MIT-CSAIL-TR-2009-010 March 26, 2009

Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit State Model Checking

Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst

Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit State Model Checking

Shay Artzi[†] Adam Kieżun[†] Julian Dolby[‡]
Frank Tip[‡] Danny Dig[*] Amit Paradkar[‡] Michael D. Ernst[⋆]
[†]MIT CSAIL, {artzi,akiezun}@csail.mit.edu
[‡]IBM T.J. Watson Research Center, {dolby,ftip,paradkar}@us.ibm.com
[*]University of Illinois at Urbana-Champaign, dig@illinois.edu
[⋆]University of Washington,mernst@cs.washington.edu

Abstract—
Web script crashes and malformed dynamically-generated web pages are common errors, and they seriously impact the usability of web applications. Current tools for web-page validation cannot handle the dynamically generated pages that are ubiquitous on today's Internet. We present a dynamic test generation technique for the domain of dynamic web applications. The technique utilizes both combined concrete and symbolic execution and explicit-state model checking. The technique generates tests automatically, runs the tests capturing logical constraints on inputs, and minimizes the conditions on the inputs to failing tests, so that the resulting bug reports are small and useful in finding and fixing the underlying faults.

Our tool Apollo implements the technique for the PHP programming language. Apollo generates test inputs for a 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 302 faults in 6 PHP web applications.

General Terms Reliability, Verification

Index Terms—Software Testing, Web Applications, Dynamic Analysis, PHP

1 Introduction

Dynamic test generation tools, such as DART [18], Cute [36], and EXE [7], generate tests 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, which pose special challenges due to the dynamism of the programming languages, the use of implicit input parameters, their use of persistent state, and their complex patterns of user interaction.

This paper extends dynamic test generation to the domain of web applications that dynamically create web (HTML) pages during execution, which are typically presented to the user in a browser. Apollo applies these techniques in the context of the scripting language PHP, one of the most popular languages for web programming. According to the internet research service, Netcraft[1], PHP powered 21 million domains as of April 2007, including large, well-known websites such as Wikipedia and WordPress.

Our goal is to find two kinds of failures in web applications: *execution failures* that are manifested as crashes or warnings during program execution, and *HTML failures* that occur when the application generates malformed HTML. As an example, execution failures may occur when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output contains an error message and execution of the application may be halted, depending on the severity of the failure. HTML failures occur when output is generated that is not syntactically well-formed HTML (e.g., when an opening tag is not accompanied by a matching closing tag). Although web browsers are designed to tolerate some degree of malformedness in HTML, several kinds of problems may occur. First and most serious is that browsers' attempts to compensate for malformed web pages may lead to crashes and security vulnerabilities[2]. Second, standard HTML renders faster[3]. Third, malformed HTML is less portable across browsers and is vulnerable to breaking or looking strange when displayed by browser versions on which it is not tested. Fourth, a browser might succeed in displaying only part of a malformed webpage, while silently discarding important information. Fifth, search engines may have trouble indexing malformed pages [45].

Web developers widely recognize the importance of creating legal HTML. Many websites are checked using HTML validators[4]. However, HTML validators can only point out problems in HTML pages, and are by themselves incapable of finding faults in applications that *generate* HTML pages. Checking *dynamic* web applications (i.e., applications that generate pages during execution) requires checking that the

1. See http://news.netcraft.com/.

2. See bug reports 269095, 320459, and 328937 at https://bugzilla.mozilla.org/show_bug.cgi?

3. See http://weblogs.mozillazine.org/hyatt/archives/2003_03.html#002904. According to a Mozilla developer, one reason why malformed HTML renders slower is that "improper tag nesting [...] triggers residual style handling to try to produce the expected visual result, which can be very expensive" [33].

4. http://validator.w3.org, http://www.htmlhelp.com/tools/validator

application creates a valid HTML page on *every* possible execution path. In practice, even professionally developed and thoroughly tested applications often contain multiple faults (see Section 6).

There are two general approaches to finding faults in web applications: static analysis and dynamic analysis (testing). In the context of web applications, static approaches have limited potential because (i) web applications are often written in dynamic scripting languages that enable on-the-fly creation of code, and (ii) control in a web application typically flows via the generated HTML text (e.g., buttons and menus that require user interaction to execute), rather than solely via the analyzed code. Both of these issues pose significant challenges to approaches based on static analysis. Testing of dynamic web applications is also challenging, because the input space is large and applications typically require multiple user interactions. The state-of-the-practice in validation for web-standard compliance of real web applications involves the use of programs such as HTML Kit[5] that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no automated tool for the validation of web applications that dynamically generate HTML pages.

This paper presents an automated technique for finding failures in HTML-generating web applications. Our technique is based on dynamic test generation, using combined concrete and symbolic (concolic) execution and constraint solving [7], [18], [36]. We created a tool, Apollo, that implements our technique in the context of the publicly available PHP interpreter.

Apollo first executes the web application under test with an empty input. During each execution, Apollo monitors the program to record the dependence of control-flow on input. Additionally, for each execution Apollo determines whether execution failures or HTML failures occur (for HTML failures, an HTML validator is used as an oracle). Apollo automatically and iteratively creates new inputs using the recorded dependence to create inputs that exercise different control flow. Most previous approaches for concolic execution only detect “standard errors” such as crashes and assertion failures. Our approach also detects such standard errors, but is to our knowledge the first to use an oracle to detect specification violations in the application’s output.

Another novelty in our work is the inference of input parameters, which are not manifested in the source code, but which are interactively supplied by the user (e.g., by clicking buttons in generated HTML pages). The desired behavior of a PHP application is usually achieved by a series of interactions between the user and the server (e.g., a minimum of five user actions are needed from opening the main Amazon page to buying a book). We handle this problem by enhancing the combined concrete and symbolic execution technique with explicit-state model checking based on automatic dynamic simulation of user interactions. In order to simulate user interaction, Apollo stores the state of the environment (database, sessions, cookies) after each execution, analyzes the output of the execution to detect the possible user options that are available, and restores the environment state before executing a new script based on a detected user option.

5. http://www.htmlkit.com

Techniques based on combined concrete and symbolic executions [7], [18], [36] may create multiple inputs that expose the same fault. 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 [8]. 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.

The contributions of this paper are the following:

- We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution [7], [18], [36], to the domain of PHP web applications. This involved the following innovations: (i) using an HTML verifier as an oracle, (ii) inferring input parameters that are not manifested in the source code, (iii) dealing with datatypes and operations specific to the PHP language, (iv) tracking the use of persistent state and how input flows through it, and (v) 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 6 real web applications and comparing the results with random testing. We show that dynamic test generation can be effective when adapted to the domain of web applications written in PHP: Apollo identified 302 faults while achieving line coverage of 50.2%.
- We present a detailed classification of the faults found by Apollo.

