An Effective Regression Testing Approach for PHP Web Applications

Aaron Marback, Hyunsook Do, and Nathan Ehresmann
Department of Computer Science
North Dakota State University
Fargo, ND
$\{aaron.marback, hyunsook.do, nathan.ehresmann\}$@*ndsu.edu*

***Abstract*—Web applications change and are upgraded frequently due to security attacks, feature updates, or user preference changes. 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. Applying regression testing to the entire product, however, can be very expensive, and often, companies cannot afford to do this because, typically, the turnaround time to release patches is expected to be short. One solution is focusing only on the areas of code that have been changed and performing regression testing on them. In this way, companies can provide quick patches more dependably whenever they encounter security breaches. In this paper, we propose a new regression testing approach that is applied to frequently patched web applications, considering security problems, and in particular, focusing on PHP programs. Our approach identifies the affected areas by code changes using impact analysis and generates new test cases for the impacted areas by changes using program slices considering both numeric and string input values. To facilitate our approach, we implemented a PHP Analysis and Regression Testing Engine (PARTE) and performed a controlled experiment using open source web applications. The results show that our approach is effective in reducing the cost of regression testing for frequently patched web applications.**

***Keywords*-Regression testing, impact analysis, test case generation, PHP web applications**

I. Introduction

Web applications change and are upgraded frequently due to security attacks, feature updates, or user preference changes. 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. Applying regression testing to the entire product, however, can require a lot of time and resources [1], and for these applications, a short turnaround time in 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 the services on which they rely. For instance, one study [2] 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 and regression test them. In this way, companies can deliver quick patches more dependably whenever they encounter security breaches.

To date, the majority of regression testing approaches have focused on utilizing existing test cases (e.g., [3], [4], [5]), but to test updated features or new functionalities throughly, we 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 [6], is one of the important regression testing problems, but only a few researchers have started working on this problem [6], [7], [8], [9]. While their work has made some progress in this area, they have only provided guidance for creating new tests [7], generated new test cases limited to numeric values [9], and considered only small desktop applications. However, web applications involve different challenges than desktop applications written in C or Java [10], and the majority of web applications heavily deal with strings in addition to numeric values. In addition, while Taneja et al. [8] 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 size applications.

To address these limitations, we propose a new test case generation approach that creates executable test cases using program slices considering both string and numeric input values. In particular, we focus on web applications written in PHP which is widely used to implement web applications [10]. To do so, the following steps are required: (1) Identifying the areas of the application impacted by the patches and (2) Generating new test cases for the impacted areas of code by using program slices and considering both string and numeric input values. Program slices have been used for many purposes, such as helping with debugging processes [11], test suite reduction [12], test selection [13], test case prioritization [14], or fault localization [15], but to our knowledge, no attempt, except for work by Samuel

and Mall [16] (whose work uses UML diagrams rather than source code), has been made to generate test cases using slices. In this work, we will utilize slices when we generate test cases.

To facilitate our approach, we implemented a PHP Analysis and Regression Testing Engine (PARTE [17]). To assess our approach, we designed and performed a controlled experiment using open source web applications. Our results showed that our approach is effective in reducing the cost of regression testing for frequently patched web applications.

In the next section of this paper, we describe our overall methodology and the technical details about PARTE. Section III presents our experiment design, results, and analysis. Section V discusses our results. Section VI describes related work relevant to web applications and regression testing, and Section VII presents conclusions and future work.

II. Methodology

To facilitate our regression test generation approach for web applications, we have developed PARTE (PHP Analysis and Regression Testing Engine). Figure 1 summarizes PARTE's three main activities (light gray boxes) and how these activities are related to each other.

While we describe each activity in detail in the following subsections, we provide an overview of our approach here.

