An Efficient Regression Testing Approach for PHP Web Applications: A Controlled Experiment

Hyunsook Do
North Dakota State University
Fargo, ND
hyunsook.do@ndsu.edu

Md. Hossain
Microsoft
Fargo, ND
md.i.hossain@ndsu.edu

April 20, 2014

Abstract

Companies that provide web applications often encounter various security attacks and frequent feature-update demands from users, and when these needs arise, companies need to fix security problems or upgrade the application with new features. These fixes often involve small patches or revisions, but still, testers need to perform regression testing on their products to ensure that the changes have not introduced new faults. Performing regression testing on the entire product, however, can be very expensive, and it is not a viable solution for companies that need a short turnaround time to release patches. One solution is focusing only on the code areas that have been changed and performing regression testing on them. By doing this, companies can provide quick patches more dependably whenever they encounter security breaches. In this paper, the authors proposed a new regression testing approach that identifies the affected areas by code changes using impact analysis and generates new test cases for the impacted areas by changes using program slices. To facilitate the approach, the researchers implemented a PHP Analysis and Regression Testing Engine (PARTE) and performed a controlled experiment using five open source web applications with multiple versions. The results showed that this approach is effective in reducing the cost of regression testing for a frequently patched web application, and exposed ways in which that effectiveness can vary with application characteristics and versioning frequencies.

Keywords: Regression testing, impact analysis, test case generation, PHP web applications, empirical studies.

1 Introduction

Current web applications offer people easy ways to deploy web sites containing blogs, shopping carts, and many other useful tools. Companies that provide web applications often encounter various security attacks and frequent feature-update demands from users, and when these needs arise, companies need to fix security problems or upgrade the application with new features. These fixes often involve small patches or revisions, but still, developers and testers need to perform regression testing on their products to detect whether these changes have introduced new faults. Performing regression testing on the entire product, however, can require a lot of time and resources [41], and with these applications, a short turnaround time for releasing patches is critical because the applications have already been deployed and used in the field, thus users could suffer a great deal of inconvenience due to the absence of services on which they rely. For instance, one study [11] reports that small and medium sized organizations experienced an average loss of $70,000 per downtime hour. Further, organizations that provide those applications could also suffer from losing customers or from damaged reputations if they do not supply patch releases in time. One solution to this

problem is to focus only on the areas of code that have been changed as well as the portions of impacted code, and to regression test them. In this way, companies can deliver quick patches more dependably whenever security breaches are encountered.

To date, the majority of regression testing approaches have focused on utilizing existing test cases (e.g., [17, 37, 49]), but to test updated features or new functionalities thoroughly, software engineers need new test cases that can cover areas that existing test cases cannot. Creating new, executable test cases for affected areas of the code, which is known as a test suite augmentation problem [8], is one of the important regression testing problems, but only a few researchers have started working on this problem [8, 39, 42, 47]. While their work has made some progress in this area, they only provided guidance to create new tests [39], generated new test cases limited to numeric values [47], and considered only small desktop applications. However, web applications involve different challenges than desktop applications that are written in C or Java [44], and most web applications deal heavily with strings in addition to numeric values. In addition, while Taneja et al. [42] propose an efficient test generation technique by pruning unnecessary paths, the dynamic, symbolic execution based test generation approach used by other researchers can still be expensive and infeasible when we apply it to large applications.

To address these limitations, this research proposes a new test case generation approach that creates test cases by using program slices considering both string and numeric input values. In particular, the focus is on web applications written in PHP which is widely used to implement web applications [44]. To do so, the following steps are required: (1) Identifying the areas of the application that are impacted by the patches and (2) Generating new test cases for the impacted code areas by using program slices and by considering both string and numeric input values. Program slices have been used for many purposes, such as the debugging processes [46], test suite reduction [20], test selection [10], test case prioritization [26], or fault localization [28], but to the researchers' knowledge, no attempt, except for work by Samuel and Mall [38] (whose work uses UML diagrams rather than source code), has been made to generate test cases using slices. In this work, slices are utilized to generate test cases. While this study focuses on PHP web applications, the approach can be applied to other language domains, and the regression testing frame can easily be extended to support them.

To facilitate this approach, the researchers implemented a PHP Analysis and Regression Testing Engine (PARTE [29]). To assess this approach, a controlled experiment was designed and performed using five open source web applications that have multiple versions, various sizes, and different characteristics (e.g., versioning frequencies).

The results showed that this approach is effective in reducing the amount of time needed to apply regression testing for frequently patched web applications. Averaging for each application, the test path reduction rates achieved by the studied approach ranged from 46% to 87%, compared to the control technique that generated test cases without considering code change information, and the results exposed ways in which that effectiveness can vary with the application characteristics and their versioning frequencies. The results also suggested that the proposed approach can be highly efficient when version revision involves small modifications due to security fixes or minor bug fixes.

The next section describes this research's overall methodology and PARTE's technical details. Section 3 presents the experiment, including the design, threats to validity, and data and analysis. Section 4 discusses the experimental results and their practical implications. Section 5 describes related work relevant to web applications and regression testing, and Section 6 presents conclusions and future work.

2 Methodology

To support the proposed regression test generation approach for web applications, the PHP Analysis and Regression Testing Engine (PARTE) was developed. Figure 1 summarizes PARTE's three main activities (light green boxes) and how these activities are related to each other. Although this research implemented the regression test generation technique that is applied to PHP web applications, the methodology utilized here can easily be applied to other languages by extending the front end file conversion functionality. When generating test cases, this research considered the test paths and input values that make test paths executable. Another important component of test cases is the test oracles that verify test results, however, in this work, test oracles are not considered. The limitation of this work related to the test oracle issue is discussed in Section 4.4.

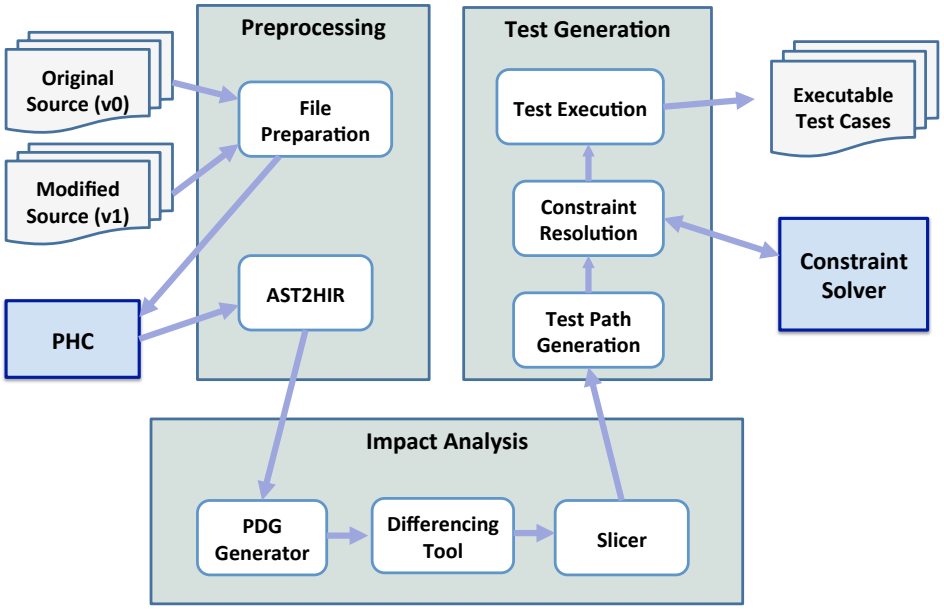

Figure 1: Overview of PARTE

Before describing each activity in detail, a brief overview of the approach is provided.

