Finding Bugs in Dynamic Web Applications

Shay Artzi[†] Adam Kieżun[†] Julian Dolby[‡]
Frank Tip[‡] Danny Dig[†] Amit Paradkar[‡] Michael D. Ernst[†]

[†]MIT CSAIL, {artzi,akiezun,dannydig,mernst}@csail.mit.edu
[‡]IBM T.J. Watson Research Center, {dolby,ftip,paradkar}@us.ibm.com

Abstract

Web script crashes and malformed dynamically-generated Web pages are common errors, and they seriously impact usability of Web applications. Current tools for Web-page validation cannot handle the dynamically-generated pages that are ubiquitous on today's Internet. In this work, we apply a dynamic test generation technique, based on combined concrete and symbolic execution, to the domain of dynamic Web applications. The technique generates tests automatically, uses the tests to detect failures, and minimizes the conditions on the inputs exposing each failure, so that the resulting bug reports are small and useful in finding and fixing the underlying faults. Our tool Apollo implements the technique for PHP. Apollo generates test inputs for the Web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. This paper presents Apollo's algorithms and implementation, and an experimental evaluation that revealed 214 faults in 4 PHP Web applications.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging;
General Terms Reliability, Verification
Keywords Software Testing, Web Applications, Dynamic Analysis, PHP

1. Introduction

Dynamic test-generation tools, such as DART [15], Cute [28], and EXE [5], find failures by executing an application on concrete input values, and then creating additional input values by solving symbolic constraints derived from exercised control flow paths. To date, such approaches have not been practical in the domain of Web applications. This paper extends dynamic test generation to scripting languages, uses an oracle to determine whether the output of the Web application is syntactically correct, and automatically sorts and minimizes the inputs that expose failures. Our Apollo system applies these techniques in the context of PHP, one of the most popular languages for Web programming. According to Netcraft, PHP powered 21 million domains as of April 2007, including large, well-known websites such as Wikipedia and WordPress.

The output of a Web application is typically an HTML page that can be displayed in a browser. Our goal is to find faults that are manifested as Web application crashes or as malformed HTML. Some faults may terminate the application, such as when a Web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output presents an error message and the application execution is halted.

More commonly in deployed applications, a Web application creates output that is not syntactically well-formed HTML, for example by generating an opening tag without a matching closing tag. Web browsers are designed to tolerate some degree of malformedness in HTML, but this merely masks underlying failures. Malformed HTML is less portable across browsers and is vulnerable to breaking on new browser releases. An application that creates invalid (but displayable) HTML during testing may create undisplayable HTML on different executions. More seriously, browsers' attempts to compensate for malformed Web pages may lead to crashes and security vulnerabilities[1]. A browser might also succeed in displaying only part of a malformed webpage, silently discarding important information. Search engines may have trouble indexing incorrect pages. Standard HTML renders on more browsers, and valid pages are more likely to look as expected, including on future versions of Web-browsers. Standard HTML renders faster[2]. For example, in Mozilla, "improper tag nesting [...] triggers residual style handling to try to produce the expected visual result, which can be very expensive" [25].

Web developers widely recognize the importance of creating legal HTML. Many websites are validated using HTML validators[3] (even the ISSTA'08 website displays the W3C HTML compliance logo). However, HTML validators are used only for static pages.

Validating *dynamic* Web applications (i.e., application that generate pages during the execution) is hard. Even professionally-developed applications often contain multiple faults (see Section 5). To prevent faults, programmers must make sure that the application creates a valid HTML page on *every* possible execution path. There are two general approaches to this problem: static and dynamic checking (testing).

Static checking of dynamic Web applications cannot fully capture their behavior. Such application are often written in languages such as PHP that enable on-the-fly creation of code and overriding of methods. In many Web applications, part (further pages) of the application is referenced from the generated HTML text (e.g., buttons and menus that require user interaction to execute), rather than from the analyzed code. Specialized analysis may be possible for a custom language, such as `<bigwig>` [3].

Testing of dynamic Web applications is also hard, because the input space is large, and application usually require multiple user in-

[1] See bug reports 269095, 320459, and 328937 at `https://bugzilla.mozilla.org/show_bug.cgi?`

[2] `http://weblogs.mozillazine.org/hyatt/archives/2003_03.html#002904`

[3] `http://validator.w3.org`, `http://www.htmlhelp.com/tools/validator`

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

teractions. The state-of-the-practice in validation for Web-standard compliance of real Web-applications is using programs such as HTML Kit[4] that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no automated validator for scripting languages that dynamically generate HTML pages.

This paper presents an automated technique for finding failures in HTML-generating Web applications. Our technique adapts the technique of dynamic test generation, based on combined concrete and symbolic execution and constraint solving [5, 15, 28], to the domain of Web applications. In our technique, the Web application under test is first executed with an empty input. During each execution, the program is monitored to record path constraints that capture the outcome of control-flow predicates. Additionally, for each execution an oracle determines whether fatal failures or HTML well-formedness failures occur, the latter via use of an HTML validator. The system automatically and iteratively creates new inputs by negating one of the observed constraints and solving the modified constraint system. Each newly-created input explores at least one additional control flow path.

Our work differs from previous approaches for testing Web applications by using an oracle to detect specification violations in the application's output, in addition to crashes or assertion failures. Another novelty in our work is inference of input parameters, which are not manifested in the source code. Our technique simulates user interaction by transforming the Web application to create additional input parameters that the execution engine interprets as user input.

Techniques based on combined concrete and symbolic executions [5, 15, 28] may create multiple inputs that expose the same failure. In contrast to previous techniques, to avoid overwhelming the developer, our technique automatically identifies the minimal part of the input that is responsible for triggering the failure. This step is similar in spirit to Delta Debugging [6]. However, since Delta Debugging is a general, *black-box*, input minimization technique, it is oblivious to the properties of inputs. In contrast, our technique is *white-box*: it uses the information that certain inputs induce partially overlapping control flow paths. By intersecting these paths, our technique minimizes the constraints on the inputs within fewer program runs. There are also significant differences in the domain and language under consideration (Web PHP applications, versus desktop C applications), as we discuss in Section 6.

We created a tool, Apollo, that implements our method in the context of the publicly available PHP interpreter. We evaluated Apollo on publicly available Web applications. In a time budget of 10 minutes per program, Apollo found 214 different faults.

In summary, the contributions of this paper are:

- We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution [5, 15, 28], to the domain of Web applications. The challenges include inferring the input parameters, which are not indicated by the source code; using an HTML verifier as an oracle; dealing with language-specific datatypes and operations; and simulating user input for interactive applications.
- We created a tool, Apollo, that implements the technique for PHP.
- We evaluated our tool by applying it to real Web applications and comparing the results with random testing. We show that dynamic test generation is highly effective when adapted to the domain of Web applications written in PHP: Apollo achieved line coverage of 58.0% and identified 214 faults.

[4] `http://www.htmlkit.com`

The remainder of this paper is organized as follows. Section 2 presents an overview of PHP, introduces our running example, and discusses classes of failures in PHP Web applications. Section 3 presents the algorithm and illustrates it on an example program. Section 4 discusses our Apollo implementation. Section 5 presents our experimental evaluation of Apollo on open-source Web applications. Section 6 gives an overview of related work, and Section 7 presents conclusions.