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 a simplified version of the algorithm and illustrates it on an example program. Section 4 presents the complete algorithm handling stateful execution with the simulation of interactive user inputs, and illustrates it on an example program. Section 5 discusses our Apollo implementation. Section 6 presents our experimental evaluation of Apollo on open-source web applications. Section 7 gives an overview of related work, and Section 8 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

```
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 ?>
```

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

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[6], 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 used in the `switch` statement on line 18, which presents the login screen or one of the teacher/student screens.

2.3 Failures in PHP Programs

Our technique targets two types of failures that can be automatically identified during the execution of PHP web applications. First, *execution failures* may be caused by a missing included file, an incorrect MySQL query, or by an uncaught exception. Such failures are easily identified as the PHP interpreter generates an error message and halts execution. Less serious execution failures, such as those caused by the use of deprecated language constructs, produce obtrusive error messages but do not halt execution. Second, *HTML failures* involve situations in which the generated HTML page is not syntactically correct according to an HTML validator. Section 1 discussed several negative consequences of malformed HTML.

As an example, the program of Figure 1 contains three faults, which cause the following failures when the program is executed:

6. http://sourceforge.net/projects/schoolmate

1) Executing the program results in 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. In particular, 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 in one of the PHP applications we studied. The second failure is caused by 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 [7], [18], [19], [36] sometimes referred to as concolic testing. For expository purposes, we will present the algorithm in two steps. First, this section presents a simplified version of the algorithm that does simulate user inputs or keep track of persistent session state. We will demonstrate this simplified algorithm on the example of Figure 1. Then, Section 4 presents a generalized version of the algorithm that handles user input simulation and stateful executions, and illustrates it on a more complex example.

The basic idea behind the technique is to execute an application on some 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 consider failures other than execution failures by using an oracle to determine whether or not program output is correct. In particular, we use 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), `header` for redirection of execution, 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.

3.1 Algorithm

Figure 2 shows pseudo-code for our algorithm. The inputs to the algorithm are: a program $\mathcal{P}$, an oracle for the output $\mathcal{O}$, and an initial state of the environment $\mathcal{S}_0$. The output of the algorithm is a set of bug reports $\mathcal{B}$ for the program $\mathcal{P}$, according to $\mathcal{O}$. The report consist of a single failure, defined by the error message and the set of statements that is related to the failure. In addition, the report contains 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 $\mathcal{O}$, Initial state $\mathcal{S}_0$
   result    : Bug reports $\mathcal{B}$;
   $\mathcal{B}$ : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
1  $\mathcal{B} := \emptyset$;
2  toExplore := emptyQueue();
3  enqueue(toExplore, ⟨emptyPathConstraint(), emptyInput⟩);
4  while not empty(toExplore) and not timeExpired() do
5      ⟨pathConstraint, input⟩ := dequeue(toExplore);
6      output := executeConcrete($\mathcal{S}_0$, $\mathcal{P}$, input);
7      foreach f in getFailures($\mathcal{O}$, output) do
8          merge ⟨f, pathConstraint, input⟩ into $\mathcal{B}$;
9      newConfigs := getConfigs(input);
10     foreach ⟨$pathConstraint_i$, $input_i$⟩ ∈ newConfigs do
11         enqueue(toExplore, ⟨$pathConstraint_i$, $input_i$⟩);
12 return $\mathcal{B}$;

13 Subroutine getConfigs(input):
14 configs := $\emptyset$;
15 $c_1 \wedge \ldots \wedge c_n$ := executeSymbolic($\mathcal{S}_0$, $\mathcal{P}$, input);
16 foreach $i = 1,\ldots,n$ do
17     newPC := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
18     input := solve(newPC);
19     if input $\neq \bot$ then
20         enqueue(configs, ⟨newPC, input⟩);
21 return configs;
```

Fig. 2: The failure detection algorithm. 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 *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\bot$ if no satisfying input exists. The *merge* auxiliary function merges the pair of pathConstraint and input for an already detected failure into the bug report for that failure.

The algorithm uses a queue of configurations. Each configuration is a pair of a path constraint and an input. A *path constraint* is a conjunction of conditions on the program's input parameters. The queue is initialized with the empty path constraint and the empty input (line 3). The program is executed concretely on the input (line 6) and tested for failures by the oracle (line 7). Then, the path constraint and input for each detected failure are merged into the corresponding bug report (lines 7–8).

Next, the algorithm uses a subroutine, newConfigs, to find new configurations. First, the program is executed symbolically on the same input (line 15). The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, that is satisfied by the path that was just executed from entry to exit of the whole program. The subroutine then creates new inputs by solving modified versions of the path constraint (lines 16–20), as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 17). 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, presumable covering new code. The algorithm uses a constraint solver to find a concrete input for each path constraint (line 18).

3.2 Example

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

Iteration 1. The first input to the program is the empty input, which is the result of solving the empty path constraint. During the execution of the program on the empty input, the condition on line 6 evaluates to `true`, and `page` is set to `0`. The condition on line 10 evaluates to `false`. The condition on line 16 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:

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

The algorithm now enters the **foreach** loop on line 16 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:

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

Iteration 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): $\texttt{page2} \leftarrow \texttt{0}, \texttt{login} \leftarrow \texttt{1}$.

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.

3.3 Path Constraint Minimization

The failure detection algorithm (Figure 2) returns bug reports. Each bug report contains a set of path constraints, and a set of inputs exposing the failure. Previous dynamic test generation tools [7], [18], [36] presented the whole input (i.e., many $\langle inputParameter, value \rangle$ pairs) 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).