- **Preprocessing** (the upper-left box in Figure 1). In this step, PARTE converts the abstract syntax trees (ASTs) generated by a PHP compiler, PHC [33], to high level intermediate representation (HIR) that preserves variable names and converts this HIR back to ASTs.

- **Impact Analysis** (the lower box in Figure 1). Using preprocessed files, a Program Dependence Graph (PDG) generator builds PDGs for two consecutive versions of a PHP web application. Then, a differencing tool and a program slicer identify the areas affected by the code changes (program slices).
- **Test Case Generation** (the upper-right box in Figure 1). A test path generator creates test paths using program slices. A constraint collector/resolver gathers string and numeric input constraints, resolving them using a constraint solver, Choco [1]. Finally, a test execution engine takes these resolved input values, executes the application through Selenium [6], and records the test execution results. The last step is done manually because the current path generator does not provide web elements for Selenium.

2.1 Preprocessing

Before performing the impact analysis on two consecutive versions of PHP web applications, two preprocessing steps were taken to address the challenges associated with analyzing a high level, dynamic programming language. The following subsections describe each step in detail.

2.1.1 File Preparation

PARTE uses abstract syntax trees (ASTs) to generate PDGs. To create ASTs from PHP programs, this research used PHC (a PHP compiler), a publicly available, open source tool. PHC was used because it generates ASTs that contain detailed information about variable data types that is necessary for input data constraint gathering and resolution.

However, there are a couple problems with PHC. PHC cannot properly parse *include* and *require* functions that contain variables instead of constant strings. Further, PHC cannot properly evaluate the expressions if any concatenation or defined values are used in the *include* or *require* function. Using defined values and runtime concatenation to include a specified file at a location defined during the web application's installation is a common practice.

Because PHC cannot resolve non-trivial *include* and *require* function calls, a parsing tool was built to search for them through the web application source code and then to locate, define, and store them in a dictionary. When the parsing tool locates *include* or *require* function names, it resolves them using the dictionary that maintains defined values. If *include* or *require* functions are completely resolved, the tool inserts their code into the associated source code. This process is repeated until the tool visits every *include* and *require* function.

Another common feature of many dynamic web applications is to specify which files to include at runtime. One typical example of this feature is that a user selects a language preference when the web application is presented. For cases like this one, a constant value is manually assigned to the environmental variable in the application.

2.1.2 AST2HIR

After obtaining the preprocessed PHP files from the previous step, ASTs for these PHP files were generated using PHC. Then, AST was converted to a high level intermediate representation (HIR) to simplify PDG generation. PHC already provides a function that generates HIR code from an AST, but the HIR code generated by PHC does not

preserve consistent variable names across versions of the same application, so it is not suitable for these regression testing tasks. PHC's HIR is focused on code optimization for performance efficiency.

PHC uses a typical three address code generation approach and creates variable names by incrementing a numerical suffix on a fixed prefix. Thus, if a single statement in a program is added, deleted, or modified, every statement that comes after it in the internal representation could have variable names that do not correspond with the variable names in the original version's internal representation. The proposed regression testing framework requires consistent variable names to properly generate an impact set given two different versions of the PHP program. To resolve this issue, an AST2HIR tool was implemented to maintain consistent variable names while generating a three address code representation for the program. Instead of simply incrementing based on the order variables appeared in a source file, a hashing function was used on the values from the control dependency information combined with the expression values within an AST statement to determine variable names.

There are two main symbol types in the AST: statements and expressions. Statements are the symbols that define classes, interfaces, and methods as well as the control flow of the program. Expressions are the rest of the symbols, some of which include arithmetic operators, conditional expressions, assignments, method invocations, and logical operators. The AST that PHC generates is in XML format, and the AST2HIR converter parses the XML tags in a depth first manner. When a complex statement ("complex" being defined as an expression that is not in formatted, three address code) is encountered while parsing, it is split into multiple expressions such that all parts are not complex. These parts are given a unique variable name, and assignment expressions are generated to assign the value of the non-complex parts to their corresponding unique variables. Some statements are also transformed to work in the PDG generator. For instance, a "for" loop is simplified into a "while" loop. "Switch" statements are converted to equivalent code using a "while" loop, "if" statements, and "break" statements.

The AST2HIR tool then produces a PHP program that is functionally equivalent to the original program but in the high level internal representation. An AST for this HIR program is then created using PHC. This HIR version of the AST is used to generate the PDGs. Throughout the paper, this HIR version of the AST is simply referred to as AST.

2.2 Impact Analysis

To analyze the impact of software modifications on a web application, the authors implemented a static program slicer for web applications that was written in the PHP programming language. The benefit of using a program slicer was that it provides a complete way to reveal which areas of the application will be affected by software modifications [7].

Using the ASTs constructed during the preprocessing steps, impact analysis is performed to identify areas of the application that have been affected by code changes. For impact analysis, three steps are involved: PDG generation, program differencing, and program slicing.

To build PDGs from ASTs, Harrold et al.'s [23] approach is utilized. First, the PDG generator constructs control flow graphs. During this phase, the PDG generator builds control flow graphs for all methods and functions that have

been declared. To construct an inter-procedural control flow graph, all method or function calls within the control flow graphs are linked to their corresponding control flow graphs. Next, the PDG generator analyzes the data by performing a def-use analysis and gathers data dependency information for variables.

After constructing PDGs for two consecutive versions, it is needed to determine which files have been changed from the previous version as well as which statements within those files have been changed. To perform this task, a textual differencing approach similar to that of the standard Unix diff tool is used. With this approach, the differencing tool compares two versions of program dependence graphs, in XML format, that contain the program structure and statement content. The differencing tool analyzes each program and applies the longest common subsequence approach on the statements' PDG representation. After the tool determines which statements have been edited, added, or deleted, it provides this information to the program slicer.

Finally, the program slicer creates program slices by using the program's modification information. Program slicing is a technique that can be used to trace control and data dependencies with an application [45], and in this work, it is used to calculate the code change impact set. This program slicer performs forward slicing by finding all statements that are data dependent on the variables defined in the modified statements and then produces a program slice following these dependencies. It then performs backward slicing on the modified statements to determine the statements upon which each modified statement is data dependent. For each modified statement, the forward and backward slices are combined into a single slice.

2.3 Test Generation

For this phase, three steps are taken: (1) test path generation, (2) input constraint collection and resolution, and (3) test execution. The following subsections describe each step in detail.

Algorithm 1 Path Generation

```
1:  Inputs: slices[]   ▷ The slices array contains all the slices that are generated by analyzing the changes made to the modified version of the application.
2:  Outputs: paths   ▷ An array of all the linearly independent paths produced from the set of slices.
3:  Declare: edges   ▷ A set of edges traveled by the path generator
4:  currentPath   ▷ A placeholder for the path currently being generator
5:  procedure PATHGENERATION(slices)
6:      paths ← ∅
7:      edges ← ∅
8:      for n ⟵ 0, n < slices.size(), n++ do
9:          while slices[n].size()> 0 do
10:             currentNode ← slice[n].firstNode()
11:             currentPath ← ∅
12:             currentPath.add(currentNode)
13:             PDGTraverser(currentPath, paths, edges, slice[n], currentNode)
14:         end while
15:     end for
16:     ResolvePaths();
17:     return paths
18: end procedure
```

