MIT-CSAIL-TR-2008-006 February 6, 2008

Finding Bugs In Dynamic Web Applications

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

Finding Bugs in Dynamic Web Applications

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

[†]MIT Computer Science and Artificial Intelligence Lab, 32 Vassar Street, Cambridge, MA 02139, USA
{artzi,dannydig,mernst,kiezun}@csail.mit.edu

[‡]IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{dolby,paradkar,ftip}@us.ibm.com

Abstract

Web script crashes and malformed dynamically-generated web pages are common errors, and they seriously impact usability of web applications. Current tools for web-page validation cannot handle the dynamically-generated pages that are ubiquitous on today's Internet. In this work, we apply a dynamic test generation technique, based on combined concrete and symbolic execution, to the domain of dynamic web applications. The technique generates tests automatically and minimizes the bug-inducing inputs to reduce duplication and to make the bug reports small and easy to understand and fix. We implemented the technique in Apollo, an automated tool that found dozens of bugs in real PHP applications. Apollo generates test inputs for the web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. This paper presents Apollo's algorithms and implementation, and an experimental evaluation that revealed a total of 214 bugs in 4 open-source PHP web applications.

1. Introduction

Dynamic test-generation tools, such as DART [14], Cute [26] or EXE [4], find bugs 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 important domain of web applications. This paper extends dynamic test generation to scripting languages, uses an oracle to determine whether the output of the web application is syntactically correct, and automatically sorts and minimizes the inputs that expose errors. Our Apollo system applies these techniques in the context of PHP, one of the most popular languages for web programming. According to Netcraft, PHP powered 21 million domains as of April 2007, including some of the largest and most well-known websites such as Wikipedia and WordPress.

The output of a web application is typically an HTML page that can be displayed in a browser. Our goal is to find errors that crash web applications, or results in a malformed HTML. Some errors may terminate the application, such as when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output presents an error message displayed inside an obtrusive table and the program execution is halted. More commonly in deployed applications, a web application creates output that is not syntactically well-formed HTML, for example by generating an opening tag without a matching closing tag. Web browsers are designed to tolerate some degree of malformedness in HTML, but this merely masks some of the underlying bugs. Malformed HTML is less portable across browsers and is vulnerable to breaking on new browser releases. An application that creates invalid (but displayable) HTML in the limited situations for which it has been tested may create undisplayable HTML on different executions. More seriously, browser's attempts to compensate for malformed web pages may lead to crashes[1], [2], and even security vulnerabilities[3]. As another serious problem, a browser might succeed in displaying only part of a malformed webpage, silently discarding important information.

Web developers widely recognize the importance of creating legal HTML. Many websites are validated using HTML validators[4],[5]. Web browsers are becoming more standard-compliant. Standard HTML renders faster. Search engines understand standard HTML better. Developers proudly display their website's compliance to the standards (even the ISSTA'08 website displays the W3C HTML compliance logo.) However, validating *dynamically* generated web pages is much harder. To prevent errors, programmers must make sure that the application creates a valid HTML page on *every* possible execution path, which is often very hard. The state-of-the-practice in validating PHP programs for web-standard compliance is using programs like HTML Kit[6] that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no widely used automated validator for dynamically generated pages.

Automatic checking of dynamically generated web applications is hard. Even professionally-developed applications often contains multiple errors (see Section 5). Dynamic checking (testing) is difficult because the input space is large—the input is a set of possible key-value pairs—and the output is highly verbose, which makes it hard to spot errors visually. Static checking is difficult because web applications are written in dynamic languages, such as PHP, which enables the creation of code and overriding of methods on the fly, making it difficult for a static analysis to capture program behavior.

This paper presents an automated technique for finding errors in HTML-generating web applications. Our approach adapts the well-established technique of dynamic test generation, based on combined concrete and symbolic execution and constraint solving [4, 14, 26], to the domain of web applications. Our work differs from these previous approaches by using an oracle to detect specification violations in the application's output, in addition to crashes or assertion failures. Another novelty in our work is inference of input parameters, which are not manifested in the source code. There

[1]`https://bugzilla.mozilla.org/show_bug.cgi?id=269095`
[2]`https://bugzilla.mozilla.org/show_bug.cgi?id=320459`
[3]`https://bugzilla.mozilla.org/show_bug.cgi?id=328937`
[4]`http://validator.w3.org`
[5]`http://www.htmlhelp.com/tools/validator`
[6]`http://www.htmlkit.com`

are also significant differences in the domain and language under consideration (web PHP applications, versus desktop C applications), as we discuss in Section 6. In our approach, the web application under test is first executed with an empty input. During each execution, the program is monitored to records path constraints that capture the outcome of control-flow predicates. Additionally, for each execution an oracle determines whether fatal errors or HTML well-formedness errors occur, the latter via use of an HTML validator. The system automatically and iteratively creates new inputs by negating one of the observed constraints and solving the modified constraint system. Each newly-created input explores at least one additional control flow path.

Many web applications create interactive HTML pages that contain user interface elements such as buttons and menus that require user interaction to execute further parts (further pages) of the application. This presents a challenge for automatic testing, because part of the application is referenced from the generated HTML text, rather than from the analyzed code. Our technique simulates user interaction by transforming the web application to create additional input parameters that the execution engine interprets as user input.

Techniques based on combined concrete and symbolic executions [4, 14, 26] may create multiple inputs that expose the same bug. 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 bug. This step is similar in spirit to Delta Debugging [5, 31]. 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 input within fewer program runs.

We implemented our method in a tool called Apollo, in the context of the publicly available PHP interpreter. We evaluated Apollo on publicly available web applications. In a short time budget of 10 minutes, Apollo found 214 bugs.

In summary, the contributions of this paper are:

- We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution [4, 14, 26], to the domain of web applications. The challenges include inferring the input parameters, which are not indicated by the source code; using an HTML verifier as an oracle; dealing with language-specific datatypes and operations; and simulating user input for interactive applications.
- We implemented the technique for PHP, in an automated tool, Apollo.
- We evaluated our techniques and tool by applying them to real web applications, and comparing with random testing. We show that dynamic test generation is highly effective when adapted to the domain of Web applications written in PHP: Apollo achieved coverage of 58.0% and identified 214 bugs.