```
   parameters: Program $\mathcal{P}$, oracle $O$, bug report $b$
   result    : Short path constraint that exposes $b.failure$
 1 $c_1 \land \ldots \land c_n := intersect(b.pathConstraints)$;
 2 $pc := true$;
 3 foreach $i = 1, \ldots, n$ do
 4     $pc_i := c_1 \land \ldots c_{i-1} \land c_{i+1} \land \ldots c_n$;
 5     if $!exposesFailures(pc_i)$ then
 6         $pc := pc \land c_i$;
 7 if $exposesFailures(pc)$ then
 8     return $pc$;
 9 return $shortest(b.pathConstraints)$;

10 Subroutine $exposesFailure(pc)$:
11 $input_{pc} := solve(pc)$;
12 if $input_{pc} \neq \bot$ then
13     $output_{pc} := executeConcrete(\mathcal{P}, input_{pc})$;
14     $failures_{pc} := getFailures(O, output_{pc})$;
15     return $b.failure \in failures_{pc}$;
16 return $false$;
```

Fig. 3: The path constraint minimization algorithm. The method *intersect* returns the set of conjuncts 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.

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 conjunct at a time (lines 3-6). If one of these shorter path constraints does not expose *b.failure*, then the removed conjunct is required for exposing *b.failure*. The set of all such required conjuncts determines the minimized path constraint. 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 6.3.2).

Our minimizer differs from *input* minimization techniques, such as delta debugging [8], [44], 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 $\texttt{page2} \neq \texttt{1337}$ 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.

$$\begin{array}{ll} \mathit{NotSet}(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} = \texttt{1} & (a) \\ \mathit{Set}(\texttt{page}) \wedge \texttt{page} = \texttt{0} \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} = \texttt{1} & (b) \end{array}$$

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

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

Then, 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 determines this by executing the program on the input (line 14 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 the minimal failure-inducing input, namely `login ← 1`.

4 Combined Concrete and Symbolic Execution with Explicit-State Model Checking

A typical PHP web application is a client-server application in which data and control flows interactively between a server that runs PHP scripts and a client, which is usually a web browser. The PHP scripts that run on the server generate HTML that includes interactive user input widgets such as buttons and menu items that, when selected by the user, invoke other PHP scripts. When these other PHP scripts are invoked, they are passed a combination of user input and constant values taken from the generated HTML. Modeling such user input is important, because coverage of the application will typically remain very low otherwise.

In Section 3, we described how to find failures in PHP web applications by adapting an existing test generation approach to consider language constructs that are specific to PHP, by using an oracle to validate the output, and by supporting database interaction. However, we did not yet supply a solution for handling user input options that are created dynamically by a web application, which includes keeping track of parameters that are transferred from one script to the next—either by persisting them in the environment, or by sending them as part of the call.

To handle this problem, Apollo implements a form of explicit-state software model checking. That is, Apollo systematically explores the state space of the system, i.e., the program under test. The algorithm in Section 3 always restarts the execution from the same initial state, and discards the state reached at the end of each execution. Thus, the algorithm reaches only 1-level deep into the application, where each level corresponds to a cycle of: a PHP script that generates an HTML form that the user interacts with to invoke the next PHP script. In contrast, the algorithm presented in this section remembers and restores the state between executions of PHP scripts. This technique, known as state matching, is widely known in model checking [22], [39] and implemented in tools such as SPIN [11] and JavaPathFinder [21]. To our knowledge, we are the first to implement state matching in the context of web applications and PHP.

4.1 Interactive User Simulation Example

Figure 4 shows an example of a PHP application that is designed to illustrate the particular complexities of finding faults in an interactive web applications. In particular, the figure shows: an `index.php` top-level script that contains static HTML in Figure 4(a), a generic login script `login.php` in Figure 4(c), and a skeleton of a data display script `view.php` in Figure 4(d). The PHP scripts in Figure 4 rely on a shared include file `constants.php` that defines some standard constants, which is shown in in Figure 4(b). Note that the code in Figure 4 is an ad-hoc mixture of PHP statements and HTML fragments. The PHP code is delimited by `<?php` and `?>` tokens (For instance lines 44 and 69 in Figure 4(c)). The use of HTML in the middle of PHP indicates that HTML is generated as if it were the argument of a print statement. The `dirname` function—which returns the directory component of a filename—is used in the `require` statements, as an example of including a file whose name is computed at run-time.

These PHP scripts are part of the client-server work flow in a web application: the user first sees the `index.php` page of Figure 4(a) and enters credentials. The user-input credentials are processed by the script in Figure 4(c), which generates a response page that allows the user to enter further input—a topic—that in turn entails further processing by the script in Figure 4(d). Note that the user name and password that are entered by the user during the execution of `login.php` are stored in special locations `$_SESSION[ $userTag ]` and `$_SESSION[ $pwTag ]`, respectively. Moreover, if the user is the administrator, this fact is recorded similarly, in `$_SESSION[ $typeTag ]`. These locations illustrate how PHP handles *session state*, which is data that persists from one page to another, typically for a particular interaction by a particular user. Thus, the updates to `_SESSION` in Figure 4(c) will be seen (as the SESSION information is saved and read locally on the server) by the code in Figure 4(d) when the user follows the link to `view.php` in the HTML page that is returned by `login.php`. The `view.php` script uses this session information to verify the username/password in line 46.

Our example program contains an error in the HTML produced for the administrative details: the `H2` tag that is opened on line 62 of Figure 4(d) is not closed. While this fault itself is trivial, finding it is not. Assume that testing starts (as an ordinary user would) by entering credentials to the script in Figure 4(c). A tester must then discover that setting `$user` to the value '`admin`' results in the selection of a different branch that records the user type '`admin`' in the session state (see lines 34–36 in `login.php`). After that, a tester would have to enter a topic in the form generated by the login script, and would then proceed to Figure 4(d) with the appropriate session

```
1  <html>
2  <head>Login</head>
3  <body>
4    <form name="login" action="exampleLogin.php">
5      <input type="text" name="user"/>
6      <input type="password" name="pw"/>
7    </form>
8  </body>
9  </html>
```

(a) `index.php`

```
10 <?php
11   userTag = 'user'
12   pwTag = 'pw';
13   typeTag = 'type';
14 ?>
```

(b) `constants.php`

```
15 <HTML>
16 <?php
17   require( dirname(__FILENAME__).'/includes/constants.php');
18
19   $user = $_REQUEST[ 'user' ];
20   $pw = $_REQUEST[ 'pw' ];
21
22   if (check_password($user, $pw) {
23     print "<HEAD>Login Successful</HEAD>\n";
24
25   $_SESSION[ $userTag] = $user;
26   $_SESSION[ $pwTag ] = $pw;
27 ?>
28   <BODY>
29     <FORM action="view.php">
30       <INPUT TYPE="text" NAME="topic"/>
31     </FORM>
32   </BODY>
33 <?php
34   if ($user == 'admin') {
35     $_SESSION[ $typeTag ] = 'admin';
36   }
37   else {
38     print "<HEAD>Login Failed</HEAD>\n";
39   }
40 ?>
41 </HTML>
```

(c) `login.php`