2. Context: PHP Web Applications

2.1 The PHP Scripting Language

This section briefly reviews the PHP scripting language, focusing on those aspects of PHP that differ from mainstream languages. Readers familiar with PHP may skip to the discussion of the running example in Section 2.2.

PHP is widely used for implementing Web applications, in part due to its rich library support for network interaction, HTTP processing, and database access. The input to a PHP program is a map from strings to strings. Each key is a parameter that the program can read, write, or check if it is set. The string value corresponding to a key may be interpreted as a numerical value if appropriate. The output of a PHP Web application is an HTML document that can be presented in a Web browser.

PHP is object-oriented, in the sense that it has classes, interfaces, and dynamically dispatched methods with syntax and semantics similar to that of Java. PHP also has features of scripting languages, such as dynamic typing, and an `eval` construct that interprets and executes a string value that was computed at run-time as a code fragment. For example, the following code fragment:

```
$code = "$x = 3;"; $x = 7; eval($code); echo $x;
```

prints the value `3` (names of PHP variables start with the `$` character). Other examples of the dynamic nature of PHP are a predicate that checks whether a variable has been defined, and class and function definitions that are statements that may occur anywhere.

The code in Figure 1 illustrates the flavor of PHP. The `require` statement that used on line 11 of Figure 1 resembles the C `#include` directive in the sense that it includes the code from another source file. However, the C version is a pre-processor directive with a constant argument, whereas the PHP version is an ordinary statement in which the file name is computed at runtime. There are many similar cases where run-time values are used, e.g., `switch` labels need not be constant. This degree of flexibility is prized by PHP developers for enabling rapid application prototyping and development. However, the flexibility can make the overall structure of program hard to discern and it can make programs prone to code quality problems.

2.2 PHP Example

The PHP program of Figure 1 is a simplified version of SchoolMate[5], which allows school administrators to manage classes and users, teachers to manage assignments and grades, and students to access their information.

Lines 6–7 read the global parameter `page` that is supplied to the program in the URL, e.g., `http://www.mywebsite.com/index.php?page=1`. Line 10 examines the value of the global parameter `page2` to determine whether to evaluate file `printReportCards.php`.

Function `validateLogin` (lines 27–39) sets the global parameter `page` to the correct value based on the identity of the user. This value is read in the `switch` statement on line 18, which presents the login screen or one of the teacher/student screens.

[5] `http://sourceforge.net/projects/schoolmate`

```
1  <?php
2
3  make_header(); // print HTML header
4
5  // Make the $page variable easy to use //
6  if(!isset($_GET['page’])) $page = 0;
7  else $page = $_GET['page’];
8
9  // Bring up the report cards and stop processing //
10 if($_GET['page2’]==1337) {
11   require(’printReportCards.php’);
12   die();  // terminate the PHP program
13 }
14
15 // Validate and log the user into the system //
16 if($_GET["login"] == 1) validateLogin();
17
18 switch ($page)
19 {
20   case 0:  require(’login.php’); break;
21   case 1:  require(’TeacherMain.php’); break;
22   case 2:  require(’StudentMain.php’); break;
23   default: die("Incorrect page number.  Please verify.");
24 }
25
26 make_footer(); // print HTML footer
27 ...
```

```
27 function validateLogin() {
28   if(!isset($_GET['username’])) {
29     echo "<j2> username must be supplied.</h2>\n";
30     return;
31   }
32   $username = $_GET['username’];
33   $password = $_GET['password’];
34   if($username=="john" && $password=="theTeacher")
35     $page=1;
36   else if($username=="john" && $password=="theStudent")
37     $page=2;
38   else echo "<h2>Login error. Please try again</h2>\n";
39 }
40
41 function make_header() { // print HTML header
42 print("
43 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
44    "http://www.w3.org/TR/html4/strict.dtd">
45 <HTML>
46  <HEAD> <TITLE> Class Management  </TITLE> </HEAD>
47  <BODY>");
48 }
49
50 function make_footer() {  // close HTML elements opened by header()
51 print("
52  </BODY>
53 </HTML>");
54 }
55 ?>
```

Figure 1: A simplified PHP program excerpt from SchoolMate. This excerpt contains three faults (2 real, 1 seeded), explained in Section 2.3.

2.3 Failures in PHP Programs

Our technique targets two types of failures that can be automatically identified as violations of the partial specification of executing PHP Web applications. (I) *execution failures* are caused by missing an included file, wrong MySQL query, and uncaught exceptions. Such failures are easily identified as the PHP interpreter generates an error message and halts execution. Less serious execution failures, such as using deprecated language constructs, produce obtrusive error messages but do not halt execution. (II) *HTML failures* involve situations in which the generated HTML page is not syntactically correct according to an HTML validator. Section 1 discussed the negative consequences of malformed HTML.

The program of Figure 1 contains three faults resulting in the following failures:

1. The program contains an *execution failure*: the file `printReportCards.php` referenced on line 11 is missing.
2. The program produces *malformed HTML* because the `make_footer` method is not executed in certain situations, resulting in an unclosed HTML tag in the output. The default case of the `switch` statement on line 23 terminates program execution when the global parameter `page` is not 0, 1, or 2 and when `page` is not written by function `ValidateLogin`.
3. The program produces *malformed HTML* when line 29 generates an illegal HTML tag `j2`.

The first failure is similar to a failure that our tool found. The second failure is the result of a fault that exists in the original code of the SchoolMate program. The third failure is the result of a fault that was artificially inserted into the example for illustration.

3. Finding Failures in PHP Web Applications

Our technique for finding failures in PHP applications is a variation on an established dynamic test generation technique [5, 15, 16, 28] sometimes referred to as concolic testing. The basic idea is to execute an application on an initial input (e.g., an arbitrarily or randomly-chosen input), and then on additional inputs obtained by solving constraints derived from exercised control flow paths. We adapted this technique to PHP Web applications as follows:

• We extend the technique to validate the correctness of the program output. We use an oracle, in the form of an HTML validator, to determine whether the output is a well-formed HTML page.

• The PHP language contains constructs such as `isset` (checking whether a variable is defined), `isempty` (checking whether a variable contains a value from a specific set), `require` (dynamic loading of additional code to be executed), and several others that require the generation of constraints that are absent in languages such as C or Java.

• PHP applications typically interact with a database and need appropriate values for user authentication (i.e., user name and password). It is not possible to infer these values by either static or dynamic analysis, or by randomly guessing. Therefore, our technique uses a pre-specified set of values for database authentication.

• The HTML pages generated by a PHP applications may contain buttons that—when pressed by the user—result in the loading and execution of additional PHP source files. We simulate such user input by transforming the source code. Specifically, for each page h that contains N buttons, we add an additional input parameter p to the PHP program, whose values may range from 1 through N. Then, at the place where page p is generated, a switch statement is inserted that includes the appropriate PHP source file, depending on the value supplied for p. The steps of the user input simulator are fully mechanical, and the required modifications are minimal, but for expediency we performed the program transformation by hand.