Algorithm 2 PDG Traverser

```
1:  procedure PDGTRAVERSER(currentPath, paths, edges, slice, currentNode)
2:      while currentPath[lastNode].occursBefore(slice.endNodes()) do
3:          if currentPath.containsNoDiffNodes() ∧ currentPath.cannotReachAdiffNode() then
4:              Return
5:          end if
6:          if slice[n].contains(currentNode) then
7:              slice[n].remove(currentNode)
8:          end if
9:          if currentPath[lastNode].getExits.Size() > 1 then
10:             for n ← currentPath[lastNode].getExits.Size()-1,0,n– do
11:                 if currentNode.exitEdge(n) ∉ edges then
12:                     edges.add(currentNode.exitEdge(n)
13:                     if n == 0 then
14:                         curentNode ⟵ currentNode.exit(n)
15:                         currentPath.add(currentNode);
16:                         continue(2);                          ▷ Go back to while loop
17:                     end if
18:                     PDGTraverser(currentPath.copy(), paths, edges, slice, currentNode.exit(n))
19:                 else
20:                     if currentNode.exitEdge(n) ∈ currentPath then
21:                         currentNode ⟵ findFirstNodeOutsideofLoop()
22:                     end if
23:                 end if
24:             end for
25:         else
26:             curentNode ⟵ currentNode.exit(0)
27:             currentPath.add(currentNode);
28:         end if
29:     end while
30:     if currentPath[lastNode].isNotASliceEndNode() then
31:         if currentPath[lastNode].canReachASliceEndNode() then
32:             currentPath.findSliceEndNode();
33:         else
34:             Return
35:         end if
36:     end if
37:     paths.add(curentPath)
38: end procedure
```

2.3.1 Path Generation

The path generator creates linearly independent test paths using the slices collected from the previous step. A *linearly independent path* includes at least one edge that has not been traversed previously (in a given set of paths under construction) [34].

Algorithm 1 shows how to generate test paths using slices. The algorithm is separated into two parts. The first part, Algorithm 1, iterates through and gathers nodes from the slice to use in the PDG traversing procedure. The second part, Algorithm 2, is the PDG traverser which follows a subpath through the PDG using the slice nodes as a guide. Algorithm 1 begins by iterating through every remaining node in each slice.

The PDG traverser (Algorithm 2) begins by analyzing a node in the PDG. If the node being processed occurs after every node in the array of remaining slice nodes (line 2), the PDG traverser checks to see if the currently processed node can reach an end node (a node where change propagation terminates) in the original slice (lines 30 and 31). If the node can, it adds the path to an end node for the current path (line 32). Otherwise, the path is discarded (line 33).

If the node being processed occurs before any of the end nodes in the slice, the node is checked to see if it can reach the changed node (line 3). If it cannot, the path is discarded. The PDG traverser then checks to see if the node is in the array of slice nodes. If it is, that node is removed from the array (lines 6 and 7). Every time the PDG traverser encounters a node with more than one edge that has not been traversed, it creates a new subpath that is a copy of itself for each of the edges and follows them (lines 9-18). Once path construction has been completed, the path is added to the path array (line 37).

After subpath construction is finished, the first node in the subpath is traced to an entry point for the program, and the last node is traced to an exit point for the program (line 16 of Algorithm 1). Algorithm 1 describes the basic construction of these paths while omitting, for brevity, all special cases that occur for cyclic graphs (loops and recursion).

2.3.2 Constraint Gathering and Resolution

Because the generated test paths only contain input parameters, not actual input values, the input values need to be assigned to the parameters to make test paths executable. The constraint collector collects constraints for the input values needed to execute a particular program's execution path. The program's inputs are the outputs from the PDG and path generators. The top level activities in the constraint collector are parsing the path and PDG XML files, generating constraints for each path, and writing the collected path's constraints to an XML file.

Constraint collection begins with path nodes that contain a conditional statement. The primary activities to collect constraints are determining the truth value of conditional statements and reducing the conditional expression in these statements. A path constraint corresponds to a path PDG node with an "if" statement. To determine the truth value of a constraint, the collector looks ahead one node in the path node list and examines the node type (which will be either *true* or *false*). After determining the constraint truth value, the constraint condition is then recursively reduced. This reduction involves parsing the expression string to determine the expression type (using regular expression matching), creating an expression object with this type, assigning any expression attributes from the parsed information (e.g., operator type) to the expression object, and generating appropriate expression objects for child expressions (if any). If there are child expressions, they are recursively reduced in the same way.

Each type of expression provides its own method for reducing child expressions in order to take advantage of information on the reduction context. For example, a compound boolean expression must have child expressions that are boolean. When generating child expressions, this information can be provided in addition to what is provided by regular expression matching. This is useful in verifying that generated expressions are of the correct type.

If regular expression matching determines that an expression is a variable (e.g., $num_of_files), the reduction process involves additional steps:

1. The collector backtracks in the path node list, starting at the variable node's index. It continues backtracking until a PDG node that provides a definition of the variable is encountered or until the beginning of the list is

reached. If a definition is found, the expression generated for the variable is of the complex type. This type is used for variables that can be defined in terms of other expressions. If, instead, no variable definition is found before reaching the start of the list, the generated expression is of the simple variable type.

2. Regular expression matching is performed on the Variable expression string to determine if it is indexed (e.g., $files[$id]). These correspond to PHP array variables. An expression is then generated for the array's index, and the generated expression for the variable is an indexed version of the type (i.e., simple or complex) determined via backtracking.

The recursive reduction process terminates on expressions of The simple type because simple type expressions do not contain child expressions that need to be reduced. After collecting a constraint, the collector records it for inclusion in the output. Once all path constraints have been generated, the collector writes the constraint information in XML format.

The tool uses an existing constraint solver, *Choco*, to determine the input values that satisfy the constraints for a given execution path (if such satisfaction is feasible). *Choco* is an open source software and consists of a set of libraries written in Java that provide many constraint solving features. It provides direct support for solving numeric and boolean constraints. It is also used to solve string constraints by mapping these constraints to equivalent constraints on integers. Once all input values have been created, they are stored in XML file format for later use by the test execution engine. These files contain information about program paths and a list of input variables for the paths. For each input variable, the variable type, name, and value are given. Resolving the inputs that use built-in PHP functions is not supported by PARTE, so those inputs require manual resolution. Further, the constraint resolution tool sets the time limit for input resolution, and it reports infeasible when it cannot find the value within the time limit. Also, if the conditions for a variable have a conflict, the tool reports the case as infeasible.

2.3.3 Test Execution

Having assigned all input values to the parameters in the test paths, a test execution engine that executes test cases over web applications using Selenium [6] was implemented. The Selenium WebDriver API (more specifically, the FirefoxDriver) was used to set the web page elements according to the variable constraints for the given execution path being tested. If test execution requires setup logic, such as authentication, this is also performed using the WebDriver API. Finally, the WebDriver API is used to execute the PHP file, and the results are passed to the test engine. As mentioned earlier, web elements are created manually because the current path generator does not provide them.

The WebDriver API was chosen because it performs all the activities that will be required: finding and setting web elements, performing any setup tasks required before the test case can be executed (such as authentication), and retrieving test execution results in HTML format.

To give a better understanding of how the approach works, an e-commerce Example is used. Assume that there are two PHP files, login.php and add_card.php, both of which are accessed on different web pages. login.php contains the login form with which users can register/authenticate, and add_card.php allows authenticated users to add their credit card information. Both login.php and add_card.php contain entry points for certain execution paths.

If add_card.php was modified between Version 1 and Version 2, the toolchain can be run on this version pair and can determine the constraints necessary to test execution paths with entry points beginning in that file. Using the Selenium WebDriver API, users can first access the login form contained on login.php to register/authenticate themselves. Afterwards, the same API is used to directly execute add_card.php after initializing the values of the web elements associated with the path constraints. It would not be necessary to start with login.php because the PHP $_SESSION variables used for authentication would still be valid when executing add_card.php, regardless of the navigational path to arrive at this page. The results of executing add_card.php are recorded as an HTML file.