```
42 <HTML>
43 <HEAD>Topic View</HEAD>
44 <?php
45   print "<BODY>\n";
46   if(check_password($_SESSION[$userTag], $_SESSION[$pwTag]) {
47     require( dirname(__FILENAME__).'/includes/constants.php');
48
49     $type = $_SESSION[ $typeTag ];
50     $topic = $_REQUEST[ 'topic' ];
51
52     if ($type == 'admin') {
53       print "<H1>Admin ";
54     } else {
55       print "<H1>Normal ";
56     }
57     print "View of $topic</H1>\n";
58
59     /* code to print topic view... */
60
61     if ($type == 'admin') {
62       print "<H2>Administrative Details\n";
63     /* code to print admin details... */
64     }
65   } else {
66       print "Please Log in\n";
67   }
68   print "</BODY>\n";
69 ?>
70 </HTML>
```

(d) `view.php`

Fig. 4: Example PHP web application.

state, which will finally generate HTML exhibiting the fault as is shown in Figure 5(a). Thus, finding the fault requires a careful selection of inputs to a series of interactive scripts, as well as making sure updates to the session state during the execution of these scripts are preserved (I.e., making sure that the execution of the different script happen during the same session).

4.2 Algorithm

Figure 6 shows pseudo-code for the algorithm, which extends the algorithm in Figure 2 with explicit-state model checking to handle the complexity of simulating user inputs. The algorithm tracks the state of the environment, and automatically discovers additional configurations based on an analysis of the output for available user options. In particular, the algorithm (i) tracks changes to the state of the environment (i.e., session state, cookies, and the database) and (ii) performs an "on the fly" analysis of the output produced by the program to determine what user options it contains, with their associated PHP scripts. By determining the state of the environment as it exists when an HTML page is produced, the algorithm can determine the environment in which additional scripts are executed as a result of user interaction. This is important because a script is much more likely to perform complex behavior when executed in the correct context (environment). For example, if the web application does not record in the environment that a user is logged in, most subsequent calls will terminate quickly (e.g., when the condition in line 46 of Figure 4(d) is false) and will not present useful information. For simplicity, the algorithm implicitly handles the fact that there are possibly multiple entry points into a PHP program. Thus, an input will contain the script to execute in addition to the values of the parameters. For instance, the first call might be to index.php script, while subsequent calls can execute other scripts.

There are four differences (underlined in the figure) with the simplified algorithm that was previously shown in Figure 2.

1) A configuration contains an explicit state of the environment (before the only state that was used was the initial state $\mathcal{S}_0$) in addition to the path constraint and the input (line 3).
2) Before the program is executed, the algorithm (method `executeConcrete`) will restore the environment to the state given in the configuration (line 7), and will return the new state of the environment after the execution.
3) When the `getConfigs` subroutine is executed to find new configurations, it analyzes the output (the actual mechanism of the analysis is explained and demonstrated in Section 5.3) to find new possible transitions from the new environment state (lines 24—26). Each transition is expressed as a pair of a path constraint and an input.
4) The algorithm uses a set of configurations that are already in the queue (line 14) and it performs state matching, in order to only explore new configurations (line 11).

```
1 <HTML>
2 <HEAD>Topic View</HEAD>
3 <BODY>
4 <H1>Admin View of A topic</H1>
 ...
5 <H2>Administrative Details
 ...
6 </BODY>
7 </HTML>
```

(a) HTML output

HTML line	*PHP lines in 4(d)*
1	42
2	43
3	45
4	53, 57
5	62
6	68
7	70

(b) output mapping

```
Error at line 6, character 7:  end tag for "H2" omitted; possible causes include a missing
end tag, improper nesting of elements, or use of an element where it is not allowed
Line 5, character 1:  start tag was here
```

(c) Output of WDG Validator

Fig. 5: **(a)** HTML produced by the script of Figure 4(d). **(b)** Output mapping constructed during execution. **(c)** Part of output of WDG Validator on the HTML of Figure 5(a).

```
parameters: Program $\mathcal{P}$, oracle $\mathcal{O}$, Initial state $\mathcal{S}_0$
result    : Bug reports $\mathcal{B}$;
$\mathcal{B}$ : $setOf(\langle$failure, $setOf$(pathConstraint), $setOf$(input)$\rangle)$
1  $\mathcal{B} := \emptyset$;
2  $toExplore := emptyQueue()$;
3  $enqueue(toExplore, \langle emptyPC(), emptyInput(), \underline{\mathcal{S}_0}\rangle)$;
4  $\underline{visited := \{\langle emptyPathConstraint(), emptyInput(), \mathcal{S}_0\rangle\}}$;
5  while not empty(toExplore) and not timeExpired() do
6      $\langle pathConstraint, input, \underline{\mathcal{S}_{start}}\rangle := dequeue(toExplore)$;
7      $\underline{\langle output, \mathcal{S}_{end}\rangle} := executeConcrete(\underline{\mathcal{S}_{start}}, \mathcal{P}, input)$;
8      foreach $f$ in $getFailures(\mathcal{O}, output)$ do
9          merge $\langle f, pathConstraint, input\rangle$ into $\mathcal{B}$;
10     $newConfigs := getConfigs(input, \underline{output, \mathcal{S}_{start}, \mathcal{S}_{end}})$;
11     $\underline{newConfigs := newConfigs - visited}$;
12     foreach $\langle pathConstraint_i, input_i, \underline{\mathcal{S}_i}\rangle \in newConfigs$ do
13         $enqueue(toExplore, \langle pathConstraint_i, input_i, \underline{\mathcal{S}_i}\rangle)$;
14         $\underline{visited := visited \cup \langle \mathcal{S}_i, input_i\rangle}$;
15 return $\mathcal{B}$;

