RC24675 (W0810-107) October 17, 2008
Computer Science

IBM Research Report

Practical Fault Localization for Dynamic Web Applications

Shay Artzi
MIT Computer Science and AI Laboratory
32 Vasser Street
Cambridge, MA 02139
USA

Julian Dolby, Frank Tip
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Practical Fault Localization for Dynamic Web Applications

Shay Artzi* Julian Dolby† Frank Tip†
* MIT Computer Science and AI Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
artzi@csail.mit.edu
†IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{dolby,ftip}@us.ibm.com

Abstract

We leverage two existing techniques—combined concrete and symbolic execution, and the Tarantula algorithm for fault localization—to create a uniquely powerful method for finding and localizing faults. The method automatically discovers inputs required to exercise paths through a program, thus overcoming the limitation of many existing fault localization techniques that a test suite be available upfront. We show how the effectiveness of Tarantula can be improved significantly by utilizing a correlation between executed statements and the output that they produce, in combination with an oracle that detects where errors occur in the output. We implemented these ideas in Apollo, a tool for testing PHP applications, using an HTML validator as our oracle. When applied to a number of open-source PHP applications, Apollo found, and precisely localized a significant number of faults.

1 Introduction

Web applications are typically written in a combination of several programming languages (e.g., JavaScript on the client side, and PHP with embedded SQL commands on the server side), and generate structured output in the form of dynamically generated HTML pages that may refer to additional scripts to be executed. Since the application is built using a complex mixture of different languages, programmers may inadvertently make mistakes and introduce faults in the applications, resulting in web application crashes and malformed dynamically-generated HTML pages that can seriously impact usability. We present the first fully automatic technique that automatically finds and localizes faults in PHP web applications.

In previous work [3], we adapted the well-established technique of *concolic* (combined concrete and symbolic) execution [4, 7, 8, 19, 22] to web applications written in PHP. In this approach, the application is first executed on an empty input, and a *path condition* is recorded that reflects the control flow predicates in the application that have been executed. By changing one of the predicates in the path condition, and solving the resulting condition, additional inputs can be obtained. Execution of the program on these inputs will result in additional control flow paths being exercised. This process is repeated until either we have sufficient coverage of the statements in the application or until the time budget is exhausted. For each execution, we determine if an execution error occurs, or if the generated HTML page is malformed, using an HTML validator as an oracle. We implemented the technique in a tool called *Apollo* (version 1.0), and in previous experiments on 4 open-source PHP applications, *Apollo 1.0* found a total of 214 failures [3].

The coverage achieved by *Apollo 1.0* was limited, since *Apollo 1.0* ignored changes to the state of the environment by the executed scripts. That is, each script was executed from a *single* initial environment state (usually a populated database). However, the desired behavior of a PHP application is often only achieved by a series of interactions between the user and the server (e.g., a minimum of five inputs are needed from opening Amazon to buying a book). In this paper we enhance concolic testing by supporting automatic dynamic simulation of user interactions, and implement it in a new version of our tool, *Apollo 2.0*. *Apollo 2.0* records the environment state (database, sessions, cookies) after executing each script, analyzes the output of the script 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.

More importantly, this paper also addresses the obvious next step of determining *where in the source code* changes need to be made in order to fix these failures. This task is commonly referred to as *fault localization*, and has been studied extensively in the literature (see, e.g., [5, 11–13, 18, 25]). In this paper we combine the *Tarantula* fault localization technique by Jones et al. [11,12] with concolic execution in order to perform fully automated failure detection and localization for web applications written in PHP. The *Tarantula* technique predicts statements that are

likely to be responsible for failures by computing for each statement, the percentage of passing tests that execute it and the percentage of failing tests that execute it. From this, a *suspiciousness rating* is computed for each executed statement. Programmers are encouraged to examine the statements in order of decreasing suspiciousness, and this has been demonstrated to be quite effective in experiments with the Siemens suite [10] of versions of small C programs into which artificial faults have been seeded [11].

The use of concolic execution to obtain passing and failing runs overcomes the limitation of *Tarantula* and many other existing fault localization techniques that a test suite with passing and failing runs be available up-front. Furthermore, the fact that PHP applications generate output in a format (HTML) that can be validated using an oracle (an HTML validator) enables us to enhance the effectiveness of fault localization. This is accomplished by maintaining, during program execution, an *output mapping* from statements in the program to the fragments of output they produce. This mapping, when combined with the report of the oracle that indicates what parts of the program's output are incorrect, provides an additional source of information about the possible location of the fault, and is used to fine-tune the suspiciousness ratings provided by *Tarantula*.

The contributions of this paper are as follows:

1. We demonstrate that the *Tarantula* technique, which was previously only evaluated on small programs from the Siemens suite with artificially seeded faults [11, 12], is effective at localizing real faults in commonly used PHP applications.
2. We present an approach for fault localization that leverages concolic execution and the *Tarantula* fault localization method. Contrary to most previous methods, ours does not require the availability of a test suite.
3. We implemented the technique in *Apollo 2.0*, a fully automated tool for finding faults in PHP applications. This included the design of a new automated technique for the simulation of user input and tracking the usage of persistent state. An experimental evaluation using 6 PHP applications demonstrates that this significantly increased coverage for 6 interactive PHP applications.
4. We used *Apollo 2.0* to localize 49 faults in 3 of the PHP applications and compared the effectiveness of: (i) *Tarantula*, (ii) a fault localization method that only uses the output mapping, and (iii) a technique that enhances *Tarantula* using the output mapping. We found that (iii) significantly outperforms (i) and (ii).