- **Preprocessing** (the upper-left box in Figure 1). To perform impact analysis, PARTE constructs program dependence graphs (PDGs) using abstract syntax trees (ASTs) generated by a PHP compiler, PHC [18]. To use PHC, two preprocessing steps are required due to the limitations with PHC (We explain these limitations in detail in the following subsections.): (1) A file preprocessor reads PHP programs and replaces the *include* and *require* function definitions in the individual PHP program with their actual code. (2) AST2HIR converts ASTs generated by PHC to high-level intermediate representation (HIR) that preserves variable names and converts this HIR back to ASTs which are used as input for a PDG generator.
- **Impact Analysis** (the lower box in Figure 1). Based on preprocessed files, a PDG generator builds PDGs for the two consecutive versions of PHP web application. Then, a program differencing tool identifies the areas of the current version's code that have been changed from the previous version. A program slicer calculates the affected areas of the code (program slices) using code change information.
- **Test Case Generation** (the upper-right box in Figure 1). Using slices obtained from the impact analysis, a test path generator creates test paths. In this process, a constraint collector gathers input constraints for both string and numeric values, and resolves them using two constraint solvers, Hampi [19] and Choco [20] (the dark-gray box in the left shown in Figure 1). If constraints cannot be resolved using these constraint solvers (e.g., interactive user inputs), a manual process is applied. Finally, a test execution engine takes these resolved input values and uses them as input for the application. The test execution engine records the results and steps taken during these tests.

A. Preprocessing

As we briefly mentioned, before we performed the impact analysis on the two consecutive versions of PHP web applications, we performed two preprocessing steps to address the challenges associated with analyzing a high-level, dynamic programming language. The following subsections describe each step in detail.

1) File Preparation: PARTE uses ASTs to generate PDGs; PHC, a publicly available, open source tool, was used to create ASTs from PHP programs. PHC creates abstract syntax trees (ASTs) that are formatted using an XML document object model (DOM) Tree. We used PHC because it generates the ASTs that contain the detailed information about variable data types, which is used for input data constraint gathering and resolution.

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

Because PHC does not resolve non-trivial *include* and *require* function calls, we built a tool that searches through the web application source code and then locates, defines, and stores them in a dictionary. When the tool encounters *include* or *require* function names, it resolves them using their 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 language preference when the web application is presented to the user. For cases like this, we manually assign a constant value to the environment variable in the application.

2) AST2HIR: Once we obtained the preprocessed PHP files from the previous step, we generated ASTs for these PHP files using PHC. The AST to HIR converter program that we implemented generates high-level intermediate representation (HIR) of a PHP program given an AST.

PHC already has the capability to generate HIR code from an AST, and it uses the HIR code to generate a compiled version of a PHP program. However, the HIR code generated

