Fault Localization for Dynamic Web Applications

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia

Abstract—In recent years, there has been significant interest in fault-localization techniques that are based on statistical analysis of program constructs executed by passing and failing executions. This paper shows how the *Tarantula*, *Ochiai*, and *Jaccard* fault-localization algorithms can be enhanced to localize faults effectively in web applications written in PHP by using an extended domain for conditional and function-call statements and by using a source mapping. We also propose several novel test-generation strategies that are geared toward producing test suites that have maximal fault-localization effectiveness. We implemented various fault-localization techniques and test-generation strategies in *Apollo*, and evaluated them on several open-source PHP applications. Our results indicate that a variant of the *Ochiai* algorithm that includes all our enhancements localizes 87.8 percent of all faults to within 1 percent of all executed statements, compared to only 37.4 percent for the unenhanced *Ochiai* algorithm. We also found that all the test-generation strategies that we considered are capable of generating test suites with maximal fault-localization effectiveness when given an infinite time budget for test generation. However, on average, a directed strategy based on path-constraint similarity achieves this maximal effectiveness after generating only 6.5 tests, compared to 46.8 tests for an undirected test-generation strategy.

Index Terms—Fault localization, statistical debugging, program analysis, web applications, PHP.

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 Structured Query Language (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. In the domain of web applications, some faults manifest themselves as web-application crashes and as malformed HTML pages that are not displayed properly in a web browser. While malformed HTML failures may seem trivial, and indeed many of them are at worst minor annoyances, they have on occasion been known to create serious vulnerabilities, e.g., via denial-of-service attacks.[1] Furthermore, such failures in the HTML code may be difficult to localize because HTML code is often *dynamically generated* by server-side code—written, for example, in PHP or Java—and so, when a failure is detected, there really is no HTML file or line number to point the developer to. In this paper, we present *Apollo*, the first fully automatic tool that efficiently finds and localizes malformed HTML and execution failures in web applications that execute PHP code on the server side.

1. For example, http://support.microsoft.com/kb/810819.

- *S. Artzi is with the IBM Software Group, 550 King St, Littleton, MA 01460. E-mail: artzi@us.ibm.com.*
- *J. Dolby, F. Tip, and M. Pistoia are with the IBM Thomas J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598. E-mail: {dolby, ftip, pistoia}@us.ibm.com.*

Manuscript received 8 Dec. 2010; revised 8 Apr. 2011; accepted 12 June 2011; published online 26 July 2011.
Recommended for acceptance by A. Orso and P. Tonella.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-12-0361.
Digital Object Identifier no. 10.1109/TSE.2011.76.

In previous work [8], [9], we adapted the well-established technique of combined concrete and symbolic execution [20], [40], [15], [21], [44] 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, dependent on that input, that have been executed. 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, 1) it integrates an HTML validator to check for failures that manifest themselves by the generation of malformed HTML, 2) it automatically simulates interactive user input, and 3) it keeps track of the interactive session state that is shared between multiple PHP scripts.

However, our previous work focused exclusively on finding *failures* by identifying inputs that cause an application to crash or produce malformed HTML. We did not address the problem of pinpointing the specific web-application instructions that cause these failures, and fixing the underlying *faults* can be very difficult and time consuming if no information is available about where they are located. 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. For an overview of the literature on fault localization, the reader is referred to Section 7.

0098-5589/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

The fault-localization algorithms explored in this paper attempt to predict the location of a fault based on a statistical analysis of the correlation between passing and failing tests and the program constructs executed by these tests. In particular, we investigate variations on three popular statistical fault-localization algorithms, known as *Tarantula* [29], [28], *Ochiai* [3], and *Jaccard* [16], [24]. These algorithms predict the location of a fault 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 executed statements in order of decreasing suspiciousness. The effectiveness of a fault-localization technique can be measured by determining how many statements need to be inspected, on average, until the fault is found. Our work differs from most previous research on fault localization in that it does not assume the existence of a test suite with passing and failing test cases. Instead, we rely on combined concrete and symbolic execution to *generate* passing and failing runs.

This paper advances the state of the art in fault localization in two ways. First, we present enhancements to previous statistical fault-localization techniques [29], [28], [3], [24] that make them significantly more effective at localizing the faults responsible for execution failures, and for the generation of malformed HTML pages in PHP web applications. These enhancements include:

- **The use of an extended domain.** We apply existing statistical fault-localization techniques on an *extended domain* of (statement, runtime value) pairs, in which the runtime value serves to differentiate occurrences of the statement at runtime. Applying this technique to conditional statements helps fault localization with the identification of errors of omission such as missing branches, and applying the technique to function calls enables us to differentiate normal return values from values such as `null` that are often correlated with erroneous behavior (e.g., incorrect handling of corner cases).
- **The use of a source mapping.** We modified the PHP interpreter to maintain a *source mapping* that records the statement in the PHP application that produced each fragment of output at runtime; thus, it is a map from `print` statements to regions of the HTML file. This mapping—when combined with the report of the HTML validator, which indicates the parts of the HTML output that are incorrect—provides an additional source of information about possible fault locations, and is used to fine-tune the suspiciousness ratings of existing fault-localization techniques.

A second main research topic explored in this paper has to do with the fact that existing fault-localization approaches assume the existence of a test suite. However, developers are often confronted with situations where a failure occurs, but where no test suite is available that can be used for fault localization. To address such situations, we present an approach for generating test suites that can be used to localize faults effectively. This approach is a variation on combined concrete and symbolic execution [20], [40], [15], [21], [44] that is parameterized by a *similarity criterion*. Such a similarity criterion measures how similar the execution characteristics associated with two tests are, and is used to *direct* the generation of tests toward tests whose execution characteristics are similar to those of a given failing test. By varying the similarity criteria being used, experiments can be conducted to determine which test-generation strategy achieves the best tradeoff between size and fault-localization effectiveness for the test suites that it generates.

