Practical Fault Localization for Dynamic Web Applications

Shay Artzi Julian Dolby Frank Tip Marco Pistoia

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

ABSTRACT

We leverage combined concrete and symbolic execution and several fault-localization techniques to create a uniquely powerful tool for localizing faults in PHP applications. The tool automatically generates tests that expose failures, and then automatically localizes the faults responsible for those failures, thus overcoming the limitation of previous fault-localization techniques that a test suite be available upfront. The fault-localization techniques we employ combine variations on the *Tarantula* algorithm with a technique based on maintaining a mapping between statements and the fragments of output they produce. We implemented these techniques in a tool called *Apollo*, and evaluated them by localizing 75 randomly selected faults that were exposed by automatically generated tests in four PHP applications. Our findings indicate that, using our best technique, 87.7% of the faults under consideration are localized to within 1% of all executed statements, which constitutes an almost five-fold improvement over the *Tarantula* algorithm.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification ; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

1. INTRODUCTION

Web applications are typically written in a combination of several programming languages, such as JavaScript on the client side, and PHP with embedded SQL commands on the server side. Such applications generate structured output in the form of dynamically generated HTML pages that may refer to additional scripts to be executed. As with any program, programmers make mistakes and introduce faults, resulting in Web-application crashes and malformed dynamically generated HTML pages. While malformed HTML errors may seem trivial, and indeed many of them are at worst minor annoyances, they have on occasion been known to enable serious attacks such as denial of service[1]. We present the first fully automatic tool that finds and localizes malformed HTML errors in Web applications that execute PHP code on the server side.

In previous work [3, 4], we adapted the well-established technique of combined concrete and symbolic execution [9, 25, 5, 10, 28] to Web applications written in PHP. With this approach, an application is first executed on an empty input, and a *path condition* is recorded that reflects the application's control-flow predicates that have been executed that depend on input. Then, by changing one of the predicates in the path condition, and solving the resulting condition, new inputs can be obtained, and executing the program on these inputs will result in additional control-flow paths being exercised. In each execution, faults that are observed during the execution are recorded. This process is repeated until either sufficient coverage of the statements in the application has been achieved, a sufficient number of faults has been detected, or the time budget is exhausted. Our previous work addresses a number of issues specific to the domain of PHP applications that generate HTML output. In particular: (i) it integrates an HTML validator to check for errors that manifest themselves by the generation of malformed HTML, (ii) it automatically simulates interactive user input, and (iii) it keeps track of the interactive session state that is shared between multiple PHP scripts. We implemented these techniques in a tool called *Apollo*. In previous experiments on 6 open-source PHP applications, *Apollo* found a total of 673 faults [4].

A problem not addressed by our previous work is that it fails to pinpoint the specific Web-application instructions that cause execution errors or the generation of malformed HTML code. Without that information, correcting these types of issues can be very difficult. This paper addresses the problem of determining *where in the source code* changes need to be made in order to fix the detected failures. This task is commonly referred to as *fault localization*, and has been studied extensively in the literature [30, 15, 23, 16, 7, 14]. Our use of combined concrete and symbolic execution to obtain passing and failing runs overcomes the limitation of many existing fault-localization techniques that a test suite with passing and failing runs be available upfront. The fault-localization techniques explored in this paper combine variations on the *Tarantula* algorithm by Jones *et al.* [15, 14] with the use of an output mapping.

The first main ingredient of our combined approach is the *Tarantula* algorithm by Jones, *et al.* [15, 14], which predicts statements that are likely to be responsible for failures. It does so by computing, for each statement, the percentages of passing and failing tests that execute that statement. From this, a *suspiciousness rating* is computed for each executed statement. Programmers are encouraged to examine the statements in order of decreasing suspicious-

[1]For example `http://support.microsoft.com/kb/810819`.

ICSE '10, May 2-8 2010, Cape Town, South Africa

ness. This has proved to be quite effective in experiments with the Siemens suite [12], consisting of versions of small C programs into which artificial faults have been seeded [14]. A variation on the basic *Tarantula* approach that we consider is an enhanced domain for conditional statements, which enables us to more accurately pinpoint errors due to missing branches in conditional statements. The second main ingredient of our approach is the use of an *output mapping* from statements in the program to the fragments of output they produce. This mapping—when combined with the report of the HTML validator, which indicates the parts of the program output that are incorrect—provides an additional source of information about possible fault locations, and is used to fine-tune the suspiciousness ratings provided by *Tarantula*.

We implemented several fault-localization techniques that combine variations on *Tarantula* with the use of the output mapping in *Apollo*, making *Apollo* a fully automated tool for failure detection and fault localization for Web applications written in PHP. We then investigated *Apollo*'s ability to localize 75 randomly selected faults that were exposed by automatically generated tests in 4 open-source PHP applications. Using the basic *Tarantula* technique, the programmer had to examine an average of 13.2% of an application's executed statements to find each of the 75 faults, when exploring the executed statements in order of decreasing suspiciousness. Using our best technique, which augments the domain of *Tarantula* for conditional statements and uses the output mapping to fine-tune *Tarantula*'s suspiciousness ratings, the programmer needs to explore only 2.2% of the executed statements, on average. More significantly, using our best technique, 87.7% of the 75 faults under consideration are localized to within 1% of all executed statements, which constitutes an almost five-fold improvement over the 17.8% for the basic *Tarantula* algorithm.