2 Context: PHP and Web Applications

PHP is widely used for implementing Web applications, in part due to its rich library support for network interaction, HTTP processing and database access. 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 forms to invoke other PHP scripts, passing them a combination of user input and constant values taken from the generated HTML.

This section briefly reviews the PHP scripting language, and discusses the kinds of failures that may occur during the execution of a PHP application, focusing on those aspects of PHP that differ from mainstream languages.

2.1 The PHP Scripting Language

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 Figures 1(b), 1(c) and 1(d) illustrates the flavor of PHP. Note first of all that the code is an ad-hoc mixture of PHP statements and HTML fragments. The PHP code is delimited by `<?php` and `?>` tokens. The use of HTML in the middle of PHP indicates that HTML is generated as if it were in a print statement. The `require` statements resemble 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. Observe that 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. 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 Failures in PHP Programs

Our technique targets two types of failures that may occur during the execution of PHP applications and that can be automatically detected:

```
<html>
<head>Login</head>
<body>
  <form name="login" action="exampleLogin.php">
    <input type="text" name="user"/>
    <input type="password" name="pw"/>
  </form>
</body>
</html>
```

(a) index.php

```
<?php
  userTag = 'user'
  pwTag = 'pw';
  typeTag = 'type';
?>
```

(b) constants.php

```
<HTML>
<?php
  require( dirname(__FILENAME__).'/includes/constants.php');

  $user = $_REQUEST[ 'user' ];
  $pw = $_REQUEST[ 'pw' ];

  if (check_password($user, $pw) {
    print "<HEAD>Login Successful</HEAD>\n";

  $_SESSION[ $userTag] = $user;
  $_SESSION[ $pwTag ] = $pw;
?>
  <BODY>
    <FORM action="view.php">
      <INPUT TYPE="text" NAME="topic"/>
    </FORM>
  </BODY>
<?php
  if ($user == 'admin') {
    $_SESSION[ $typeTag ] = 'admin';
  }
  else {
    print "<HEAD>Login Failed</HEAD>\n";
  }
?>
</HTML>
```

(c) login.php

```
<HTML>
<HEAD>Topic View</HEAD>
<?php
  print "<BODY>\n";
  if(check_password($_SESSION[$userTag], $_SESSION[$pwTag]) {
    require( dirname(__FILENAME__).'/includes/constants.php');

    $type = $_SESSION[ $typeTag ];
    $topic = $_REQUEST[ 'topic' ];

    if ($type == 'admin') {
      print "<H1>Admin ";
    } else {
      print "<H1>Normal ";
    }
    print "View of $topic</H1>\n";

    /* code to print topic view... */

    if ($type == 'admin') {
      print "<H2>Administrative Details\n";
    /* code to print admin details... */
    }
  } else {
      print "Please Log in\n";
  }
  print "</BODY>\n";
?>
</HTML>
```

(d) view.php

Figure 1: Example PHP web application.

- *execution failures* are caused by missing included files, incorrect MySQL queries, and uncaught exceptions. Such failures are easily identified as the PHP interpreter generates an error message and halts execution. Less serious execution failures, such as those caused by the use of deprecated language constructs produce obtrusive error messages but do not halt execution.
- *HTML failures* involve situations in which the generated HTML page is not syntactically correct according to an HTML validator. This may result in pages being rendered incorrectly in a browser, it may cause portability problems, and the resulting pages may render slower when browsers attempt to compensate for the malformedness.

2.3 Fault Localization

Detecting failures only demonstrates that a fault exists; the next step is to find the *location* of the fault that causes each failure. There are at least two pieces of information that might help:

1. For HTML failures, validators provide locations in the HTML file that have problems, and one could correlate malformed HTML fragments with the portions of the scripts that produced them.
2. For both kinds of failures, one could look at runs that do *not* exhibit the error, and record what set of statements such runs execute. Comparing that set of statements with the set of statements executed by the failing run could provide clues as to the fault location. The extensive literature on fault localization algorithms that exploit such information, is discussed in Section 7.

2.4 PHP Example

Figure 1 shows an example of a PHP application that is designed to illustrate the particular complexities of finding and localizing faults in PHP web applications. In particular, the figure shows: an `index.php` top-level script that contains static HTML in Figure 1(a), a generic login script `login.php` in Figure 1(c), and a skeleton of a data display script `view.php` in Figure 1(d). The two PHP scripts rely on a shared include file `constants.php` that defines some standard constants, which is shown in in Figure 1(b).

These fragments are part of the client-server work flow in a Web application: the user first sees the `index.php` page

of Figure 1(a) and enters credentials. The user-input credentials are processed by the script in Figure 1(c), which generates a response page that allows the user to enter further input—a topic—that in turn generates further processing by the script in Figure 1(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 1(c) will be seen by the code in Figure 1(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 5.