The remainder of this paper is organized as follows. Section 2 presents an overview of PHP, introduces our running example and discusses classes of bugs in PHP web applications. Section 3 presents the algorithm and illustrates it on an example program. Section 4 discusses Apollo, an automated tool that we developed to implement our technique. Section 5 presents the experimental evaluation of Apollo on open-source web applications. Section 6 gives an overview of related work and Section 7 concludes.

2. PHP Web Applications

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 2.1.

The input to a PHP program is a map from strings to strings. Each key is a parameter that the program can read, write, or check if it is set. The string value corresponding to a key may be interpreted as a numerical value if appropriate. 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 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. More interestingly, PHP also has a number of signature features of scripting languages, such as dynamic typing, and an `eval` construct that interprets a string value that was computed at run-time as a code fragment, and then executes it. 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 the fact that there is a predicate that checks whether a variable has been defined, and that class and function definitions are statements that may occur anywhere.

The code in Figure 1 illustrates the flavor of PHP. There are recognizable forms of the usual program constructs such as `if` and `switch`. PHP's `require` statement that is 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 and need not be constant. There are many similar cases where run-time values are used, such as, e.g., the fact that `switch` labels need not be constant. This degree of flexibility is prized by PHP developers for enabling rapid application prototyping and development. However, as we shall see, it can make the overall structure of program hard to discern and is prone to code quality problems.

2.1 PHP Example

The PHP program of Figure 1 is a simplified version of SchoolMate[7], a PHP/MySQL solution for managing elementary, middle and high schools. The program allows administrators to manage classes and users, teachers to manage assignments and grades, and students to access their information.

The program starts (line 3) by calling the `make_header` function to create the HTML header. It then reads the global variable `page` that is supplied to the program in the URL, i.e., `http://www.mywebsite.com/index.php?page=1`. The if statement that follows (line 11) examines the value of the global parameter `page2` to determine if the interpreter should execute the `require` statement to include and evaluate a given file `printReportCards.php`[8] and terminate program execution (the function `die`).

On line 16, the program calls function `validateLogin` to determine if the user supplied the required login information. This function `validateLogin` (lines 28–40) sets the global variable `page` to the correct value based on the identity of the user. This value is read in the switch statement on line 18, which will either present the login screen to the user (created by a file `login.php`

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

[8] The reader may have noticed a typing error in the filename on line 11, which will be used later to illustrate our bug detection technique.

```
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 ...
28 function validateLogin() {
29   if(!isset($_GET['username'])) {
30     echo "<j2> username must be supplied.</h2>\n";
31     return;
32   }
33   $username = $_GET['username'];
34   $password = $_GET['password'];
35   if($username=="john" && $password=="theTeacher")
36     $page=1;
37   else if($username=="john" && $password=="theStudent")
38     $page=2;
39   else echo "<h2>Login error. Please try again</h2>\n";
40 }
41
42 function make_header() { // print HTML header
43 print("
44 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
45    "http://www.w3.org/TR/html4/strict.dtd">
46 <HTML>
47  <HEAD> <TITLE> Class Management  </TITLE> </HEAD>
48  <BODY>");
49 }
50
51 function make_footer() {  // close HTML elements opened by header()
52 print("
53  </BODY>
54 </HTML>");
55 }
56 ?>
```

Figure 1: A simplified PHP program excerpt from SchoolMate. This program contains three bugs, which are explained in Section 3.1.

execution problem	PHP interpreter displays an obtrusive error message, May also stop execution of the script.
HTML problem	PHP generates HTML for which an HTML validator issues a problem report

Table 1: Classification of bugs in PHP programs.

not shown here) or present one of the teacher/student screens (created by `TeacherMain.php` and `StudentMain.php` respectively). Execution ends with a call to the function `make_footer` that creates, in the output, the closing HTML elements opened by the `make_header` function.

2.2 Bugs in PHP Programs

We distinguish two kinds of problems that may arise when executing PHP web applications, as Table 1 shows. An *execution problem* occurs when the interpreter detects a problem during the execution of a program, such as missing included file, wrong MySQL queries or uncaught exceptions. This results in the generation of an obtrusive error message and might result in stop the execution of the script. Some execution problems are less serious, and occur when the interpreter encounters code that is questionable, such as using deprecated language constructs. In such cases, an error message is also generated, but will only be presented in a deployed application. An *HTML problem* involves situations in which the generated HTML page is not syntactically correct. The severity of malformed HTML varies depending on the kind of malformedness as well on whether or not the browser can gracefully handle the error. Web browsers vary in the degree to which they handle HTML syntax errors. Therefore, a malformed web-page may display differently under different browsers, which is undesirable. For example, on one browser, a part of the page may be missing while another, more lenient, browser may be able to display the part of the page with syntax errors.

The program of Figure 1 contains the following three bugs, explained in Section 3.1:

1. The program contains an *execution error* that is triggered when the `require` statement on line 11 is executed because of a typo in the referenced filename. Specifically, the code refers to a string `'printReportCards.php['` that should be `'printReportCards.php'`.
2. The program produces *malformed HTML* because the `make_footer` method is not executed in certain situations, resulting in an unclosed HTML tag in the output. The bug is triggered when the default case of the `switch` statement on line 23 is executed, which terminates program execution by calling `die()`. Note that this line can only be executed 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 30 is executed, resulting in the generation of an illegal HTML tag `j2`.

The first and third bug were artificially inserted into the example for illustrative purposes, although Apollo does manage to expose a bug resulting from a missing include file in schoolmate. The second bug, however, exists in the original code. Observe that the conditions under which the second bug is triggered are complicated and that this bug might have gone undetected even if the developer tried to verify the output of her program using an HTML verifier.

