report the elapsed time and CPU time used in testing per application and per branch.

RQ3: How does each of our enhancements affect fault finding ability?

To answer this question we use an automated oracle to compare the fault finding ability of the test suites produced by each enhancement. The automated oracle parses PHP error logs and the HTML output for execution errors and warnings. We perform a Wilcoxon unpaired one-sided signed rank test at the 95% confidence level to determine the statistical significance of the observed results.

A. Web Applications Studied

For the evaluation we used the same PHP applications used by other research on web testing using non search based approaches [5]. These applications range from small to medium-sized applications. Table I provides a brief quantitative and qualitative description of each application.

TABLE I
THE WEB APPLICATIONS USED IN THE STUDY

App Name	Version	PHP Files	PHP ELoC	Description
FAQForge	1.3.2	19	834	FAQ management tool
Schoolmate	1.5.4	63	3,072	School admin system
Webchess	0.9.0	24	2,701	Online chess game
PHPSysInfo	2.5.3	73	9,533	System monitoring tool
Timeclock	1.0.3	62	14,980	Employee time tracker
PHPBB2	2.0.21	78	22,280	Customisable web forum

B. Experimental Set-up

We ran each of the 3 versions of the tool 30 times on each of the 6 PHP applications and collected coverage data. We provide data for repeated runs in order to cater for the stochastic nature of the search based optimization that lies at the heart of our approach. Multiple runs are samples from the space of all possible runs. With 30 runs of each algorithm, a sufficient sample size for statistical significance testing is provided. The tool was executed on an Intel Core 2 Duo CPU, running at 2 GHz with 2 GB RAM.

The applications were installed and set-up locally on the same machine used to generate inputs. The database for each of these applications was set up following an identical systematic strategy as follows: All tables are populated in a minimal manner, but such that each table contains at least one record and a record is created for each possible value of a column of an enumerated type. All applications except one (PHPSysInfo) use a database. For configurable applications, all features were enabled where the application permitted it. Where login is required to use the web application, a valid username and password pair was supplied to the tool.

C. Branch Coverage

Table II summarizes the results obtained by the experiment. Coverage results are reported together with the number of test cases generated to achieve this coverage.

The number of covered branches increases with **SCS** for all applications. Webchess displays the highest improvement with an 85% increase in coverage. Overall, **SCS** recorded an average increase in coverage of 25.1% compared to **NMS**.

Covered branches also increase with **DMV** for all applications studied. An average improvement of 22.3% was observed in branch coverage over all applications with FaqForge showing the highest improvement.

Figure 3 shows the variation in branch coverage achieved over 30 runs of each approach. We notice that **DMV**'s lowest coverage is higher than the highest coverage of other approaches for 5 of the 6 applications. For 3 of the applications **NMS** shows little or no variation over 30 runs. For all applications studied we found a statistically significant increase in branch coverage at the 95% confidence level for **DMV** compared to **SCS** and **NMS**.

In three applications, running the same test suite twice can produce different branch coverage levels on each occasion. Eliminating this non-determinism is not desirable because it manifests important aspects of the application. Investigating the branches covered, we made the following observations: When a new game is initiated in Webchess, the user can either choose his/her preferred colour or choose random. A similar issue is found in PHPSysInfo where the user can select the template of the output or select random. Timeclock has a weather display feature that, when enabled, fetches the current weather information from the internet and displays it. The current weather information affects coverage of the branches that control the format in which this weather information is displayed.

D. Efficiency

To measure efficiency, we recorded average execution times and measured effort for each approach over 30 runs. Our reported CPU time does not include time spent calling and processing the PHP Application (which our reported elapsed time does). Effort is the ratio between total number of fitness evaluations and branches covered. Effort is the more reliable empirical assessment of the search algorithm time complexity since it is unconfounded with difficulties of measuring time in a multi process environment. However, figures for elapsed time are more useful as a rough guide to likely overall test data generation time performance for each application, which is affected by the system under test as well as the performance of the algorithms.

Effort and time results are reported in Table II. Effort decreases with each of the two algorithmic enhancements for all applications except Timeclock, for which effort increases. Overall 6 applications studied, **SCS** decreased effort per covered branch by an average of 11.2% while **DMV** decreased effort by a further 30.17%.