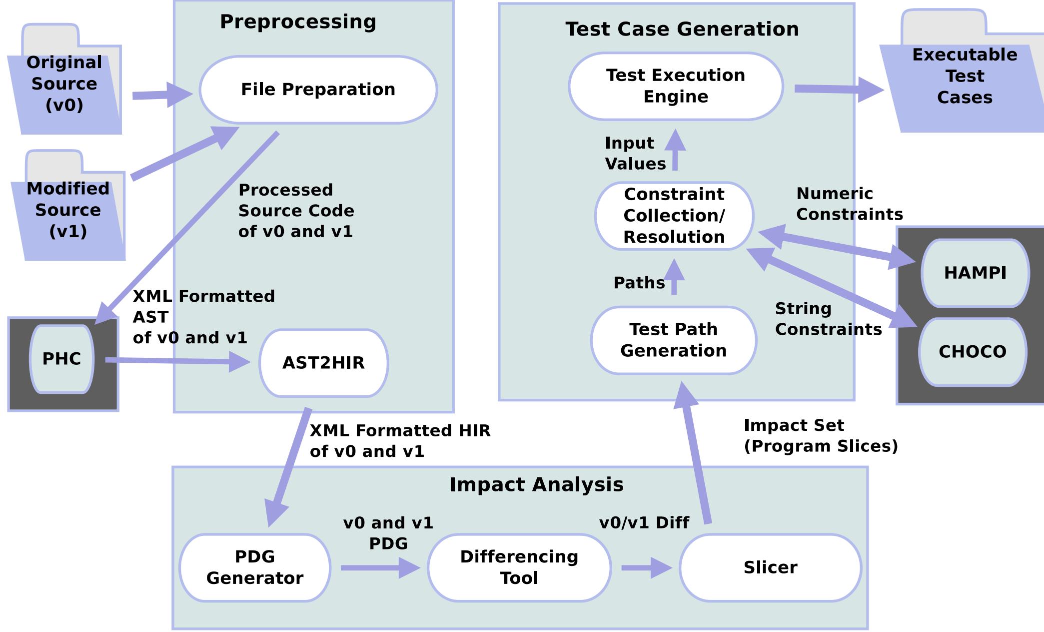

Figure 1: Overview of PARTE

by PHC does not preserve consistent variable names across versions of the same application, so it is not suitable for our 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. This means that, if a single statement in a program is added, deleted, or changed, every statement that comes after it in the internal representation may have variable names that do not correspond with the variable names in the original program version's internal representation. Our regression testing framework requires consistent variable names to properly generate an impact set given two different versions of the PHP program.

To address this problem, we built an AST2HIR tool that maintains consistency between variable names while generating a three-address code representation of the program. Instead of simply incrementing based on the appearance order of variables in a source file, we used a hashing function on the values from the control dependency information combined with the expression values within a statement from the AST to determine variable names.

When parsing the AST of the PHP program, there are two main types of symbols: statements and expressions. Statements are the symbols that define classes, interfaces, and methods as well as the control flow of the program. Expressions comprise 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. There is a difference between the parsing method of statements and expressions. Statements are parsed before their child nodes are parsed while expressions are only parsed after their child nodes are parsed.

When a complex statement (complex being defined as an expression that is not in formatted three-address code) is encountered during parsing, it is split into multiple expressions such that all parts are not complex. These parts are assigned 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 into 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 this paper, we refer to this HIR version of the AST as simply AST.

B. *Impact Analysis*

Having constructed ASTs through the preprocessing step, we perform the impact analysis that identifies the areas of the application that have been affected by code changes. For impact analysis, we use control and data flow information from two consecutive versions of the program to gather differences and to perform program slicing. The following subsections provide further details.

1) PDG Generation: To build PDGs from ASTs, we followed the same approach used by Harrold et al. [21]. 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. The PDG generator uses a bit-vector to track the definitions and uses of variables throughout the application.

2) Program Differencing: Next, we need to determine which files have been changed and which statements within these files have been changed. To do so, we use a textual differencing approach similar to that of the standard Unix diff tool. In our approach, the differencing tool compares two program dependence graphs, in XML format, that contain the program structure and statement content. To do this task, the tool analyzes each program and applies a longest common subsequence approach on the PDG representation of the statements. Once the tool determines which statements have been edited, added, or deleted, it provides this data to the program slicer.

3) Program Slicing: The final step is to gather program slices for the modification made on a program. Program sling is a technique than can be used to trace control and data dependencies with an application [22] and, in this work, it is used for calculating the code change impact set. Our program slicer performs forward slicing by finding all statements that are data dependent on the variables defined in the modified statements and produces a program slice following these dependencies. It then performs backwards slicing on the modified statements to determine upon which statements each modified statement is data dependant. For each modified statement, the forward and backward slices are combined into a single slice.

C. Test Generation

To generate executable test cases, three steps are required: (1) test path generation, (2) input constraint collection and resolution, and (3) test execution and test result collection. The following subsections describe each step in detail.

1) Path Generation: To generate test paths using the slices the tool has collected from the previous step, a tool was constructed to generate linearly independent test paths. A *linearly independent path* is a path that includes at least one edge that has not been traversed previously (in a given set of paths under construction) [23].