To summarize, the contributions of this paper are as follows:

1. We present an approach for fault localization that uses combined concrete and symbolic execution to generate a suite of passing and failing tests. This overcomes the limitation of previous methods by not requiring the upfront availability of a test suite.
2. We demonstrate that automated techniques for fault localization, which were previously only evaluated on programs with artificially seeded faults, is effective at localizing real faults in open-source PHP applications.
3. We present 6 fault localization techniques that combine variations on the *Tarantula* algorithm with the use of an output mapping from statements to the fragments of program output that they produce.
4. We implemented these 6 techniques in *Apollo*, an automated tool for detecting failures and localizing faults in PHP.
5. We used *Apollo* to localize 75 randomly selected faults in 4 PHP applications and compared the effectiveness of the 6 fault localization techniques. Our findings show that, using our best technique, 87.7% of the faults are localized to within 1% of all executed statements, which constitutes an almost five-fold improvement over the *Tarantula* algorithm.

2. PHP WEB APPLICATIONS

PHP is a widely used scripting language 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 program in which data and control flow interactively between a server, which runs PHP scripts, and a client, which is a Web browser. The PHP scripts generate HTML code, which gets pushed to the client. Such code often includes forms that invoke other PHP scripts and pass them a combination of user input and constant values taken from the generated HTML.

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 those 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`. Other examples of the dynamic nature of PHP are the presence of the `isset()` function, which checks whether a variable has been defined, and the fact that statements defining classes and functions may occur anywhere.

The code in Figure 1 illustrates the flavor of a PHP Web application and the difficulty in localizing faults. As can be seen, 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 occurred in a `print` statement. The `require` statements resemble the C `#include` directive by causing the inclusion of code from another source file. However, while `#include` in C is a preprocessor directive that assumes a constant argument, `require` in PHP is an ordinary statement in which the filename is computed at run time; for example, the arguments of the `require` statements in line 6 of the PHP script of Figure 1(c) and in line 6 of the PHP script of Figure 1(d). are dynamically computed at run time based on the output of the `dirname` function, which returns the directory component of a filename. Similarly, `switch` labels in PHP need not be constant but, unlike in other languages, can be dynamically determined at run time. 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 render programs prone to code-quality problems that are difficult to localize.

2.2 Failures in PHP Programs

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

- **Execution Failures.** These are caused by missing included files, incorrect SQL queries and uncaught exceptions. Such failures are easily identified since 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.** These involve situations in which generated HTML code is not syntactically correct, causing them to be rendered incorrectly in certain browsers. This may not only lead to portability problems, but also decrease performance since the resulting pages may render slower when browsers attempt to compensate for the malformed HTML code.

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  <HTML>
2    <HEAD><TITLE>Login</TITLE></HEAD>
3    <BODY>
4      <FORM METHOD = "post" NAME = "login" ACTION = "login.php">
5        User Name: <INPUT TYPE = "text" NAME = "user"/><BR>
6        Password: <INPUT TYPE = "password" NAME = "pw"/><BR>
7        <INPUT TYPE = "submit" VALUE = "submit" NAME = "Submit"/>
8      </FORM>
9    </BODY>
10 </HTML>
```

(a) index.php

```
1  <?php
2  $userTag = 'user';
3  $pwTag = 'pw';
4  $typeTag = 'type';
5  $topicTag = 'topic';
6  $authTag = 'auth';
7  function check_password($user, $pw) {
8    /* authentication code... */
9  }
10 ?>
```

(b) constants.php

```
1  <HTML>
2    <HEAD><TITLE>Topic Selection Page</TITLE></HEAD>
3    <BODY>
4  <?php
5  session_start();
6  require(dirname(__FILE__).'/includes/constants.php');
7  $user = $_REQUEST[$userTag];
8  $pw = $_REQUEST[$pwTag];
9  $_SESSION[$authTag] = false;
10 if (check_password($user, $pw)) {
11 ?>
12     <FORM action="view.php">
13       Topic: <INPUT TYPE="text" NAME="topic"/><BR>
14       <INPUT TYPE="submit" VALUE="submit" NAME="Submit"/>
15     </FORM>
16   </BODY>
17 <?php
18   $_SESSION[$userTag] = $user;
19   $_SESSION[$pwTag] = $pw;
20   $_SESSION[$authTag] = true;
21   if ($user == 'admin') {
22     $_SESSION[$typeTag] = 'admin';
23   } else {
24     $_SESSION[$typeTag] = 'regularUser';
25   }
26 }
27 ?>
28 </HTML>
```

(c) login.php