2.4 An Example of Test Path and Input Generation

v0.php (original version)

```
1. $a = $_POST['inputA'];
2. $b = $_POST['inputB'];
3. if ($a < 12)
   {
4.  $b = 5;
5.  $a = a-1;
   }
   else
6.  $b = b+3;
7. if ($a > 7)
   {
8.  if ($b == 5)
9.    echo "a\n";
    else
10.   $b= 7;
   }
11. echo "$b\n";
12. echo "done processing\n";
```

v1.php (modified version)

```
1. $a = $_POST['inputA'];
2. $b = $_POST['inputB'];
3. if ($a < 12)
   {
4.  $b = 5;
5.  $a = a-3; //changed statement
   }
   else
6.  $b = b+3;
7. if ($a > 7)
   {
8.  if ($b == 5)
9.    echo "a\n";
    else
10.   $b= 7;
   }
11. echo "$b\n";
12. echo "done processing\n";
```

PDG for v0 and v1

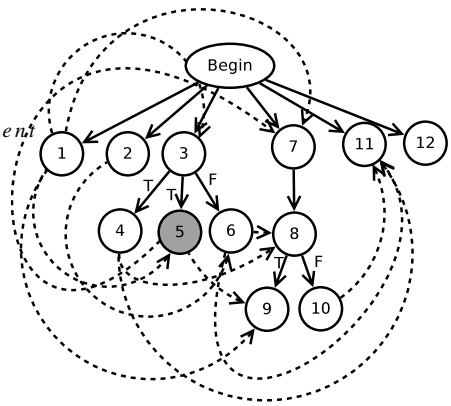

Figure 2: Path Generation Example Program

Test paths are generated using program slices by taking the following steps. Suppose there is a simple PHP program (v0.php) and its modified program (v1.php) as shown in Figure 2. The rightmost graph shows a PDG for v0.php and v1.php.

In the PDG, the solid lines represent control dependence edges, and the dashed lines represent data dependence edges. As the example shows, statement 5 has changed, so the slicing algorithm takes node 5 and variable a ($<5, a>$) as a slicing criterion. By performing forward and backward slicing, the slice with respect to $<5, a>$ includes nodes 1, 5, 7, and 9 (node 1 from backward slicing, and nodes 7 and 9 from forward slicing).

Having obtained this slice, the path generator creates subpaths starting with its first node (in this example, node 1). As the tool walks the path using control flow information, it adds nodes 2 and 3 to a subpath ($\{1, 2, 3\}$). Once the path generator reaches node 3, it has a choice to explore one of the control successors (4 and 6). When the tool analyzes the path containing edge $3 \rightarrow 4$, it sees that the next control successor is node 5, yielding a subpath of $\{1, 2, 3, 4, 5\}$. When the tool analyzes the path containing edge $3 \rightarrow 6$, it discovers that there is no path from node 6 to node 5. The tool then discards path $\{1, 2, 3, 6\}$.

Continuing with the remaining edges, the tool produces a subpath of $\{1, 2, 3, 4, 5, 7, 8\}$. Because node 8 is another branching node, the tool yields subpaths $\{1, 2, 3, 4, 5, 7, 8, 9\}$ and $\{1, 2, 3, 4, 5, 7, 8, 10\}$. Edges $8 \rightarrow 9$ and $8 \rightarrow 10$ are marked as covered by the path generator. The tool then recognizes that subpath $\{1, 2, 3, 4, 5, 7, 8, 10\}$ ends with a node that is outside the impact set. (It is impossible to go from node 10 to nodes $\{1, 5, 7, 9\}$.) The tool also recognizes that there is no path from node 7 to node 9 that goes through node 10. The tool then discards this path. This process is repeated until all nodes in a slice have been visited.

Having generated subpaths that contain all nodes in a slice, the tool walks the program dependence graph from the first node to the beginning of the program (In this case, node 1 is at the beginning.) and walks the last node in the path to the end of the program by adding nodes 11 and 12 to it. In this example, the tool generates one final path, $\{1, 2, 3, 4, 5, 7, 8, 9, 11, 12\}$. Note that, without applying this approach, four linearly independent paths need to be generated to test the modified program, v1.php.

The next step is to generate inputs for the created path. First, the constraint collector gathers constraint information by analyzing all the path's branching nodes. In the example for path $\{1, 2, 3, 4, 5, 7, 8, 9, 11, 12\}$, three constraints are collected: \$ POST['inputA'] < 12 , \$ POST['inputA']-3 > 7, and \$ POST['inputB'] $== 5$. Using a constraint solver, the user can choose input values 11 and 5 for \$ POST['inputA'] and \$ POST['inputB'], respectively. Once these values are resolved, the test execution engine can simply use them as inputs for the web application to walk the desired path.

3 Empirical Study

As stated in Section 1, the goal of this study is to assess the proposed regression testing approach in terms of the number of new test cases that are generated. In particular, the following research question is addressed:

RQ: Is the PARTE approach effective in reducing the number of new test cases that are necessary to regression test web applications?

To investigate this research question, an empirical study was performed. The following subsections present the objects of analysis, study setup, threats to validity, and data analysis.

3.1 Objects of Analysis

In this study, five open source web applications, written in PHP, were used as objects of analysis; the applications were obtained from SourceForge.

osCommerce [4] (open source Commerce) is a web based store-management and shopping cart application. *FAQ-Forge* [18] is a web application used to create FAQ (frequently asked questions) documents, manuals, and HOWTOs. Three versions were used for these two applications.

phpScheduleIt [5] is a web application that attempts to solve the problem of scheduling and managing resource utilization. It provides a permission-based calendar that allows users to self-register and to reserve resources and tools to manage those reservations. Some typical applications are a conference room, equipment, or work shift scheduling. Four versions of this application were used.

Mambo [2] is a content management system that can be used for everything from simple websites to complex corporate applications. It is used worldwide to power government portals; corporate intranets and extranets; e-commerce sites; and nonprofit outreach, school, church, and community sites. Seven versions of *Mambo* were used for this study.

Mantis [3] is a web-based bugtracking system. Mantis supports multiple DBMS, such as the MySQL, MS SQL, and PostgreSQL databases, and works on multiple platforms. Mantis provides various bugtracking functionalities, including changelog support, source control integration, and time tracking. Due to its complicated functionalities, Mantis is the largest one among the applications with which this research experimented. (The latest version is over 200KLOC.) For this object program, eight versions were used for this study.

They are real, non-trivial web applications that have been utilized by a large number of users. Table 1 lists, for each studied object, the associated "Version," "Lines of Code," and "No. of Files." The lines of code counted in this table include both PHP code and HTML markups.

Table 1: Objects of Analysis

Application	Version	Lines of Code	No. of Files	Application	Version	Lines of Code	No. of Files
FAQForge	1.3.0	1806	20	*phpScheduleIt*	1.0.0	35045	90
	1.3.1	1837	20		1.1.0	59753	143
	1.3.2	1671	18		1.2.0	63138	178
osCommerce	2.2MS1	53510	302		1.2.12	72396	192
	2.2MS2	68330	506	*Mantis*	1.1.6	139124	496
	2.2MS2-060817	78892	502		1.1.7	139196	496
Mambo	4.5.5	149868	703		1.1.8	139194	496
	4.5.6	150967	719		1.2.0	206150	748
	4.6.1	127309	771		1.2.1	206492	747
	4.6.2	129235	659		1.2.2	207123	746
	4.6.3	130716	654		1.2.3	209104	753
	4.6.4	133420	663		1.2.4	209345	753
	4.6.5	133475	663				

3.2 Variables and Measures

3.2.1 Independent Variables

This empirical study manipulated one independent variable, test generation technique. The research considered one control technique and one heuristic technique.