3.1 Algorithm

Figure 2 shows the pseudo-code of our algorithm. The inputs to the algorithm are: a program $\mathcal{P}$ and an output oracle $\mathcal{O}$. The output of the algorithm is a set of bug reports $\mathcal{B}$ for the program $\mathcal{P}$, according to $\mathcal{O}$. Each bug report contains: identifying information about the failure, the set of all inputs under which the failure was exposed, and the set of all path constraints that lead to the inputs exposing the failure.

parameters: Program $\mathcal{P}$, oracle O
result : Bug reports $\mathcal{B}$;
$\mathcal{B}$: *setOf*(⟨failure, *setOf*(pathConstraint), *setOf*(input)⟩)

1 $\mathcal{P}' := simulateUserInput(P)$;
2 $\mathcal{B} := \emptyset$;
3 $pcQueue := emptyQueue()$;
4 $enqueue(pcQueue, emptyPathConstraint())$;
5 **while** *not empty(pcQueue) and not timeExpired()* **do**
6 $\quad pathConstraint := dequeue(pcQueue)$;
7 $\quad input := solve(pathConstraint)$;
8 $\quad$ **if** $input \neq \perp$ **then**
9 $\quad\quad output := executeConcrete(\mathcal{P}', input)$;
10 $\quad\quad failures := getFailures(O, output)$;
11 $\quad\quad$ **foreach** *f in failures* **do**
12 $\quad\quad\quad$ merge $\langle f, pathConstraint, input\rangle$ into $\mathcal{B}$;
13 $\quad\quad c_1 \wedge \ldots \wedge c_n := executeSymbolic(\mathcal{P}', input)$;
14 $\quad\quad$ **foreach** $i = 1,\ldots,n$ **do**
15 $\quad\quad\quad newPC := c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
16 $\quad\quad\quad enqueue(pcQueue, newPC)$;
17 **return** $\mathcal{B}$;

Figure 2: The failure detection algorithm. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\perp$ if no satisfying input exists. The output of the algorithm is a set of bug reports. Each bug report contains a failure, a set of path constraints exposing the failure, and a set of input exposing the failure.

The algorithm uses a queue of path constraints. A *path constraint* is a conjunction of conditions on the program's input parameters. The queue is initialized with the empty path constraint (line 4). The algorithm uses a constraint solver to find a concrete input that satisfies a path constraint taken from the queue (lines 6– 7). The program is executed concretely on the input and tested for failures (lines 9–10). The path constraint and input for each detected failure are merged into the corresponding bug report (lines 11–12). Next, the program is executed symbolically on the same input (line 13). The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, that is fulfilled if the given path is executed (here, the path constraint reflects the path that was just executed). The algorithm then creates new test inputs by solving modified versions of the path constraint (lines 14–16), as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 15). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along a prefix of the original execution path, and then take the opposite branch.

3.2 Example

Let us now consider how the algorithm of Figure 2 exposes the third fault in the example program of Figure 1.

Execution 1. The first input to the program is the empty input, which is the result of solving the empty path constraint. During execution, the condition on line 6 evaluates to `true` (which sets `page` to `0`) and the condition on line 10 evaluates to `false`. The condition on line 16 evaluates evaluates to `false` because parameter `login` is not defined. The `switch` statement on line 18 selects the case on line 20 because `page` has the value of `0`. Execution terminates on line 26. The HTML verifier determines that the output is legal, and *executeSymbolic* produces the following path constraint:

$$NotSet(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} \neq \texttt{1} \quad \text{(I)}$$

The algorithm now enters the **foreach** loop on line 14 of Figure 2, and starts generating new path conditions by systematically traversing subsequences of the above path constraint, and negating the last conjunct. Hence, from (I), the algorithm derives the following three path constraints:

$$NotSet(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} = \texttt{1} \quad \text{(II)}$$
$$NotSet(\texttt{page}) \wedge \texttt{page2} = \texttt{1337} \quad \text{(III)}$$
$$Set(\texttt{page}) \quad \text{(IV)}$$

Execution 2. For path constraint (II), the constraint solver may find the following input (the solver is free to select any value for `page2`, other than 1337): `page2` ← `0`, `login` ← `1`.

When the program is executed with this input, the condition of the if-statement on line 16 evaluates to `true`, resulting in a call to the `validateLogin` method. Then, the condition of the if-statement on line 28 evaluates to `true` because the `username` parameter is not set, resulting in the generation of output containing an incorrect HTML tag `j2` on line 29. When the HTML validator checks the page, the failure is discovered and a bug report is created and added to the output set of bug reports.

parameters: Program $\mathcal{P}$, oracle O, bug report b
result : Short path constraint that exposes *b.failure*

1 $c_1 \wedge \ldots \wedge c_n := intersect(b.pathConstraints)$;
2 $pc := true$;
3 **foreach** $i = 1, \ldots, n$ **do**
4 $\quad pc_i := c_1 \wedge \ldots c_{i-1} \wedge c_{i+1} \wedge \ldots c_n$;
5 $\quad input := solve(pc_i)$;
6 $\quad$ **if** $input \neq \perp$ **then**
7 $\quad\quad output := executeConcrete(\mathcal{P}, input)$;
8 $\quad\quad failures := getFailures(O, output)$;
9 $\quad\quad$ **if** $b.failure \notin failures$ **then**
10 $\quad\quad\quad pc := pc \wedge c_i$;
11 $input_{pc} := solve(pc)$;
12 **if** $input_{pc} \neq \perp$ **then**
13 $\quad output_{pc} := executeConcrete(\mathcal{P}, input_{pc})$;
14 $\quad failures_{pc} := getFailures(O, output_{pc})$;
15 $\quad$ **if** $b.failure \in failures_{pc}$ **then**
16 $\quad\quad$ **return** pc;
17 **return** $shortest(b.pathConstraints)$;

Figure 3: The path constraint minimization algorithm. The method *intersect* returns a conjunction containing the conditions that are present in all given path constraints, and the method *shortest* returns the path constraint with fewest conjuncts. The other auxiliary functions are the same as in Figure 2.

3.3 Path Constraint Minimization

The failure detection algorithm (Figure 2) returns bug reports for different failures. Each bug report contains a set of path constraints leading to inputs exposing the failure. Previous dynamic test generation tools [5, 15, 28] presented the whole input to the user without an indication of the subset of the input responsible for the failure. As a postmortem phase, our minimizing algorithm attempts to find a shorter path constraint for a given bug report (Figure 3). This eliminates irrelevant constraints, and a solution for a shorter path constraint is often a smaller input.

For a given bug report b, the algorithm first intersects all the path constraints exposing *b.failure* (line 1). The minimizer systematically removes one condition at a time (lines 3-10). If one of these shorter path constraints does not expose *b.failure*, then the removed condition is required for exposing *b.failure*. The final path constraint is the conjunction of all such required conditions. From the minimized path constraint, the algorithm produces a concrete input that exposes the failure.

The algorithm in Figure 3 does not guarantee that the returned path constraint is the shortest possible that exposes the failure. However, the algorithm is simple, fast, and effective in practice (see Section 5.3.2).

Our minimizer differs from *input* minimization techniques, such as delta debugging [6,34], in that our algorithm operates on the *path constraint* that exposes the failure, and not the *input*. A constraint concisely describes a class of inputs (e.g., the constraint $page2 \neq 1337$ describes all inputs different than 1337). Since a concrete input is an instantiation of a constraint, it is more effective to reason about input properties in terms of their constraints.

Each failure might be encountered along several execution paths that might partially overlap. Without any information about the properties of the inputs, delta debugging minimizes only a *single* input at a time, while our algorithm handles *multiple* path constraints that lead to a failure.