FAQForge is processed the fastest (1 min) while PHPBB2 is the slowest with an average of approximately 97 minutes per run for **DMV**. Of course, our tool is only a research prototype. However, even this worst case can be a accommodated within a daily build cycle, with overnight automated test data generation.

E. Fault Finding Ability

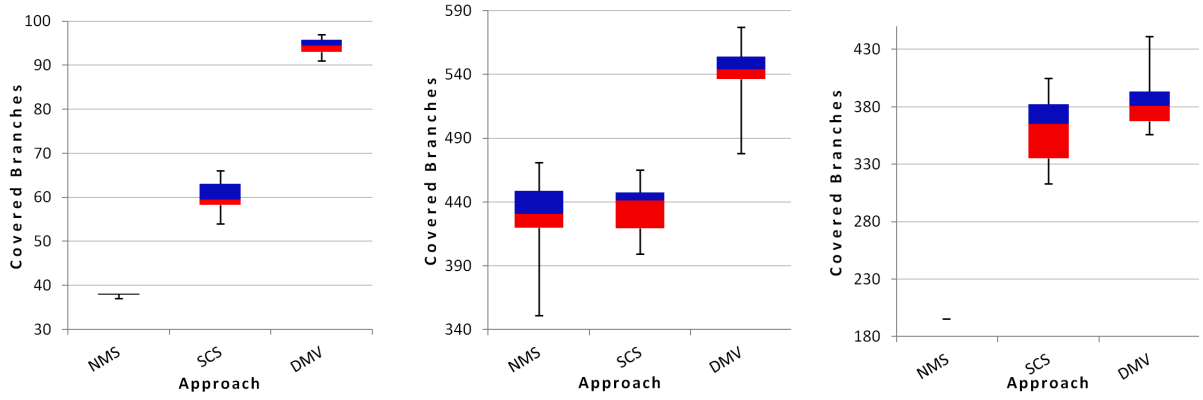
In order to ensure that our determination of whether or not SWAT has found a fault is free from experimenter bias and subjectivity we use an automated oracle. Therefore our reported results for faults found are lower bounds, guaranteed to be free from false positives. Our oracle parses PHP error log files and the output HTML page of each test case for faults. Only distinct faults counted.

In Table II crashes indicate PHP interpreter fatal execution errors; these are errors that cause the execution of scripts

TABLE II

AVERAGE COVERAGE AND EXECUTION TIME RESULTS OBTAINED BY RUNNING EACH ALGORITHM 30 TIMES FOR EACH APPLICATION WITH THE SAME BUDGET OF EVALUATIONS PER BRANCH FOR EACH VERSION. EFFORT IS THE NUMBER OF EVALUATIONS PER BRANCH COVERED. RESULTS IN BOLD ARE STATISTICALLY SIGNIFICANTLY BETTER THAN THE RESULTS ABOVE THEM USING THE WILCOXON'S TEST (95% CONFIDENCE LEVEL).