16 Subroutine $getConfigs(input, \underline{output, \mathcal{S}_{start}, \mathcal{S}_{end}})$:
17 $configs := \emptyset$;
18 $c_1 \wedge \ldots \wedge c_n := executeSymbolic(\underline{\mathcal{S}_{start}}, \mathcal{P}, input)$;
19 foreach $i = 1,\ldots,n$ do
20     $newPC := c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
21     $input := solve(pathConstraint)$;
22     if $input \neq \bot$ then
23         $enqueue(configs, \langle newPC, input, \underline{\mathcal{S}_{start}}\rangle)$;
24 $\underline{\textbf{foreach } \langle newInput_i, newPC_i\rangle \in analyzeOutput(output) \textbf{ do}}$
25     $\underline{\textbf{if } newInput \neq \bot \textbf{ then}}$
26         $\underline{configs := configs \cup \langle newPC_i, newInput_i, \mathcal{S}_{end}\rangle}$;
27 return $configs$;
```

Fig. 6: The failure detection algorithm. The output of the algorithm is a set of bug reports, each reports a failure and the set of tests exposing that failure. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\bot$ if no satisfying input exists. The *merge* auxiliary function merges the pair of pathConstraint and input for an already detected failure into the bug report for that failure. The *analyzeOutput* auxiliary function performs an analysis of the output to extract possible transitions from the current environment state.

4.3 Example

We will now illustrate the algorithm of Figure 6 using the example application of Figure 4. The inputs to the algorithm are: $\mathcal{P}$ is the code from Figure 4, the initial state of the environment is empty, the first script to execute is the script in Figure 4(a), and $\mathcal{O}$ is the WDG HTML validator[7]. The algorithm begins on line 3 by initializing the work queue with one item: an empty input to the script of Figure 4(a) with an empty path constraint and an empty initial environment.

iteration 1. The first iteration of the outer loop (lines 5–14) removes that item from the queue (line 6), restores the empty initial state, and executes the script (line 7).

No failures are observed. The call to *executeSymbolic* on line 18 returns an empty path constraint, so the function *analyzeOutput* on line 24 is executed next, and returns one user option; $\langle$`login.php`$, \emptyset, \emptyset\rangle$ for executing `login.php` with no input, and the empty state. This configuration is added to the queue (line 13) since it was not seen before.

iteration 2-5. The next iteration of the top-level loop dequeues the new work item, and executes `login.php` with empty input, and empty state. No failures are found. The call to *executeSymbolic* in line 18 returns a path constraint `user` $\neq$ `admin` $\wedge$ `user` $\neq$ `reg`, indicating that the call to `check_password` on line 22 in Figure 4(c) returned false[8]. Given this, the loop at lines 19–23 will generate several new work items for the same script with the following path constraints: `user` $\neq$ `admin` $\wedge$ `user` = `reg`, and `user` = `admin` which are obtained by negating the previous path constraint. The loop on lines 24—26 is not entered, because no user input options are found. After several similar iterations, two inputs are discovered: `user` = `admin`$\wedge$`pw` = `admin`, and `user` = `reg`$\wedge$`pw` = `reg`. These corresponds to alternate control flows in which the `check_password` test succeeds.

iteration 6-7. The next iteration of the top-level loop dequeues an item that allows the `check_password` call to succeed (assume it selected `user` = `reg`...). Once again, no failures are observed, but now the session state with *user* and *pw* set is recorded at line 7. Also, this time *analyzeOutput* (line 24) finds the link to the script in Figure 4(d), and so the loop at lines 24—26 adds one item to the queue, executing `view.php` with the current session state.

7. http://htmlhelp.com/tools/validator/
8. For simplicity, we omit the details of this function. It compares the user name and password to some constants '`admin`' and '`reg`'.

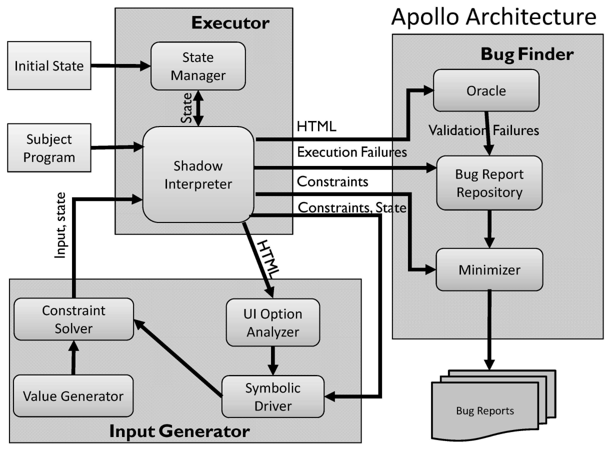

Fig. 7: The architecture of Apollo.

The next iteration of the top-level loop dequeues one work item. Assume that it takes the last one described above. Thus, it executes the script in Figure 4(d) with a session that defines *user* and *pw* but not *type*. Hence, it produces an execution with no errors.

iteration 8-9. The next loop iteration takes that last work item, containing a user and password pair for which the call to `check_password` succeeds, with the user name as '`admin`'. Once again, no failures occur, but now the session state with *user*, *pw* and *type* set is recorded at line 7. This time, there are no new inputs to be derived from the path constraint, since all prefixes have been covered already. Once again, parsing the output finds the link to the script in Figure 4(d) and adds a work item to the queue, but with a different session state (in this case, the session state also includes a value for *type*). The resulting execution of the script in Figure 4(d) with the session state that includes *type* results in an HTML failure.

5 Implementation

We created a tool called Apollo that implements our technique for PHP. Apollo consists of three major components, **Executor**, **Bug Finder**, and **Input Generator** illustrated in Figure 7. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation.

The inputs to Apollo are the program under test and an initial value for the environment. The environment will usually consist of a database with some values, and additional information about username/password pairs for the database. 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.

The **Executor** is responsible for executing a PHP script with a given input in a given state. The executor contains two sub-components:

- The **Shadow Interpreter** is a PHP interpreter that we have modified to propagate and record path constraints and positional information associated with output. This positional information is used to determine which failures are likely to be symptoms of the same fault.
- The **State Manager** restores the given state of the environment (database, session, cookies) before the execution, and stores the new environment after the execution.

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 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** implements the algorithm described in Figure 6. The Input Generator contains the following sub-components:

- The **UI Option Analyzer** analyzes the HTML output of each execution to convert the interactive user options into new inputs to execute.
- The **Symbolic Driver** generates new path constraints from the constraints found during the 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.

5.1 Executor

We modified the Zend PHP interpreter 5.2.2[9] to produce symbolic path constraints for the executed program, using the "shadow interpreter" approach [9]. The shadow interpreter performs the regular (concrete) program execution using the concrete values, and simultaneously performs symbolic execution. Creating the shadow interpreter required five alterations to the PHP runtime:

1) **Associating Symbolic Parameters with Values**
   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,

9. http://www.php.net/

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 [7], [18], [19], [36], our interpreter does not associate complex symbolic expressions with all runtime values, but only symbolic variables, which exist only for input-derived values. This design keeps the constraint solver simple and reduces the performance overhead. As our results (Section 6) indicate, this lightweight approach is sufficient for the analyzed PHP programs.

2) **Storing Constraints at Branch Points**
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) **Handling PHP Native Functions**
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.

4) **Propagating Inputs through Sessions and Cookies**
The use of session state allows a PHP application to store user-supplied information on the server for retrieval by other scripts. We enhanced the PHP interpreter to record when input parameters are stored in session state. This enables Apollo to track constraints on input parameters in all scripts that use them.

5) **Web Server Integration**
Dynamic web applications often depend on information supplied by a web-server (such as Apache), and some PHP constructs are simply ignored by the command line interpreter (e.g., *header*). In order to allow Apollo to analyze more PHP code, Apollo supports execution through the Apache web-server in addition to the stand-alone command line executor. A developer can use Apollo to silently analyze the execution and record any failure found while manually using the subject program on an Apache server.

The modified interpreter performs symbolic execution along with concrete execution, i.e., every variable during 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[10]. 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.

State Manager. PHP applications make use of persistent state such as the database, session information, and cookies. The State Manager is in charge of (i) restoring the environment prior to each execution, and (ii) storing the new environment after each execution.

5.2 Bug Finder

The bug finder is in charge of transforming the results of the executed inputs into bug reports. Below is a detailed description of the components of the bug finder.

Bug Report Repository This 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 execution failures, and validator message for HTML failures), and the PHP statement generating the problematic HTML fragments identified 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, (error message and statement involved in the failure) and the sets of path constraints and inputs exposing failures with the same characteristics.

Oracle. PHP Web applications output HTML/XHTML. 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[11] and the online W3C markup validation service[12]. Both oracles identified the same HTML failures. Our experiments use the faster WDG validator.

10. 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.
11. http://htmlhelp.com/tools/validator/offline
12. http://validator.w3.org

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.

5.3 Input Generator

UI Option Analyzer

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. There are two challenges involved in dealing with such interactive applications. First, we need to analyze the HTML output to find the referenced scripts, and the different values that can be supplied as parameters. Second, Apollo needs to be able to follow input parameters through the shared global information (database, the `session`, and the `cookie` mechanisms)

Apollo approach to the above challenges is to simulate user interaction by analyzing the dynamically created HTML output, and tracking the symbolic parameters through the environment (with the exception of the database). Apollo automatically extracts the available user options from the HTML output. Each option contains the script to execute, along with any parameters (with default values if supplied) for that script. Apollo also analyzes recursive static HTML documents that can be called from the dynamic HTML output, i.e. Apollo traverses hyperlinks in the generated dynamic HTML that link to other HTML documents on the same site.

Since additional code on the client side (for instance, Java script) 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.

For example after analyzing the output of the program of Figure 8, the UI Option Analyzer will return the following two options:

1) Script: “mainmenu.php”
   PathConstraint: `txtNick` = ”*Admin*” ∧ Exist (`pwdPassword`)
2) Script: “newuser.php”
   PathConstraint: ∅

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 5), 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 [7] and SAGE [19]. 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.