Our example program contains an error in the HTML produced for the administrative details: the `H2` tag that is opened on line 21 of Figure 1(d) is not closed. While this fault itself is trivial, finding it and localizing its cause is not. Assume that testing starts (as an ordinary user would) by entering credentials to the script in Figure 1(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 20–22 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 1(d) with the appropriate session state, which will finally generate HTML exhibiting the fault as is shown in Figure 2(a). Thus, finding the fault requires a careful selection of inputs to a series of interactive scripts, and tracking updates to session state during the execution of these scripts.

The next step is to determine the cause of the malformed HTML. Consider the two sources of information suggested in Section 2.3:

- Our validator produces the output shown in Figure 2(c) for this fault, indicating that lines 5 and 6 in the malformed HTML of Figure 2(a) are associated with the HTML failure. These lines correspond to the `H2` heading and the following `/BODY` tags, respectively. By correlating this information with the output mapping shown in Figure 2(b), we can determine that lines 21 and 27 in `view.php` produced these lines of output.
- The second source of information is obtained by comparing the statements executed in passing and failing runs. The HTML failure only occurs when `$type` is equal to '`admin`', and the difference between passing and failing runs therefore consists of all code that is guarded by the two conditionals on lines 11 and 20 in `view.php`. Consequently, we may conclude that the statements on lines 12, 14, and 21 are suspect.

Neither of these estimates is precise, since the fault is clearly in the printing of the `H2` line itself (line 21). We can, however, combine the results of the validator and the sets of statements. Specifically, we could observe that the printing of `/BODY` on line 27 in `view.php` occurs in both passing and failing executions, and is therefore unlikely to be the location of the fault. Furthermore, we can observe that lines 12 and 14, each of which is only executed in one of the executions, is not associated with the failure according to the information we received from the oracle. Therefore, we can conclude that the fault is most closely associated with line 21 in `view.php`.

3 Concolic Execution in the Presence of Interactive User Input

Our technique for finding failures in PHP applications is a variation on *concolic* (combined concrete and symbolic) execution [4, 7, 8, 19, 22], a well-established test generation technique. The basic idea behind this technique is to execute an application on some initial (e.g., empty or randomly-chosen) input, and then on additional inputs obtained by solving constraints derived from exercised control flow paths. Failures that occur during these executions are reported to the user.

In a previous paper [3], we described how this technique can be adapted to the domain of dynamic web applications written in PHP. The resulting *Apollo 2.0* tool takes into account language constructs that are specific to PHP, uses an oracle to validate the output, and supports database interaction. However, we previously relied on a *manual solution* for the challenging problem of interactive user input that we already described in Section 2: PHP applications typically generate HTML pages that contain user-interface features such as buttons that—when selected by the user—result in the execution of additional PHP scripts. Modeling such user input is important, because coverage of the application will typically remain very low otherwise. In our previous paper [3], we relied on a manually performed program transformation that translates interactive user input into additional script parameters. This manual step has several limitations:

- It was performed only once before the analysis, and thus did not take into account user input options that are created dynamically by the web application.
- More importantly, while *Apollo 1.0* was able to execute additional parts of the program, it did so without any knowledge of parameters that are transferred from one executable component to the next by persisting them in the environment, or sending them as part of the call.

```
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 1(d)*
1	1
2	2
3	4
4	12, 16
5	21
6	27
7	29

(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

Figure 2: **(a)** HTML produced by the script of Figure 1(d). **(b)** Output mapping constructed during execution. **(c)** Part of output of WDG Validator on the HTML of Figure 2(a)

.

In this paper, we replace this manual step with an automatic method that (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 HTML output produced by PHP scripts 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, we 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 scripts will present only vanilla information and terminate quickly (e.g., when the condition in line 5 of Figure 1(d) is false). The new automated approach has increased coverage and the number of faults found, and we envision it could be utilized in other tools as well (e.g., in the context of the work by Wassermann et al. [22], who use concolic execution to find SQL injection vulnerabilities in PHP applications).

3.1 Algorithm

Figure 3 shows pseudo-code for our algorithm, which extends the algorithm of *Apollo 1.0* [3] by tracking the state of the environment, and automatically discovering additional scripts based on an analysis of available user options. The inputs to the algorithm are: a program $\mathcal{P}$ composed of any number of executable components (PHP scripts), the initial state of the environment before executing any component (e.g, database), a set of executable components reachable from the initial state C, and an output oracle O. The output of the algorithm is a set of bug reports $\mathcal{B}$ for the program $\mathcal{P}$, according to O. Each bug report contains the identifying information about the failure (message, and generating program part), and the set of tests exposing the failure.

The algorithm uses a queue of tests. Each test contains the program component to execute, a *path constraint* which is a conjunction of conditions on the program's input parameters, and the environment state before the execution. The queue is initialized with one test for each of the components executable from the initial state, and the empty path constraint (lines 3– 5). The algorithm uses a constraint solver to find a concrete input that satisfies a path constraint from the selected test (lines 7–9). The algorithm restores the environment state (line 11), then executes the program component concretely on the input and checks if failures occurred (lines 12–14). Any detected failure is merged into the corresponding bug report (lines 15–16). Next, the program is executed symbolically on the same input (line 17). The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, that is fulfilled if the given path is executed (here, the path constraint reflects the path that was just executed). The algorithm then creates new test inputs by solving modified versions of the path constraint (lines 18–21) as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 19). 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. Finally, the algorithm analyzes the output to find new transitions from the new environment state (line 22). Each transition is expressed as a pair of path constraints and an executable component. The algorithm then adds new tests for each transition that was not explored before (line 23– 27).

3.2 Example

We will now illustrate the algorithm of Figure 3 using the example application of Figure 1. The inputs to the algorithm are: $\mathcal{P}$ is the code from Figure 1, the initial state of the environment is empty, C is the script in Figure 1(c), and O is the WDG HTML validator[1]. The algorithm begins on lines 3–5 by initializing the work queue with one item: the script of Figure 1(a) with an empty path constraint and an empty initial environment.

iteration 1. The first iteration of the outer loop (lines 6–27)

[1] http://htmlhelp.com/tools/validator/