```
1  <HTML>
2    <HEAD><TITLE>Topic View</TITLE></HEAD>
3
4  <?php
5  session_start();
6  require(dirname(__FILE__).'/includes/constants.php');
7  print "<BODY>\n";
8  if ($_SESSION[$authTag]) {
9    $type = $_SESSION[$typeTag];
10   $topic = $_REQUEST[$topicTag];
11   if ($type == 'admin') {
12     print "<H1>Administrative ";
13   } else {
14     print "<H1>Normal ";
15   }
16   print "View of topic " . $_REQUEST[$topicTag] . "</H1>\n";
17   if ($type == 'admin') {
18     print "<H2>Administrative Details\n";
19     /* code to print administrative details... */
20   } else {
21     print "<H2>Details</H2>\n";
22     /* code to print normal details... */
23   }
24 }
25 print "</BODY>\n";
26 ?>
27
28 </HTML>
```

(d) view.php

Figure 1: Sample PHP Web Application

1. For HTML failures, HTML validators provide the problematic locations in the HTML code. Malformed HTML fragments can then be correlated with the portions of the PHP 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 runs can then provide clues that can help localizing the fault. The extensive literature on fault-localization algorithms that exploit such information is discussed in Section 7.

2.4 Motivating 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. The top-level `index.php` script in Figure 1(a) contains static HTML code. The `login.php` script in Figure 1(c) attempts to authenticate the user. The `view.php` script in Figure 1(d) is for data display. The two PHP scripts `login.php` and `view.php` rely on a shared `constants.php` include file, shown in Figure 1(b); this file defines some standard constants and an authentication function.

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 the required credentials. The user-input credentials are processed by the `login.php` script in Figure 1(c). This script generates a response page that allows the user to enter further input, causing further processing by the `view.php` 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 in `$_SESSION[$typeTag]`. This illustrates how PHP handles *session state*, which is data that persists from one page to another to keep track of the interactions with the application 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 in line 8 to verify the username and password provided by the user.

Our sample program contains an error in the HTML code it produces: the `H2` tag opened in line 18 of Figure 1(d) is never closed. While this fault itself is trivial, localizing its cause is not. An ordinary tester would likely start executing this application by entering credentials into the script in Figure 1(c). The tester must then discover that setting `$user` to the value `admin` results in the selection of a different branch that records the user type `$typeTag` as `admin` in the session state, as shown in lines 21–23 of `login.php`. After that, the tester would have to enter a topic in the form generated by the `login.php` script, and would then proceed to execute the code in Figure 1(d) with the appropriate session state, which will finally generate the HTML code exhibiting the fault, as shown in Figure 2(a). Thus, finding the fault requires careful selection of inputs to a series of interactive scripts, and tracking updates to the session state during the execution of those scripts.

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

1. Our validator produces the output shown in Figure 2(c) for this fault, indicating that lines 5 and 6 in the malformed HTML of

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

(a) HTML output

HTML line	*PHP lines in 1(d)*
1	1
2	2
3	7
4	12, 16
5	18
6	25
7	28

(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)

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 18 and 25 in `view.php` produced these lines of output.

2. The second source of information is obtained by comparing the statements executed in passing and failing runs. The HTML failure only occurs when the value of `$type` is `admin`. Thus, the difference between passing and failing runs consists of all code that is guarded by the two conditionals in lines 11 and 17 in `view.php`. We can conclude that the statements in lines 12, 14, 18 and 21 are suspect.

Neither of these estimates is precise, because the fault is clearly the omission of the printing of an `/H2` tag in line 18. We can, however, combine the results of the validator and the sets of statements executed in passing and failing runs. Specifically, we could observe that the printing of `/BODY` in line 25 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, are 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 18 in `view.php`.

Another thing to observe about the PHP Web application in Figure 1 is that the `login.php` script in Figure 1(c) has an `if` statement in line 10 for which there is no matching `else` branch. This implies that the `BODY` tag is closed (in line 16) only if the authentication check of line 10 succeeds; if that check fails, the `BODY` tag will never be closed, giving rise to a malformed HTML page. This problem may not be discovered during testing, since it manifests itself only if invalid authentication credentials are provided. Furthermore, since the code that should close the `BODY` tag is missing, there is no line that is only executed by a failing run, and the *Tarantula* fault-localization technique will fail to pinpoint the exact program point responsible for the malformed HTML code. This paper introduces a novel condition-modeling technique to address such *errors by omission*. In Section 4.4, we will discuss condition modeling and its applicability to the example of Figure 1.

Note that, due to the necessarily small size of this example, the HTML errors it contains are localized and could likely be found with grep or a similar tool; however, in more complex applications, we encounter HTML errors that combine HTML generated by multiple statements that are not all in one place.

3. COMBINED CONCRETE AND SYMBOLIC EXECUTION

Our technique for finding failures in PHP applications is a variation on combined concrete and symbolic execution [9, 25, 5, 10, 28], 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. Our *Apollo* tool takes into account language constructs that are specific to PHP, uses an oracle to validate the output, and supports database interaction. In [4], we extended the work to address interactive user input (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. *Apollo* tracks the state of the environment, and automatically discovers additional scripts that the user may invoke based on an analysis of available user options. This is important because a script is much more likely to perform complex behavior when executed in the correct context (environment). For example, if a 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 8 of Figure 1(d) is false).