Algorithm 1 shows a path generation algorithm that generates linearly independent paths using slices. This algorithm is separated into two parts. The first part iterates through and gathers the nodes from the slice to use in the PDG traversing procedure. The second part is the PDG traverser, which follows a subpath through the PDG using the nodes in the slice 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 to the current path (line 32). Otherwise, the path is discarded (line 33).

If the node that is 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 a 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 these edges and follows them (lines 9-18). Once construction of the path has been completed, the path is added to the path array (line 37).

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

2) Constraint Gathering and Resolution: Because the generated test paths contain only input parameters, not actual input values, the input values need to be assigned to the parameters to make test paths executable. To do this, the tool gathers the constraints for these input values. During this process, the constraint gatherer analyzes each

Algorithm 1 Path Generation

1: **Inputs:** ***slices[]*** ▷ *The slices array contains all of the slices that are generated by the analyzing changes made to the modified version of the application.*
2: **Outputs:** ***paths*** ▷ *An array of all of 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: $paths \leftarrow \emptyset$
7: $edges \leftarrow \emptyset$
8: **for** $n \longleftarrow 0$, $n < slices$.size(), n++ **do**
9: **while** $slices[n]$.size()> 0 **do**
10: $currentNode \leftarrow slice[n]$.firstNode()
11: $currentPath \leftarrow \emptyset$
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 ⟵ findFirstNodeOutside-ofLoop()
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
```

place in the paths where a conditional expression occurs. Then, the tool determines which value (*true* or *false*) the conditional expression must have for the desired path to be executed. After the tool has found each of these conditional expressions, it proceeds backward, starting from the last conditional expression encounter, through the path substituting each variable used in the conditional expressions until it is only left with expression on input variables, constants, or combinations of the two. Next, the tool determines which expressions require numeric values and which ones require string values. It then breaks these expressions apart if they contain any logical connectors (*and* and *or*).

The tool uses two existing constraint solvers, *Choco* and *Hampi*, to help solve these expressions and to determine input values. *Choco* is a numeric constraint solver and consists of a set of libraries written in Java that provides many constraint solving features. *Hampi* is a string constraint solver that uses its a predefined constraint syntax. For values that take too long for a constraint solver to resolve, manual constraint resolution has been applied. Once all input values have been created, they are stored in XML file format because all tools in PARTE manipulate input and output files in XML format. This XML file contains information about which path and argument each value is associated.

3) Test Execution: Having assigned all input values to the parameters in the test paths, we implemented a test execution engine based on our existing tool that creates automated tests from Threat Models [24] Threat Models are a way for software designers to define and describe possible threats to the security of a software system [25]. The testing engine parses the input values file and enters data into the web application using a framework, written in the Perl programming language that uses the LWP and WWW:Mechanize libraries to simulate user interaction. Once we have executed each test, we record the file output (typically an HTML file) and also create an executable Perl script associated with the execution of that task.

D. An Example of Test Path and Input Generation

We illustrate how we generate test paths using program slices. Suppose we have a simple PHP program (v0.php) and its modified program (v1.php) as shown in Figure 2. The rightmost graph shows a PDG for 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 been changed, so our 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 from the first node in it (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. It then discards the 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 the 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 of the impact set (It is impossible to go from node 10 back to nodes {*1, 5, 7, 9*}.). The tool

v0.php (original version)

```
1. $a = $_POST['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4.   $b = 6;
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['input'];
2. $b = $_POST['input2'];
3. if ($a < 12)
   {
4.   $b = 6;
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 v1