```
   parameters: P Program, S0 Initial environment state, C Components
         executable from S0, O oracle;
   P,C : setOf(Executable component);
   result      : Bug reports B;
   B : setOf(⟨failure, setOf(T test)⟩);
   T : ⟨ Executable component, Path constraint, Environment State⟩
 1 B := ∅;
 2 pcQueue := emptyQueue();
 3 foreach component in C do
 4       test := ⟨component, emptyPathConstraint(), S0⟩;
 5       enqueue(pcQueue, test);
 6 while not empty(pcQueue) and not timeExpired() do
 7       test := dequeue(pcQueue);
 8       component := test.component;
 9       input := solve(test.pathConstraint);
10       if input ≠ ⊥ then
11             restoreState(test.state);
12             output := executeConcrete(component, input);
13             newState := getCurrentState();
14             failures := getFailures(O, output);
15             foreach f in failures do
16                   merge ⟨f, test⟩ into B;
17             c1 ∧ ... ∧ cn := executeSymbolic(component, input);
18             foreach i = 1,...,n do
19                   newPC := c1 ∧ ... ∧ ci−1 ∧ ¬ci;
20                   newTest := ⟨test.component, newPC, test.state⟩;
21                   enqueue(pcQueue, newTest);
22             ⟨PC1, component1⟩ ∧ ... ∧ ⟨PCm, componentm⟩ :=
               analyzeOutput(output);
23             foreach i = 1,...,m do
24                   newPC := c1 ∧ ... ∧ cn ∧ PCi;
25                   newTest := ⟨componenti, newPC, newState⟩;
26                   if pcQueue not contains newTest then
27                         enqueue(pcQueue, newTest);
28 return B;
```

Figure 3: The failure detection algorithm. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\perp$ if no satisfying input exists. The auxiliary functions *restoreState* and *getCurrentState* create a given environment state, or load the current state of the environment respectively. The *analyzeOutput* auxiliary function performs an analysis of the output to extract possible transitions from the current environment state. The output of the algorithm is a set of bug reports, each reports a failure and the set of tests exposing that failure.

removes that item from the queue (line 7), uses an empty input to satisfy the empty path constraint (line 9), restores the empty initial state (line 11), and executes the script (line 12).

No failures are observed, so the next few lines (line 13–16) do nothing. The call to *executeSymbolic* on line 17 returns an empty path constraint, so the function *analyzeOutput* on line 22 is executed next, and returns one user option; ⟨`login.php`, ∅, ∅⟩ for executing `login.php` with no input, and the empty state. This test is added to the queue (line 27).

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 17 returns a path constraint `user` $\neq$ `admin` $\wedge$ `user` $\neq$ `reg`, indicating that the call to `check_password` on line 8 in Figure 1(c) returned false[2]. Given this, the loop at lines 18–21 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 23—27 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 13. Also, this time *analyzeOutput* (line 22) finds the link to the script in Figure 1(d), and so the loop at line 23–27 adds one item to the queue, executing `view.php` with the current session state.

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 1(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 13. 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 1(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 1(d) with the session state that includes *type* results in an HTML failure.

There are a few other things that happen, but at this point the reader should note that we have observed one successful and one failing execution for the script in Figure 1(d). We will discuss in Section 4.5 how this information will be used for fault localization.

4 Fault Localization

In this section, we first review the *Tarantula* fault localization technique. We then present an alternative technique that is based on the output mapping and positional information obtained from an oracle. Finally, we present a technique that combines the former with the latter.

[2]For simplicity, we omit the details of this function. It compares user and password to some constants '`admin`' and '`reg`'.

4.1 *Tarantula*

Jones et al. [11, 12] presented *Tarantula*, a fault localization technique that associates with each statement a *suspiciousness rating* that indicates the likelihood that it contributes to a failure. The suspiciousness rating $S_{tar}(l)$ for a statement that occurs at line[3] l is a number between 0 and 1 that is defined as follows:

$$S_{tar}(l) = \frac{Failed(l)/TotalFailed}{Passed(l)/TotalPassed + Failed(l)/TotalFailed}$$

where $Passed(l)$ is the number of passing executions that execute statement l, $Failed(l)$ is the number of failing executions that execute statement l, *TotalPassed* is the total number of passing test cases, and *TotalFailed* is the total number of failing test cases. After suspiciousness ratings have been computed, each of the executed statements is assigned a *rank*, in order of decreasing suspiciousness. Ranks do not need to be unique: The rank of a statement l reflects the maximum number of statements that would have to be examined if statements are examined in order of decreasing suspiciousness, and if l were the last statement of that particular suspiciousness level chosen for examination.

Jones and Harrold [11] conducted a detailed empirical evaluation in which they apply *Tarantula* to faulty versions of the Siemens suite [10], and compare its effectiveness to that of several other fault localization techniques (see Section 7). The Siemens suite consists of several versions of small C programs into which faults have been seeded artificially. Since the location of these faults is given, one can evaluate the effectiveness of a fault localization technique by measuring its ability to identify these faults. In the fault localization literature, this is customarily done by reporting the percentage of the program that needs to be examined by the programmer, assuming statements are inspected in decreasing order of suspiciousness [1, 5, 11, 18].