We implemented these techniques 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 115 randomly selected faults that were exposed by automatically generated tests in five open-source PHP applications, using enhanced versions of the *Tarantula*, *Ochiai*, and *Jaccard* techniques. In each case, we found the enhancements improve the fault-localization effectiveness significantly. For example, using the basic *Ochiai* technique, the programmer had to examine an average of 5.6 percent of an application's executed statements to find each of the 115 faults, when exploring the executed statements in order of decreasing suspiciousness. Using our best technique, which augments the domain of *Ochiai* for conditional and function-call statements and which uses the source mapping to fine-tune *Ochiai*'s suspiciousness ratings, the programmer needs to explore only 0.7 percent of the executed statements, on average. More significantly, using our best technique, 87.8 percent of the 115 faults under consideration are localized to within 1 percent of all executed statements, compared to only 37.4 percent for the unenhanced *Ochiai* algorithm.

We also implemented several strategies for directed test generation in *Apollo*, and we measured the fault-localization effectiveness of test suites generated according to each strategy, using the enhanced version of *Ochiai* that we found to be the most effective fault-localization technique. Our results show that a new, directed test-generation technique based on *path-constraint similarity* (PCS) yields the smallest test suites with the same excellent fault-localization characteristics as test suites generated by other techniques. In particular, when compared to test generation based on the undirected test-generation strategy in [7], which aims to maximize code coverage, our directed technique reduces test-suite size by 86.1 percent and test-suite generation time by 88.6 percent.

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

1. We present two mechanisms, the use of an extended domain and the use of a source mapping, that significantly enhance the effectiveness of existing fault-localization techniques such as *Tarantula*, *Ochiai*, and *Jaccard*.
2. To evaluate these fault-localization techniques in *Apollo*, we implemented each of the fault localization techniques, localized 115 randomly selected faults in five PHP applications, and compared the technique's effectiveness. Our findings show that, using our best technique, an enhanced version of *Ochiai*, 87.8 percent of the faults are localized to within 1 percent of all executed statements, compared to only 37.4 percent for the unenhanced

Ochiai algorithm. Similar improvements were obtained for enhanced versions of the *Jaccard* and *Tarantula* algorithms.

3. We present a family of directed test-generation techniques, based on combined concrete and symbolic execution, that are capable of generating small test suites with high fault-localization effectiveness. These techniques overcome the important limitation of many previous fault-localization methods that a test suite be available upfront.
4. To evaluate directed test generation, we implemented these directed test-generation techniques in *Apollo*. Our evaluation shows that a directed technique based on path-constraint similarity reduces test suite size by 86.1 percent and generation time by 88.6 percent when compared to an existing undirected test-generation technique, without compromising fault-localization effectiveness.

These findings show that automated techniques for fault localization, which were previously primarily evaluated on programs with artificially seeded faults, are effective at localizing real faults in open-source PHP web applications.

The remainder of this paper is organized as follows: Section 2 reviews the PHP language and the kinds of failures that may arise in PHP programs. Section 3 reviews the *Tarantula*, *Ochiai*, and *Jaccard* fault-localization algorithms and presents our extensions to these algorithms. In Section 4, we present algorithms for test generation that are directed by similarity criteria toward the generation of test suites with high fault-localization effectiveness. Section 5 presents an implementation of our work in the context of the *Apollo* tool. Section 6 presents an evaluation of our fault-localization and test-generation algorithms on a set of open-source PHP programs. Related work is discussed in Section 7. Finally, conclusions are presented in Section 8.

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 runtime 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 Fig. 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 runtime; for example, the arguments of the `require` statements on line 2 of the PHP script of Fig. 1c and on line 9 of the PHP script of Fig. 1e are dynamically computed at runtime 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 runtime. 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, for example, by missing included files 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 and incorrect SQL queries, 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. 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.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN http://www.w3.org/TR/html4/loose.dtd">
2  <HTML>
3    <HEAD><TITLE>Login</TITLE></HEAD>
4    <BODY>
5    <SCRIPT  type="text/javascript">
6     function signon() {
7      var form = document.forms[0];
8      var elts = form.operation;
9      for (var elt in elts) {
10      if (elts[elt].checked==true) {
11        open(elts[elt].value + ".php?user=" + form.user.value + "&pw=" + form.pw.value);
12      }
13     }
14    }
15   </SCRIPT>
16     <FORM ID="theform" METHOD = "post" NAME = "login" ACTION="javascript: signon();">
17       User Name: <INPUT TYPE = "text" NAME = "user"/><BR>
18       Password: <INPUT TYPE = "password" NAME = "pw"/><BR>
19       <INPUT TYPE="radio" NAME="operation" VALUE="query"/>Query
20       <INPUT TYPE="radio" NAME="operation" VALUE="update"/>Update <BR>
21       <INPUT TYPE="submit" NAME="Submit"/>
22     </FORM>
23   </BODY>
24 </HTML>
```

(a) `index.html`