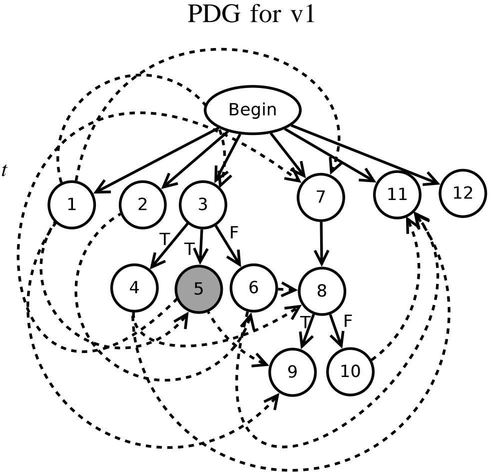

Figure 2: Path Generation Example Program

also recognizes that there is no path from 7 to 9 that goes through node 10. It 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 our approach, we need to generate four linearly independent paths 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 branching nodes of the path. In our example for path {*1, 2, 3, 4, 5, 7, 8, 9, 11, 12*}, three constraints are collected: $ POST['input'] < 12 , *$ POST['input']-3 > 7*, and *$ POST['input2']==5*. Using a constraint solver, we can choose input values 11 and 5 for $ POST['input'] and $ POST['input2'], respectively. Once these values have been resolved, the test execution engine can simply use them as inputs for the web application to walk the desired path.

III. Empirical Study

As stated in Section I, we wished to assess our regression testing approach in terms of the number of new test cases generated by our approach.

To investigate our research question, we performed an empirical study. The following subsections present our objects of analysis, study setup, and data and analysis.

A. Objects of Analysis

For this study, we used two open source web applications written in PHP as objects of analysis; the applications were obtained from SourceForge. We used three versions for both *osCommerce* [26] and *FAQForge* [27]. *osCommerce* (open source Commerce) is a web based store-management and shopping cart application. *FAQForge* is a web application used to create FAQ (frequently asked questions) documents, manuals, and HOWTOs. They are real, non-trivial web applications that have been used by a large number of users. Table I lists, for each of our objects, its associated "Version", "Lines of Code", and "No. of Files".

Table I: Objects of Analysis

Application	Version	Lines of Code	No. of Files
FAQForge	1.3.0	1806	20
	1.3.1	1837	20
	1.3.2	1671	18
osCommerce	2.2MS1	53510	302
	2.2MS2	68330	506
	2.2MS2-060817	78892	502

B. Variables and Measures

1) Independent Variables: Our empirical study manipulated one independent variable, test generation technique. We 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.

2) Dependent Variable and Measures: Our dependent variable is the number of tests generated by the techniques.

C. Experiment Setup

We performed our study using a virtual machine on multiple hosts. Because we used several virtual machine hosts and their performance capabilities are different, we did not collect data associated with the time. 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. We used PHP version 5.2.13.

All but one of the modules in the tools of PARTE 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 Perl. 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.

To generate the test paths for the version being tested, we generated all linearly independent paths for the entire program (for the control technique), and then generated all linearly independent paths for the areas impacted by changes made to the version under test. Next, we collected input constraints and resolved them using two existing constraint solvers, *Hampi* and *Choco*. If the solvers could not resolve the constraints, we resolved them manually. We used an XML format to store input values, and the test execution engine executed test cases by accessing the input file.

D. Threats to Validity

In this section, we describe the internal and external threats to the validity of our study. We also describe the approaches we used to limit the effects of these threats.

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

External Validity. We used open source web applications for our study, so these programs are not representative of the applications used in practice. However, we tried to minimize this threat by using non-trivial size web applications that have been utilized by many users.

IV. Data and Analysis

Table II shows the total number of linearly independent paths for each application. The first column shows the application name and the version analyzed. The second column shows the number of paths generated. The base versions of these applications are not listed in the table because regression testing starts from the second release of the application. For the latest version of *FAQForge*, 1418 test paths were generated, and in the case of *osCommerce*, 35775 test paths were generated. *osCommerce* generated a large number of paths compared to *FAQForge*, but this finding is not surprising considering the lines of code associated with each application.