3.4 Minimization Example

The malformed HTML failure described in Section 3.2 can be triggered along different execution paths. For example, both of the following path constraints lead to inputs that expose the failure. Path constraint (*a*) is the same as (II) in Section 3.2.

$$NotSet(\texttt{page}) \wedge \texttt{page2} \neq 1337 \wedge \texttt{login} = 1 \qquad (a)$$
$$Set(\texttt{page}) \wedge \texttt{page} = 0 \wedge \texttt{page2} \neq 1337 \wedge \texttt{login} = 1 \qquad (b)$$

First, the minimizer computes the intersection of the path constraints (line 1). The intersection is

$$\texttt{page2} \neq 1337 \wedge \texttt{login} = 1 \qquad (a \cap b)$$

The minimizer creates two shorter path constraints (by removing each of the two conjuncts in turn). First, the minimizer creates path constraint `login = 1`. This path constraint corresponds to an input that reproduces the failure, namely `login` ← `1`. The minimizer knows this by executing the program on the input (line 8 in Figure 3). Second, the minimizer creates path constraint `page2` ≠ `1337`. This path constraint does not correspond to an input that exposes the failure. Thus, the minimizer concludes that the condition `login = 1`, that was removed from $(a \cap b)$ to form the second path constraint, is required. In this example, the minimizer returns `login = 1`. The result is the minimal path constraint that describes this failure-inducing input.

4. Implementation

We created a tool called Apollo that implements our technique for PHP. Apollo consists of three major components, **Executor**, **Input Generator**, and **Bug Finder**, illustrated in Figure 4. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation.

The **User Input Simulator** component performs a transformation of the program that models interactive user input.

The **Executor** is responsible for executing a given PHP file with a given input. Before each execution, the executor creates the appropriate database for the application. The executor contains two sub-components:

- The **Shadow Interpreter** is a PHP interpreter that we have modified to record path constraints and positional information associated with output.
- The **Database Manager** initializes the database used by a PHP application, and restores it before each execution.

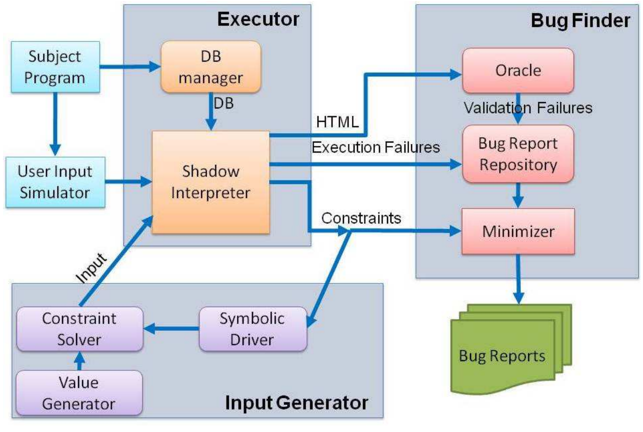

Figure 4: The architecture of Apollo.

The **Bug Finder** uses an oracle to find HTML failures, stores the all bug reports, and finds the minimal conditions on the input parameters for each bug report.

The Bug Finder has the following sub-components:

- The **Oracle** finds HTML failures in the output of the program.
- The **Bug Report Repository** stores all bug reports containing execution failures and HTML failures found during all executions.
- The **Input Minimizer** finds, for a given bug report, the smallest path constraint on the input parameters that results in inputs inducing the same failure as in the report.

The **Input Generator** contains the implementation of the algorithm described in Section 3. The Input Generator contains the following sub-components:

- The **Symbolic Driver** generates new path constraints, and selects the next path constraint to solve for each execution.
- The **Constraint Solver** computes an assignment of values to input parameters that satisfies a given path constraint.
- The **Value Generator** generates values for parameters that are not otherwise constrained, using a combination of random value generation, and constant values mined from the program source code.

4.1 User Input Simulator

Many PHP Web applications create interactive HTML pages that contain user interface elements such as buttons and menus that require user interaction to execute further parts of the application. In such cases, pressing the button may result in the execution of additional PHP source files. For example, Figure 5 contains a simplified main entry file (`index.php`)[6] of Webchess, one of the programs evaluated in Section 5, which contains user interface elements that refer to two additional scripts, `mainmenu.php` and `newuser.php`. There are two challenges involved in dealing with such interactive applications. First, our basic approach does not automatically analyze the referenced files, because these are referenced from within HTML output (as opposed to being referenced from within PHP source code). Second, global information is shared between the different scripts using the `SESSION` global table.

[6] A PHP file is a combination of text (usually HTML) and PHP snippets. When this file is processed the PHP interpreter output the HTML parts as is, and executed the PHP parts when appropriate.

Our current approach to the above challenges is to simulate user interaction by transforming the script in the following way: (i) adding an integer-valued parameter `_btn` to the main PHP script, whose value denotes the button that has been selected, and (ii) adding a `switch` statement to the main PHP file that uses the value of `_btn` is used to select an additional PHP file to be included (using a `require_once` statement). For example, for the program of Figure 5, we add:

```
switch($_GET["_btn"]) {
case 1:
        require_once("mainmenu.php");
        break;
case 2:
        require_once("newuser.php");
        break;
}
```

This approach has the advantages that `SESSION` state is automatically shared between the different files, since the session is now an ordinary array used in the normal way by both the included and the original code. Our algorithm is then able to find the values that correspond to each of the user interface elements. Since code might be executed when a button is pressed, this approach might induce false positive bug reports. In our experiments, this limitation produced no false positive bug reports. A slight disadvantage of this approach is that the transformed PHP application will output a *sequence* of HTML pages rather than a single page, so that some post processing is needed before the HTML validator can be invoked. However, the transformation is mechanical and the required source code changes are minimal. We currently perform the transformation manually and are investigating a solution where the transformation is performed automatically.

4.2 Executor

We modified the Zend PHP interpreter 5.2.2[7] to produce symbolic path constraints for the executed program, using the "shadow interpreter" approach [7]. The shadow interpreter performs the regular (concrete) program execution using the concrete values, and simultaneously performs symbolic execution. Creating the shadow interpreter required three alterations to the PHP runtime:

(1) A symbolic variable may be associated with each value. Values derived from the input—that is, either read directly as input or computed from input values—have symbolic variables associated with them. Values not derived from the input do not. These associations arise when a value is read from one of the special arrays `_POST`, `_GET`, and `_REQUEST`, which store parameters supplied to the PHP program. For example, executing the statement `$x = $_GET["param1"]` results in associating the value read from the global parameter `param1` and bound to parameter `x` with the symbolic variable `param1`. Values maintain their associations through assignments and function calls (thus, the interpreter performs symbolic execution at the inter-procedural level). Importantly, during program execution, the concrete values remain, and the shadow interpreter does not influence execution.

Unlike other projects that perform concrete and symbolic execution [5, 15, 16, 28], our interpreter does not associate complex symbolic expressions with any runtime values, but only symbolic variables and those only for input-derived values. This design keeps the constraint solver very simple and reduces the performance overhead. As our results (Section 5) indicate, this lightweight approach is sufficient for the analyzed PHP programs.