Specifically, Jones and Harrold compute for each failing test run a *score* (in the range of 0%-100%) that indicates the percentage of the application's executable statements that the programmer need not examine in order to find the fault. This score is computed by determining a set of examined statements that initially contains only the statement(s) at rank 1. Then, iteratively, statements at the next higher rank are added to this set until at least one of the faulty statements is included. The score is now computed by dividing the number of statements in the set by the total number of executed statements. Using this approach, Jones and Harrold found that 13.9% of the failing test runs were scored in the 99-100% range, meaning that for this percentage of the failing tests, the programmer needs to examine less than 1% of the program's executed statements to find the fault. They also report that for an additional 41.8% of the failing tests, the programmer needs to inspect less than 10% of the executed statements.

[3] We use line numbers to identify statements, because that enables us to present the different fault localization techniques in a uniform manner.

4.2 Fault Localization using the Output Mapping

An oracle that determines whether or not a failure occurs can often provide precise information about which parts of the output are associated with that failure. For instance, an HTML validator will typically report the location of malformed HTML. Such information can be used as a heuristic to localize faults in the program, provided that we can determine which portions of the program produced which portions of the output. The basic idea is that the code that produced the erroneous output is a good place to start looking for the causative fault. This is formalized as follows. Assume we have the following two functions:

- $O_n(f)$ returns output line numbers reported by the oracle O for failure f, and
- $\mathcal{P}_n(o)$ returns the set of program parts of the source program responsible for output line o

Given these two functions, we define a suspiciousness rating $S_{map}(l)$ of the statement at line l for failure f as follows:

$$S_{map}(l) = \begin{cases} 1 & \text{if } l \in \bigcup_{o \in O_n(f)} \mathcal{P}_n(o) \\ 0 & \text{otherwise} \end{cases}$$

Note that this is a "binary" rating: program parts are either highly suspicious, or not suspicious at all.

4.3 Combined Technique

The *Tarantula* algorithm presented in Section 4.1 localizes failures based on how often statements are executed in failing and passing executions. However, in the web applications domain, a significant number of lines are executed in *both* cases, or only in failing executions. Thus, the fault localization technique presented in Section 4.2 can be used to enhance the *Tarantula* results by giving a higher rank to statements that are blamed by both *Tarantula* and the mapping technique. More formally, we define a new suspiciousness rating $S_{comb}(l)$ for the statement at line l as follows:

$$S_{comb}(l) = \begin{cases} 1.1 & \text{if } S_{map}(l) = 1 \wedge S_{tar}(l) > 0.5 \\ S_{tar}(S) & \text{otherwise} \end{cases}$$

Informally, we give the suspiciousness rating 1.1 to any statement that is identified as highly suspicious by the oracle, and for which *Tarantula* indicates that the given line is positively correlated with the fault (indicated by the fact that *Tarantula*'s suspiciousness rating is greater than 0.5).

line(s)	executes	$S_{tar}(l)$	$S_{map}(l)$	$S_{comb}(l)$
4,6,8,9,11	*both*	0.5	0.0	0.5
12	*failing only*	1.0	0.0	1.0
13	*passing only*	0.0	0.0	0.0
14	*passing only*	0.0	0.0	0.0
16, 20	*both*	0.5	0.0	0.5
21	*failing only*	1.0	1.0	1.1
27	*both*	0.5	1.0	0.5
28,29	*both*	0.5	0.0	0.5

Figure 4: Suspiciousness ratings for lines in the PHP script of Figure 1(d), according to three techniques. The columns of the table show, for each line l, when it is executed (in the passing run, in the failing run, or in both runs), and the suspiciousness ratings $S_{tar}(l)$, $S_{map}(l)$, and $S_{comb}(l)$.

4.4 Generating Inputs for *Tarantula*

As we discussed previously, *Tarantula* computes suspiciousness ratings using a formula that considers how many times a statement is executed by passing and failing executions. But *which* passing executions and failing executions should be supplied as inputs to *Tarantula*?

To answer this question, assume that the algorithm of Section 3 has exposed a number of failing executions. This set can be partitioned into subsets that pertain to the same failure. Here, two failures are assumed to be "equivalent" (i.e., due to the same fault) if the oracle produces the same message for them, and if the same program constructs are correlated with these messages according to the output mapping. In Section 6, we will conduct separate fault localization experiments for each subset of equivalent failing executions.

This leaves the question of what set of passing executions we should supply to *Tarantula* as inputs along with these failing executions. We currently consider two options:

1. We supply *all* passing executions that were identified by the algorithm of Section 3.
2. We supply a *randomly selected subset* of 10% of the passing tests that were identified by the algorithm of Section 3.

Note that the above strategies can be applied to both the *Tarantula* and the combined algorithms.

4.5 Example

As described in Section 3.2, the test input generation algorithm produced two runs of the script in Figure 1(d): one that exposed an HTML error and one that did not. Figure 4 shows the suspiciousness ratings $S_{tar}(l)$, $S_{map}(l)$, and $S_{comb}(l)$ that are computed for each line l in the PHP script in Figure 1(d), according to the three fault localization techniques under consideration.