Table II: Total Number of Linearly Independent Paths

Application	Number of Paths
FAQForge 1.3.1	612
FAQForge 1.3.2	1418
osCommerce 2.2 MS2	34924
osCommerce 2.2 MS2 - 060817	35775

Table III summarizes the data gathered from running PARTE on *FAQForge* and *osCommerce*. The table lists, for each of the applications, "Version Pair" (two versions of the applications analyzed), "Paths" (the number of linearly independent paths generated using program slices for the latter version), and "Inputs" (the number of input values required for executable test paths). It should be noted that many generated inputs were reused for multiple test paths. Therefore, for most cases, there are more test paths than inputs. Table IV shows test path reduction rates for our approach over the control technique.

Table III: Linearly Independent Paths Generated Using Program Slices

Application	Version Pair	Paths	Inputs
FAQForge	1.3.0 & 1.3.1	20	27
	1.3.1 & 1.3.2	335	219
osCommerce	2.2MS1 & 2.2MS2	1639	526
	2.2MS2 & 2.2MS2-060817	2464	781

Table IV: Test Path Reduction Rates over the Control Technique

Application	Version Pair	Reduction Rate
FAQForge	1.3.0 & 1.3.1	96%
	1.3.1 & 1.3.2	76%
osCommerce	2.2MS1 & 2.2MS2	95%
	2.2MS2 & 2.2MS2-060817	94%

A. Results for FAQForge

For *FAQForge*, our test path generator created 20 and 335 paths for the two pairs of versions, respectively. Upon manual inspection of the source for the first pair (versions 1.3.0 and 1.3.1), we discovered that only one of the files in version 1.3.1 has been changed from version 1.3.0. The file that was modified was a library file that contained functions included by the main index file. During path generation, we did not analyze any of the library files directly. Instead, the path generator analyzed files that the user would execute directly. If FAQForge the files were not executed directly, they were located in the "lib" directory. The changes made to the library file were propagated to the index file during the file preparation phase of preprocessing.

The inputs generated for the first pair consisted of 23 string inputs and 4 numeric inputs. For numeric inputs, *Choco* was able to resolve three (the other required input

from a database). Of the 23 string inputs, *Hampi* was able to solve 11. The other 12 inputs were resolved manually using information from the application's database.

The second pair of *FAQForge* (versions 1.3.1 and 1.3.2) yielded 335 paths using the path generator. Upon manual inspection of the source code, we found that 12 files in version 1.3.2 had changed from version 1.3.1. Among these 12 different 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.

The inputs generated for the second pair consisted of 198 string inputs and 21 numeric inputs. *Choco* was able to resolve 15 numeric inputs, and *Hampi* was able to resolve 93 string inputs. For the remaining inputs, we resolved them using information from the application's database.

As shown in Table IV, our technique produced a relatively small number of test paths compared to the control technique. Our technique reduced the number test paths by 96% and 76% for versions 1.3.1 and 1.3.2, respectively.

B. Results for osCommerce

For *osCommerce*, the test path generator created 1639 and 2434 paths for the two pairs of versions, respectively. Upon manual inspection of the source for the first pair (versions 2.2MS1 and 2.2MS2), we found that 279 of the 506 files had changed. The modified files were in every module of the application and the files with the largest diffs were the library files that were included in the executable files. Again, during path generation, we did not analyze any library files (files in *osCommerce*'s "include" directory) directly. Instead, the path generator analyzed only files that the user would execute directly.

The inputs generated for the first pair consisted of 474 string inputs and 52 numeric inputs. For numeric inputs, *Choco* was able to resolve 31 (the other required input from a database). Of the 474 string inputs, *Hampi* was able to solve 193. The rest of inputs were either too complex for *Hampi* to solve or required input from the application's database. For instance, situations that often required manual resolution were strings that were manipulated by native functions in PHP. Compared to the number of paths generated, a relatively small number of inputs were required. This 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 was changed, we marked the statement as a difference.

The second pair of *osCommerce* (versions 2.2MS2 and 2.2MS2-060817) yielded 2464 paths using the path generator. Upon manual inspection of the source code, we found that 105 of the 502 files had changed.