(2) At branching points (i.e., value comparisons) that involve values associated with symbolic variables, the interpreter extends the initially empty path constraint with a conjunct that corresponds to the branch actually taken in the execution. For example, if the program executes a statement `if ($name == "John")` and this condition succeeds, where `$name` is associated with the symbolic variable `"username"`, then the algorithm appends the conjunct `username = "John"` to the path constraint.

(3) Our modified interpreter records conditions for PHP-specific comparison operations, such as `isset` and `empty`, which can be applied to any variable. Operation `isset` returns a boolean value that indicates whether or not a value different from `NULL` was supplied for a variable. The `empty` operator returns true when applied to: the empty string, `0`, `"0"`, `NULL`, `false`, or an empty array. The interpreter records the use of `isset` on values with an associated symbolic variable, and on uninitialized parameters.

The `isset` comparison creates either the *NotSet* or the *Set* condition. The constraint solver chooses an arbitrary value for a parameter `p` if the only condition for `p` is *Set* (`p`). Otherwise, it will also take into account other conditions. The *NotSet* condition is used only in checking the feasibility of a path constraint. A path constraint with the *NotSet* (p) condition is feasible only if it does not contain any other conditions on `p`. The `empty` comparison creates equality or inequality conditions between the parameter and the values that are considered empty by PHP.

The modified interpreter performs symbolic execution along with concrete execution, i.e., every variable during the program execution has a concrete value and may have additionally a symbolic value. Only the concrete values influence the control flow during the program execution, while the symbolic execution is only a "witness" that records, but does not influence, control flow decisions at branching points. This design deals with exceptions naturally because exceptions do not disrupt the symbolic-value mapping for variables.

Our approach to symbolic execution allows us to handle many PHP constructs that are problematic in a purely static approach. For instance, for computed variable names (e.g., `$x = ${$foo}`), any symbolic information associated with the value that is held by the variable named by `foo` will be passed to `x` by the assignment[8]. In order to heuristically group HTML failures that may be manifestations of the same fault, Apollo records the output statement (i.e. `echo` or `print`) that generated each fragment of HTML output.

Database Manager. Most PHP applications use a database and to execute such programs some initial values need to be supplied. Apollo's Database Manager is responsible for: (i) (re)initializing the database prior to each execution (i.e., filling it with some initial values), and (ii) supplying additional information about username/password pairs. Attempting to retrieve information from the database using randomly chosen values for username/password is unlikely to be successful. Symbolic execution is equally helpless without the database manager because reversing cryptographic functions is beyond the state-of-the-art for constraint solvers.

4.3 Bug Finder

Bug Report Repository The repository stores the bug reports found in all executions. Each time a failure is detected, the corresponding bug report (for all failures with the same characteristics) is updated with the path constraint and the input inducing the failure. A failure is defined by its characteristics, which include: the type of the failure (execution failure or HTML failure), the corresponding message (PHP error/warning message for exe-

[7] `http://www.php.net/`

[8] On the other hand, any data flow that passes outside PHP, such as via JavaScript code in the generated HTML, will not be tracked by this approach.

cution failures, and validator message for HTML failures), and the PHP statement generating the problematic HTML part pointed by the validator (for HTML failures), or the PHP statement involved in the PHP interpreter error report (for execution failures). When the exploration is complete, each bug report contains one failure characteristics, and the sets of path constraints and inputs exposing failures with the same characteristics.

Oracle. PHP Web applications output HTML/XTHML. Therefore, in Apollo, we use as oracle an HTML validator that returns syntactic (malformed) HTML failures found in a given document. We experimented with both the offline WDG validator[9] and the online W3C markup validation service[10]. Both oracles identified the same HTML failures. Our experiments use the laster WDG validator.

Input Minimizer. Apollo implements the algorithm described in Figure 3 to perform *postmortem* minimization of the path constraints. For each bug report, the minimizer executes the program multiple times, with multiple inputs that satisfy different path constraints, and attempts to find the shortest path constraint that results (executing the program with an input satisfying the path constraint) in the same failure characteristics.

4.4 Input Generator

The **Symbolic Driver** implements the combined concrete and symbolic algorithm of Figure 2. The driver has two main tasks: select which input to consider next (line 6), and create additional inputs from each executed input (by negating conjuncts in the path constraint). To select which input to consider next, the driver uses a *coverage heuristic*, similar to those used in EXE [5] and SAGE [16]. Each conjunct in the path constraint knows the branch that created the conjunct, and the driver keeps track of all branches previously executed and favors inputs created from path constraints that contain un-executed branches.

Constraint Solver. The interpreter implements a lightweight symbolic execution, in which the only constraints are equality and inequality with constants. Apollo transforms path constraints into integer constraints in a straightforward way, and uses `choco`[11] to solve them.

This approach still allows us to handle values of the standard types (integer, string), and is straightforward because the only constraints are equality and inequality[12].

In cases where parameters are unconstrained, Apollo uses a combination of values that are randomly generated and values that are obtained by mining the program text for constants (in particular, constants used in comparison expressions).

5. Evaluation

We experimentally measured the effectiveness of Apollo in finding faults in PHP Web applications. We designed the experiments to answer the following research questions:

Q1. How many faults can Apollo find, and of what varieties?

Q2. How effective is the fault localization technique of Apollo compared to alternative approaches such as randomized testing, in terms of the number and severity of discovered faults and the line coverage achieved?

Q3. How effective is our minimization in reducing the size of inputs parameter constraints and failure-inducing inputs?

[9] `http://htmlhelp.com/tools/validator/offline`

[10] `http://validator.w3.org`

[11] `http://choco-solver.net/index.php?title=Main_Page`

[12] Floating-point values can be handled in the same way, though none of the examined programs required it.

```
<?php
   echo "<h2>WebChess ".$Version." Login"</h2>;
?>
<form method="post" action="mainmenu.php">
<p>
   Nick: <input name="txtNick" type="text" size="15"/>
   <br />
   Password: <input name="pwdPassword" type="password" size="15"/>
</p>
<p>
   <input name="login" value="login" type="submit"/>
   <input name="newAccount" value="New Account"
     type="button" onClick="window.open('newuser.php', '_self')"/>
</p>
</form>
```

Figure 5: A simplified version of the main entry point (`index.php`) to a PHP program. The HTML output of this program contains a form with two buttons. Pressing the `login` button executes `mainmenu.php` and pressing the `newAccount` button will execute the `newuser.php` script.

program	#files	total LOC	PHP LOC	#downloads
faqforge	19	1712	734	14164
webchess	24	4718	2226	32352
schoolmate	63	8181	4263	4466
phpsysinfo	73	16634	7745	492217
total	179	31245	14968	543199

Figure 6: Characteristics of subject programs. The **#files** column lists the number of `.php` and `.inc` files in the program. The **total LOC** column lists the total number of lines in the program files. The **PHP LOC** column lists the number of lines that contain executable PHP code. The **#downloads** column lists the number of downloads from `http://sourceforge.net`.

For the evaluation, we selected the following four open-source PHP programs (from `http://sourceforge.net`):

faqforge(1.3.2) : tool for creating and managing documents.

webchess(0.9.0) : online chess game.

schoolmate(1.5.4) : PHP/MySQL solution for administering elementary, middle, and high schools.

phpsysinfo(2.5.3) : displays system information, e.g., uptime, CPU, memory, etc.

Figure 6 presents the characteristics of the subject programs.