The inputs to *Apollo*'s 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 identification information of the failure (message, and generating program part), and the set of tests exposing the failure.

The algorithm uses a queue of tests[2]. Each test is comprised of three components: (i) the program component to execute, (ii) a *path constraint* which is a conjunction of conditions on the program's input parameters, and (iii) 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. The algorithm then processes each element of this queue as follows:

1. Using a constraint solver to find a concrete input that satisfies a path constraint from the selected test.
2. Restoring the environment state, then executing the program component on the input and checking for failures. Detected failures are merged into the corresponding bug reports. The program is also executed symbolically on the same input. The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, which

[2]The criteria of selecting tests from the queue prefers tests that will cover additional code. More details can be found in [4].

is satisfied if the given path is executed (here, the path constraint reflects the path that was just executed).
3. Creating new test inputs by solving modified versions of the path constraint as follows: for each prefix of the path constraint, the algorithm negates the last conjunct. 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.
4. Analyzing the output to find new transitions (referenced scripts, and parameter values) from the new environment state. Each transition is expressed as a pair of path constraints and an executable component.
5. Adding new tests for each transition not explored before.

For instance, an execution of *login.php* that did not define `$user` would generate a path constraint noting that `$user` is not set, i.e. !*isset*(*$user*). A subsequent execution could be constructed by negating this constraint to *isset*(*$user*). An execution satisfying this new constraint will define `$user` to some value.

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. Next, a technique is presented that combines the former with the latter. Finally, we discuss how the use of an extended domain for conditional expressions can help improve *Tarantula*'s effectiveness.

4.1 Tarantula

Jones, *et al.* [15, 14] presented *Tarantula*, a fault-localization technique that associates with each statement a *suspiciousness rating* that indicates the likelihood for that statement to contribute 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, 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 the order of decreasing suspiciousness. Ranks need not be unique: The rank of the 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 [14] conducted a detailed empirical evaluation in which they apply *Tarantula* to faulty versions of the Siemens suite [12], 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 those faults is given, one can evaluate the effectiveness of a fault-localization technique by measuring its ability to identify those 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 that statements are inspected in decreasing order of suspiciousness [7, 1, 23, 14].

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

More 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.

4.2 Fault Localization Using Output Mapping

An oracle that determines whether or not a failure occurs can often provide precise information about the parts of the output that are associated with that failure. For instance, given an HTML page, an HTML validator will typically report the locations in the corresponding HTML code where the code is syntactically incorrect. Such information can be used as a heuristic to localize faults in the program, provided that it is possible to determine which portions of the program produced the faulty 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:

- $O(f)$ returns output line numbers reported by the oracle O for failure f, and
- $\mathcal{P}(o)$ 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:

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

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

This output mapping depends on an oracle that can provide a good mapping of an error to the location that generated it; the HTML validator is a good example of such an oracle, but, in general, not all errors will have such an oracle available. Thus, we combine this approach with others to handle the full range of errors.

4.3 Tarantula with Output Mapping

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-application domain, a significant number of lines are executed in *both* cases, or only in failing executions. 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

line(s)	executed by	$S_{tar}(l)$	$S_{map}(l)$	$S_{comb}(l)$
5,6,7,8,9,10,11	*both*	0.5	0.0	0.5
12	*failing only*	1.0	0.0	1.0
13,14	*passing only*	0.0	0.0	0.0
16,17	*both*	0.5	0.0	0.5
18	*failing only*	1.0	1.0	1.1
20,21	*passing only*	0.0	0.0	0.0
25	*both*	0.5	1.0	0.5

Figure 3: 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 $S_{comb}(l)$.

Tarantula indicates that the given line is positively correlated with the fault (indicated by a *Tarantula* suspiciousness rating greater than 0.5).

Example.

As described in Section 2.4, the test input generation algorithm produces two runs of the script in Figure 1(d): one that exposes an HTML error and one that does not. Figure 3 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 the passing and the failing run 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 18 and 25 in the script of Figure 1(d). Consequently, the suspiciousness ratings for lines 18 and 25 are 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 18 as the most suspicious ones, and the output mapping based technique identifies lines 18 and 25 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 18 as the faulty statement.

4.4 Tarantula with Condition Modeling

As we observed in Section 4.1, the *Tarantula* algorithm works by associating a suspiciousness rating with each statement present in the program under analysis. Sometimes, however, it is the absence of a statement that causes a failure. For example, a `switch` statement in which the `default` case is omitted can cause a failure if the missing `default` case was supposed to close certain HTML tags. Similarly, an `if` statement for which the matching `else` branch is missing can cause the resulting HTML code to be malformed if the `boolean` predicate in the `if` statement is `false`, as we noticed in Section 2.4 when discussing the `if` statement in line 10 of the `login.php` script. The *Tarantula* fault-localization technique, as previously applied to statements, cannot rank a missing statement since that will never be executed.