```
1  <HTML>
2  <HEAD>
3  <TITLE>Vehicle Queries</TITLE>
4  </HEAD>
5  <BODY>
6  <?php
7  session_start();
8  $user = $_REQUEST["user"];  {
9  $pw = $_REQUEST["pw"];
10
11 function check_permissions($user, $pw) {
12   ...
13 }
14
15 $foo = check_permissions($user, $pw);
16 $_SESSION['readOK'] = $foo->readOK;
17 $_SESSION['writeOK'] = $foo->writeOK;
18 session_write_close();
19 ?>
20     <FORM action="request.php">
21       Query ('Make' or 'Model'): <INPUT TYPE="text" NAME="query"/>
22       Name: <INPUT TYPE="text" NAME="key"/><BR>
23       <HR>
24       <INPUT TYPE="submit" VALUE="submit" NAME="Submit"/>
25     </FORM>
26 </BODY>
27 </HTML>
```

(b) `query.php`

```
1  <?php
2  require_once dirname(__FILE__) . "/../config/settings.php";
3
4  mysql_connect("localhost",$username,$password);
5  mysql_select_db($database);
6  ?>
```

(c) `db.php`

```
1  <?php
2  $username="root";
3  $password="example";
4  $database="example";
5  ?>
```

(d) `settings.php`

```
1  <HTML>
2  <HEAD>
3  <TITLE>Results</TITLE>
4  </HEAD>
5  <?php
6  error_reporting( E_WARNING );
7
8  session_start();
9  require_once(dirname(__FILE__) . "/util/db.php");
10
11 if (isset($_REQUEST['query']) ){
12  echo ' <BODY>';
13  if ($_SESSION['readOK'] == 1) {
14   switch ($_REQUEST['query']) {
15   case "Model":
16     $kf="model"; break;
17   case "Make":
18     $kf="make"; break;
19   }
20   $v = $_REQUEST['key'];
21
22   $sql="SELECT * FROM VEHICLES WHERE " . $kf . " = '" . $v . "'";
23   $result = mysql_query($sql);
24   $n = mysql_numrows($result);
25   echo '<H1>' . $n . ' results for ' . $v . '</H1>';
26 ?>
27   <UL>
28 <?php
29   $i = 0;
30   while ($i < $n) {
31     $descr=mysql_result($result,$i,'descr');
32     $name=mysql_result($result,$i,'model');
33     $make=mysql_result($result,$i,'make');
34     echo '<LI> ' . $make . ' ' . $name . ': ' . $descr;
35     $i++;
36   }
37   echo '</UL>';
38   echo ' </BODY>';
39  } else {
40   echo '<H2>Permission Denied</H2>';
41  }
42 } else if (isset($_REQUEST['update']) ){ {
43   ... update code ...
44 }
45 ?>
46 </HTML>
```

(e) `request.php`

Fig. 1. Sample PHP web application.

2. For both kinds of failures, one could look at runs that do *not* exhibit the failure, 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

Fig. 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.[2]

The code fragments shown in Fig. 1 are part of the client/server workflow in a web application: The user first sees the

2. This code has been tested using Safari 5.0.2, Apache 2.2.16, and PHP 5.3.3.

index.html page of Fig. 1a and enters the required credentials. These user-input credentials are processed by the query.php script in Fig. 1b. This script generates a response page that allows the user to enter further input, causing further processing by the request.php script in Fig. 1e. Note that the permissions obtained for the user during the execution of query.php are stored in special locations $_SESSION['readOK'] and $_SESSION ['writeOK']. This illustrates how PHP handles *session state*, which are data that persist from one page to another to keep track of the interactions with the application by a particular user. Thus, the updates to _SESSION in Fig. 1b will be seen by the code in Fig. 1e when the user follows the link to request.php in the HTML page that is returned by query.php. We now discuss some aspects of Fig. 1 in further detail:

1. index.html. The top-level index.html script in Fig. 1a contains static HTML and JavaScript code; this file accepts user credentials and determines whether to invoke query.php, shown in Fig. 1b, or update.php (not shown) based on the radio button selected. When the form is submitted, the action is a call to the JavaScript function signon (lines 6-14). This kind of computed URL is becoming ever more common, and is difficult for a pure web crawler to understand.
2. *query.php*. The query.php script in Fig. 1b checks the permissions of the user and stores that information in session state for later use (lines 11-17). Observe that the script also contains calls to session_start(), which is required in PHP for scripts to use session state, and to session_write_close(), which is an optimization to indicate writing to the session is done. The rest of the script is a static HTML form in which the user enters parameters for a database query. When this form is submitted, the data are passed to request.php, shown in Fig. 1e.
3. *db.php*. The db.php script is an include file that isolates the code required to connect to the MySQL database that holds the vehicle information that the web application is accessing. It is used by being included into request.php, and it depends on settings.php to define the system configuration for accessing the database.
4. *settings.php*. This file isolates the site-specific configuration for the web application. It is used by being included into db.php.
5. *request.php*. The request.php script in Fig. 1e carries out the database request made by the user. This script can be called either from query.php or update.php (not shown) that provides access to update the data. When it receives a query parameter (checked at line 11), it first checks the read permission stored in the session state (line 13). If that check passes, it constructs an SQL query using the kind of query requested to determine the key field (lines 14-19). It then executes the query (line 23) and prints out the results (lines 24-37). If the permission check fails, it prints a permission denied message (line 40).

2.5 Faults in the Motivating Example

Our sample program may exhibit both execution failures and HTML failures. Specifically, there are two faults exhibiting HTML failures and one fault exhibiting an execution failure:

- The BODY tag opened at line 12 of Fig. 1e may not be closed since the corresponding /BODY tag is produced at line 38 in a different control context. In particular, a closing tag will not be generated in the case when permission is denied (line 40). Exhibiting this failure requires that the page be accessed in a way that produces a "permission denied" message. This means that a tester would have to start at index.html and enter a set of credentials that do not have read permission and choose the query option on the radio buttons. After that, the tester must fill in and submit the form on query.php. 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.
- An HTML failure occurs in case the database query line 23 of Fig. 1e returns no elements. In this case, $n will be 0 on line 24 of Fig. 1e so that the body of the loop will not execute. This results in an empty <UL> list, which is a violation of the HTML specification. Exposing this failure requires a similar process of filling in forms as for the first one.
- An execution failure may occur in request.php if it is called with an unexpected query parameter. Then, the $kf variable will be left undefined by the switch (lines 14-19), resulting in invalid SQL which will, in turn, result in errors in statements like mysql_numrows (line 24). Exposing this failure requires a similar process of filling in forms as for the previous ones.

3 Fault Localization

In this section, we first review the basic fault-localization algorithms we implemented and extended: *Tarantula*, *Jaccard*, and *Ochiai*. We then present an alternative technique that is based on source 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 and return-value expressions can help improve the basic algorithm's effectiveness.

3.1 Fault Localization Algorithms

Jones et al. [29], [28] 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, calculated with the following similarity coefficient:

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

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

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.

In the field of data clustering, other similarity coefficients have been proposed that can also be used to calculate suspiciousness ratings. These include the *Jaccard* [24] coefficient used in the pinpoint program [16]:

$$S_{jac}(l) = \frac{Failed(l)}{TotalFailed + Passed(l)},$$

and the *Ochiai* [3] coefficient used in the molecular biology domain:

$$S_{och}(l) = \frac{Failed(l)}{\sqrt{Failed(l)*(Passed(l) + Failed(l))}}.$$

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 each statement is the number of statements with greater than or equal suspiciousness:

$$rank(l) = |l' : S(l') >= S(l)|.$$

The rank of 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.

Abreu et al. [1] conducted a detailed empirical evaluation in which they applied the *Tarantula*, *Ochiai*, and *Jaccard* algorithms to faulty versions of the Siemens suite [23], and compare their effectiveness (see Section 7). The Siemens suite consists of several versions of small C programs into which faults have been seeded artificially, and a very extensive test suite exposing them. 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 [18], [3], [38], [28].

In a similar experiment, Jones and Harrold [28] computed for each failing test run a *score* (in the range of 0-100 percent) 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 percent of the failing test runs were scored in the 99-100 percent range, meaning that for this percentage of the failing tests, the programmer needs to examine less than 1 percent of the program's executed statements to find the fault. They also report that for an additional 41.8 percent of the failing tests, the programmer needs to inspect less than 10 percent of the executed statements.

3.2 Fault Localization Using Source Mapping

An oracle that determines whether or not a failure occurs in the output 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 file where the HTML 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:

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

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

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

The effectiveness of using the source mapping for fault localization may vary depending on the types of failures and applications. In our experience, while the HTML validator often produces good positional information, this is not always the case. Therefore, it makes sense to combine the use of the source mapping with the use of statistical fault-localization methods such as those discussed previously.

3.3 Enhancing Statistical Fault Localization with Source Mapping

The algorithms presented in Section 3.1 localize 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 3.2 can be used to enhance the effectiveness of statistical algorithms by giving a higher rank to statements that are blamed by both the statistical method and the source-mapping technique. More formally, we define a new suspiciousness rating $S_{comb}(l)$ for the statement at line l as follows:

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

Informally, we give the suspiciousness rating 1.1 to any statement that is identified as highly suspicious according to the source mapping, and for which the original algorithm indicates that the given line is positively correlated with the fault (indicated by a suspiciousness rating greater than 0.5).

3.4 Example

As an example, suppose we have two runs of the script in Fig. 1e, one of which exposes the HTML of the empty ⟨UL⟩ tag previously discussed in Section 2.5, and one which does