3. Algorithm

Our algorithm for finding bugs in PHP applications is a variation on a well-established dynamic test generation technique [4, 14, 15, 26] (sometimes referred to as concolic testing). The basic idea is to execute an application on an initial input (e.g., an unconstrained or randomly chosen input), and then on additional inputs obtained by solving constraints derived from exercised control flow paths. We adapted this technique to web applications written in PHP as follows:

- We extend the technique to validate the correctness of the program output. We use an oracle, in the form of an HTML val-

```
   parameters: Program $\mathcal{P}$, initial input $\mathcal{I}$, oracle $O$
   result     : Bugs in $\mathcal{P}$
1  $\mathcal{P}' := simulateUserInput(P)$;
2  $\mathcal{I}.bound := 0$;
3  $bugs := \emptyset$;
4  $inputQueue := emptyQueue()$;
5  $enqueue(inputQueue, \mathcal{I})$;
6  while $not\ empty(inputQueue)\ and\ not\ timeExpired()$ do
7      $input := dequeue(inputQueue)$;
8      $output := executeConcrete(\mathcal{P}', input)$;
9      $bugs := bugs \cup checkForBugs(O, output)$;
10     $c_1 \wedge \ldots \wedge c_n := executeSymbolic(\mathcal{P}', input)$;
11     foreach $i = input.bound, \ldots, n$ do
12         $pc := c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
13         if $satisfiable(pc)$ then
14             $newInput := solve(pc)$;
15             $newInput.bound := i$;
16             $enqueue(inputQueue, newInput)$;
17         end
18     end
19 end
20 return $bugs$;
```

Figure 2: Pseudo-code for algorithm. The algorithm uses auxiliary functions *simulateUserInput*, *emptyQueue*, *enqueue*, *dequeue*, *executeConcrete*, *executeSymbolic*, *checkForBugs*, *timeExpired* and *satisfiable*. Each input has an associated attribute *bound*, used to prevent multiple exploration of the same input.

idator, to determine whether or not the output is a well-formed HTML page.

- The PHP language contains a number of constructs such as `isset` (checking whether a variable is defined), `isempty` (checking whether a variable contains a value from a specific set), `require` (dynamic loading of additional code to be executed), and several others that require the generation of constraints that are absent in languages such as C or Java.
- PHP applications typically interact with a database and need appropriate values for user authentication (i.e., user name and password). It is not possible to infer these values by either static or dynamic analysis, or by randomly guessing. Therefore, our technique uses a pre-specified set of values for database authentication.
- The HTML pages generated by a PHP applications may contain buttons that—when pressed by the user—result in the loading and execution of additional PHP source files. We simulate such user input by transforming the source code. Specifically, for each page h that contains N buttons, we add an additional input parameter p to the PHP program, whose values may range from 1 through N. Then, at the place where page p is generated, a switch statement is inserted that includes the appropriate PHP source file, depending on the value supplied for p. The steps of the user input simulator are fully mechanic, and the required modifications are minimal, but for the evaluation we performed the program transformation by hand (due to time constraints).

Figure 2 shows the pseudo-code of our algorithm. The inputs to the algorithm are: (i) a program $\mathcal{P}$, (ii) an output oracle O, and (iii) an initial input $\mathcal{I}$. The algorithm associates attribute *bound* with each input, to prevent solving the same path constraint multiple times, similarly to SAGE [13, 15]. The algorithm begins (line 1) by transforming the program to simulate user input, as discusses before. Then, the algorithm initializes the collection of bugs found (line 3. Next, the algorithm associates bound 0 with the initial input (line 2). Variable *inputQueue*, which represents a priority queue of inputs that have yet to be explored, is initialized to a singleton queue containing the initial input.

The algorithm then enters an iterative phase that continues as long as there are more inputs to explore, and as long as the time budget has not been exceeded (lines 6-19). As long as this is the case, an input is retrieved (line 7). Next, the program is executed concretely on the input (line 8). Additionally, the algorithm consults the oracle to find bugs in the output (line 9).

Next, the program is executed symbolically on the input (line 10). The result of symbolic execution is a *path constraint*, which is a conjunction of conditions on the program's input parameters, $\bigwedge_{i=1}^{n} c_i$, that is fulfilled on a given executed path (here, the path constraint reflects the path that was just executed). The algorithm then creates new test inputs by modifying the path constraint (lines 11-18), as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 12). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along the prefix of the original execution path, and then take the opposite branch. The algorithm consults the constraint solver to check satisfiability of the alternative path constraint (line 13), and to find a concrete input that satisfies the alternative path constraint (line 14). The algorithm then adds the new input to the queue (line 16).

3.1 Algorithm Example

Let us now consider how the algorithm of Figure 2 finds one of the previously discussed bugs in the example program of Figure 1.

execution 1. The first input to the program is the empty input (i.e., empty mapping of parameter names to values). When the program is executed with this input, the else-branch of the if-statement on line 6 is selected because the `page` parameter is not set. Furthermore, since parameter `page2` is not defined, the condition of the if-statement on line 10 evaluates to `false`, thus bypassing the body of that if-statement. When execution reaches the if-statement on line 16, its condition evaluates to `false` because parameter `login` is not defined. Execution then continues with the switch statement on line 18, and the case on line 20 is selected because `page` has value `0`. Execution eventually reaches line 26 and terminates. The algorithm then invokes an HTML verifier and determines that the output is legal, and *executeSymbolic* produces the following path constraint:

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

To understand how this path constraint is generated, note that the execution of an `isset` condition in the program gives rise to a *NotSet* conjunct or a *Set* conjunct in the path constraint, depending on whether the condition succeeds. Furthermore, because neither parameter `page2` nor `login` were defined, the interpreter gave each the default value of `0`. The comparisons of those default values on lines 10 and 16 of Figure 1 give rise to the path constraint.