App Name	Alg	Total #branches	Test cases	Fitness Evals	Covered Branches			Elapsed Time		CPU Time		Average Faults Found		
					Num	%	Effort	Time	per Br	Time	per Br	Crash	Error	Warning
FAQForge	NMS	142	25	4809	38.0	26.7	126.7	104	2.75	28	0.734	0	0	20.0
	SCS		22	1453	**60.2**	**42.4**	**24.1**	40	**0.66**	16	**0.266**	0	**1.0**	**25.0**
	DMV		34	2177	**94.4**	**66.5**	**23.1**	64	0.69	23	0.249	0	**5.9**	**46.1**
Schoolmate	NMS	828	164	21840	428.9	51.8	50.9	949	2.21	228	0.531	2.1	13.5	75.5
	SCS		167	19037	435.5	52.6	**43.7**	712	**1.63**	229	0.525	**2.6**	**15.7**	75.5
	DMV		172	12641	**542.3**	**65.5**	**23.3**	549	**1.01**	211	**0.388**	**3.6**	**21.5**	**87.4**
Webchess	NMS	1051	21	9542	195.0	18.6	48.9	705	3.62	41	0.209	0	6.1	9.0
	SCS		43	10650	**360.9**	**34.3**	**29.5**	922	**2.55**	83	0.230	0	**16.9**	**41.7**
	DMV		45	8953	**382.6**	**36.4**	**23.4**	879	**2.30**	82	0.215	0	**20.0**	**55.3**
PHPSysInfo	NMS	1451	8	1529	300.0	20.7	5.1	5891	19.64	13	0.043	0	0	3.0
	SCS		11	1398	**315.4**	**21.7**	**4.4**	5302	**16.81**	12	0.039	0	**2.9**	**3.9**
	DMV		20	1337	**333.4**	**23.0**	**4.1**	4459	**16.37**	53	0.160	0	3.2	**4.0**
Timeclock	NMS	3567	116	7212	543.6	15.2	13.3	1083	1.99	56	0.103	0	0	155.0
	SCS		248	8445	**548.5**	**15.4**	15.4	1135	2.07	58	0.106	0	0	**155.9**
	DMV		244	12239	**655.3**	**18.4**	19.4	754	**1.15**	77	0.117	0	**1.6**	**173.2**
PHPBB2	NMS	5680	116	24690	816.6	14.4	30.2	5956	7.29	259	0.317	0	3.0	41.1
	SCS		248	22981	821.6	14.5	**28.0**	5533	**6.73**	251	0.306	0	3.0	**41.8**
	DMV		244	24080	**1007.3**	**17.7**	**23.9**	5821	**5.78**	252	**0.248**	0	**4.6**	**58.4**