```
1 <HTML>
2 <HEAD>
3 <TITLE>Results</TITLE>
4 </HEAD>
5 <BODY>
6 <H1> results for </H1>  <UL>
7 </UL></BODY>
8 </HTML>
```

(a) HTML output

HTML line	*PHP lines in 1(e)*
1	1
2	2
3	3
4	4
5	12
6	25, 27
7	37
8	38
9	4

(b) source mapping

```
Line 7, Character 5:

</UL>

Error: missing a required sub-element of UL
```

(c) Output of WDG Validator

Fig. 2. (a) HTML produced by the script of Fig. 1e. (b) Output mapping constructed during execution. (c) Part of output of WDG validator on the HTML of Fig. 2a.

not. Furthermore, assume that these executions have the following characteristics:

- Both the passing and the failing executions have `readOK` set to true by the `query.php` script of Fig. 1b.
- Both the passing and the failing executions have the `query` parameter set to `Model` by the `query.php` script of Fig. 1b.
- In the passing run, some value is used for the `key` form field that matches some item in the database of vehicles.
- In the failing run, the `key` form field has a garbage value because the user did not enter a specific value.

In this case, the failing run will produce an empty list of matches, which results in the previously discussed HTML failure, whereas the passing run will not. Hence, the first is a failing run and the second a passing run with respect to the failure in Fig. 2.

Fig. 3 presents the suspiciousness rating given by the three techniques. 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 runs 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 Fig. 2c, along with the HTML in Fig. 2a and the mapping from lines of HTML to the lines of PHP that produced them in Fig. 2b. The validator says the error is in line 7 of the output, and those output fragments were produced by line 37 in the script of Fig. 1e. Consequently, the suspiciousness rating for line 37 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 3.3.

line(s)	**executed by**	$S_{tar}(l)$	$S_{map}(l)$	$S_{comb}(l)$
1-14	*both*	0.5	0.0	0.5
15-16	*both*	0.5	0.0	0.5
20-30	*both*	0.5	0.0	0.5
31-36	*passing only*	0.0	0.0	0.0
37	*both*	0.5	1.0	0.5
38, 45-46	*both*	0.5	0.0	0.5

Fig. 3. Suspiciousness ratings for lines in the PHP script of Fig. 1e, 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 $S_{comb}(l)$.

As can be seen from the table, the *Tarantula* technique identifies many lines as the most suspicious ones, and the source-mapping-based technique only identifies line 37 as such. In this particular example, the use of the source mapping does not result in greater fault-localization effectiveness. However, Section 3.5 will introduce another enhancement that will help with the localization of this particular fault, as we shall see in Section 3.6.

3.5 Fault Localization Using an Extended Domain

As we observed in Section 3.1, fault-localization algorithms work 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 file to be malformed if the `boolean` predicate in the `if` statement is `false`. For instance, in Section 2.5, we observed a failure caused by a missing `default` case when discussing the `switch` statement in lines 14-19 in Fig. 1e. Traditional fault-localization techniques, as previously applied to statements, cannot rank a missing statement that will never be executed.

We enhance the effectiveness of fault localization 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`.

line(s)	executed by	$S_{tar}(l)$	$S_{map}(l)$	$S_{comb}(l)$
1-9	*both*	0.5	0.0	0.5
(11,1)	*both*	0.5	0.0	0.5
12	*both*	0.5	0.0	0.5
(13,1)	*both*	0.5	0.0	0.5
(14,1)	*both*	0.0	0.0	0.0
15-16	*both*	0.5	0.0	0.5
20-23	*both*	0.5	0.0	0.5
(24,0)	*failing only*	1.0	0.0	1.0
(24,positive int)	*passing only*	0.0	0.0	0.0
25-29	*both*	0.5	0.0	0.5
(30,0)	*both*	0.5	0.0	0.5
(30,1)	*passing only*	0.0	0.0	0.0
31-36	*passing only*	0.0	0.0	0.0
37	*both*	0.5	1.0	0.5
38, 45-46	*both*	0.5	0.0	0.5

Fig. 4. Suspiciousness ratings for lines in the PHP script of Fig. 1(3) with conditional and return-value modeling, 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. $(s, 1)$. 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 $(s, \ldots)$, the conditional statement s is assigned the maximum of these ratings, from which its rank is computed in the normal manner.[4] 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 example below.

Another fault-localization enhancement is our *return-value modeling* technique, which consists of dynamically storing, for each procedure call, the line number of the caller along with an abstract model of the value. This allows the underlying fault-localization technique to distinguish, for example, `null` values from non-`null` values, zero `int`, and `double` values from nonzero ones, `true` Boolean values from `false` Boolean values, and empty arrays, strings, and resources from nonempty ones. These distinctions are useful for the purpose of fault localization because faults often arise in situations where, for example, a program mistakenly attempts to dereference a `null` value, divide by 0, or extract values from empty collections. Liblit et al. [31] sampled a simpler model of return values (targeted at functions returning scalars) for bug isolation.

3.6 Example Revisited

We will now consider the example of Section 3.4 again, to study the effects of conditional and return-value modeling. Fig. 4 shows the suspiciousness ratings of lines in the script in Fig. 1e with the addition of conditional outcome and return-value modeling. For simplicity, only the return value of line 24 is shown since that is the only one where there are multiple abstract return values. Observe that now there are lines that only occur in the failing run, and they get given the maximum blame. In particular, the blame falls on the call to `numrows` when it returns 0, that is line 24 with result 0. And these are really the appropriate lines to blame since the empty list can be returned only when there are no results.

4. Alternatively, the programmer may choose to compute separate suspiciousness ratings for all pairs $(s, \ldots)$, which has the advantage of providing context information from the extended domain. For instance, a ranking may denote a particular conditional statement and that its outcome was `false`, which may give the programmer a better notion of why this particular statement is causing a failure.

4 Directed Test Generation

The fault-localization techniques presented in the previous section require a collection of tests, some passing and some failing. However, if a failure occurs and no test suite is available, such techniques are powerless. To address such situations, this section presents a variation on combined concrete and symbolic execution [9] for generating test suites that can be used to localize faults effectively. When generating tests, it is desirable to keep the size of the generated test suites small in order to keep the cost of running these tests manageable. This is not of immediate concern for the subject programs used in Section 6, for which the tests have running times in the order of up to a few minutes. However, for industrial-sized applications, running the tests may require many hours, or require other resources or human involvement. For these reasons, our objective is to generate small test suites with excellent fault-localization characteristics.