The algorithm now enters the **foreach** loop on line 11 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:

$$\begin{array}{lr} NotSet(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} = \texttt{1} & \text{(II)} \\ NotSet(\texttt{page}) \wedge \texttt{page2} = \texttt{1337} & \text{(III)} \\ Set(\texttt{page}) & \text{(IV)} \end{array}$$

execution 2. For path constraint (II), the constraint solver may find

```
parameters: Program $\mathcal{P}$, oracle $O$, bug $\mathcal{B}$
result    : Short path constraint that exposes $\mathcal{B}$
1  $pathConstraints := allExposing(\mathcal{B})$;
2  $c_1 \wedge \ldots \wedge c_n := intersect(pathConstraints)$;
3  $pc := true$;
4  foreach $i = 1, \ldots, n$ do
5      $pc_i := c_1 \wedge \ldots c_{i-1} \wedge c_{i+1} \wedge \ldots c_n$;
6      $input_i := solve(pc_i)$;
7      $output_i := executeConcrete(\mathcal{P}, input_i)$;
8      $bugs_i := checkForBugs(O, output_i)$;
9      if $\mathcal{B} \notin bugs_i$ then
10         $pc := pc \wedge c_i$;
11     end
12 end
13 $input_{pc} := solve(pc)$;
14 $output_{pc} := executeConcrete(\mathcal{P}, input_{pc})$;
15 $bugs_{pc} := checkForBugs(O, output_{pc})$;
16 if $\mathcal{B} \in bugs_{pc}$ then
17     return $pc$;
18 else
19     return $shortest(pathConstraints)$;
20 end
```

Figure 3: Pseudo-code for the path constraint minimization heuristic algorithm. The algorithm uses an auxiliary functions *allExposing* (returns all path constraints that expose a bug), *intersect* (returns the conjunction of conditions that are present in all given path constraints), and *shortest* (returns the path constraint with fewest conjuncts). The other auxiliary functions, *solve*, *executeConcrete* and *checkForBugs* are the same as in Figure 2.

the following input (the solver is free to select any value for `page2`, other than 1337)

$$\texttt{page2} \leftarrow \texttt{0}, \texttt{login} \leftarrow 1$$

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 29 evaluates to `true`, because the `username` parameter is not set, resulting in the generation of output containing an incorrect HTML tag `j2` on line 30. When the HTML validator checks the page, the bug is discovered and added to the list.

3.2 Input Minimization

Our technique creates complex inputs that expose hard-to-find bugs. Our technique presents to the user the concrete input on which the bug was encountered. Previous dynamic test generation tools [4, 14, 26] presented the whole input to the user, without indicating which part is responsible for the bug, even though the bug might have occurred for a small subset of the input. This verbosity overloads the developer with unnecessary information, obfuscating the real cause of the bug.

The goal of input minimization is to aid the developer to quickly find the location of bugs. The minimizer finds a short path constraint and input that exposes each bug. The idea is that a shorter path constraint describes a more general condition under which the bug is revealed. Figure 3 shows the pseudo-code of the minimization algorithm. Our technique minimizes bug-inducing inputs *post-mortem*, i.e., after the main algorithm exposes the bugs.

Our minimizer differs from *input* minimization techniques, such as delta debugging [5,31], in that our algorithm operates on the *path constraint* that exposes the bug, and not the *input*. A constraint concisely describes a class of inputs (e.g., the constraint $page2 \neq 1337$ describes all inputs different than 1337). Since a concrete input is an instantiation of a constraint, it is more effective to reason about input properties in terms of their constraints. For example, the minimizer finds the overlap of two inputs, $page2 \leftarrow 0$ and $page2 \leftarrow 1$, by looking at the constraints ($page2 \neq 1337$) that generated these inputs. Without the constraint information, the minimizer could not decide whether the two inputs overlap, other than by executing both.

Each bug 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 bug.

For each bug, the minimizer knows all path constraints that lead to inputs that exposed the bug. For each bug, the algorithm first intersects all those path constraints (line 2 in Figure 3). Starting from this intersection path constraint, of length n, the minimizer systematically creates path constraints of length $n-1$ by removing one condition at a time (lines 4-12). For each of these shorter path constraints, the minimizer invokes a constraint solver to find a concrete input. The minimizer executes the program on this input and if the program output does not expose the bug, then it means that the removed condition is required for exposing the bug. The final path constraint is the conjunction of all such required conditions. Along with the final path constraint, Apollo returns a concrete input that exposes the bug.

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

3.3 Minimization Example

We illustrate the minimization on the following example. The malformed HTML bug described in Section 3.1 can be triggered along different execution paths. For example, both of the following path constraints lead to inputs that expose the bug. Path constraint (a) is the same as (II) in Section 3.1.

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

First, the minimizer computes the intersection of the path constraints (line 2). The intersection is

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

The minimizer creates two shorter path constraints (by removing each of the two conjunct in turn). First, `login = 1`. This path constraint corresponds to an input that reproduces the bug, namely `login` $\leftarrow$ `1`. Second, `page2` $\neq$ `1337`. This path constraint does not correspond to an input that exposes the bug. Thus, the minimizer concludes that the condition `login = 1`, that was removed from ($a \cap b$) to form the second path constraint, is required. The result of minimization is the conjunction of all required conditions. In this example, the minimizer returns `login = 1`. Note that our algorithm executed the program only 2 times.

The result is the minimal path constraint that describes bug-inducing inputs. In this example, the path constraint also corresponds to the smallest input that exposes the bug.