(a) FAQForge (b) Schoolmate (c) Webchess

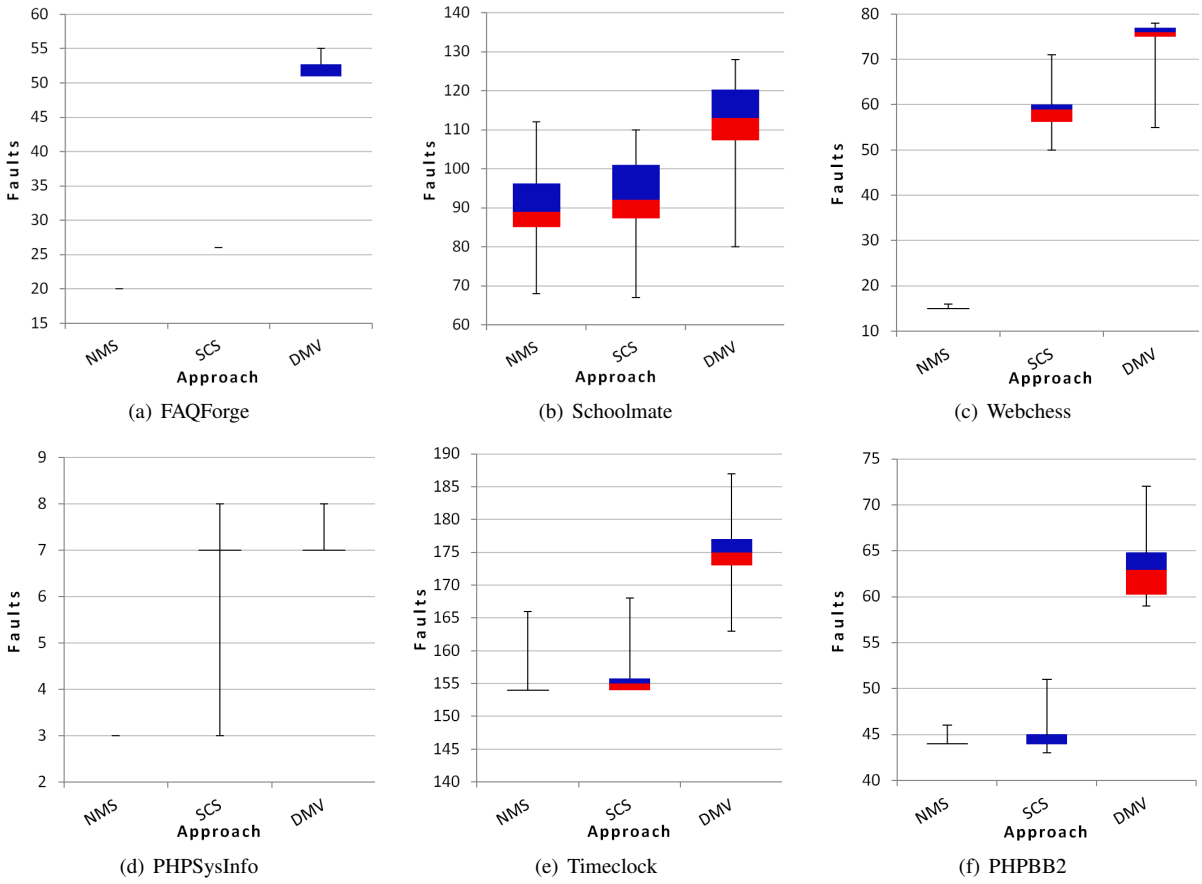
(d) PHPSysInfo (e) Timeclock (f) PHPBB2

Fig. 3. Coverage results over 30 runs of each of the 3 algorithms on each of the 6 web applications.

to abort. Errors indicate PHP interpreter 'warnings'; these are defined in the PHP manual as non-fatal errors.[2] Errors also include those parsed from the output HTML. Warnings indicate PHP interpreter notices and strict warnings.

Figure 4 shows the variation in total faults found for each application over 30 runs. Some of the faults found indicate

[2]http://www.php.net/manual/



(a) FAQForge (b) Schoolmate (c) Webchess

(d) PHPSysInfo (e) Timeclock (f) PHPBB2

Fig. 4. Fault results over 30 runs of each of the 3 algorithms on each of the 6 web applications.

a lack of adequate validation of inputs before use in critical operations such as database queries. Inputs are concatenated directly to SQL statements which could cause a security threat. In some cases where faults were parsed from the generated HTML, the SQL statement that caused the error was displayed to the user, giving hackers the opportunity to analyse the statements and form an SQL injection attack. PHPBB2 uses an input field to redirect requests to other parts of the site (e.g. after login). This input field is displayed to the user (in the query string) and can be modified to potentially gain access to server files. Other faults found include inclusion of non-existent files and wrong use of functions.

F. Answers to Research Questions

In this section we answer the research questions we posed at the start of this section, based on the empirical evidence from our experiments on the 6 web applications.

1) Answer to RQ1: How does each of our enhancements affect branch coverage?: The results of the evaluation provide evidence to support the claim that each enhancement improved branch coverage for all of the 6 applications under test. In particular, we note that **DMV** statistically significantly outperforms **NMS** and **SCS** for all 6 applications studied.

SCS also, in turn, achieved higher coverage than **NMS** for all 6 applications. Wilcoxon's test indicated these results to be statistically significant for 4 of the 6 applications. A closer look at the type of branches that were additionally covered by **SCS** reveals that they are predominantly string predicates that involve constants. In Schoolmate and PHPBB2 the improvement in branch coverage was not statistically significant. This can be attributed to the fact that those two applications have relatively fewer constant-using string predicates. The average percentage of predicates that involve a string constant overall applications is 27% while for Schoolmate for example the average is only 15%.

Branch coverage results for **DMV** compared to **SCS** statistically significantly increase for all 6 applications. By analysing additionally covered branches, we find that **DMV** performs better than **SCS** at covering constant-using string predicates. It also covers string predicates that are variable-using rather than constant-using.

As might be expected, both algorithms appear to achieve higher coverage when the application contains key string predicates that uncover unexplored parts of the application under test. Our empirical results suggest that this situation may be sufficiently common for seeding to be very effective.

2) Answer to RQ2: How does each of our enhancements affect efficiency of the approach?: Effort decreases statistically significantly by using **SCS** rather than **NMS** for all applications except one. Timeclock is the only application were effort

increased instead of decreasing. This may be caused by the nature of constants mined from Timeclock: Float constants mined from Timeclock had a precision as high as 16 decimal places while the highest for all other applications was 3 decimal places. Assigning these constants to input variables when initializing the input vector may have, in some cases, not assisted and possibly even impeded the search process for Timeclock.