4.1 Combined Concrete and Symbolic Execution

Our technique for generating test suites is a variation on combined concrete and symbolic execution [20], [40], [15], [21], [44], 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 [8], 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 [9], we extended the work to address interactive user input: 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. Furthermore, a script is much more likely to perform complex behavior when executed in the correct context or 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 without executing much of the program logic (e.g., when the condition on line 13 of Fig. 1e is `false`; note that this condition is derived indirectly from user information on lines 11-17 of Fig. 1b).

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 that can be executed from the initial state $\mathcal{C}$, and an output oracle $\mathcal{O}$. The output of the algorithm is a set of bug reports $\mathcal{B}$ for the program $\mathcal{P}$, according to $\mathcal{O}$. Each bug report contains 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.[5] Each test is comprised of three components: 1) the program component to execute, 2) a *path constraint* which is a conjunction of conditions on the program's input parameters, and 3) 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 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 previously explored.

For instance, an execution of `query.php`, shown in Fig. 1b, that did not provide `user` as an input parameter would generate a path constraint noting that `$user` is not set, i.e.,`!isset($user)`, after attempting to read `$user` when executing line 8. A subsequent execution could be constructed by negating this constraint to `isset($user)`. An execution satisfying this new constraint will define `$user` to some value.

4.2 Similarity Criteria

Given a failing execution, the general intuition behind our test-generation technique is that localizing the corresponding fault can be done more effectively if a passing test is generated whose characteristics are "similar" to those of the failing execution because that maximizes the chances that the fault is correlated with the difference between the path constraints of the generated passing test and those of the faulty execution; the smaller the difference, the higher the precision with which the fault can be localized.

To make this more precise, we need to formalize the concept of "similarity" between two tests. This leads us to introducing a similarity criterion, which is a function that takes as an input two tests and produces a fraction that indicates how similar the two tests are; more formally, if E is an element in the queue of tests, a *similarity criterion* is a function $\sigma : E \times E \rightarrow [0, 1]$. Note that there can be multiple similarity criteria, each based on what characteristics are considered.

In order to guide our test-generation technique toward generating similar executions, a similarity function is used as the selection methodology. The selection methodology is responsible for selecting the next input to explore, thus directing the test-generation process to explore similar executions. In this paper, we consider two different similarity metrics, one based on path constraints and one based on inputs, which are described in the following sections.

4.3 Path-Constraint Similarity

One element of each test in our queue is the path constraint used to generate its input. Accordingly, one of our similarity criteria measures the amount of similarity between the path constraints associated with two executions. We have implemented two techniques for path-constraint similarity: *subset comparison* and *subsequence comparison*. With subset comparison, execution similarity is computed based on the cardinality of the largest subset of identically evaluating conditional statements that are traversed in the two executions; with subsequence comparison, execution similarity is computed based on the length of the largest contiguous subsequence of conditions that evaluate to the same value in both executions.

To better understand the difference between these two metrics, consider, for example, two program executions $e_1, e_2 \in E$ that evaluate conditions $\langle C_1, C_2, C_3, C_4, C_5, C_6 \rangle$, and assume that condition C_3 evaluates to `true` in e_1 and `false` in e_2, but C_1, C_2, C_4, C_5, C_6 evaluate to the same Boolean value in both executions. In this case, $\sigma(e_1, e_2) = \frac{5}{6}$ if the similarity criterion is based on subset comparison, and $\sigma(e_1, e_2) = \frac{3}{6}$ if the similarity criterion is based on subsequence comparison. In practice, we observed that these two similarity metrics lead to very similar results. Therefore, in the remainder of this paper, we concentrate only on path-constraint similarity based on subset comparison.

4.4 Input Similarity (IS)

With this approach, we compare the inputs to different tests. The path constraints of each test are solved to provide the actual inputs to be used, and these inputs are compared. Input similarity is based on *subset comparison*: The similarity between two executions is computed based on the number of inputs that are identical for both executions. For example, consider two executions e_1 and e_2 with inputs $\langle S_1, S_2, S_3, S_4, S_5, S_6 \rangle$ and $\langle T_1, T_2, T_3, T_4, T_5, T_6 \rangle$, respectively, such that $S_3 \neq T_3$, but $S_i = T_i, \forall i \neq 3$. In this case, $\sigma_\alpha(e_1, e_2) = \frac{5}{6}$.

5. The criteria of selecting tests from the queue give preference to tests that will cover additional code. More details can be found in [9].

path constraint	example input	PCS	IS
$isset(query) \wedge query \neq' make' \wedge query \neq' model' \wedge isset(key)$	query=garbage&key=something		
$isset(query) \wedge query \neq' make' \wedge query \neq' model' \wedge \neg isset(key)$	query=rubbish	3/4	0/2
$isset(query) \wedge \neg (query \neq' make' \wedge query \neq' model')$	query=make&key=something	1/4	1/2
$\neg isset(query)$	<empty>	0/4	0/2

Fig. 5. Example path constraints for similarity metrics.

4.5 Example

Fig. 5 illustrates the input-based and path-constraint-based test-generation strategies using the example program of Fig. 1e. Each row in the table corresponds to an execution. Let us assume that the first row in the table shows an execution that leads to the failure resulting from the invalid SQL created when `$kf` is not set and that the remaining rows correspond to passing executions generated by our test-generation strategies. Observe that the latter three rows in the table correspond to executions that can be derived from the first by our test-generation strategies since each is derived from the path constraint for the failing execution by removing a number of conjuncts at the end and negating the last conjunct in the sequence that remains.