The inputs generated for this pair consisted of 63 numeric inputs and 718 string inputs. *Choco* was able to resolve 33 numeric inputs, and *Hampi* was able to resolve 205 string inputs. The remaining inputs were either too complex for *Hampi* to solve or required input from the application's database. Again, the small number of inputs needed compared to the number of paths because a large number of changes in PHP statements were primarily responsible for printing static HTML.

Considering the number of files that were changed for both versions (279 of 506 for version 2.2MS2 and 105 of 502 for version 2.2MS2-060817), the number of paths generated was relatively small (1649 for version 2.2MS2 and 2464 for version 2.2MS2-060817). The reason for this is that there were numerous changes in statements that contained no variables, and therefore, there were no data dependencies.

As shown in Table IV, our technique produced a relatively small number of test paths compared to the control technique. Our technique reduced the number of test paths by 95% and 94% for versions 2.2MS2 and 2.2MS2-060817, respectively.

V. Discussion

Our results strongly support the conclusion that the program slice based test case generation approach can generate significantly fewer test cases needed to test the modified version of the program compared to the control technique.

As we observed from the data analysis section, the test path reduction rates achieved by our approach over the control technique were significantly high for all cases (over 94% for three of four cases and 76% for one case). Because the test paths require actual inputs to create executable test cases, by reducing the number of test paths necessary for the modified program, we expect to produce further savings of the costs associated with collecting test input constraints and resolving constraints.

In particular, in the case of *osCommerce*, for the test paths generated using our approach (1639 and 2434 test paths for versions 2.2MS2 and 2.2MS2-060817), we manually resolved 302 of 526 and 543 of 781 input values for those versions, respectively. Considering the number of test paths generated by the control technique (34924 and 35775 test paths), we expect that a large number of inputs would require manual resolution, a time consuming process. In addition to this cost, other costs, such as the time required for validating results as also as the maintenance cost for test cases and their associated artifacts accumulated over time, could be added to the overall regression testing cost.

There are also some security implications for this approach. We observed that test cases generated with our 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. Our test generating tool discovered the difference and generated several test cases that traversed these changes.

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

security features for website administration. Also, there was a security fix that corrects an SQL injection problem between versions 2.2MS2 and 2.2MS2-060817. For both cases, our tool was able to generate test cases that exercised all these fixes.

One interesting security related issue in *osCommerce* was a feature added to version 2.2MS2. This feature allowed users to inject shell commands into the web server. Our tool generated several test cases that covered this change. However, without proper inputs for these test cases, we were not able to test this security feature correctly. Thus, by identifying the correct malicious input, our test execution engine was able to reveal this particular security vulnerability.

VI. 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., [4], [5]) and test case prioritization (e.g., [3], [28]). Recently, researchers have started working on test suite augmentation techniques which create new test cases for areas that have been affected by changes [6], [7], [9], [8], [29]. Apiwattanapong et al. [6] and Santelices et al. [7] present an approach to identify areas affected by code changes using program dependence analysis and symbolic execution, and provide test requirements for changed software. Xu et al. [9] present an approach to generate test cases by utilizing the existing test cases and adapting a concolic test case generation technique. Taneja et al. [8] propose an efficient test generation technique that uses dynamic symbolic execution, eXpress, by pruning irrelevant paths. These approaches focus on desktop applications such as C or Java, but our approach applies regression testing to web applications which create different challenges (e.g. dynamic programming languages deployed using multitier architecture). Chen et al. [29] present a model-based approach that generates a regression test suite using dependence analysis of the modified elements in the model.

Creating executable test cases automatically has been one of the challenging tasks in the software testing community [30], and recent research in this area has proposed a promising approach which involves concolic testing (a combination of concrete and symbolic execution) [31]. Since then, other researchers have extended the concolic technique to handle PHP code [10], [32]. Wassermann et al. [10] and Artzi et al. [32] 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 [33] use a crawler to discover the link structure of web applications and to produce test case specifications from this structure. Deng et al. [34] use static analysis to extract information related to URLs and their parameters for web applications, and generate executable test cases using the collected information. Halfond et al. [35] present an approach that uses symbolic execution to identify precise interfaces for web applications.