4. Implementation

We implemented our technique in a tool called Apollo, which consists of three major components, **Executor**, **Input Generator**, and **Bug Finder**, illustrated in Figure 4. This section first provides

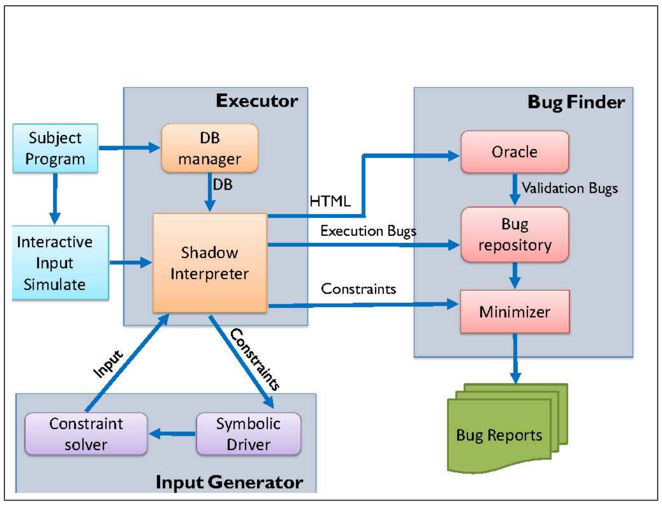

Figure 4: Illustration of the architecture of Apollo.

a high-level overview of the components and then discusses the pragmatics of the implementation,

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

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

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

- The **Symbolic Driver** generates new path constraints, and selects the next path constraint to solve for each execution.
- The **Constraint Solver** computes an assignment of values to input parameters that satisfies a given path constraint.
- The **Value Generator** (not presented in Figure 4) 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.

The **Bug Finder** stores the bugs found in all executions. After the exploration is completed, the bug finder analyzes all the bugs to remove duplicate reports and minimize the conditions that exposes each bug. The Bug Finder has the following sub-components:

- The **Oracle** finds syntactic problems in the output of the program.
- The **Bug repository** stores all reports containing syntactic and execution bugs found during all executions.
- The **Input Minimizer** finds, for a given error-inducing input, the minimal part of the input that induces the same error.

The stand-alone component of the **User Input Simulator** performs a transformation of the program that models interactive user input by way of additional parameters.

In the remainder of this section, we describe each of these components, and discusses the pragmatics of implementing our technique for PHP.

Interpreter. We modified the Zend PHP interpreter, version 5.2.2, [9] to produce symbolic path constraints for the executed program, using the "shadow interpreter" approach [6]. The shadow interpreter performs the regular (concrete) program execution using the concrete values, and simultaneously performs symbolic execution. This involves the following steps:

- A symbolic variable may be associated with each value. 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 = $_GET["param1"]` results in associating the value read from the global parameter `param1` and bound to parameter `x` with the symbolic variable `param1`. Values maintain their associations through assignments and function calls.
  Unlike other projects that perform concrete and symbolic execution [4,14,15,26], our interpreter does not associate complex symbolic expressions with runtime values, only the (optional) symbolic variables. This keeps the constraint solver very simple and reduces performance overhead. As our results (Section 5) indicate, this lightweight approach is sufficient for the analyzed PHP programs.
- At branching points (i.e., value comparisons) that involve values associated with symbolic variables, the interpreter extends the (initially empty) path constraint with a conjunct that corresponds to the branch actually taken in the execution. For example, if the program executes a statement `if ($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.
- 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 whether 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* conditions. 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.

In addition to the above modifications in support of path constraint generation, we also modified the interpreter to record positional information for output-generating statements such as `echo` and `print`. This positional information is used to detect redundancy (duplicates) in the HTML problem reports. Every such report (e.g., unknown tag) contains the position of the problem in the output. However, the same problem can manifest itself in different places over different executions. To overcome this, Apollo maps the position of the problem in the output to the PHP statement that produced the text in that position. Apollo treats two syntactic bugs as equivalent if they contain the same message without the output position, and they are produced by the same statement.