The first two columns of the table show, respectively, the path constraint for each execution, and an example input that satisfies that path constraint. The last two columns in the table show the similarity of each of the passing executions to the failing execution, according to the path-constraint similarity and input similarity criteria. For example, the second row has a PCS of 3 since only the last element of the constraint is different and the other three are the same. For input similarity, we compare the number of inputs that are the same among the example inputs; for instance, the third row has one element that is the same, `key`.

We can see from the example that in this case, the input similarity heuristic is selecting the second possible input. In this case, that input will result in a nonbuggy but very similar execution that will allow fault localization to determine the cause of the fault precisely.

5 Implementation

In *Apollo* , we implemented a *shadow interpreter* based on the Zend PHP interpreter 5.2.2[6] that simultaneously performs concrete program execution using concrete values, and a symbolic execution that uses symbolic values that are associated with variables. Furthermore, *Apollo* uses the `choco`[7] constraint solver to solve path constraints during the combined concrete and symbolic test generation. This process is orchestrated by a standard Apache[8] webserver that uses the instrumented PHP interpreter.

We implemented the following extensions to the shadow interpreter to support fault localization:

- **Statement coverage.** All fault-localization techniques use the percentage of failing and passing tests executing a given statement to calculate the statement's suspiciousness score. To this end, 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[9] and the CSE HTML Validator V9.0.[10]
- **Source mapping.** The source-mapping technique, described in Section 3.2, correlates a fault found in the HTML output with the statements producing the erroneous output fragments. Our shadow interpreter creates the mapping by recording the line number of the originating PHP statement whenever output is produced by `echo` and `print` statements.
- **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 3.5. For each comparison, it records a pair consisting of the statement's line number and the relevant Boolean result. For each execution of a `switch` statement, it records a pair consisting of the `switch`'s line number and a set of all executed `case` blocks during that execution.
- **Return-value modeling.** For this feature, the shadow interpreter stores the line number of the call and an abstract model of the value. The model allows the fault localization technique to distinguish between `null` and non-`null` values, zero and nonzero `int` and `double` values, `true` and `false` Boolean values, as well as empty and nonempty arrays, strings, and resources.[11]

6 Evaluation

This section presents an evaluation of the fault-localization techniques that we presented in the previous sections.

6.1 Research Questions

With respect to the enhancements to the basic fault-localization techniques described in Section 3, we are interested in answering the following research questions:

- RQ1. How effective are the basic fault-localization algorithms *Tarantula* [28], *Jaccard* [24], and *Ochiai* [3] in the domain of PHP web applications?
- RQ2. How much more effective do these basic fault-localization algorithms become when combined with the use of a *source mapping* and/or with the

6. http://www.php.net/.
7. http://www.emn.fr/z-info/choco-solver/choco-publications.html.
8. http://www.apache.org/.

9. http://htmlhelp.com/tools/validator/.
10. http://www.htmlvalidator.com/.
11. A PHP resource is a special type holding a reference to an external resource, such as a stream, socket, or the result of an SQL query.

TABLE 1
Characteristics of Subject Programs

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
phpsysinfo	2.5.3	73	7,745	492,217

*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.*

use of an extended domain for conditional statements and function-call statements, as presented in Sections 3.3 and 3.5?

Test cases do not always exist. To enable fault localization in the cases where they do not, this paper has described an efficient way to generate test suites that can be used for fault localization. In such cases, it is reasonable to expect that a limited amount of time will be available for test generation. Therefore, we are interested in determining how quickly each of the test-generation strategies under consideration converges toward its maximal effectiveness. This leads us to formulate two more research questions:

- RQ3. What is the maximal fault-localization effectiveness of test suites, measured as the percentage of well-localized faults, generated by each of the test-generation strategies?
- RQ4. How many tests need to be generated by each test-generation strategy in order to reach its maximal fault-localization effectiveness?

We will answer these research questions in the reminder of this section based on a concrete evaluation of our tool on a number of production-level PHP applications.

6.2 Subject Programs

In order to determine the effectiveness of automated fault-localization algorithms such as the ones studied in this paper, it is customary to apply the algorithms to programs that contain faults at known locations. To this end, we selected the following five open-source PHP programs from http://sourceforge.net, with which we were already familiar from our prior work on test generation [9]:

- **faqforge.** A tool for creating and managing documents.
- **webchess.** An online chess game.
- **schoolmate.** A PHP/MySQL solution for administering elementary, middle, and high schools.
- **timeclock.** A web-based timeclock system.
- **phpsysinfo.** A utility for displaying system information, such as uptime, CPU, memory, etc.

Table 1 presents some characteristics of these subject programs.

6.3 Methodology

In order to answer questions RQ1 and RQ2 about the effectiveness of different fault-localization techniques, a set of localized faults, and a test suite exposing them are needed for each subject program. Since neither a test suite nor a set of known faults existed for our subject programs,

TABLE 2
Characteristics of the Test Suites and Localized Faults in the Subject Programs

program	tests	localized faults		
		HTML	exec.	total
faqforge	1028	22	8	30
webchess	678	7	17	24
schoolmate	676	24	16	40
timeclock	577	14	3	17
phpsysinfo	178	2	2	4
total	3137	69	46	115

The columns of the table indicate: (1) the subject program, (2) the number of tests in the test suite generated for that program, and (3) the number of faults manually localized for that program (three columns: HTML faults, execution faults, and total).

we generated a test suite using the combined concrete and symbolic execution technique of *Apollo* [9] (see Section 4.1). 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, it is necessary to know where faults are located. Unlike most previous research on automated fault-localization techniques [29], [28], [39], 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. For each fault, we devised a patch and ensured that applying this patch fixed the problem. This was done by running the tests again and making sure that the associated failures[12] did not recur. The patch altered some statements in the code, which we will call the *faulty statements* later. For cases such as a missing `else` or `switch` clause, the faulty statement was deemed to be the corresponding `if` or `switch` with the appropriate value in the extended domain.