```
<?php
   echo "<h2>WebChess ".$Version." Login"</h2>;
?>
<form method="post" action="mainmenu.php">
<p>
   Nick: <input name="txtNick" type="text" size="15" default="admin"/>
   <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>
```

Fig. 8: 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.

To avoid redundant exploration of similar executions, Apollo performs state matching (performed implicitly in Line 11 of Figure 6) by not adding already-explored transitions.

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`[13] 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[14].

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).

6 Evaluation

We experimentally measured the effectiveness of Apollo by using it to find faults in PHP web applications. We designed experiments to answer the following research questions:

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

Q2. How effective is the fault detection technique of Apollo compared to alternative approaches in terms of the number and severity of discovered faults and the line coverage achieved?

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

For the evaluation, we selected 6 open-source PHP programs from http://sourceforge.net (see Figure 9):

- **faqforge**: tool for creating and managing documents.
- **webchess**: online chess game.
- **schoolmate**: PHP/MySQL solution for administering elementary, middle, and high schools.
- **phpsysinfo**: displays system information, e.g., uptime, CPU, memory, etc.

13. http://choco-solver.net/index.php?title=Main_Page

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

program	version	#files	PHP LOC	#downloads
faqforge	1.3.2	19	734	14,164
webchess	0.9.0	24	2,226	32,352
schoolmate	1.5.4	63	4,263	4,466
phpsysinfo	2.5.3	73	7,745	492,217
timeclock	1.0.3	62	13,879	23,708
phpBB2	2.0.21	78	16,993	18,668,112

Fig. 9: Subject programs. **#files** counts the `.php` and `.inc` files. **PHP LOC** is the number of executable PHP lines, computed by the interpreter as the number of lines with PHP opcodes. **#downloads** is the number of downloads from `http://sourceforge.net`.

Fault Category	Faults	Percentage
Malformed SQL	60	71.4
Array index out of bound	5	6.0
Resource used as offset	4	4.8
Failed to open stream	4	4.8
File not found	2	2.6
Can't open connection	2	2.6
Assigning reference	2	2.6
Undefined function	1	1.2

Fig. 11: The execution faults found by Apollo.

- **timeclock** is a web-based timeclock system.
- **phpBB2** is a discussion forum.

6.1 Generation Strategies

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** is an approach similar to the one proposed by Halfond and Orso [20] for JavaScript. The test input generation strategy 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.

6.2 Methodology

To answer the first research question (Q1) we applied Apollo to 6 subject programs and we classified the discovered failures into five groups based on their different failure characteristics:

- **execution crash:** the PHP interpreter terminates with an exception.
- **execution error:** the PHP interpreter emits an error message that is visible in the generated HTML.
- **execution warning:** the PHP interpreter emits an error message that is invisible in the generated HTML.
- **HTML error:** the program generates HTML for which the validator produces an error report.
- **HTML warning:** the program generates HTML for which the validator produces a warning report.

This classification is a refinement of the one presented in Section 2.3.

To answer the second research question (Q2) we compared our technique to two other approaches. We compared both the coverage achieved and the number of faults found with the **Randomized** generation strategy. Coverage was measured using the line coverage metric, i.e., the ratio of the number of executed lines to the total number of lines with executable PHP code in each application. We ran each test input generation strategy for 10 minutes on each subject program. This time limit was chosen arbitrarily, but it allows each strategy to generate hundreds of inputs and we have no reason to believe that the results would be materially 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 an oracle, and line coverage measurement. To avoid bias, we ran both strategies inside the same experimental harness. This includes the Database Manager (Section 5), which supplies user names and passwords for database access. For our experiments, we use the WDG offline HTML validator, version 1.2.2.

We also compared Apollo's results to the results reported by Minamide's static analysis [31] on four subject programs (Section 6.3.1 presents the results).

To answer the third research question, about the effectiveness of the input minimization, we performed the following experiments. Recall that several execution paths and inputs may expose the same failure. Our input minimization algorithm attempts to produce the shortest possible input that exposes each failure. The inputs to the minimizer are the failure found by the algorithm in Figure 6 along with all the execution paths that expose each failure.

6.3 Results

Figure 10 tabulates the faults (we manually inspected most of the reported failures and, to the best of our knowledge, all reported faults are counted only once). and line coverage results of running the two test input generation strategies on the subject programs. The **Apollo** strategy found 302 faults in the subject applications, versus only 95 faults for **Randomized**. Moreover, the **Apollo** test generation strategy achieved an average line coverage of 50.2%, versus only 11.6% for **Randomized**.

The coverage of **phpbb2** and **timeclock** is relatively small as the output of these applications contains client-side scripts written in JavaScript which Apollo currently does not analyze.

Figures 11 and 12 classify the faults reported by Apollo. The execution errors (Figure 11) are dominated by database-related errors, 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". The two SQL-related error messages quoted above occurred in faqforge (9

program	strategy	#inputs generated	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	717	92.4	0	9	0	46	19	74
webchess	**Randomized**	1805	5.9	1	13	2	3	0	19
	Apollo	557	42.0	1	20	2	7	0	35
schoolmate	**Randomized**	1396	8.3	1	0	0	18	0	19
	Apollo	724	64.9	2	21	9	58	0	100
phpsysinfo	**Randomized**	406	21.3	0	5	3	2	0	10
	Apollo	143	56.2	0	5	4	2	0	11
timeclock	**Randomized**	928	3.2	0	1	1	29	1	32
	Apollo	789	14.1	0	1	1	64	1	67
phpbb2	**Randomized**	2497	11.4	0	0	3	1	0	4
	Apollo	649	31.7	0	0	5	21	0	26
Total	**Randomized**	8493	11.6	2	19	9	63	2	95
	Apollo	3579	50.2	3	56	21	198	20	302

Fig. 10: 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.

Fault Category	Faults	Percentage
Element not allowed	40	17.5
Missing end tag	39	17.1
Can't generate system identifier	25	11.0
No attribute	25	11.0
Unopened close tag	21	9.2
Missing attribute	21	9.2
character not allowed	11	4.8
End tag for unfinished element	11	4.8
Incorrect attribute	8	3.5
Element declaration finished prematurely	8	3.5
Unfinished tag	7	3.1
Duplicate specification	4	1.8
Undefined element	4	1.8
Incorrect attribute value	4	1.8

Fig. 12: The HTML faults found by Apollo.

cases of error 1) and webchess (19 cases of error 1 and 1 case of error 2), schoolmate (20 cases of error 1 and 9 cases of error 2), timeclock (1 case of error 1), and phpbb2 (1 case of error 1).

These failures 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 return an invalid result. The subject programs failed to check the return value of `mysql_query`, and simply assume that a valid result is returned. These faults are indications of a potentially serious problem: the concatenation of user-supplied strings into SQL queries makes these programs vulnerable to SQL injection attacks [10]. Thus our testing approach indicates possible SQL injection vulnerabilities despite not being specifically designed to look for security issues.