The control technique generates all possible linearly independent paths without using any particular regression test case generation heuristics. This technique serves as an experimental control. The heuristic technique generates regression test cases using the program slices explained in earlier sections.

3.2.2 Dependent Variable and Measures

The dependent variable is the number of test paths generated by the techniques.

3.3 Experiment Setup

This study was performed using a virtual machine on multiple hosts. Because several virtual machine hosts with different performance capabilities were used, the data associated with time was not collected. The operating system for the virtual machine was Ubuntu Linux version 10.10. The server ran Apache as its HTTP server and MySQL as its database backend. PHP version 5.2.13 was used.

All but one of the modules in the PARTE tools were written in Java, and the Oracle/Sun JRE and JDK version 6 were used as the development and execution platforms. The test execution engine was written in Java. PHC version 0.2.0.3 was used to parse the PHP files. Perl and Bash scripts were used to control the modules and to pass data.

3.4 Threats to Validity

In this section, the internal and external threats to the validity of this study are explained. The approaches used to limit these threats' effects are also described.

Internal Validity: The outcome of this study could be affected by the choice of program analysis. The authors applied static analysis on a dynamic programming language, PHP. To do so, they had to change many statements in the program during the file preparation phase, removing and fixing many dynamic environment variables. However, the process was carefully examined to minimize the adverse effect that might be introduced into the converted files.

External Validity. This study used open source web applications, so these programs are not representative of the applications used in practice. Therefore, the results cannot be generalized. However, the researchers tried to reduce this threat by using five non-trivial sized web applications with multiple versions that have been utilized by many users.

3.5 Data and Analysis

In this section, the results of the experiment and data analyses are presented. Further implications of the data and results for the experiment are discussed in Section 4.

Table 2: Results for Path Generation

Application	Version Pair	Number of Paths for Entire Application (path-entire)	Number of Paths Generated by PARTE (path-parte)	Inputs Required for path-parte	Reduction Rate (path-parte over path-entire)
FAQForge	1.3.0 & 1.3.1	73	5	25	93%
	1.3.1 & 1.3.2	73	19	210	74%
osCommerce	2.2MS1 & 2.2MS2	2403	1719	4392	28%
	2.2MS2 & 2.2MS2-060817	2409	58	813	98%
phpScheduleIt	1.0.0 & 1.1.0	481	338	2389	30%
	1.1.0 & 1.2.0	518	314	3877	39%
	1.2.0 & 1.2.12	529	154	2339	71%
Mambo	4.5.5 & 4.5.6	1357	65	490	95%
	4.5.6 & 4.6.1	1388	1388	4446	0%
	4.6.1 & 4.6.2	1416	236	2075	83%
	4.6.2 & 4.6.3	1409	92	1236	93%
	4.6.3 & 4.6.4	1444	114	1567	92%
	4.6.4 & 4.6.5	1444	20	324	99%
Mantis	1.1.6 & 1.1.7	3482	221	1524	94%
	1.1.7 & 1.1.8	3482	185	1238	95%
	1.1.8 & 1.2.0	4345	2802	20425	36%
	1.2.0 & 1.2.1	4373	166	1772	96%
	1.2.1 & 1.2.2	4389	106	1042	98%
	1.2.2 & 1.2.3	4403	103	890	98%
	1.2.3 & 1.2.4	4419	199	2643	95%

Table 2[1] summarizes the data gathered from the experiment running on *FAQForge*, *osCommerce*, *phpScheduleIt*, *Mambo*, and *Mantis*. The table lists, for each application, "Version Pair" (two versions of the applications analyzed), "Number of Paths for Entire Application (path-entire)" (the total number of linearly independent paths for the latter version), "Number of Paths Generated by PARTE (path-parte)" (the number of linearly independent paths generated using program slices for the latter version), "Inputs Required for path-parte" (the number of input values required for executable test paths), and "Reduction Rate" (test path reduction rate for the proposed approach (path-parte) over the control technique (path-entire)). The number of inputs presented in the table is the summation for the number of inputs required for each path, thus redundant inputs may exist.

When considering the entire application for test path generation, for the latest version of *FAQForge*, 73 test paths were generated; in the case of *osCommerce*, 2,409 test paths were generated; for *phpScheduleIt*, there were 529 test

[1]There were a few bugs in the implementation of the path generation algorithm in the authors' previous work; these bugs were fixed before collecting data for this study. Therefore, the data values presented in this paper do not match those published in the researchers' previous work [30].

paths; *Mambo* had 1444 test paths; and for *Mantis*, 4,419 test paths were generated. As expected, the number of paths is proportional to the size of each application. When the test paths were generated by applying the proposed approach (path-parte), on average, the test path reduction rates over path-entire for the five applications were 84%, 63%, 46%, 77%, and 87%, respectively (in the order the applications appeared in the table). The following subsections present the data analysis for each application.

3.5.1 Results for FAQForge

For *FAQForge*, the test path generator created 5 and 19 paths for the two pairs of versions, respectively, while the control technique produced 73 paths for both versions. For *FAQForge*, the size of the application is relatively small compared to other applications, and the changes between versions are also very small; therefore, the number of test paths required for the new version is relatively small.

Upon manual inspection of the source for the first pair (versions 1.3.0 and 1.3.1), only one of the files in version 1.3.1 had been changed from version 1.3.0. The modified file was a library file that contained functions included by the main index file. During path generation, none of the library files were analyzed directly. Instead, the path generator analyzed files that the user would execute directly. If *FAQForge* files were not executed directly, they were located in the "lib" directory. Changes made to the library file were propagated to the index file during the file preparation phase of preprocessing.

The second pair for *FAQForge* (versions 1.3.1 and 1.3.2) yielded 19 paths using the path generator. Upon manual inspection of the source code, unlike the first version pair, the second pair had many changes in the source files; 12 files in version 1.3.2 had changed from version 1.3.1. Among these 12 files, only three were analyzed directly. The rest of files were, again, library files (in the "lib" directory) invoked using the *include* and *require* functions.

As shown in Table 2, the proposed technique produced a relatively small number of test paths compared to the control technique. The experimental technique reduced the number of test paths by 93% and 74% for versions 1.3.1 and 1.3.2, respectively. The number of inputs required for path-parte for *FAQForge* was small compared to other applications; for each version pair, the number of inputs was 25 and 210, respectively.

3.5.2 Results for osCommerce

For *osCommerce*, the test path generator created 1,719 and 58 paths for the two pairs of versions, respectively. Manual inspection of the source for the first pair (versions 2.2MS1 and 2.2MS2) showed that 279 of the 506 files had changed. The modified files were in every module of the application, and the files with the largest differences were the library files included in the executable files. Again, during path generation, no library files (files in *osCommerce*'s "include" directory) were studied directly. Instead, the path generator only analyzed files that the user executed directly.

The second pair for *osCommerce* (versions 2.2MS2 and 2.2MS2-060817) yielded 58 paths using the path generator. Manual inspection of the source code showed that 105 of the 502 files had changed. The number of changed files was smaller than with the first version pair. Considering how many files were changed for the second version pair (105 of

502 files), the number of generated paths was relatively small (58) because there were numerous changes in statements that contained no variables, and therefore, there were no data dependencies.