We enhance *Tarantula*'s effectiveness by employing a new *condition-modeling* technique. This new technique uses an augmented domain for modeling conditional statements: instead of assigning a suspiciousness rating and rank to a conditional statement itself, it assigns a rating and rank to pairs of the form (*statement*, *index of first* `true` *case*).

The number of pairs associated with a `switch` statement is equal to the number of cases in the statement plus 1. For example, if a `switch` statement s has three `case` predicates, then the pairs considered by the condition-modeling technique are as follows:

1. $(s, 0)$, modeling the fact that all `case` predicates evaluate to `false`, causing the `default` branch—if it exists—to be executed
2. $(s, 3)$, modeling the fact that both the first and second `case` predicates evaluate to `false`, and the third one to `true`
3. $(s, 2)$, modeling the fact that the first `case` predicate evaluates to `false` and the second one to `true`
4. $(s, 1)$, modeling the fact that the first `case` predicate evaluates to `true`

If s is an `if` statement, there are two pairs associated with s:

1. $(s, 0)$, modeling the fact that the predicate evaluates to `false`
2. $(s, 1)$, modeling the fact that the predicate evaluates to `true`

After computing suspiciousness ratings for all pairs $(s, \cdots)$, the conditional statement s is assigned the maximum of these ratings, from which its rank is computed in the normal manner. This technique allows us to rank a `switch` statement with a missing `default` case and an `if` statement with a missing `else` branch, as explained in the following example.

Example.

In the `login.php` script of Figure 1(c), if s is the `if` statement in line 10, then $(s, 1)$ is going to be assigned rank 0 because when its predicate is `true`, s is never going to participate in a faulty execution. On the other hand, $(s, 0)$ is assigned rank 1 because executing s with its predicate set to `false` leads to a faulty execution, as discussed in Section 2.4. Our enhancement of *Tarantula* with condition modeling will assign to s the higher of the two ranks, 1. This is in contrast to the rank 0.5 that the statement-based *Tarantula* algorithm would have assigned to s.

5. IMPLEMENTATION

In Apollo [4], we implemented a *shadow interpreter* based on the Zend PHP interpreter 5.2.2[4] that simultaneously performs concrete program execution using concrete values, and a symbolic execution that uses symbolic values that are associated with variables. We implemented the following extensions to the shadow interpreter to support fault localization:

- **Statement Coverage.** All fault localization techniques based on *Tarantula* use the percentage of failing and passing tests executing a given statement to calculate the statement's suspiciousness score. Our shadow interpreter records the set of executed statements for each execution by hooking into the `zend_execute` and `compile_file` methods.
- **HTML Validator.** Apollo has been configured to use one of the following HTML validators as an oracle for checking HTML output: the Web Design Group (WDG) HTML validator[5] and the CSE HTML Validator V9.0[6].

[4] `http://www.php.net/`
[5] `http://htmlhelp.com/tools/validator/`
[6] `http://www.htmlvalidator.com/`

program	version	#files	PHP LOC	#downloads
faqforge	1.3.2	19	734	14,164
webchess	0.9.0	24	2,226	32,352
schoolmate	1.5.4	63	4,263	4,466
timeclock	1.0.3	62	13,879	23,708

Table 1: Characteristics of subject programs. The #files column lists the number of `.php` and `.inc` files in the program. 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`.

- **Output Mapping.** The output mapping technique, described in Section 4.2, localizes a fault found in the output to the statements producing the erroneous output part. Our shadow interpreter creates the mapping by recording the line number of the originating PHP statement whenever output is written out using the `echo` and `print` statements. The producing statement is found in the map using the positional information reported by an oracle checking the output for faults.
- **Condition Modeling.** Our shadow interpreter records the results of all comparisons in the executed PHP script for the conditional modeling technique, as described in Section 4.4. For each comparison, it records a pair consisting of the statement's line number and the relevant boolean result. The only exception is the execution of a `switch` statement. For this, the shadow interpreter stores the set of results for all executed `case` blocks together with the `switch` line number.

6. EVALUATION

This evaluation aims to answer two questions:

Q1. How effective is the *Tarantula* [14] fault localization technique in the domain of PHP web applications?

Q2. How effective is *Tarantula*, when combined with the use of an *output mapping* and/or when modeling the outcome of conditional expressions, as presented in Section 4?

6.1 Subject Programs

For the evaluation, we selected 4 open-source PHP programs[7]:

- **faqforge** is a tool for creating and managing documents.
- **webchess** is an online chess game.
- **schoolmate** is a PHP/MySQL solution for administering elementary, middle, and high schools.
- **timeclock** is a web-based timeclock system.

Figure 1 presents some characteristics of these programs.

6.2 Methodology

In order to answer our research questions, a set of localized faults, and a test suite exposing them is needed for each subject program. Since neither a test suite nor a set of known faults exists for our subject programs, we use *Apollo*'s combined concrete and symbolic execution technique that was presented in Section 3 to generate a test suite, and to detect failures. For this initial experiment, we gave the test generator a time budget of 20 minutes, and during this time hundreds of tests were generated and many failures were found for each subject program.

In order to investigate the effectiveness of an *automatic* fault localization technique such as *Tarantula*, it is necessary to know where faults are located. Unlike previous research on automated fault localization techniques [15, 14, 24], where the location of faults was known (e.g., because faults were seeded), we did not know where the faults were located, and therefore needed to localize them manually. Manually localizing and fixing faults is a very time-consuming task, so we limited ourselves to 20 faults in each of the subject programs. In **webchess**, only 15 faults were found to cause the 35 failures, so we use a total of 75 faults as the basis for the experiments discussed in this section. For each fault, we devised a patch and ensured that applying this patch fixed the problem, by running the tests again, and making sure that the associated failures[8] did not recur. Table 2 summarizes the details of the generated test suites, and the localized faults used in the remainder of this section. We used the following fault localization techniques to assign suspiciousness ratings to all executed statements:

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

program	tests	failures			localized faults		
		HTML	exec.	total	HTML	exec.	total
faqforge	748	121	9	130	17	3	20
webchess	503	15	20	35	8	7	15
schoolmate	583	105	42	147	18	2	20
timeclock	562	435	3	438	19	1	20
total	2393	676	74	750	62	13	75

Table 2: Characteristics of the test suites, failures and localized faults in the subject programs. The columns of the table indicate: (i) the subject program, (ii) the number of tests in the test suite generated for that program, (iii) the number of failures exposed by the test suite (three columns: HTML failures, execution errors, total), and (iv) the number of faults manually localized for that program (three columns: HTML faults, execution faults, total).

$\mathcal{T}$ The *Tarantula* algorithm that was presented in Section 4.1

$\mathcal{O}$ The technique of Section 4.2 based on using an output mapping in combination with positional information obtained from an oracle (HTML validator).

$\mathcal{T}$+$\mathcal{O}$ The combined technique described in Section 4.3 that combines *Tarantula* with the use of the output mapping.

$\mathcal{TC}$ The variation on *Tarantula* presented in Section 4.4 in which conditional expressions are modeled as (condition, value) pairs.

$\mathcal{TC}$+$\mathcal{O}$ The variation on *Tarantula* presented in Section 4.4, combined with the use of the output mapping.

$\mathcal{TC}$ **or** $\mathcal{O}$ A combined fault localization technique that uses $\mathcal{TC}$ for execution errors, and $\mathcal{O}$ for HTML failures.

We computed suspiciousness ratings separately for each localized fault, by applying each of these fault localization techniques to a test suite that comprised the set of failing tests associated with the fault under consideration, and the set of all passing tests.

Similar to previous fault-localization studies [15, 7, 14, 24], we measured the effectiveness of a fault localization algorithm as the minimal number of statements that needs to be inspected until the first faulty line is detected, assuming that statements are examined in order of decreasing suspiciousness.

6.3 Results

Table 3 shows experimental results for each of the six techniques ($\mathcal{T}$, $\mathcal{O}$, $\mathcal{T}$+$\mathcal{O}$, $\mathcal{TC}$, $\mathcal{TC}$+$\mathcal{O}$ and $\mathcal{TC}$ **or** $\mathcal{O}$) discussed above. The table shows, for each subject program (and for the subject programs in aggregate) a group of six rows of data, one for each technique. Each row shows, from left to right, the average number (percentage) of statements that needs to be explored to find each fault, followed by 11 columns of data that show how many of the faults were localized by exploring up to 1% of all statements, up to 10% of all

[8] In general, a single fault may be responsible for multiple failures.

program	technique	# statements(%)	0-1	1-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100
	$\mathcal{T}$	54.6(7.7)	15.0	50.0	35.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	O	142.5(20.1)	80.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	20.0
faqforge	$\mathcal{T}$+O	9.3(1.3)	85.0	10.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{TC}$	58.4(8.2)	10.0	55.0	35.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{TC}$ **or** O	79.5(11.2)	75.0	10.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.0
	$\mathcal{TC}$+O	7.4(1.0)	85.0	10.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{T}$	203.1(23.0)	15.4	30.8	0.0	15.4	23.1	7.7	0.0	0.0	0.0	0.0	0.0
	O	339.8(38.5)	61.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	38.5
webchess	$\mathcal{T}$+O	135.8(15.4)	69.2	0.0	0.0	7.7	7.7	7.7	0.0	0.0	0.0	0.0	0.0
	$\mathcal{TC}$	139.1(15.8)	30.8	30.8	0.0	0.0	30.8	0.0	7.7	0.0	0.0	0.0	0.0
	$\mathcal{TC}$ **or** O	133.8(15.2)	53.8	23.1	0.0	0.0	7.7	0.0	7.7	0.0	0.0	0.0	7.7
	$\mathcal{TC}$+O	61.8(7.0)	84.6	0.0	0.0	0.0	7.7	0.0	7.7	0.0	0.0	0.0	0.0
	$\mathcal{T}$	508.5(18.2)	25.0	45.0	10.0	0.0	0.0	0.0	0.0	10.0	0.0	5.0	5.0
	O	976.5(35.0)	65.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	35.0
schoolmate	$\mathcal{T}$+O	149.5(5.4)	80.0	15.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0
	$\mathcal{TC}$	382.8(13.7)	30.0	45.0	10.0	0.0	0.0	0.0	0.0	10.0	0.0	0.0	5.0
	$\mathcal{TC}$ **or** O	837.5(30.0)	70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	30.0
	$\mathcal{TC}$+O	9.1(0.3)	90.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{T}$	136.5(3.8)	15.0	85.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	O	360.0(10.0)	90.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.0
timeclock	$\mathcal{T}$+O	18.3(0.5)	90.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{TC}$	138.5(3.9)	15.0	85.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{TC}$ **or** O	183.1(5.1)	90.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0
	$\mathcal{TC}$+O	18.4(0.5)	90.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$\mathcal{T}$	227.8(13.2)	17.8	54.8	12.3	2.7	4.1	1.4	0.0	2.7	0.0	1.4	2.7
	O	465.7(25.9)	75.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	24.7
aggregated	$\mathcal{T}$+O	72.7(5.6)	82.2	9.6	1.4	1.4	1.4	1.4	0.0	0.0	0.0	0.0	2.7
	$\mathcal{TC}$	183.6(10.4)	20.5	56.2	12.3	0.0	5.5	0.0	1.4	2.7	0.0	0.0	1.4
	$\mathcal{TC}$ **or** O	352.2(15.4)	74.0	8.2	1.4	0.0	1.4	0.0	1.4	0.0	0.0	0.0	13.7
	$\mathcal{TC}$+O	20.5(2.2)	87.7	8.2	1.4	0.0	1.4	0.0	1.4	0.0	0.0	0.0	0.0

Table 3: Results of fault localization using the different fault localization techniques. The columns of the table indicate (i) the subject program, (ii) the fault localization technique used (iii) the average number of statements to inspect, and average percentage of statements to inspect, (iv)-(xiv) indicate the percentage of faults in each range of percentage of statements to inspect.

statements, up to 20% of all statements, and so on. Consider, for example, the case where the $\mathcal{T}$+O technique is used to localize faults in **faqforge**. If a programmer inspects the statements reported by this technique in decreasing order of suspiciousness, then on average, he will need to inspect 9.3 statements until he has found the first faulty statement, and this corresponds to 1.3% of the executed statements. Furthermore, we can see that for 85% of the faults in **faqforge**, less than 1% of the executed statements needs to be inspected, for an additional 10% of the faults, between 1% and 10% of the executed statements needs to be inspected, and that the remaining 5% of the faults requires inspecting between 10% of 20% of the executed statements.