5.1 Generation Strategies

We compared our technique to two other approaches. First, we implemented an approach similar to that proposed by Halfond and Orso [17] for JavaScript (described as **Randomized** below). Second, we compared our results to those reported by Minamide's static analysis [23] on the same subject programs (Section 5.3.1 presents the results).

We use the following test input generation strategies in the remainder of this section:

- **Apollo** generates test inputs using the technique described in Section 3.
- **Randomized** generates test inputs by giving random values to parameters. The values are chosen from constant values that appear textually in the program source and from default values. A difficulty is that the parameters' names and types are not immediately clear from the source code. The randomized strategy infers the parameters' names and types from dynamic traces—any variable for which the user can supply a value, is classified as a parameter.

5.2 Methodology

We ran each test input generation strategy for 10 minutes on each subject program. The time limit was chosen arbitrarily, but it allows each strategy to generate hundreds of inputs and we have no reason to think the results would be much affected by a different time limit. This time budget includes all experimental tasks, i.e., program execution, harvesting of constant values from program source, test generation, constraint solving (where applicable), output validation via oracle, and line coverage measurement. To avoid bias, we ran both strategies inside the same experimental harness. This includes the Database Manager (Section 4), that supplies user names and passwords for database access. For our experiments, we use the WDG offline HTML validator, version 1.2.2.

We measured line coverage, i.e., the ratio of the number of executed lines to the total number of lines with executable PHP code in the application. We statically computed the total number of executable PHP lines in the subject programs by counting, in the interpreter, the number of lines with PHP opcodes. Figure 6 gives the size of the subject programs.

We classify the discovered faults into five groups based on their different failure characteristics. These are a refinement of the classification in Section 2.3.

execution crash: PHP interpreter terminates with an exception.

execution error: PHP interpreter emits a warning visible in the generated HTML.

execution warning: PHP interpreter emits a warning invisible in the generated HTML.

HTML error: program generates HTML for which the validator produces an error report.

HTML warning: program generates HTML for which the validator produces a warning report.

5.3 Results

Figure 7 tabulates the faults (each fault is counted only once) and line coverage results of running the two test input generation strategies on the subject programs. From these results, it is clear that the Apollo test generation strategy outperforms the randomized testing by achieving an average line coverage of 58.0%, versus 15.2% for **Randomized**. The Apollo strategy significantly outperforms the **Randomized** strategy by finding a total of 214 faults in the subject applications, versus a total of 59 fault for **Randomized**.

We examined the detailed results for schoolmate:

The 2 execution crashes happen when the program tries to load two files that are missing from the distribution of schoolmate. Since schoolmate contains 63 PHP source files and compilation is done on-the-fly when the interpreter loads a new file, such faults can be hard to detect. The developer needs to execute all possible paths to make sure the program loads all relevant files.

The 30 execution errors are all database-related, where the application had difficulties accessing the database, resulting in error messages such as (1) "supplied argument is not a valid MySQL result resource" and (2) "Unable to jump to row 0 on MySQL result".

These error messages have the same cause: user-supplied input parameters are concatenated directly into SQL query strings; leaving these parameters blank results in malformed SQL causing the `mysql_query` functions to not return a valid result. The subject programs failed to check the return value of `mysql_query`, and simply assume that a valid result was returned. These are potentially serious faults, since they are symptoms of a worse problem: the concatenation of user-supplied strings into SQL queries makes these programs vulnerable to SQL injection attacks [8]. Thus our testing approach indicates these serious vulnerabilities despite not being specifically designed to look for security issues.

All 14 execution warnings were about an unset time zone (which results in the interpreter using an arbitrary timezone).

The 58 faults in `schoolmate` that manifested resulted in generation of malformed HTML can be classified as follows:

- 7 cases where an invalid attribute is used, e.g., "there is no attribute BORDERCOLOR",
- 7 cases where a required attribute is missing, e.g., "required attribute TYPE not specified",
- 2 cases where an undefined element is used, e.g., "element EMPTY undefined",
- 1 case where an incorrect value for an attribute is supplied: "value of attribute ALIGN cannot be CENTER; must be one of TOP, MIDDLE, BOTTOM, LEFT, RIGHT",
- 28 cases where a tag is not properly closed, e.g., "found end tag for element FONT which is not open",
- 10 cases where an element is used in a place where it is not allowed, e.g., "document type does not allow element BODY here",
- 3 cases where a duplicate attribute is supplied, e.g., "duplicate specification of attribute CELLPADDING".

The breakdown of the faults for the other PHP applications that we analyzed is similar. Indeed, we noticed that the two SQL-related error messages quoted above for schoolmate recurred in faqforge (9 cases of error 1) and webchess (19 cases of error 1 and 1 of error 2). The other severe faults Apollo discovers in webchess happen when the interpreter tries to call an undefined function. The call to include the PHP files that defines the function is not executed due to a value supplied as one of the parameters.

5.3.1 Comparison with Static Analysis

Minamide [23] presents a static analysis for discovering HTML malformedness faults in PHP applications. His analysis tool approximates the string output of a program with a context-free grammar. His tool was able to discover unclosed tags in the subject programs by intersecting this grammar with the regular expression of matched pairs of delimiters (open/closed tags). Our analysis uses an HTML validator and covers the whole language standard. We performed our evaluation on a set of applications overlapping with Minamide's (webchess, faqforge, schoolmate).

However, Apollo cannot be applied to two of Minamide's subject programs (phpwims and timeclock) because the programs need to be executed in a Web-browser and the current implementation of Apollo does not support this mode of execution (see section 5.5).

For the three overlapping subject programs, Apollo is both more *effective* and more *efficient* than Minamide's tool. Apollo found 2.7 times as many HTML validation faults found by Minamide's tool(120 vs. 45). Faults found by Minamide's tool are not publicly available so we do not know whether Apollo discovered all faults that Minamide's tool discovered. However, Apollo found 83 execution faults, which are out of reach for Minamide's tool. Apollo is also more scalable—on schoolmate, the largest of the programs, Apollo found 104 faults in 10 minutes, while Minamide's tool found only 14 faults in 126 minutes. The time spent in Minamide's tool is due to constructing large automata and to the expensive algorithm for checking disjointness between regular expressions and context-free languages.

5.3.2 Path Constraint Minimization

We measure the effectiveness of the minimization algorithm of Section 3.3 via the reduction ratio between the size of the shortest original (un-minimized) path constraint and the minimized path constraint.

program	strategy	#inputs	line coverage %	execution crash	execution error	execution warning	HTML validation error	HTML validation warning	Total faults
faqforge	Randomized	1461	19.2	0	0	0	10	1	11
	Apollo	429	86.8	0	9	0	38	17	64
webchess	Randomized	1805	5.9	1	13	2	3	0	19
	Apollo	557	42.0	1	25	2	7	0	35
schoolmate	Randomized	1396	8.3	1	0	0	18	0	19
	Apollo	724	64.9	2	30	14	58	0	104
phpsysinfo	Randomized	406	21.3	0	5	3	2	0	10
	Apollo	143	56.2	0	5	4	2	0	11
Total	Randomized	5211	15.2	2	18	5	33	1	59
	Apollo	1853	58.0	3	69	20	105	17	214