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

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

<p>
   <input name="login" value="login" type="submit"/>
   <input name="newAccount" value="New Account"
     type="button" onClick="window.open('newuser.php', '_self')"/>
</p>
</form>
```

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

User Input Simulator. Many PHP web applications create interactive HTML pages that contain user interface elements such as buttons and menus that require user interaction to execute further parts of the application. In such cases, pressing the button may result in the execution of additional PHP source files. For example, Figure 5 contains a simplified main entry file (`index.php`)[10] of webchess program, one of the programs evaluated in Section 5, which contains user interface elements that refer to two additional scripts, `mainmenu.php` and `newuser.php`. There are two challenges involved in dealing with such interactive applications. First, our basic approach does not automatically analyze the referenced files, because these are referenced from within HTML output (as opposed to being referenced from within PHP source code). Second, global information is shared between the different scripts using the `SESSION` global table.

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

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

This approach has the advantages that `SESSION` state is automatically shared between the different files, and that our algorithm is then able to find the values that correspond to each of the user interface elements. Since code might be executed when a button is pressed, this approach might induce false positive bug reports. We manually checked that our results do not contain false positive bug reports that are due to this limitation. Another slight disadvantage of this approach is that the transformed PHP application will output a *sequence* of HTML pages rather than a single page, so that some post processing is needed before the HTML validator can be invoked. However, the transformation is mechanical and the required source code changes are minimal. We currently perform the transformation manually, and are investigating a solution where the transformation is performed automatically.

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

Database Manager. Most PHP applications use a database, and to find bugs in such programs, some initial values need to be supplied. Apollo's Database Manager is responsible for: (i) (re)initializing the database prior to each execution (i.e., filling it with some initial values), and (ii) supplying additional information about username/password pairs that Apollo should use. The latter needs to be supplied, because 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 **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 7), and to 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 [4] and SAGE [15]; each conjunct in the path constraint knows the branch that created the conjunct, the driver keeps track of all branches executed so far and favors inputs created from path constraints that contain previously un-executed branches.

Constraint Solver. In Apollo, path constraints are transformed into integer constraints in a straightforward way, and solved using the `choco` solver[11].

Value Generator. 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).

Oracle. PHP web applications output HTML/XTHML. Therefore, in Apollo, we use as oracle an HTML validator that returns syntactic (malformed) HTML problems found in a given document. We experimented with both the offline WDG validator[12] and the online W3C markup validation service[13]. Both oracles identified the same syntactic bugs. In our experiments, we report the results obtained using WDG because that validator is faster.

Input Minimizer. Apollo minimizes inputs *postmortem*, after the main algorithm of Section 3 exposes the bugs. Apollo uses the algorithm described in Section 3.2. For every bug, the minimizer executes the program multiple times, with multiple inputs, and attempts to shorten the path constraints that expose the same bugs.

5. Evaluation

In this section, we report on experiments in which we measured the effectiveness of Apollo in finding bugs in PHP web applications. We designed the experiments to answer the following research questions:

Q1. How many bugs can Apollo find, and how can these bugs be classified according to the classification of Table 1?

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

Q3. How effective is our input minimization in reducing the size of bug-inducing inputs?

[11]`http://choco-solver.net/index.php?title=Main_Page`
[12]`http://htmlhelp.com/tools/validator/offline`
[13]`http://validator.w3.org`

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

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

For the evaluation, we selected the following open-source PHP programs (from `http://sourceforge.net`): faqforge 1.3.2 (tool for creating and managing documents), webchess 0.9.0 (online chess game), schoolmate 1.5.4 (PHP/MySQL solution for administering elementary, middle and high schools), phpsysinfo 2.5.3 (widely used, customizable script that displays system information, e.g., uptime, CPU, memory, etc.) Figure 6 presents the characteristics of the subject programs.

5.1 Generation Strategies

We compared our technique to a more conventional approach, which learns the input parameters and applicable values from the source. Halfond and Orso [16] present such a technique which statically discovers parameters and an approximation of the value domain for each parameter. Their tool can only analyze Java Script code, and therefore we compared to a similar technique which dynamically detects the input parameters and their types, and randomly assigns values mined from the program's source to each parameter.

Additionally, we compared our results to those reported by Minamide's static analysis [22] on the same subject programs (Section 5.3.1 presents the results). We use the following test input generation strategies in the remainder of this section:

Apollo is a strategy that generates test inputs using a the algorithm in Figure 2.

Randomized is a strategy that generates test inputs by giving random values to parameters. Random testing of PHP scripts is not trivial, however, because the parameters are not immediately clear from the source code. The randomized strategy infers the parameters' names and types from the dynamic traces—any variable for which the user can supply a value for is classified as a parameter. The randomized strategy assigns random values for each parameter. The values are chosen from a set of applicable (for the parameter's type) constant values that appear textually in the program source.

5.2 Methodology

We run each test input generation strategy for 10 minutes on each subject program. This time budget includes all experimental tasks, i.e., program execution, harvesting of constant values from program source, test generation, constraint solving (where applicable), output validation via oracle, and coverage measurement. To avoid bias, we run both strategies inside the same experimental harness. This includes the Database Manager (Section 4), that supplies username and password for the database access.

We measure line coverage, i.e., the ratio of the number executed lines to the total number of lines with PHP executable code in the application. We computed the total number of PHP lines in the application by counting, in the interpreter, the number of lines with PHP opcodes. Figure 6 presents the total number of lines for each of the subject programs.

To discover bugs in the PHP applications, Apollo executes the applications on the generated inputs and uses a validator to validate the correctness of the output using an HTML validator. For our experiments, we use the WDG offline HTML validator, version 1.2.2. Additionally, we intercept errors and warnings emitted by the PHP interpreter. We classify the discovered problems into five groups that are a refinement of Table 1:

execution crash: PHP interpreter terminates with an exception.

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

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

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

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

5.3 Results

Figure 7 tabulates the bug finding and coverage results of running the different test input generation strategies on the subject programs. From these results, it is clear that the Apollo test generation strategy outperforms the randomized testing by achieving an average coverage of 58.0%, versus 15.2% for **Randomized**. The Apollo strategy significantly outperforms the **Randomized** strategy by finding a total of 214 bugs in the subject applications, versus a total of 59 bugs for **Randomized**.

To get a better understanding of the types of bugs uncovered by Apollo, we examined the detailed results for schoolmate. The two most severe execution crashes happen when the program tries to load two files that are missing from the distribution of schoolmate. Since schoolmate contains 63 PHP source files and compilation is done on the fly when the interpreter needs to load a new file, it is not trivial to detect such problems. The developer needs to execute all possible paths to make sure the program loads all relevant files.