To understand how the *Tarantula* ratings are computed, consider statements that are only executed in the passing run. Such statements obtain a suspiciousness rating of 0/(1+0) = 0.0. By similar reasoning, statements that are only executed in the failing run obtain a suspiciousness rating of 1/(0+1) = 1.0, and statements that are executed in both cases obtain a suspiciousness rating of 1/(1+1) = 0.5.

The suspiciousness ratings computed by the mapping-based technique can be understood by examining the output of the validator in Figure 2(c), along with the HTML in Figure 2(a) and the mapping from lines of HTML to the lines of PHP that produced them in Figure 2(b). The validator says the error is in line 5 or 6 of the output, and those were produced by lines 21 and 27 in the script of Figure 1(d). Consequently, the suspiciousness ratings for lines 21 and 27 is 1.0, and all other lines are rated 0.0 by the mapping-based technique. The suspiciousness ratings for the combined technique follow directly from its definition in Section 4.3.

As can be seen from the table, the *Tarantula* technique identifies lines 12 and 21 as the most suspicious ones, and the output mapping based technique identifies lines 21 and 27 as such. In other words, each of these fault localization techniques—when used in isolation—reports one non-faulty statement as being highly suspicious. However, the combined technique correctly identifies only line 21 as the faulty statement.

5 Implementation

We extended the *Apollo 1.0* tool [3] with the algorithm for combined concrete and symbolic execution in the presence of interactive user input and persistent session state that was presented in Section 3, and with the fault localization algorithm that was presented in Section 4. This section discusses some key features of the implementation.

interactive user input and session state. As was mentioned in Section 3, it is important to determine what PHP scripts the user may invoke by selecting buttons, checkboxes, etc. in the HTML output of previously executed scripts. To this end, *Apollo 2.0* automatically extracts the available user options from the HTML output. Each option contains the script to execute, along with any parameters (with default value if supplied) for that script. *Apollo 2.0* also analyzes recursive static HTML documents that can be called from the dynamic HTML output, i.e. *Apollo 2.0* traverses hyperlinks in the generated dynamic HTML that link to other HTML documents on the same site. To avoid redundant exploration of similar executions, *Apollo 2.0* perform state matching (performed implicitly in Line 26 of Figure 3) by not adding already-explored transitions.

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, to enable

program	version	#files	total LOC	PHP LOC
faqforge	1.3.2	19	1712	734
webchess	0.9.0	24	4718	2226
schoolmate	1.5.4	63	8181	4263
phpsysinfo	2.5.3	73	16634	7745
timeclock	1.0.3	62	20792	13879
phpBB2	2.0.21	78	34987	16993

Figure 5: Characteristics of subject programs. The columns of the table indicate (i) the version of the program we used, (ii) the number of source files with PHP scripts, (iii) the total number of lines in each program, and the number of lines with executable PHP code.

Apollo 2.0 to track constraints on input parameters in all scripts that use them.

web server integration. *Apollo 1.0* [3] only supported the execution of PHP scripts using the PHP command line interpreter. However, dynamic web applications often depend on information supplied by a web-server, and some PHP constructs are simply ignored by the command line interpreter (e.g., *header*). *Apollo 2.0* supports execution through the Apache web-server in addition to the stand-alone command line executor. A developer can use *Apollo 2.0* to silently analyze the execution and record any failure found while manually using the subject program on an Apache server.

6 Evaluation

This evaluation aims to answer two questions:

Q1. What is the effect of automatically simulating user input interaction on coverage and on the number of failures exposed?

Q2. How effective are the three fault localization techniques presented in Section 4 in practice?

6.1 Subject Programs

For the evaluation, we selected six open-source PHP programs (from `http://sourceforge.net`), for which the characteristics are shown in Figure 5. **faqforge** is a tool for creating and managing documents. **webchess** is an online chess game. **schoolmate** is an PHP/MySQL solution for administering elementary, middle, and high schools. **phpsysinfo** displays system information, e.g., uptime, CPU, memory, etc. **timeclock** is a web-based timeclock system. **phpBB2** is an open source discussion forum.

6.2 Coverage/Failures Detected

We ran *Apollo* with and without the simulation of user interaction 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 much affected by a different time limit. This time budget includes all experimental tasks. We measured line coverage, i.e., the ratio of the number of executed lines to the total number of lines with executable PHP code that was shown in Figure 5. Furthermore, we classified the discovered failures as execution failures and HTML failures, as was discussed previously in Section 2.2.

program	strategy	%cov	failures		
			exec.	HTML	total
faqforge	**No Simulated UI**	86.8	9	55	64
	Simulated UI	92.4	9	63	72
webchess	**No Simulated UI**	37.8	20	7	27
	Simulated UI	39.4	26	8	34
schoolmate	**No Simulated UI**	65.0	35	60	95
	Simulated UI	65.0	35	61	96
phpsysinfo	**No Simulated UI**	55.5	3	1	4
	Simulated UI	55.7	6	2	8
timeclock	**No Simulated UI**	3.2	2	30	32
	Simulated UI	14.1	2	117	119
phpBB2	**No Simulated UI**	11.4	3	1	4
	Simulated UI	28.0	5	19	24

Figure 6: Experimental results for 10-minute test generation runs. The table presents results each of the **No Simulated UI** and the **Simulated UI** strategies. The **%cov** column lists the line coverage achieved by the generated inputs. The next three columns show the number of execution errors, HTML failures, and the total number of failures.