As shown in Table 2, the proposed technique produced a relatively small number of test paths compared to the control technique. The researched technique reduced the number of test paths by 28% and 98% for versions 2.2MS2 and 2.2MS2-060817, respectively. The input numbers were proportional to the test path number, so similar to the test path generation results, the number of inputs required for the version pairs was quite different: 4,392 for the first version pair and 813 for the second version pair. However, when considering the ratio between input numbers and test path numbers, the first version required a relatively small number of inputs per path. This result can be attributed to many changes in a simple output statement that has no data dependencies. A common example in PHP would be to *echo* or *print* static HTML statements. If the static text changed, the statement was marked as a difference.

3.5.3 Results for phpScheduleIt

In the case of *phpScheduleIt*, overall, the test path reduction rates achieved by applying the proposed approach were low compared to those of other object programs. In particular, the first two version pairs produced 30% and 39% reduction rates, respectively. Manual inspection of the source code for the first two version pairs showed that they involved large functionality changes and bug fixes. For the first version pair, among 90 files, 71 files were changed between the pair. For the second version pair, among 143 files, 105 files were changed. Due to these extensive changes among files, many code components needed to be tested, and this was the major reason for the low test path reduction rates.

For the last version pair, the reduction rate was 71%, which was better than the previous version pairs, but it was slightly low compared to other applications. Manual inspection showed that, among 178 files, 139 files had changed. While the number of modified files was similar to the second version pair, the files that contained a large amount of code involved minor changes, so the affected code area was not large; as a consequence, the test path reduction rate was higher than it was for other version pairs. The number of inputs required for path-parte for each version pair was 2,389; 3,877; and 2,339, respectively.

3.5.4 Results for Mambo

For *Mambo*, the test path reduction rates for all version pairs but one were high, ranging from 0% to 99%. In particular, for the last version pair, the reduction rate achieved by applying the proposed approach was 99%. This fact indicated that less than 1% of test cases are actually required for the modified web application, a huge savings considering the application. (Note that the size for the last version of *Mambo* is over 133 KLOC.)

Manual inspection showed that a few minor changes were introduced between the versions (e.g., 4.6.4 and 4.6.5). Only 14 files were changed among the 663 files for the version pair. As a result, only 20 test paths were needed for the new version. Version pair 4.5.5 and 4.5.6 also produced a high reduction rate. There were 319 files that changed among the 703 files for this version pair, so a high reduction rate was unlikely. Further inspection of the source code

revealed that most changes were not in the code part but in the top comment section for the disclaimer. Thus, these changes did not affect the number of test paths required for version 4.5.6.

Unlike other version pairs, for 4.5.6 and 4.6.1, no benefits were gained by applying the proposed technique. By looking at the version number changes, it can be inferred that there were major changes between these two versions. In fact, version 4.5.6 was released on 22 Jan 2008, and the next version (4.6.1) was released after 10 months (04 Oct 2008), indicating that there was a major change between these two versions. Manual inspection of two versions showed that a large area of code had changed. Among the 719 files, 317 files were changed, and most of the changes were extensive in nature. Most of the functions changed as part of refactoring and the feature update. These changes required to generate test paths for the entire application. As a result, a large number of inputs (4,446) was also required for those test paths.

3.5.5 Results for Mantis

Similar to the *Mambo* results, the test path reduction rates for *Mantis* were very high with one exception (the version pair 1.1.8 and 1.2.0). The rates ranged from 36% to 98%, and except for the third version pair, the proposed approach achieved over 90% of the reduction rate. For the version pairs that produced over 90% of the reduction rate, there were minor functionality changes and bug fixes in the source code. There were no big changes in any module of the source code. Version pair 1.1.7 and 1.1.8 was inspected, producing a 95% reduction rate. There were only four minor bug fixes and one translational update. Version 1.1.8 was intended for wrapping up minor things before developing the next major release, 1.2.0.

For only one version pair (1.1.8 and 1.2.0), the reduction rate (36%) was low. Inspecting the release date and release notes for this version pair showed that it was a major release which included large functionality improvement, a large number of bug fixes, and architectural restructuring. Manual inspection of the source code also reaffirmed the extensive changes. Among the 496 files, 260 files changed. For these reasons, almost all components in the modified version needed to be tested.

4 Discussion

To further explore the results of this experiment, three issues were considered: (1) a summary of the test path generation results obtained in this experiment and the practical implications, (2) a discussion of test input constraints, and (3) a discussion of the security implications for this approach. Following the discussion, the limitations of this work are addressed.

4.1 Test Path Generation Results and Practical Implications

The results from the experiment strongly support the conclusion that the program slice based test case generation approach needs fewer test cases to examine the modified version of the program compared to the control technique.

To show these results visually, bar graphs are shown in Figure 3. The figure contains five subfigures that present results for each object program, and each subfigure contains bar graphs that show the total number of test paths produced with two techniques (the control (path-entire) and the proposed technique (path-parte)).

As the figure shows, for all version pairs of all object programs, the proposed approach generated a smaller number of test paths than the control technique, and for the majority of the cases, the difference between the control technique and the proposed approach was very large. In total, 12 of 20 version pairs produced over a 90% reduction rate, for *Mambo* and *Mantis*, all cases but two produced substantial reduction rates. However, for some cases, the difference between these approaches was not outstanding. For instance, with the first version pair in *osCommerce*, the first two version pairs in *phpScheduleIt*, the second version pair in *Mambo*, and the third version pair in *Mantis*, the savings produced by the proposed approach were smaller than average. The researchers speculated that the version release affected this outcome. When a software system goes through major version changes, more functionalities tend to be added, and code refactoring can take place; therefore, more code modifications are involved with the major revisions than for small bug fixes or feature updates. In fact, the version pairs that did not produce huge savings went through major changes, and large portions of code were refactored.

The version pairs that produced substantial savings, indeed, involved minor bug fixes and small feature upgrades. This situation is where the authors aim to apply their approach as was described in the Introduction section. Revisions caused by bug fixes or security problems tend to be unexpected and unplanned compared to major revisions, thus a short turnaround time for releasing these revisions is a dire issue. By providing regression testing approaches that can save time and effort, applications can be deployed as early as possible.

4.2 Test Input Constraints

Because the test paths require actual inputs to create executable test cases, reducing the number of test paths necessary with the modified program should produce further cost savings when collecting test input constraints and resolving constraints. By automating the constraint resolution process for the majority of inputs, a substantial amount of time and effort could be reduced when resolving constraints manually.

To provide some ideas about how much time can be saved with automatic constraint resolution, the time taken to assign actual values to the input variables obtained by the proposed constraint collector were investigated. Two cases from *osCommerce* that contain multiple numeric and string inputs were examined. One case had 16 inputs (2 numeric and 14 string), and the other case had 10 inputs (2 numeric and 8 string). To resolve all input values, it took 15 and 20 minutes, respectively. The tester had to go through the source files and the application database to figure out the proper values. When applying the implemented tool for the entire version of *osCommerce*, it took less than 5 minutes. The input values assigned by human testers could be more realistic, but when the number of inputs is prohibitively large to be resolved manually, automatic resolution would be the viable solution. Not all inputs can be resolved with the automatic resolution tool, but still, it would provide a good baseline that testers can utilize.