The three execution crashes (when the interpreter terminates with an exception) in Figure 10 happen when the interpreter tries to load files or functions that are missing. For instance, for some inputs that can be supplied to the schoolmate subject program, the PHP interpreter attempts to load a file that does not exist in the current distribution of schoolmate. Since schoolmate has 63 files, and PHP is an interpreted language that allows the use of run-time string values when loading files, it is hard to detect such faults. Apollo also discovers a severe fault in the webchess subject program. This fault occurs when the interpreter tries to call to a function that is undefined since the PHP file implementing it is not included due to a value supplied as one of the parameters.

The 228 malformed HTML faults can be divided into several categories (Figure 12), These faults are mainly concerned with HTML elements that occur in the wrong place, HTML elements with incorrect values, and with unclosed tags. The breakdown of HTML faults is similar across the different PHP applications.

6.3.1 *Comparison with Static Analysis*

Minamide [31] presents a static analysis for discovering HTML malformedness faults in PHP applications. Minamide's analysis tool approximates the string output of a program with a context-free grammar, then discovers unclosed tags by intersecting this grammar with the regular expression of matched pairs of delimiters (open/closed tags). By contrast, our analysis uses an HTML validator and covers the entire language standard.

We performed our evaluation on a set of applications overlapping with Minamide's (webchess, faqforge, schoolmate, timeclock). For these four overlapping subject programs, Apollo is both more *effective* and more *efficient* than Minamide's tool. Apollo found 3.4 times as many HTML validation faults as Minamide's tool (195 vs. 56). The 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 80 execution faults, which are out of reach for Minamide's tool. Apollo is also more scalable—on schoolmate, Apollo found 58 malformed HTML faults in 10 minutes, while Minamide's tool found only 14 faults in 126 minutes. The considerably longer running time of Minamide's tool is due to the construction of large

program	success rate%	path constraints orig. size	path constraints reduction	inputs orig. size	inputs reduction
faqforge	64	22.3	78%	9.3	69%
webchess	91	23.4	81%	10.9	60%
schoolmate	51	22.9	62%	11.5	42%
phpsysinfo	82	24.3	82%	17.5	74%

Fig. 13: 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 **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 amount by which the minimized size is smaller than the unminimized size (i.e., $1 - \frac{minimized}{unminimized}$). The higher the percentage, the more successful the minimization.

automata and to the expensive algorithm for checking disjointness between regular expressions and context-free languages.

6.3.2 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 input) and the minimized path constraint (and input).

Figure 13 tabulates the results. The results show that our input minimization technique effectively reduces the size of inputs by at least 42%, for more than 50% of the faults.

6.4 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 have encountered cases when malformed HTML crashed the popular Internet Explorer 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 for sites related to financial transactions. Many 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. Line coverage indicates how much of the application was explored by the analysis. An analysis can only find faults in lines that are covered, so more coverage generally leads to more faults being detected.

Why do we present the user with minimized path constraints and inputs in addition to the inputs exposing the failure? Although an input that corresponds to a longer path constraint still exposes the same failure, in our experience, the removal of superfluous information helps programmers with 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 6 PHP projects. These may have serious quality problems, or may be unrepresentative in other ways. Four of the subject programs were also used as subject programs by Minamide [31]. 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 9), 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 11 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.

6.5 Limitations

Simulating user inputs based locally executed JavaScript 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 that is not visible. Apollo might also report a 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.

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. Similar to [41], 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.

7 Related Work

An earlier version of this paper was presented at ISSTA'08 [2]. The Apollo tool presented there did not handle the problem of automatically simulating user interactions in web applications. Instead, it relied on a manual transformation of the program under test to enable the exploration of a few selected user inputs. The current paper also extends [2] by providing a more extensive evaluation, which includes two new large web applications, and by presenting a detailed classification of the faults found by Apollo. In addition, the Apollo tool presented in [2] did not yet support web server integration,

In the remainder of this section, we discuss three categories of related work: (i) combined concrete and symbolic execution, (ii) techniques for input minimization, and (iii) testing of web applications.

7.1 Combined Concrete and Symbolic Execution

DART [18] is a tool for finding combinations of input values and environment settings for C programs that trigger errors such as assertion failures, crashes and nontermination. DART combines random test generation with symbolic reasoning to keep track of constraints for executed control-flow paths. A constraint solver directs subsequent executions towards uncovered branches. CUTE [36] is a variation (called *concolic testing*) on the DART approach. The authors of CUTE introduce a notion of approximate pointer constraints to enable reasoning over memory graphs and handle programs that use pointer arithmetic.

Subsequent work extends the original approach of combining concrete and symbolic executions to accomplish two primary goals: 1) improving scalability [1], [5], [16], [17], [19], [29], and 2) improving execution coverage and fault detection capability through better support for pointers and arrays [7], [36], better search heuristics [19], [24], [28], or by encompassing wider domains such as database applications [14].

Godefroid [16] proposed 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.

Exploiting the structure of the program input may improve scalability [17], [29]. Majumdar and Xu [29] abstract context-free grammars that represent the program inputs to produce a symbolic grammar. This grammar reduces the number of input strings to enumerate during test generation.

Majumdar and Sen [28] describe hybrid concolic testing, interleaves random testing with bounded exhaustive symbolic exploration to achieve better coverage. Inkumsah and Xie [24] combine evolutionary testing using genetic mutations with concolic testing to produce longer sequences of test inputs. SAGE [19] also uses improved heuristics, called *white-box fuzzing*, to achieve higher branch coverage.

Emmi *et al.* [14] extend concolic testing to database applications. This approach creates and inserts database records and enables testing program code that depends on embedded SQL queries.

Wassermann et al. [42] present a concolic testing tool for PHP. The goal of their work is to automatically identify security vulnerabilities caused by injecting malicious strings into SQL commands. Their tool uses a framework of finite-state transducers and a specialized constraint solver.

Some approaches aim at checking functional correctness. A number of tools [4], [6] use a separate implementation of the function being tested to compare outputs. This limits the approach to situations where a second implementation exists.

While our work builds on this significant body of research, there are two significant differences. First, our work goes beyond simple assertion failures and crashes by using 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. Second, our work addresses PHP's complex execution model, that involves multiple scripts invoked via user-interface options in generated HTML pages, and communicating values via session state and cookies. The only other concolic testing approach for PHP [42] does not present a fully automatic solution for dealing with multiple interrelated PHP scripts.