Figure 6 tabulates the line coverage results and observed failureson the subject programs for each of the two test input generation strategies (with simulated user interaction and without). Although the effect varies, it is clear that the user input simulation allows *Apollo* to achieve better results on all subject programs. For example, on **phpsysinfo** the effect on coverage is marginal (55.7% vs. 55.5%) because this program is not interactive. On the other hand, for **phpBB2** the effect is significant for both the coverage obtained (28.0% vs. 11.4%), and for the number of failures detected (24 instead of 4), and similarly for **timeclock** because these applications only performs most operations when starting in the correct state (e.g., when the user is logged in).

6.3 Localizing Faults

To answer the second research question, we created a localized faults database by manually localizing up to 20 faults in 3 of the subject programs (**webchess** contained only 9 faults that caused the 34 failures reported previously). We then applied the three fault localization methods that were discussed in Section 4 to each failure f: (i) our implementation of *Tarantula* (see Section 4.1), (ii) a fault localization technique that uses only the output mapping (see Section 4.2), and (iii) a fault localization technique that combines *Tarantula* with the output mapping (see Section 4.3). As mentioned in Section 4.4, we tried two sets of inputs for each technique: (a) the set of executions exposing f in combination with *all* passing executions, and (b)

program	failing/passing	*Tarantula*	mapping	combined
webchess	**All**	19.00	25.12	6.94
	Random	24.35	25.12	7.20
schoolmate	**All**	29.94	15.06	5.09
	Random	30.21	15.06	5.10
timeclock	**All**	16.09	5.12	2.24
	Random	21.54	5.12	2.41

Figure 7: Average percentage of the program a developer would need to inspect in order to locate the failures using different fault localization techniques. The **failing/passing** column indicates the method that was used to select the sets of passing and failing tests (one of All, Random) used for the fault localization. *Tarantula* is the fault localization technique described in 4.1. **mapping** is the fault localization based only on the output mapping (see Section 4.2). **combined** is the combined fault localization technique described in 4.3.

the set of executions exposing f in combination with 10% of randomly selected passing executions.

We measure the effectiveness of these fault localization algorithms as the minimal number of statements that need to be inspected until all the faulty lines are detected, assuming that statements are examined in order of decreasing suspiciousness (Section 4.1). Figure 7 tabulates the results.

The results show that the **combined** technique is clearly superior to each of the *Tarantula* and mapping-based techniques that it builds upon. For **webchess**, the programmer would need to inspect 19.00% of the statements on average when *Tarantula* is supplied with all passing executions, 25.12% when the mapping-based technique is used, but only 6.94% using the combined technique. Using the same set of executions, the programmer needs to inspect 29.94% of **schoolmate**'s statements using *Tarantula*, 15.06% using the output mapping, and only 5.09% using the combined technique. Similar results are obtained for timeclock. The use of a randomly selected subset of the passing tests yields slightly worse results for each of the techniques.

7 Related Work

This section only presents a summary of the literature on fault localization, and in particular on fault localization techniques that use information from passing and failing executions to predict the likelihood that statements are responsible for failures. For a review of the literature on concolic execution, we refer the reader to our previous paper [3].

Early work on fault localization relied on the use of program slicing [21]. Lyle and Weiser [16] introduce *program dicing*, a method for combining the information of different program slices. The basic idea is that, when a program computes a correct value for variable x and an incorrect value for variable y, the fault is likely to be found in statements that are in the slice w.r.t. y, but not in the slice w.r.t. x. Variations on this idea technique were later explored by Pan and Spafford [17], and by Agrawal et al. [2].

In the spirit of this early work, Renieris and Reiss [18] use *set-union* and *set-intersection* methods for fault localization, that they compare with their *nearest neighbor* fault localization technique (discussed below). The set-union technique computes the union of all statements executed by passing test cases and subtracts these from the set of statements executed by a failing test case. The resulting set contains the suspicious statements that the programmer should explore first. In the event that this report does not contain the faulty statement, Renieris and Reiss propose an SDG-based ranking technique in which additional statements are considered based on their distance to previously reported statements along edges in a System Dependence Graph [9]. The set-intersection technique identifies statements that are executed by all passing test cases, but not by the failing test case, and attempts to address errors of omission, where the failing test case neglects to execute a statement.

The *nearest neighbors* fault localization technique by Renieris and Reiss [18] assumes the existence of a failing test case and many passing test cases. The technique selects the passing test case whose execution spectrum most closely resembles that of the failing test case according to one of two distance criteria[4], and reports the set of statements that are executed by the failing test case but not by the selected passing test case. In the event that the report does not contain the faulty statement, Renieris and Reiss use the SDG-based ranking technique mentioned above to identify additional statements that should be explored next. *Nearest Neighbor* was evaluated on the Siemens suite [10], a collection of small C programs for which faulty versions and a large number of test cases are available, and was found to be superior to the *set-union* and *set-intersection* techniques.