In order to ease the discussion of the relative effectiveness of the techniques, we will say that a fault is *well-localized* by a fault-localization technique if inspecting the statements in decreasing order of suspiciousness according to that technique implies that all faulty statements are found after inspecting fewer than 1% of all executed statements. Using this terminology, we can see that:

- Using the basic *Tarantula* algorithm, only 17.8% of all faults are well-localized, on average (see the first row of data in the set of rows labeled **aggregated**).
- Using the oracle-based technique O, 75.3% of all faults are well-localized, on average over all subjects.
- Combining *Tarantula* with the oracle ($\mathcal{T}$+O) yields a technique that outperforms either of its constituents, with 82.2% of all faults being well-localized on average.
- Adapting *Tarantula* to operate on statements and (condition,value) pairs ($\mathcal{TC}$) is slightly helpful, by well-localizing 20.5% of all faults, versus the previously mentioned 17.8% for the statement-based *Tarantula*-algorithm.
- The most effective fault localization technique is obtained by using the variant of *Tarantula* that operates on statements and (condition,value) pairs, in combination with the oracle ($\mathcal{TC}$+O). Using this technique, 87.7% of all faults are well-localized, on average over all subjects.
- For the combined technique that uses the $\mathcal{TC}$ technique for execution errors and the O technique for HTML failures, 74.0% of all faults are well-localized.

6.4 Discussion

While we have discussed only aggregated data so far, the results appear to be consistent across the four subject applications. It is interesting to note that the effectiveness of the more precise modeling of conditionals depends on whether the subject program contains any faults that consist of missing branches in conditions. For one subject (**webchess**) this accounts for a 15.4% improvement in well-localized faults over the basic *Tarantula* algorithm, whereas for another (**timeclock**), it makes no difference at all. In summary, we found that the $\mathcal{TC}$+O yields an almost five-fold increase in the percentage of well-localized bugs, when compared with the basic *Tarantula* algorithm. Most of this increment is due to the use of the output mapping in combination with positional information obtained from the oracle. This is undoubtedly due to the fact that many of the localized faults manifest themselves via malformed HTML output. Our treatment of conditional expressions accounts for a much smaller part of the gains in precision, but is still helpful in cases where the fault consists of a missing branch in a conditional statement.

It is interesting to note that, since the oracle provides a binary suspiciousness rating, it tends to either be very helpful, or not helpful at all. This argues strongly for a fault-localization method that combines a statistical method such as *Tarantula*, with one based on an output mapping. One could consider using different techniques for different kinds of faults (e.g., use *Tarantula* for execution errors, and the oracle-based technique for HTML errors). However, the example that we discussed previously in Section 2.4 shows that

the two techniques can reinforce each other in useful ways. This is confirmed by our experimental results: the combined technique $\mathcal{TC}$ **or** $\mathcal{O}$ is significantly less effective (74.0% of all statements being well-localized) than the combined technique $\mathcal{TC}$+$\mathcal{O}$ (87.7%).

Figure 4 shows a graph depicting the aggregated data of Table 3. The *X*-axis represents the percentage of statements that need to be examined in decreasing order of suspiciousness until the first fault has been found, and the *Y*-axis the number of faults localized. A line is drawn for each of the six fault localization techniques under consideration. From these lines, it is clear that the $\mathcal{TC}$+$\mathcal{O}$ technique outperforms all other techniques. In particular, note that, for *any* percentage *n* between 0% and 100%, $\mathcal{TC}$+$\mathcal{O}$ localizes more faults than any of the other algorithms when up to *n*% of all statements are examined in decreasing order of suspiciousness.

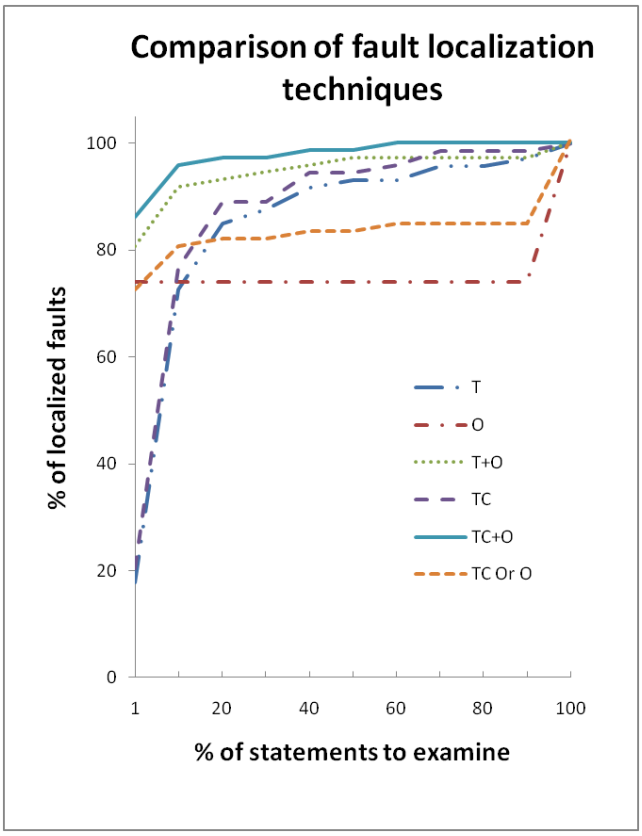

Figure 4: Effectiveness comparison of different fault-localization techniques. X-axis: percentage of statements that need to be inspected. Y-axis: percentage of faults.

6.5 Threats to Validity

There are several objections a critical reviewer might raise to the evaluation presented in this section. First, one might argue that the benchmarks are not representative of real-world PHP programs. While this may be the case, we selected open-source PHP applications that are widely used, as is evidenced by the number of downloads reported in Figure 1. The same subject programs were also used as subject programs by Minamide [20]. Second, it could be the case that the faults we exposed and localized are not representative. We do not consider this to be a serious risk, because we were previously unfamiliar with the faults in these subject programs, and all of them were exposed by automatic and systematic means. A potentially more serious issue is that any given fault may be fixed in multiple different ways. The fixes we devised were mostly one-line code changes, for which we attempted to produce the simplest possible solution. A possible experiment (which we plan to pursue as future work) is to submit patches with our fixes to the developers of the subject programs, and observe if they are accepted. The most serious criticism to our evaluation, in our own opinion, is the assumption that programmers would inspect the statements strictly in decreasing order of suspiciousness. In practice, it is very likely that programmers who try to follow this discipline would automatically look at adjacent statements, so the assumption is probably not completely realistic. Our rebuttal to this argument is that we evaluate all techniques in exactly the same way, and that this approach to measuring the effectiveness of fault localization methods has been used in previous research in the area (e.g., [15, 14, 24]).

7. RELATED WORK

This section reviews the literature on fault localization, focusing on Tarantula and other approaches.

7.1 Tarantula