7.2 Minimizing Failure-Inducing Inputs

Our work minimizes the constraints on the input parameters. This shortens the failure-inducing inputs and to help to pinpoint the cause of faults. Godefroid *et al.* [19] faced this challenge since their technique produces several distinct inputs that expose the same fault. Their approach hashes all such inputs and returns an example failure-inducing input. Our work also addresses another issue: identifying the minimal set of program variables that are essential to induce the failure. In this regard, our work is similar to *delta debugging* [8], [44] and its extension *hierarchical delta debugging* [32]. These approaches modify the failure inducing input directly, thus leading to a single, minimal failure-inducing input. In contrast, our technique modifies the set of constraints on the failure-inducing input. This creates minimal *patterns* of failure-inducing inputs, which facilitates debugging. Moreover, our technique is more efficient, because it takes advantage of the (partial) overlapping of different inputs.

7.3 Testing of Web Applications

The language under consideration in this paper, PHP, is quite different from the focus of previous testing research. PHP poses several new challenges such as 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* (XSS) attacks [23], [26], [31], [40], [43]. In particular, Minamide [31] 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. However, 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-insensitive and context-insensitive approximations in the static analysis techniques used in this method result in false positives, while our method reports only real faults.

Kieżun *et al.* present a dynamic tool, Ardilla [27], to create SQL and XSS attacks. Their tool uses dynamic tainting, concolic execution, and attack-candidate generation and validation. Like ours, their tool reports only real faults. However, Kieżun *et al.* focus on finding security faults, while we concentrate on functional correctness. Their tool builds on and extends the input-generation component of Apollo but does not address the problem of user interaction. It is an interesting area of future research to combine Apollo's user-interaction and state-matching with Ardilla's exploit-detection capabilities.

McAllister *et al.* [30] also tackle the problem of testing interactive web application. Their approach attempts to follow user interactions. Their method relies on pre-recorded traces of user interactions, while our approach automatically discovers allowable interactions. Moreover, their approach to handling persistent state relies on instrumenting one particular web application framework, Django. In contrast, our approach is to instrument the PHP runtime system and observe database interactions. This allows handling state of PHP applications regardless of any framework they may use.

Benedikt *et al.* [3] present a tool, VeriWeb, for automatically testing dynamic webpages. They use a model checker to systematically explore all paths (up to a certain bound) of user navigate in a web site. When the exploration encounters HTML forms, VeriWeb uses *SmartProfiles*. SmartProfiles are user-specified attribute-value pairs that are used to automatically populate forms and supply values that should be provided as inputs. Although VeriWeb can automatically fill in the forms, the human 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. Benedikt *et al.* do not report any faults found, while we report 302.

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*) [25], [34], [38]. As far as we know, our work is the first attempt at *proactive* fault detection in PHP web applications using dynamic analysis.

Finally, our work is related to *implementation based* (as opposed to *specification based* e.g., [35]) testing of web applications. These works abstract the application behavior using a) client-side information such as user requests and corresponding application responses [12], [15], or b) server-side monitoring information such as user session data [13], [37], or c) static analysis of server-side implementation logic [20]. 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 [20] use static analysis of the server-side implementation logic to extract a web application's interface, i.e., the 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 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 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 6) a version of Halfond and Orso's technique for PHP. Compared to that re-implementation, Apollo achieved higher line coverage (50.2% vs. 11.6%) and found more faults (302 vs. 95).

8 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 6 open-source PHP web applications. Apollo's test generation strategy achieves over 50% line coverage. Apollo found a total of 302 faults in these applications: 84 execution problems and 218 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.

References

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

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA*, pages 261–272, 2008.

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

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[5] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

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

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

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

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

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

[11] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software — Practice and Experience*, 29(7):577–603, June 1999.

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

[13] 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.

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

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

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

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

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

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

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

[21] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[22] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Ku. Verifying Web applications using bounded model checking. In *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

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

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

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

[27] A. Kieżun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of International Conference of Software Engineering (ICSE)*, 2009.

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

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

[30] S. McAllister, E. Kirda, and C. Kruegel. Leveraging user interactions for in-depth testing of web applications. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 191–210, Berlin, Heidelberg, 2008. Springer-Verlag.

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

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

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

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

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

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

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

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

[39] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA*, 2006.

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

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

[42] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 249–260, 2008.

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

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

[45] F. Zoufaly. Web standards and search engine optimization (seo) – does google care about the quality of your markup?, 2008.