Cleve and Zeller [5, 25] present a fault-localization technique based on Delta Debugging [24], a binary search and minimization technique. Delta debugging is first employed to identify the variables responsible for a failure, by selectively introducing values that occur in the program state of a failing run into the state obtained during a passing run, and observing whether or not the failure reoccurs. Then, delta debugging is applied again in order to identify *cause transitions*, i.e., points in the program where one variable ceases to be the cause for a failure, and where another variable starts being the origin of that failure. Cleve and Zeller report finding a real failure in GCC using the technique, and also evaluate their work on the Siemens suite.

Dallmeier et al. [6] present a fault localization technique in which differences between method call sequences that

[4] One similarity measure defines the distance between two test cases as the cardinality of the symmetric set difference between the statements that they cover. The other measure considers the differences in the relative execution frequencies.

occur in passing and failing executions are used to identify suspicious statements. They evaluate the technique on buggy versions of the NanoXML Java application.

Two recent papers by Jones and Harrold [11] and by Abreu et al. [1] present empirical evaluations of several fault localization techniques, including several of the techniques discussed above, using the Siemens suite. Yu et al. [23] evaluated the sensitivity of several of the fault localization techniques discussed above to test suite reduction. Here, the goal was to determine to what extent the effectiveness of fault localization techniques was reduced as a result of applying several test-suite minimization techniques.

Other fault localization techniques analyze statistical correlations between control flow predicates and failures (see, e.g., [14, 15]), and correlations between changes made by programmers and test failures [20].

In this paper, we apply the *Tarantula* technique in a different domain (open-source web applications written in PHP instead of C programs), and adapted it to take into account positional information that we obtained from the PHP interpreter. Instead of using artificially seeded faults such as the ones in the Siemens suite, we study real faults that were exposed by our *Apollo 2.0* tool. Moreover, we do not use an existing test suite but rely on *Apollo 2.0* to generate a large number of (passing and failing) test cases instead.

8 Conclusions and Future Work

We have presented an approach for failure detection and fault localization thats leverages concolic execution [4, 7, 8, 19, 22] and the *Tarantula* algorithm [11, 12] to automatically find and localize failures in PHP web applications. Our algorithm adapts concolic execution to the domain of web applications by performing dynamic simulation of user interaction in different environment states.

Unlike previous fault localization methods, ours does not require a test-suite with passing and failing test cases to be available up front. We use an output mapping between PHP statements and the output they produce in combination with positional information about HTML errors obtained from the oracle to improve on *Tarantula*'s fault localization.

We implemented the technique in *Apollo 2.0*. In experiments on 6 open-source PHP applications, we found that our new automatic method for simulating user input significantly improved line coverage and the number of failures found. We also found that a fault localization technique that *combines Tarantula* with information retrieved from the output mapping is significantly more precise than either *Tarantula* or the output mappings alone.

The main topic for future work is to explore the use of concolic execution to generate passing test cases that are highly similar to failing test cases, to further improve the effectiveness of *Tarantula*.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC 2006*, pages 39–46, 2006.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, pages 143–151, Toulouse, France, 1995.

[3] 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.

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

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, May 2005.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP*, pages 528–550, 2005.

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

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

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

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[11] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI'05*, pages 15–26, 2005.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *FSE*, pages 286–295, 2005.

[16] J. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, Beijing (Peking), China, 1987.

[17] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, July 1992.

[18] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

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

[20] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *FSE*, pages 57–68, Portland, OR, USA, Nov. 7–9, 2006.

[21] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[22] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.

[23] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *ICSE*, pages 201–210, 2008.

[24] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, September 1999.

[25] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10. ACM Press, November 2002.