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

Both of these error messages have the same basic cause: user-supplied input parameters are concatenated directly into SQL query strings; leaving these parameters blank results in malformed SQL causing the `mysql_query` functions to not return a valid result. All three applications fail to check the return value of `mysql_query`, and simply assume that a valid result was returned. This causes functions that use the result to give the two above-quoted SQL-related errors. These are potentially serious bugs, since they are symptoms of a worse problem: the concatenation of user-supplied strings into SQL queries makes these programs vulnerable to SQL injection attacks [7], a well-studied class of security holes. Thus our testing approach points to these serious vulnerabilities despite not being specifically designed to look for security issues.

All 14 execution warnings were about unset time zone (which results in the interpreter using an arbitrary timezone). The 58 bugs in `schoolmate` that manifested themselves by the generation of malformed HTML can be classified as follows: 7 cases where an invalid attribute is used, e.g., "there is no attribute BORDERCOLOR", 7 cases where a required attribute is missing, e.g., "required attribute TYPE not specified", 2 cases where an undefined element is used, e.g., "element EMPTY undefined", 1 case where an incorrect value for an attribute is supplied: "value of attribute

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

Figure 7: Experimental results for 10-minute test generation runs. The table presents results for each subject program, and each strategy, separately. The **#inputs** column presents the number of inputs that each strategy created in the given time budget. The **coverage** column lists the line coverage achieved by the generated inputs. The **execution crashes**, **errors**, **warnings** and **HTML errors**, **warnings** columns list the number of bugs in the respective categories (see Section 5.2). The **Total bugs** columns sums up the number of discovered bugs.

ALIGN cannot be CENTER; must be one of TOP, MIDDLE, BOTTOM, LEFT, RIGHT",28 cases where a tag is not properly closed, e.g.,"found end tag for element FONT which is not open",10 cases where an element is used in a place where it is not allowed, e.g., " document type does not allow element BODY here",3 cases where a duplicate attribute is supplied, e.g., "duplicate specification of attribute CELLPADDING".

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

5.3.1 Comparison with Static Analysis

Minamide [22] presents a static analysis for discovering HTML malformedness bugs in PHP applications. His analysis tool is limited to finding unmatched pairs of delimiters (open/closed tags), while ours uses the official HTML validator and covers the whole language standard. We performed our evaluation on a set of applications overlapping with Minamide's (webchess, faqforge, schoolmate). For two of Minamide's subject programs (phpwims and timeclock) Apollo cannot be applied because the programs need to be executed in a web-browser and the current implementation of Apollo does not support this mode of execution.

Our tool achieves better results. Apollo found 2.7× more HTML validation bugs in the applications (122 vs. 45). Moreover, Apollo found 83 execution problems, which are out of reach for Minamide's tool. Apollo is also more scalable—on schoolmate, the largest of the programs, Apollo found 104 bugs in 10 minutes, while Minamide's tool found only 14 bugs in 126 minutes.

5.3.2 Input Minimization

To answer the third research question, about the effectiveness of the input minimization, we performed additional experiments. Recall that for each bug, there may be several execution paths, and inputs, that expose the bug. Our input minimization algorithm attempts to produce the shortest possible input that exposes each bug. The input minimizer takes the bugs found by the algorithm in Figure 2 along with all the execution paths that expose a bug. In our experiments, we measure the effectiveness of minimization, i.e., the reduction ratio. We use the size of the shortest original (unminimized) path constraint as the base for comparison. We consider the minimization of path constraints to have succeeded only if it produces a path shorter than the shortest un-minimized path.

program	success rate%	path constraint size	path constraint reduction	input size	input reduction
faqforge	64	22.3	4.5×	9.3	3.2×
webchess	91	23.4	5.1×	10.9	2.5×
schoolmate	51	22.9	2.6×	11.5	1.7×
phpsysinfo	82	24.3	5.3×	17.5	3.8×

Figure 8: Results of input minimization. The **success rate** indicates the percentage of bugs whose exposing input was successfully minimized (i.e., the minimizer found a shorter exposing input). The **size** columns list the average size of original (un-minimized) path constraints and their corresponding inputs. The **reduction** columns list the minimization ratio (i.e., how many times smaller is the minimized path constraint or input). The greater the ratio, the more successful is the minimization.

Figure 8 tabulates the results of the experiment. The results show that our input minimization technique effectively reduces the size of inputs by up to a factor of 5.3×, for more than 50% of the bugs. With this reduction, the programmer is liberated from having to analyze superfluous inputs.

5.4 Threats to Validity

Construct Validity. One could argue why we count malformed HTML as a defect in dynamically generated webpages. Does a webpage with malformed HTML pose a real problem or this is an artificial problem generated by the overly conservative specification of the HTML language? Although web browsers are resilient to malformed HTML, we encountered cases (in a different project) when malformed HTML crashed a widely popular web browser. More importantly, even though the browser might tolerate malformed HTML, different browsers or different versions of the same browser will not display all information that a user needs. This becomes crucial for some website, for example banking. Many informational and functional websites provide a button for verifying the validity of statically generated HTML. The challenges of dynamically generated webpages prevent the same institutions from validating the content. Finally, Apollo can discovers execution bugs in addition to malformed HTML problems.

One could question the use of coverage as a quality metric. We use line coverage only as a *secondary* metric, our *primary* metric begin the number of bugs found. The experimental results show that Apollo achieves a significantly better results, in both metrics, than randomized testing and static analysis.

One could ask why we minimize the path constraints presented to the programmer. Although a longer path constraint still exposes the bug, by removing superfluous information, Apollo can better assist the programmer in pinpointing the location of the bug.

Internal Validity. One could ask whether the design of evaluation and the results truly represent a cause-and-effect. Since we used subject projects developed by others, we could not influence the quality of the subject programs. Apollo does not search for seeded bugs, but it finds *real* bugs in real programs.

External Validity. One could ask whether our results are generalizable besides the subject programs that we chose. We only used Apollo to find bugs in four PHP projects. These may be immature projects that have serious quality problems, and are not representative. Three of the subject programs are also used as subject programs by Minamide [22]. 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, and it is ranked in the top 0.5% projects on sourceforge. Nevertheless, Apollo finds bugs in phpsysinfo.

Reliability. One could ask whether the results we present are reproducible. The subject programs that we used are publicly available from sourceforge. The bugs that we found are available for examination at `pag.csail.mit.edu/apollo`.

6. Related Work

Godefroid *et al.* [14] present DART, a tool for finding combinations of input values and environment settings for C programs that trigger assertion failures and crashes when these programs are executed. DART combines random test generation with the use of a symbolic reasoning component for keeping track of path constraints that capture the outcome of executed control flow predicates. A constraint solver is used to determine from the recorded path constraints how subsequent executions can be directed towards uncovered branches. Experimental results indicate that DART is highly effective at finding large numbers of errors in several C applications and frameworks, including important and previously unknown security vulnerabilities.

The core approach combining concrete and symbolic executions has been extended to accomplish two primary goals: 1) To improve the scalability of the approach [1, 12, 13, 15, 21] and 2) To improve execution coverage and bug detection capability through better support for pointers and arrays [4, 26], better search heuristics [15, 17, 20], or encompassing wider domains such as database applications [10].

Godefroid [12] proposes a compositional approach to improve the scalability of DART significantly. 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 available summary. This approach is shown to be capable of handling much larger programs than DART capabilities. Anand *et al.* [1] extend this compositional approach to be demand-driven to reduce the summary computation effort.