Using **DMV** caused effort to decrease for all applications except one. Wilcoxon's test indicated that this reduction was statistically significant in the 5 cases where the reduction was observed. Like **SCS**, effort for Timeclock increased with **DMV**. This could be caused by the fact that the applications were transformed before running the tool to decompose 'And' and 'Or' statements. However, coverage is measured on the untransformed version. Our transformation is merely an enabling testability transformation [14] and it would be unreasonable to attach any importance to coverage of our own internal representation. However, this does account for the increase in effort for Timeclock: While for the untransformed version of Timeclock the effort for **NMS** is 13.3 increasing to 19.4 for **DMV**, for the transformed version of Timeclock this effort reduces from 13.0 for **NMS** to 12.4 for **DMV**.

3) Answer to RQ3: How does each of our enhancements affect fault finding ability?: The number of errors and crashes statistically significantly increased by using **SCS** for all applications where errors and crashes were found. The same is observed for warnings for all applications except Schoolmate. This may be tied to the fact that coverage for Schoolmate using **SCS** does not increase statistically significantly.

All error types record an increase in numbers when using **DMV** for all applications where faults are found. Wilcoxon's test indicates that this increase is statistically significant for 5 out of 6 applications.

G. Threats to Validity and Limitations

The internal threats that could affect the validity of results depend on the set-up of the applications. Results could be affected by database state and configuration. To minimize bias, a systematic procedure was defined for populating the database and configuring the application. This procedure ensures that no prior knowledge about the applications under test can be exploited and is performed in the same manner for all applications.

External threats are related to the choice of applications and the degree to which one can generalize from the results obtained from those chosen for the study. The applications were selected to provide compatibility with previous research on testing web applications. However, they are real applications used by real users as the high number of downloads from Sourceforge indicates.

We took steps to insure that our results would be reproducible. The state of the application was initialized before each test case is called. The applications used are open source and thus publicly available. Bug reports are available online.[3]

[3]http://www.cs.ucl.ac.uk/staff/nalshahw/swat

Limitations Our overall aim is to produce a fully automated testing approach that generates tests, runs them and reports faults found entirely automatically. However, there are some aspects of the overall approach that are not, as yet, fully automated. Deciding input types is partially manual. Username and password information also needs to be provided by the user. Some data types used in predicates are not yet supported by the Instrumenter. Enhancing the tool to handle all data types and defining better fitness functions for arrays and objects may further improve coverage.

VI. Related Work

Search Based Software Engineering (SBSE) has been widely used in testing both functional and non-functional properties of applications [1], [21]. However, despite much work on SBSE, search based test data generation has not previously been applied to automated test data generation for web applications.

Marchetto and Tonella [20] extended their state based approach for testing Ajax web applications by using a search based technique to select test case sequences. The approach used a Hill Climbing algorithm to construct test sequences that maximize the diversity of the test suite. This is important because of the asynchronous nature of Ajax application communication with the server, which is absent from non-Ajax applications such as the ones we study.

In the present paper we applied search based testing to web applications and introduced using Dynamically Mined Value to the search process. We also imbued our work with ideas collected and adopted from several previous approaches in the SBST literature. Our approach retains the Alternating Variable Method (AVM) introduced by Korel [18] in adopted form. Our approach to search exploration adopts a similar systematic technique for branch order to that used by Michael et al. [22] for C programs using evolutionary algorithms. Michael et al. also keep track of inputs that caused the branch to be reached to use as seeds. However, the overall algorithm and application domain, being stand alone C applications, was very different to ours. Seeding constants gathered from the source code to the search space and using the Levenshtein distance to measure fitness for strings was first proposed by Alshraideh and Bottaci [3]. We used the same idea to enhance our tool and applied it to larger scale web applications implemented in PHP. Zhao et al. [32] also used search based techniques to generate string test data for boundary value testing.

Halford et al. [12], [13] introduced an algorithm that uses symbolic execution of the source code to group inputs into interfaces. The approach was applied to Java applications, while our approach is applied to PHP applications. Wassermann et al. [30], [31] also used symbolic execution to generate test data for web applications. However, their work focused on SQL injection attacks and examined only functions that call database queries.

Artzi et al. [5] automatically generated test cases for dynamic web applications using Dynamic Symbolic Execution. Their approach also targeted PHP applications . However, the

two approaches differ in the test adequacy criteria (statement coverage vs. branch coverage). Their approach also produces a different number of test cases since all test cases generated during the run of the tool are collected. Their algorithm minimizes the test suite in regards of faults found using an automated oracle (similar to the one we use in the evaluation) for fault localization. While our approach seeks to produce a test suite that achieves branch coverage.