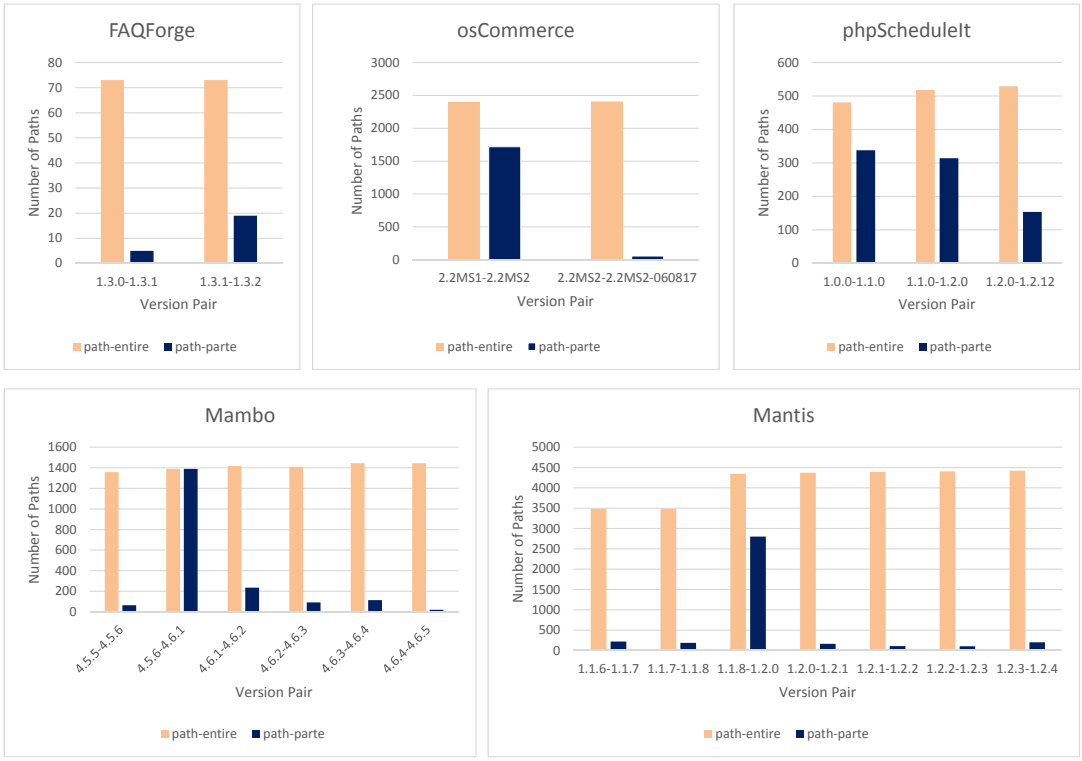

Figure 3: Experiment Results: The Total Number of Test Paths Generated by the Control and PARTE techniques

4.3 Security Implications

There are some security implications for this approach. Through the experiment, the researchers observed that the applications contained many security fixes and added security features; the test cases generated with this approach could help testers verify that a security patch or fix has been properly implemented.

In *FAQForge*, there was a security patch implemented between versions 1.3.1 and 1.3.2. The test generating tool discovered the difference and generated several test cases that traversed these changes. For version 1.3.1, changes were made to the file that contains code for a login page that allows administrators to perform security associated tasks. The change made to this file introduced an HTML meta tag with an http-equiv attribute. This allowed the page to load on all major browsers. Because the change happens in the header, it was important to test all code paths for the login page, such as page load, validate username and password, and count failed login attempts. The test paths generated in this experiment included all these code paths or blocks.

In *osCommerce*, a similar scenario occurred between versions 2.2MS1 and 2.2MS2. The developers added some

security features for website administration. For example, features supporting SSL validation and forcing cookie usage were implemented. Another interesting security related feature was added to version 2.2MS2. This feature allowed users to inject shell commands into the web server. The tool generated several test cases that covered this change. However, without proper inputs for these test cases, this security feature could not be tested correctly. Because the current constraint resolution tool does not generate malicious inputs, the researchers' previous tool [31] was used to generate malicious inputs. With these inputs, the tool was able to reveal this particular security vulnerability. In version 2.2MS2-060817, only a few minor security fixes were made. For example, the fix prevented a session ID from being passed in Tell-A-Friend E-Mails. This fix was a minor change to the statement that prepares the body of the email to be sent. Also, there was a security fix that corrected an SQL injection problem between versions 2.2MS2 and 2.2MS2-060817. The tool was able to generate test cases that exercised all these fixes.

In the case of *phpScheduleIt*, version 1.2.0 introduced a security feature where application administrators can grant permissions to other users. These permissions give users the access to make, modify, or delete an existing reservation. Also, administrators can remove permissions to prevent users from accessing a resource. The regression tests for this version should cover code that was added to enable user profile management. The results from this experiment show that the new code blocks associated with the new security features and fixes were included in the regression test paths that were generated.

In version 4.6.2 of *Mambo*, a security fix was introduced. One such fix is about the Captcha (completely automated public turing test to tell computers and humans apart) feature. If a session is not initiated, then it is regenerated, and a new session code is assigned to the Captcha code. This produces new challenge text and audio for verification. Given the type and location of this fix, it is important to cover all viable paths in this code. All these code blocks were included in the regression paths generated for this experiment.

In the case of *Mantis*, the version pair 1.1.6 and 1.1.7 contained one security bug fix. The fix for this bug required changes to the files that handle logging out of the current session and reloading the verification page. These changes impacted the code blocks that perform logout, initialize a new session, and validate user credentials. Another version pair, 1.1.8 and 1.2.0, contained one security fix. The bug was about the possibility of a cross site scripting attack through permalink_page.php. The fix was to check if the input URL is safe. Because this change happened at the root level, all paths in the permalink_page.php file should be included to test the fix, and the tool was able to generate those paths.

4.4 Limitations

Although the proposed approach and empirical results are promising, the method has some limitations that need to be addressed. One limitation involves test oracles. As mentioned in Section 2, this work focused on generating test cases, and this approach did not generate the oracles. Automating test oracle generation or verification of the results could be investigated in the future.

Another limitation involves test case execution. To execute test cases automatically, the test execution engine needs to pass the web elements to Selenium along with test paths and input values. However, the current path generator does not provide web elements, so they had to be provided manually. As a result, not all test cases that were generated through the implemented tool were able to be run. However, this issue has been identified, and the feature addition is being developed.

5 Related Work

To date, many regression testing techniques have been proposed, and most of them have focused on reusing the existing test cases for regression test selection (e.g., [37, 49]) and test case prioritization (e.g., [17, 43]). Existing test cases, however, are often insufficient to retest code or system behaviors that are affected by code changes.

To address this problem, recently, researchers started working on test suite augmentation techniques which create new test cases for areas that have been affected by changes [8, 39, 48, 47, 42, 12]. Apiwattanapong et al. [8] and Santelices et al. [39] presented an approach to identify areas affected by code changes using program dependence analysis and symbolic execution, and provided test requirements for changed software. Xu et al. [48, 47] presented an approach to generate test cases by utilizing the existing test cases and adapting a concolic test case generation technique. Taneja et al. [42] proposed an efficient test generation technique that uses dynamic symbolic execution, eXpress, by pruning irrelevant paths. These approaches focused on desktop applications, such as C or Java, but the proposed approach applied regression testing to web applications, creating different challenges (e.g., dynamic programming languages deployed using multi-tier architecture).

Creating executable test cases automatically has been one of the challenging tasks for the software testing community [13], and recent research in this area has proposed a promising approach which involves concolic testing (a combination of concrete and symbolic execution) [40]. Since then, other researchers have extended the concolic technique to handle PHP code [44, 9]. Wassermann et al. [44] and Artzi et al. [9] have also utilized a concolic approach to generate test cases for PHP web applications. Other test case generation techniques for web applications use crawlers and spiders to identify and test web application interfaces. Ricca and Tonella [35] use a crawler to discover the link structure of web applications and to produce test case specifications from this structure. Deng et al. [14] use static analysis to extract information related to URLs and their parameters for web applications, and to generate executable test cases using the collected information. Halfond et al. [22] present an approach that uses symbolic execution to identify precise interfaces for web applications.