Majumdar and Xu [21] exploit the structure of the program input to improve scalability. In this approach, context free grammars which represent the program inputs are abstracted to produce a symbolic grammar. This symbolic grammar helps reduce the number of input strings that need to be enumerated during the combined concrete and symbolic test generation phase. The approach is shown have better scalability than the *concolic* version [26] of the combined concrete and symbolic exeuction approach.

Majumdar and Sen [20] describe a hybrid concolic testing approach which exploits the capability of random testing to explore deeper and longer inputs to achieve better coverage. Hybrid concolic testing interleaves random testing until saturation with bounded exhaustive symbolic exploration. This approach is demonstrated to result in doubling the coverage results achieved by concolic testing alone. Inkumsah and Xie [17] combine evolutionary testing using genetic mutations with concolic testing to produce longer sequences of test inputs. The SAGE system developed by Godefroid *et al.* [15] also uses improved heuristics, called *white-box fuzzing*, to achieve higher branch coverage faster.

Emmi *et al.* [10] extend the concolic testing to encompass database applications. This approach enables insertion of any needed database records to ensure execution of program code which depends on embedded SQL queries. A string constraint solver which can decide string equality, inequality, and membership in a regular language is used to facilitate the task.

Our work can benefit from these extensions to the combined concrete and symbolic execution approach. However, our work differs from the prior works in several respects. Most importantly, our work goes beyond simple assertion failures and crashes by relying on an oracle (in the form of an HTML validator) to determine correctness, which means that our tool can handle situations where the program has functionally incorrect behavior without relying on programmer assertions to determine that this is the case. Cadar and Engler [3] also recognize the issue of functional correctness, but address it by using a separate implementation of the function being tested to compare outputs. This limits the approach to situations where a second implementation exists.

Our work also minimizes the error inducing input to aid developer to pinpoint the cause of bugs. Godefroid *et al.* [15] faced this challenge since their technique produces several distinct inputs which may excite the same bug at a particular code location. They addressed the issue by hashing all such inputs and returning one exemplar of error inducing input to the developer. Our work addresses this issue as well as a different one: identifying the minimal set of program variables in an input that are essential to induce the error. In this regard, our work is similar in spirit to *delta debugging* [5, 31] and its extension *hierarchical delta debugging* [23]. These approaches modify the error inducing input directly, thus leading to a singular, minimal exemplar of such input. Our approach, on the other hand, modifies the set of constraints on error inducing input. This enables our approach to provide minimal *patterns* of error inducing inputs, thus aiding the bug fixing activities even further. Moreover, since our technique is aware of the (partial) overlapping of different inputs, it is more efficient.

The language under consideration, PHP, is also quite different, posing several new challenges such as the dynamic inclusion of files, and function definitions that are statements. Existing techniques for bug detection in PHP applications use static analysis and target security vulnerabilities such as *SQL injection* or *cross-site scripting* attacks [19, 22, 29, 30]. In particular, Minamide [22] uses static string analysis and language transducers to model PHP string operations to generate *potential* HTML output—represented by a context free grammar—from the web application. This method can be used to generate HTML document instances of the resulting grammar and to validate them using an existing HTML validator. As a more complete alternative, Minamide proposes a *matching validation* which checks for containment of the generated context free grammar against a regular subset of the HTML specification. Unfortunately, this approach can only check for matching start and end tags in the HTML output, while our technique covers the entire HTML specification. Also, flow and context insensitive approximations in the static analysis techniques used in this method result in false positives, whereas our method reports only real bugs.

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

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*) [18,24,28]. To the best of our knowledge, our work is the first attempt at *proactive* bug detection in PHP web applications using dynamic analysis.

Finally, our work is related to the growing body of work in *implementation based* (as opposed to *specification based* e.g., [25]) testing of web applications. These works abstract the application behavior using either a) client side information such as user requests and corresponding application responses [8,11], or b) server side monitoring information such as user session data [9,27], or c) static analysis of server side implementation logic [16]. 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 [16] use static analysis of the server side implementation logic to extract web application interface—a set of input parameters and their potential values. They obtained better code coverage with test cases based on the interface extracted using their technique as compared to the test cases based on the interface extracted using a conventional web crawler. However, the resulting coverage may depend on the choices made by the test generator to combine parameter values—an exhaustive combination of values may be needed to maximize the code coverage. In contrast, our work uses dynamic analysis of server side implementation logic for bug detection and minimizes the number of inputs needed to maximize the coverage. Furthermore, we include results on bug detection capabilities of our approach.

7. Conclusions

We have presented a technique for finding bugs 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 by the application. 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 bug-inducing inputs.

We implemented the analysis in a tool called Apollo, and evaluated it on four open-source PHP web applications. We found a total of 214 bugs in these applications, including 92 execution problems, and 122 cases where malformed HTML was generated. We also found that Apollo's test generation strategy achieves good coverage and find many bugs. Finally, Apollo also minimizes the size of bug-inducing inputs: the minimized inputs are up to 5.3× shorter than the unminimized ones. This reduction can help programmers to better diagnose the bugs.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08 (to appear)*.

[2] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *WWW'02*.

[3] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05*.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS'06*.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE'05*.

[6] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. Technical report, Microsoft Research, 2007. MSR-TR-2007-151.

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

[8] S. Elbaum, K.-R. Chilakamarri, M. Fisher, and G. Rothermel. Web application characterization through directed requests. In *WODA'06*.

[9] 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.

[10] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA'07*.

[11] M. Fisher, S. G. Elbaum, and G. Rothermel. Dynamic characterization of web application interfaces. In *FASE'07*.

[12] P. Godefroid. Compositional dynamic test generation. In *POPL'07*.

[13] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI'08 (To appear)*.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*.

[15] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS'08 (to appear)*.

[16] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *ESEC-FSE'07*.

[17] K. Inkumsah and T. Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE'07*.

[18] M. Johns and C. Beyerlein. SMask: preventing injection attacks in web applications by approximating automatic data/code separation. In *SAC'07*.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP'06: Security and Privacy*.

[20] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*.

[21] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE'07*.

[22] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW'05*.

[23] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *ICSE'06*.

[24] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID'05*.

[25] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'01*.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE'05*.

[27] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05*.

[28] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL'06*.

[29] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*.

[30] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*.

[31] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6), 1999.