In this paper, we apply the *Tarantula* technique [15, 14], which was previously discussed in Section 4, in a new domain (web applications written in PHP). Previous evaluations of the *Tarantula* algorithm have primarily focused on the Siemens suite, a collection of small C programs into which artificial faults have been seeded and for which a large number of test cases is available. By contrast, we study real faults in open-source PHP web applications. Moreover, unlike previous work on *Tarantula*, we do not assume the availability of a test suite but rely on combined concrete and symbolic execution to generate a large number of (passing and failing) test cases instead.

Santelices, *et al.* [24] investigate the tradeoffs of applying the *Tarantula* algorithm to different types of program entities: statements, branches, and def-use pairs. The results for the branch-based and def-use-based variants are mapped to statements, so that their effectiveness can be compared. The outcome of this comparison is that the branch-based algorithm is more precise than the statement-based one, and that def-used based variant is more precise still. Santelices, *et al.* also present algorithms that combine the variants by computing an overall suspiciousness rating for each statement that is derived from the underlying suspiciousness ratings, and report that one of these combined algorithms is even more precise than the def-use based algorithm. In this paper, we also explore special treatment of branches, but unlike Santelices, *et al.*, we do not compute a separate suspiciousness rating based on branch information but instead extend the statement-based approach by treating each control-flow predicate as several distinct statements, one for each branch.

7.2 Other Approaches for Fault Localization

Early work on fault localization relied on the use of program slicing [27]. Lyle and Weiser [19] 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 [21], and by Agrawal, *et al.* [2].

In the spirit of this early work, Renieris and Reiss [23] 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 a ranking technique in which additional statements are considered based on their distance to previously reported statements along edges in the System Dependence Graph (SDG) [11]. 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 [23] 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[9], 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 [12], and was found to be superior to the *set-union* and *set-intersection* techniques.

Recent papers by Jones and Harrold [14] and by Abreu, *et al.* [1] empirically evaluate various fault-localization techniques (including many of the ones discussed above) using the Siemens suite. Dallmeier, *et al.* [8] present a technique in which differences between method-call sequences that occur in passing and failing executions are used to identify suspicious statements. They evaluate the technique on buggy versions of the NanoXML Java application. Cleve and Zeller [7, 30], Zhang et al. [31], and Jeffrey, *et al.* [13] present fault localization techniques that attempt to localize faults by modifying the program state at selected points in a failing run, and observing whether or not the failure reoccurs. Other fault localization techniques analyze statistical correlations between control flow predicates [17, 18] or path profiles [6] and failures, time spectra [29], and correlations between changes made by programmers and test failures [26, 22]. In recent work by Zhang et al. [32], suspiciousness scores are associated with basic blocks and control-flow edges, and computed by solving (e.g., using Gaussian elimination) a set of equations that reflect control flow between basic blocks.

8. CONCLUSIONS AND FUTURE WORK

We have leveraged combined concrete and symbolic execution and several fault-localization techniques to create a uniquely powerful tool that automatically detects failures and localizes faults in PHP Web applications. The fault-localization techniques that we evaluated combine variations on the *Tarantula* algorithm with a technique based on maintaining a mapping between executed statements and the fragments of output they produce. We implemented these techniques in a tool called *Apollo*, and evaluated them by localizing 75 randomly selected faults that were exposed by automatically generated tests in four PHP applications. Our findings indicate that, using our best technique, 87.7% of the faults under consideration are localized to within 1% of all executed statements, which constitutes an almost five-fold improvement over the *Tarantula* algorithm.

For future work, we plan to investigate if the effectiveness of our techniques can be enhanced by generating additional tests whose execution characteristics are similar to those of failing tests. We also plan to explore the effectiveness of variations on the *Ochiai* fault localization technique [1], and to submit the patches we devised to remedy the localized faults to the developers of our test subjects.

[9]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.

9. 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*, Toulouse, France, 1995.

[3] S. Artzi, A. Kieżun, 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] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering*, 2010. To appear.

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

[6] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

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

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

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

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

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

[12] 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.

[13] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, pages 167–178, 2008.

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

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

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

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

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

[19] J. Lyle and M. Weiser. Automatic bug location by program slicing. In *ICCEA*, pages 877–883, Beijing (Peking), China, 1987.

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

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

[22] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA*, pages 239–249, 2007.

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

[24] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

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

[26] 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.

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

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

[29] C. Yilmaz, A. M. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *ICSE*, pages 81–90, 2008.

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

[31] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[32] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *ESEC/FSE*, pages 43–52, 2009.