Figure 7: Experimental results for 10-minute test generation runs. The table presents results for each subject program, and each strategy, separately. The **#inputs** column presents the number of inputs that each strategy created in the given time budget. The **coverage** column lists the line coverage achieved by the generated inputs. The **execution crashes**, **errors**, **warnings** and **HTML errors**, **warnings** columns list the number of faults in the respective categories. The **Total faults** columns sums up the number of discovered faults.

program	success rate%	path constraints orig. size	path constraints reduction	inputs orig. size	inputs reduction
faqforge	64	22.3	0.22	9.3	0.31
webchess	91	23.4	0.19	10.9	0.40
schoolmate	51	22.9	0.38	11.5	0.58
phpsysinfo	82	24.3	0.18	17.5	0.26

Figure 8: Results of minimization. The **success rate** indicates the percentage of faults whose exposing input was successfully minimized (i.e., the minimizer found a shorter exposing input). The **path constraint** columns list the statistics for path constraints, while the **inputs** list the statistics for inputs that correspond to those path constraints. The **orig. size** columns list the average size of original (un-minimized) path constraints and inputs. The size of a path constraint is the number of conjuncts. The size of an input is the number of key-value pairs in the input. The **reduction** columns list the ratio of minimized to unminimized size. The lower the ratio, the more successful the minimization.

Figure 8 tabulates the results. The results show that our input minimization technique effectively reduces the size of inputs by up to a factor of 0.18, for more than 50% of the faults.

5.4 Threats to Validity

Construct Validity. Why do we count malformed HTML as a defect in dynamically generated webpages? Does a webpage with malformed HTML pose a real problem or this is an artificial problem generated by the overly conservative specification of the HTML language? Although web browsers are resilient to malformed HTML, we encountered cases (in a different project) when malformed HTML crashed a widely popular Web browser. More importantly, even though a particular browser might tolerate malformed HTML, different browsers or different versions of the same browser may not display all information in the presence of malformed HTML. This becomes crucial for some websites, for example banking. Many informational and functional websites provide a button for verifying the validity of statically generated HTML. The challenges of dynamically generated webpages prevent the same institutions from validating the content.

Why do we use line coverage as a quality metric? We use line coverage only as a *secondary* metric, our *primary* metric being the number of faults found. The experimental results show that Apollo achieves significantly better results, in both metrics, than randomized testing and static analysis.

Why do, in addition to inputs exposing the failure, we present the user with minimizes path constraints? Although an input that corresponds to a longer path constraint still exposes the same failure, by removing superfluous information, Apollo may be able to better assist the programmer in pinpointing the location of the fault.

Internal Validity. Did Apollo discover real, unseeded, and unknown faults? Since we used subject projects developed by others, we could not influence the quality of the subject programs. Apollo does not search for known or seeded faults, but it finds *real* faults in real programs. For those subject programs that connect to a database, we populated the database with random records. The only thing that is "seeded" into the experiment is a username/password combination, so that Apollo can access the records stored in the database.

External Validity. Will our results generalize beyond the subject programs? We only used Apollo to find faults in four PHP projects. These may have serious quality problems, or be unrepresentative in other ways. Three of the subject programs are also used as subject programs by Minamide [23]. We chose the same programs to compare our results. We chose an additional subject program, phpsysinfo, since it is almost double the size of the largest subject that Minamide used. Additionally, phpsysinfo is a mature and active project in sourceforge. It is widely used, as witnessed by almost half a million downloads (Figure 6), and it is ranked in the top 0.5% projects on sourceforge (rank 997 of 176,575 projects as of 7 May 2008). Nevertheless, Apollo found faults in phpsysinfo.

Reliability. Are the results reproducible? The subject programs that we used are publicly available from sourceforge. The faults that we found are available for examination at `http://pag.csail.mit.edu/apollo`.

5.5 Limitations

Simulating user inputs based only on static information.

In the current strategy for simulating user input, Apollo instruments the PHP script statically, i.e., without executing the script. In general, the HTML output of a PHP script might contain buttons and arbitrary snippets of JavaScript code that are executed when the user presses the corresponding button. The actions that the JavaScript might perform are currently not analyzed by Apollo. For instance, the JavaScript code might pass specific arguments to the PHP script. As a result, Apollo might report false positives. Apollo might report a false positive if Apollo decides to execute a PHP script as a result of simulating a user pressing a button but the button is not visible. Apollo might also report false positive if it attempts to set an input parameter that would have been set by the JavaScript code. In our experiments Apollo did not report any false positives. However, we are currently exploring a way to handle simulated user input dynamically.

Limited tracking in native methods. Apollo has limited tracking of input parameters through PHP native methods. PHP native methods are implemented in C, which make it difficult to automatically track how input parameters are transformed into output parameters. We have modified the PHP interpreter to track parameters across a very small subset of the PHP native methods. Similarly to [32], we plan to create an external language to model the dependences between inputs and outputs for native methods to increase Apollo line coverage when native methods are executed.

Limited sources of input parameters. Apollo currently considers as parameters only inputs coming from the global arrays _POST, _GET and _REQUEST. Supporting other global parameters such as _ENV and _COOKIE is straightforward.

Running as a stand-alone application. For implementation convenience, Apollo currently invokes the PHP runtime as a stand-alone child process which means that certain functionality provided by Web server integration is limited. Running via a Web server is ongoing work.

6. Related Work

Godefroid *et al.* [15] present DART, a tool for finding combinations of input values and environment settings for C programs that trigger assertion failures and crashes when these programs are executed. DART combines random test generation with the use of symbolic reasoning to track of constraints for executed control flow paths. A constraint solver directs subsequent executions towards uncovered branches. Experimental results indicate that DART is highly effective at finding large numbers of faults in several C applications and frameworks, including important and previously unknown security vulnerabilities.

Subsequent work extends the original approach of combining concrete and symbolic executions to accomplish two primary goals: 1) improving scalability [1, 13, 14, 16, 22], and 2) improving execution coverage and fault detection capability through better support for pointers and arrays [5, 28], better search heuristics [16, 18, 21], or by encompassing wider domains such as database applications [11].

Godefroid [13] proposes a compositional approach to improve the scalability of DART. In this approach, summaries of lower level functions are computed dynamically when these functions are first encountered. The summaries are expressed as pre- and post-conditions of the function in terms of its inputs. Subsequent invocations of these lower level functions reuse the summary. Anand *et al.* [1] extend this compositional approach to be demand-driven to reduce the summary computation effort.

Majumdar and Xu [22] exploit the structure of the program input to improve scalability. In this approach, context-free grammars that represent the program inputs are abstracted to produce a symbolic grammar. This symbolic grammar reduces the number of input strings that need to be enumerated during the combined concrete and symbolic test generation phase. The approach is shown have better scalability than the *concolic* version [28] of the combined concrete and symbolic execution approach.

Majumdar and Sen [21] describe a hybrid concolic testing approach that exploits the capability of random testing to explore deeper and longer inputs to achieve better coverage. Hybrid concolic testing interleaves random testing until saturation with bounded exhaustive symbolic exploration. This approach doubles the coverage achieved by concolic testing alone. Inkumsah and Xie [18] combine evolutionary testing using genetic mutations with concolic testing to produce longer sequences of test inputs. SAGE [16] also uses improved heuristics, called *white-box fuzzing*, to achieve higher branch coverage faster.