Other researchers have investigated test case problems considering models with the system specifications rather than source code [12, 27, 19, 32, 38]. Chen et al. [12] present a model-based approach that generates a regression test suite using dependence analysis of the modified elements in the model. Similarly, Fourneret et al. [19] generate necessary test cases for the modified portion of the model (UML) using requirement information and impact analysis. Samuel and Mall [38] generate test cases from UML activity diagrams using a dynamic slicing technique. Some

work on model-based regression testing with existing test cases has been done. Korel et al. [27] use dependence analysis of Extended Finite State Machine (EFSM) models to reduce the size of regression test suites. Naslavsky et al. [32] present a test selection technique that chooses test cases for the modified models by establishing traceability relationships among entities in the models and existing test cases.

While these existing techniques and approaches have motivated this study, this work is different from previous research in the following aspect. This work focuses on evolving web applications that require frequent patches and regression testing. Therefore, unlike other approaches, this study only focuses on the areas affected by code changes and generates test cases using program slicing. Some work on regression testing for web applications [15, 16] has been done, but the focus for those studies is different than this research. Dobolyi and Weimer [15] present an approach that automatically compares the output from similar web applications to reduce the regression testing effort. Elbaum et al. [16] present a web application testing technique that utilizes user session data considering regression testing contexts.

Another research area that is relevant to this work is program slicing, a heavily researched area of software engineering. Weiser [45] formally defined program slicing as a way to determine which areas of a program are affected by what variables in a specific line of source code. Agrawal et al. [7] expanded this work by determining the statements which can be affected by a variable occurrence for a specific input value. Horwitz et al. [24, 25] also contributed by developing a way to generate slices that traversed multiple procedures using system dependence graphs. Gupta et al. [21, 20] applied program slicing techniques to the area of selective regression testing. Ricca and Tonella [36] presented an approach to construct program dependency graphs for web applications and addressed problems with handling events/hyperlinks and the dynamic nature of web applications.

6 Conclusions and Future Work

This paper presented a new regression test case generation approach that creates test cases using program slices for web applications written in PHP. To evaluate this approach, a controlled experiment was conducted using five non-trivial and open source web applications with multiple versions, and the results showed that this approach can be effective in reducing the number of test paths necessary for the modified application by focusing on the areas affected by the modified code.

While this approach can reduce the amount of time needed to apply regression testing for patched web application software, it can also reduce the time and effort as well as improve the testing effectiveness when major releases are tested because new test cases and other associated artifacts are accumulated over time.

The results of this research suggested several avenues for future work. First, the approach was evaluated utilizing five widely used, open source web applications. Because this approach provided promising results, the next natural step is to perform additional studies that apply this approach to industrial contexts (e.g., using industrial sized applications or considering constraints imposed by an industry's regression testing practice) to see whether the approach can address

an industry's practical problems.

Second, this work did not consider the use of the existing test cases. However, by utilizing the existing test cases when testing the modified program, additional savings can be achieved. Thus, the authors plan to investigate test case selection approaches that choose test cases that exercise the modified code areas to help reduce the cost of generating new tests.

Third, as discussed earlier, resolving constraints could require a lot of time and effort. Thus, the researchers plan to develop a technique that identifies reusable constraint values for regression test cases from the previous version. This action will accommodate further savings for regression testing web applications that require frequent patches and short regression testing cycles.

Acknowledgments

Mike Delaney, Kailash Joshi, Justin Anderson, Joshua Tan, Aaron Marback, and Nathan Ehresmann helped construct parts of the tool infrastructure used for the experimentation. This work was supported, in part, by NSF Awards CNS-0855106 and CCF-1050343, and NSF CAREER Award CCF-1149389 to North Dakota State University.

References

[1] Choco solver website. http://www.emn.fr/z-info/choco-solver/.

[2] Mambo web page. http://mambo-foundation.org/.

[3] Mantis web page. http://www.mantisbt.org/.

[4] osCommerce web page. http://www.oscommerce.com/.

[5] phpScheduleIt web page. http://phpscheduleit.org/.

[6] Selenium web page. http://seleniumhq.org/projects/.

[7] H. Agrawal and R. Horgan. Dynamic program slicing. In *SIGPLAN*, June 1990.

[8] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. MATRIX: Maintenance-oriented testing requirements identifier and examiner. In *TAIC PART*, 2006.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the International Conference on Software Engineering*, May 2010.

[10] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symp. on Prin. of Programming Languages*, January 1993.

[11] R. Boggs, J. Bozman, and R. Perry. Reducing downtime and business loss: Adressing business risk with effective technology. Technical report, IDC, August 2009.

[12] Y. Chen, R. Probert, and H. Ural. Model-based regression test suite generation using dependence analysis. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, July 2007.

[13] R. DeMillo and J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[14] Y. Deng, P. Frankl, and J. Wang. Testing web database applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[15] K. Dobolyi and W. Weimer. Harnessing web-based application similarities to aid in regression testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2009.

[16] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 49–59, 2003.

[17] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[18] FaqForge. Faqforge homepage. http://sourceforge.net/projects/faqforge/.

[19] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon. Selective test generation method for evolving critical systems. In *1st Int. Workshop on Regression Testing - co-located with ICST'2011*, Berlin, Germany, March 2011. IEEE Computer Society Press.

[20] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6(2):270–285, June 1996.

[21] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308, November 1992.

[22] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the International Conference on Software Testing and Analysis*, July 2009.

[23] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. *SIGSOFT Softw. Eng. Notes*, 18, July 1993.

[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM.

[25] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Sys.*, 12(1):26–60, 1990.

[26] D. Jeffrey and N. Gupta. Experiments with test case prioritization using relevant slice. *Journal of Systems and Software*, 81(2):196–221, 2008.

[27] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 214–, Washington, DC, USA, 2002. IEEE Computer Society.

[28] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, March 2002.

[29] A. Marback and H. Do. A Regression Testing Engine for PHP Web Applications: PARTE (fast abstract). In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 404–405, November 2010.

[30] A. Marback, H. Do, and N. Ehresmann. An effective regression testing approach for PHP web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 221–230, April 2012.

[31] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. Security test generation using threat trees. In *Proceedings of the Automation of Software Test*, May 2009.

[32] L. Naslavsky, H. Ziv, and D. Richardson. $MbSRT^2$: Model-based selective regression testing with traceability. In *International Conference on Software Testing, Verification and Validation*, April 2010.

[33] phc. phc - the open source php compiler. http://www.phpcompiler.org/.

[34] R. S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[35] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, 2001.

[36] F. Ricca and P. Tonella. Construction of the system dependence graph for web application slicing. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 123–132, 2002.

[37] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2), April 1997.

[38] P. Samuel and R. Mall. Slicing-based test case generation from uml activity diagrams. *ACM SIGSOFT Software Engineering Notes*, 34(6), 2009.

[39] R. Santelices and M. J. Harrold. Applying agressive propagation-based strategies for testing changes. In *International Conference on Software Testing, Verification and Validation*, April 2011.

[40] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 263–272, September 2005.

[41] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[42] K. Taneja, T. Xie, N. Tillmann, and J. Halleux. eXpress: Guided path exploration for efficient regression test generation. In *Proceedings of the International Conference on Software Testing and Analysis*, July 2011.

[43] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 1–12, July 2006.

[44] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the International Conference on Software Testing and Analysis*, July 2008.

[45] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[46] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[47] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2011.

[48] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2010.

[49] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 140–150, July 2007.