While these existing techniques and approaches have motivated us, our work is different from theirs in the following aspect. Our work focuses on evolving web applications that require frequent patches and regression testing. This means that, unlike other approaches, we focus only on the areas affected by code changes and generate test cases using program slicing. Some work on regression testing for web applications [36], [37] has been done, but their focus is different than ours. Dobolyi and Weimer [36] present an approach that automatically compares the outputs from similar web applications to reduce the regression testing effort. Elbaum et al. [37] present a web application testing technique that utilizes user session data considering regression testing contexts.

VII. Conclusions and Future Work

In this paper, we presented a new regression test case generation approach that creates executable test cases using program slices for web applications written in PHP. To evaluate our approach, we conducted an experiment using non-trivial, open source web applications, and our results showed that our approach can be effective in reducing the number of test cases necessary for the modified application by focusing only on the areas affected by the modified code.

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

The results of our studies suggest several avenues for future work. First, we evaluated our approach using two widely used open source web applications. Because our approach provided promising results, the next natural step is to perform additional studies that apply our approach to industrial contexts (e.g., using industrial-size applications or considering constraints imposed by an industry's regression testing practice) to see whether our approach can address scalability and practicality.

Second, when we resolved input constraints using the existing string constraint solver, *Hampi*, some cases were not resolved automatically due to *Hampi*'s limitations. Thus, we plan to build a tool that further automates the constraint resolution process by addressing some of the current problems with the existing tool.

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

Acknowledgments

Cesar Ramirez helped construct parts of the tool infrastructure used in the experimentation. This work was supported in part by NSF under Awards CNS-0855106 and CCF-1050343 to North Dakota State University.

References

[1] A. Srivastava and J. Thiagarajan, " Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, Jul. 2002, pp. 97–106.

[2] R. Boggs, J. Bozman, and R. Perry, "Reducing downtime and business loss: Adressing business risk with effective technology," IDC, Tech. Rep., Aug. 2009.

[3] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[4] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, Apr. 1997.

[5] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2007, pp. 140–150.

[6] 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.

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

[8] 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*, Jul. 2011.

[9] 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*, Nov. 2010.

[10] 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*, Jul. 2008.

[11] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[12] R. Gupta, M. Harrold, and M. Soffa, "Program slicing-based regression testing techniques," *Journal of Software Testing, Verification, and Reliability*, vol. 6, no. 2, pp. 270–285, Jun. 1996.

[13] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th ACM Symp. on Prin. of Programming Languages*, Jan. 1993.

[14] D. Jeffrey and N. Gupta, "Experiments with test case prioritization using relevant slice," *Journal of Systems and Software*, vol. 81, no. 2, pp. 196–221, 2008.

[15] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, no. 1, pp. 49–76, Mar. 2002.

[16] P. Samuel and R. Mall, "Slicing-based test case generation from uml activity diagrams," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 6, 2009.

[17] 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*, Nov. 2010, pp. 404–405.

[18] phc, http://www.phpcompiler.org/.

[19] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2009, pp. 105–115.

[20] Choco, http://www.emn.fr/z-info/choco-solver/.

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

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

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

[24] 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.

[25] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[26] osCommerce, http://www.oscommerce.com/.

[27] FaqForge, http://sourceforge.net/projects/faqforge/.

[28] 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*, Jul. 2006, pp. 1–12.

[29] 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*, Jul. 2007.

[30] R. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[31] 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*, Sep. 2005, pp. 263–272.

[32] 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.

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

[34] Y. Deng, P. Frankl, and J. Wang, "Testing web database applications," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–10, 2004.

[35] 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*, Jul. 2009.

[36] 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*, Nov. 2009.

[37] 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*, 2003, pp. 49–59.