Emmi *et al.* [11] extend concolic testing to encompass database applications. This approach enables insertion of database records to enable execution of program code that depends on embedded SQL queries. Their system uses a string constraint solver that can decide string equality, inequality, and membership in a regular language is used to facilitate the task.

Our work benefits from these extensions to the combined concrete and symbolic execution approach. However, our work differs from the prior work in several respects. Most importantly, our work goes beyond simple assertion failures and crashes by relying on an oracle (in the form of an HTML validator) to determine correctness, which means that our tool can handle situations where the program has functionally incorrect behavior without relying on programmer assertions. Cadar and Engler [4] also recognize the issue of functional correctness, but address it by using a separate implementation of the function being tested to compare outputs. This limits the approach to situations where a second implementation exists.

Our work also minimizes the constraints on the input parameters, which in turn results in shorter inputs inducing each failure, to help the developer pinpoint the cause of faults. Godefroid *et al.* [16] faced this challenge since their technique produces several distinct inputs which may expose the same fault at a particular code location. They addressed the issue by hashing all such inputs and returning one exemplar of failure inducing input to the developer. Our work addresses this issue as well as a different one: identifying the minimal set of program variables in an input that are essential to induce the failure. In this regard, our work is similar in spirit to *delta debugging* [6, 34] and its extension *hierarchical delta debugging* [24]. These approaches modify the failure inducing input directly, thus leading to a singular, minimal exemplar of such input. Our technique, on the other hand, modifies the set of constraints on failure inducing input. This enables us to provide minimal *patterns* of failure inducing inputs, further aiding the fault fixing activities. Moreover, since our technique is aware of the (partial) overlapping of different inputs, it is more efficient.

The language under consideration, PHP, is also quite different from what previous testing research focused on. PHP poses several new challenges such as the dynamic inclusion of files and function definitions that are statements. Existing techniques for fault detection in PHP applications use static analysis and target security vulnerabilities such as *SQL injection* or *cross-site scripting* attacks [20, 23, 31, 33]. In particular, Minamide [23] uses static string analysis and language transducers to model PHP string operations to generate *potential* HTML output—represented by a context-free grammar—from the Web application. This method can be used to generate HTML document instances of the resulting grammar and to validate them using an existing HTML validator. As a more complete alternative, Minamide proposes a *matching validation* which checks for containment of the generated context free grammar against a regular subset of the HTML specification. Unfortunately, this approach can only check for matching start and end tags in the HTML output, while our technique covers the entire HTML specification. Also, flow and context insensitive approximations in the static analysis techniques used in this method result in false positives, whereas our method reports only real faults.

Benedikt *et al.* [2] present a tool, VeriWeb, for automatically testing dynamic Webpages. They use a model checker to systematically explore all paths (up to a certain bound) that a user could navigate in a Web site. When the exploration encounters forms, VeriWeb uses *SmartProfiles* to collect values that should be provided as inputs to forms. Although VeriWeb can automatically fill in the forms, the tester needs to pre-populate the user profiles with values that a user would provide. In contrast, Apollo automatically discovers input values by looking at the branch conditions along an execution path. Also, Benedikt *et al.* do not report any faults found, while we report 214.

Dynamic analysis of string values generated by PHP Web applications has been considered in a *reactive* mode to prevent the execution of insidious commands (*intrusion prevention*) and to raise an alert (*intrusion detection*) [19, 26, 30]. To the best of our knowledge, our work is the first attempt at *proactive* fault detection in PHP Web applications using dynamic analysis.

Finally, our work is related to the growing body of work in *implementation based* (as opposed to *specification based* e.g., [27]) testing of Web applications. These works abstract the application behavior using a) client-side information such as user requests and corresponding application responses [9, 12], or b) server-side monitoring information such as user session data [10, 29], or c) static analysis of server-side implementation logic [17]. The approaches that use client-side information or server-side monitoring information are inherently incomplete, and the quality of generated abstractions depends on the quality of the tests run.

Halfond and Orso [17] use static analysis of the server-side implementation logic to extract Web application interface—a set of input parameters and their potential values. They implemented their technique for JavaScript. They obtained better code coverage with test cases based on the interface extracted using their technique as compared to the test cases based on the interface extracted using a conventional Web crawler. However, the resulting coverage may depend on the choices made by the test generator to combine parameter values—an exhaustive combination of values may be needed to maximize the code coverage. In contrast, our work uses dynamic analysis of server side implementation logic for fault detection and minimizes the number of inputs needed to maximize the coverage. Furthermore, we include results on fault detection capabilities of our technique. We implemented and evaluated (Section 5) a version of Halfond and Orso's technique for PHP. Compared to that re-implementation, Apollo achieved higher line coverage (58.0% vs. 15.2%) and found more faults (214 vs. 59).

7. Conclusions

We have presented a technique for finding faults in PHP Web applications that is based on combined concrete and symbolic execution. The work is novel in several respects. First, the technique not only detects run-time errors but also uses an HTML validator as an oracle to determine situations where malformed HTML is created. Second, we address a number of PHP-specific issues, such as the simulation of interactive user input that occurs when user interface elements on generated HTML pages are activated, resulting in the execution of additional PHP scripts. Third, we perform an automated analysis to minimize the size of failure-inducing inputs.

We created a tool, Apollo, that implements the analysis. We evaluated Apollo on four open-source PHP Web applications. Apollo's test generation strategy achieves over 50% line coverage. Apollo found a total of 214 faults in these applications: 92 execution problems and 122 cases of malformed HTML. Finally, Apollo also minimizes the size of failure-inducing inputs: the minimized inputs are up to 5.3× smaller than the unminimized ones.

8. References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, 2008.

[2] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic Web sites. In *WWW*, 2002.

[3] C. Braband, A. Moller, and M. Schwartzbach. Static validation of dynamically generated HTML. In *PASTE*, 2001.

[4] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[7] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[8] D. Dean and D. Wagner. Intrusion detection via static analysis. In *Symposium on Research in Security and Privacy*, May 2001.

[9] S. Elbaum, K.-R. Chilakamarri, M. Fisher, and G. Rothermel. Web application characterization through directed requests. In *WODA*, 2006.

[10] S. Elbaum, S. Karre, G. Rothermel, and M. Fisher. Leveraging user-session data to support Web application testing. *IEEE Trans. Softw. Eng.*, 31(3), 2005.

[11] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[12] M. Fisher, S. G. Elbaum, and G. Rothermel. Dynamic characterization of Web application interfaces. In *FASE*, 2007.

[13] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[14] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[16] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[17] W. G. J. Halfond and A. Orso. Improving test case generation for Web applications using automated interface discovery. In *ESEC-FSE*, 2007.

[18] K. Inkumsah and T. Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE*, 2007.

[19] M. Johns and C. Beyerlein. SMask: preventing injection attacks in Web applications by approximating automatic data/code separation. In *SAC*, 2007.

[20] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Security and Privacy*, 2006.

[21] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[22] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[23] Y. Minamide. Static approximation of dynamically generated Web pages. In *WWW*, 2005.

[24] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *ICSE*, 2006.

[25] R. O'Callahan. Personal communication, 2008.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2005.

[27] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *ICSE*, 2001.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.

[29] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for Web applications. In *ASE*, 2005.

[30] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *POPL*, 2006.

[31] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

[32] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, 2008.

[33] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*, 2006.

[34] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, 1999.