Using session data to test web applications was first proposed by Elbaum et al. [10] and later extended by others [2], [26]. In the study by Elbaum et al. [10] a comparison between structural testing and testing using session data was performed. The results showed that structural testing such as ours and reuse of session data can be expected to be complementary testing techniques.

The nature of web applications makes it easily possible for users to bypass input validations coded in the client-side interface (e.g. JavaScript) by submitting requests directly to the server. This poses a threat that bypass testing exposes to help prevent such attacks. Therefore, Offutt et al. [24] introduced the idea of bypass testing. In our work we also 'bypass' the interface to generate data for the server-side code directly. The test data, although not specifically generated for bypass testing, could be used for that purpose.

VII. Conclusion

In this paper we introduced a set of related search based testing algorithms, adapted for web application testing and augmented the approach with static and dynamic seeding. We introduced a tool, SWAT, that implements our automated test data generation approach for PHP web applications. Our approach draws on more than ten years of results reported for search based testing, as applied to conventional stand–alone applications, seeking to exploit and build upon best practice and proven results where possible. However, as the paper shows, there are many issues raised by web application testing, such as dynamic type binding and user interface inference that create novel challenges for search based testing that have not previously been addressed. We also show how our novel Dynamically Mined Value seeding approach can significantly reduce effort and increase effectiveness for web application testing.

We report on an empirical study that evaluates our approach on 6 PHP web applications ranging in size up to 20k LoC, presenting results concerning coverage, various measures of test effort and also an analysis of fault detection ability. Our tool detected an average of 60 faults and 424 warnings over all 6 applications studied.

References

[1] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51:957–976, June 2009.

[2] Nadia Alshahwan and Mark Harman. Automated session data repair for web application regression testing. In *ICST '08*, pages 298–307, 2008.

[3] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[4] Rolph E. Anderson and Srini S. Srinivasan. E-satisfaction and e-loyalty: A contingency framework. *Psychology and Marketing*, 20(2):123–138, 2003.

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36:474–494, 2010.

[6] Eric Bouwers and Martin Bravenboer. PHP-front: Static analysis for PHP. strategoxt.org/PHP/PhpFront.

[7] Thomas G. Brashear, Vishal Kashyap, Michael D. Musante, and Naveen Donthu. A profile of the internet shopper: Evidence from six countries. *The Journal of Marketing Theory and Practice*, 17(3):267–282, 2009.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[9] Naveen Donthu and Garcia Adriana. The internet shopper. *Journal of Advertising Research*, 39(3):52–58, 1999.

[10] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[11] Paul Gerrard. Risk-based e-business testing. Part 1: Risks and test strategy. www.gerrardconsulting.com, 2000.

[12] William Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA '09*, pages 285–296, 2009.

[13] William Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *SIGSOFT '08/FSE-16*, pages 181–191, 2008.

[14] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30:3–16, 2004.

[15] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07*, pages 73–83, 2007.

[16] Internet Retailer. Top 500 e-retailers take a bigger bite of the pie. www.internetretailer.com, June 2009.

[17] Internet World Stats. World internet users and population stats. www.internetworldstats.com, December 2009.

[18] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[19] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *TAIC-PART '09*, pages 95–104, 2009.

[20] Alessandro Marchetto and Paolo Tonella. Search-based testing of Ajax web applications. In *SSBSE '09*, pages 3–12, 2009.

[21] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[22] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27:1085–1110, 2001.

[23] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[24] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. *ISSRE '04*, pages 187–197, 2004.

[25] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[26] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *ASE '04*, pages 132–141, 2004.

[27] TechCrunch. Forrester forecast: Online retail sales will grow to $250 billion by 2014. techcrunch.com, March 2010.

[28] TIOBE Software. Tiobe programming community index. www.tiobe.com/tpci.htm, January 2011.

[29] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *ASE '98*, pages 285–288, 1998.

[30] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. *PLDI '07*, 42(6):32–41, 2007.

[31] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *ISSTA '08*, pages 249–260, 2008.

[32] Ruilian Zhao, Michael R. Lyu, and Yinghua Min. Automatic string test data generation for detecting domain errors. *Software Testing, Verification and Reliability*, 20:209–236, 2010.