Table 2 summarizes the details of the generated test suites, and the localized faults that we will use in the remainder of this section. For **phpsysinfo**, **timeclock**, and **webchess**, we took all the faults we found. For **faqforge**, we took all the execution faults we found, and a random set of HTML faults. For **schoolmate**, we took a random set of faults. Finally, we used the following fault-localization techniques to assign suspiciousness ratings to all executed statements:

- $Alg = Tar, Jac, Och$. The "basic" fault-localization algorithms: *Tarantula*, *Jaccard*, and *Ochiai* presented in Section 3.1.
- SM. The technique of Section 3.2 based on using a source mapping in combination with positional information obtained from an oracle (HTML validator).
- $Alg{+}SM$. The combined technique described in Section 3.3 that combines a basic fault-localization algorithm with the use of the source mapping.
- $Alg{+}Mod$. The variation on a basic fault-localization algorithm presented in Section 3.5 in which conditional expressions are modeled as (condition, value) pairs, and return-value expressions are modeled as (expression, abstract value).

12. In general, a single fault may be responsible for multiple failures.

- $Alg{+}Mod{+}SM$. The variation on the basic fault-localization algorithm presented in Section 3.5, in which conditional expressions are modeled as (condition, value) pairs, and return-value expressions are modeled as (expression, abstract value) pairs, and combined with the use of the source mapping.
- $Alg{+}Mod$ **or** *SM*. A combined fault-localization technique that uses $Alg{+}Mod$ for execution failures, and *SM* 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 was comprised of the set of failing tests associated with the fault under consideration and the set of all passing tests.

Similarly to previous fault-localization studies [29], [18], [28], [39], 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.

In order to answer questions RQ3 and RQ4, we restricted our attention to execution failures for which the location of the fault is not immediately obvious from an error message.[13] We concentrated on execution failures because most HTML failures were already localized very effectively using the source mapping. In particular, we found that 60 of the 69 HTML failures that we reported in Table 2 were already well localized to $<1\%$ of executed statements using the source mapping. In these cases, the quality of the test suite has negligible impact. Execution failures, however, are localized exclusively using a statistical technique and are therefore highly suitable for evaluating the effectiveness of directed test-generation techniques.

We used the following four test-generation strategies to generate test suites used for fault localization:

- *Base.* Test generation using the combined concrete and symbolic execution algorithm in [9], which starts from an empty input, and aims to maximize branch coverage. We call this algorithm *Base* because we will use it as the baseline for comparison with the new similarity-based directed generation algorithms.
- *Coverage.* Test generation using the combined concrete and symbolic execution algorithm in [9], but starting test generation from the failing test.
- *PCS*. Test generation using the subset-based path-constraint similarity metric that was described in Section 4.3.
- *IS*. Test generation using the input similarity metric that was described in Section 4.4.

For each strategy and for each fault, we used *Apollo* to generate test suites. Then, for each test suite and each localized fault, we computed suspiciousness ratings for all executed statements using our "best" version of the *Ochiai* algorithm ($Och{+}Mod{+}SM$).[14] Finally, we computed the percentage of faults that are "well localized."

13. This error message includes some information about where the error occurred, and we are focused on crashes in which this information did not make the bug apparent.

14. We only report results for the *Ochiai*-based algorithm *Och+Mod+SM* because we found that algorithm to be superior, but results obtained using *Tar+Mod+SM* and *Jac+Mod+SM* are similar.

Our results were obtained on a MacBook Pro with a 2.4 GHz Intel Core i5 processor, 3 GB of Random Access Memory (RAM), and the Mac OS 10.6.5 (Snow Leopard) operating system. Apollo was run on a Sun Microsystems Java Standard Edition (SE) V1.6.1 Runtime Environment.

6.4 RQ1 and RQ2—Fault Localization Effectiveness

Tables 3, 4, and 5 show experimental results for each of the three underlying fault localization algorithms (*Tarantula*, *Ochiai*, and *Jaccard*), and variations on each algorithm (*Alg* SM, *Alg+SM*, $Tar{+}Mod$, *Alg+*$Mod{+}SM$, and *Alg+*Mod **or** SM) as discussed above. Each 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 percent of all statements, up to 10 percent of all statements, up to 20 percent of all statements, and so on. Consider, for example, the case where the $Tar{+}Mod{+}SM$ technique is used to localize faults in **faqforge** (Table 3). If a programmer inspects the statements reported by this technique in decreasing order of suspiciousness, then on average, he will need to inspect 4.1 statements until he has found the first faulty statement, and this corresponds to 0.6 percent of the executed statements. Furthermore, we can see that for 93.3 percent of the faults in **faqforge**, less than 1 percent of the executed statements needs to be inspected, and for the remaining 6.7 percent of the faults, between 1 and 10 percent of the executed statements need to be inspected. If the underlying algorithm is *Ochiai* (Table 5), a programmer would only need to inspect 3.0 statements on average for **faqforge**, which constitutes 0.4 percent of all 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 percent of all executed statements. Using this terminology, we can see that:

- Using the basic *Tarantula* algorithm, only 27.0 percent of all faults are well localized, on average (see the first row of data in the set of rows labeled **aggregated**). The basic *Jaccard* and *Ochiai* algorithms fare slightly better, with 37.4 percent of faults being well localized in each case.
- Using the source-mapping technique SM, 57.4 percent of all faults are well localized, on average over all subjects.
- Combining any fault-localization algorithm with the oracle ($Tar{+}SM, Jac{+}SM, Och{+}SM$) yields a technique that outperforms either of its constituents, with 71.3 percent of all faults being well localized on average, for each of the three techniques.
- Adapting *Tarantula* to use the modeling techniques (return value, conditionals) ($Tar{+}Mod$) is helpful by well localizing 47.0 percent of all faults versus the previously mentioned 27.0 percent for the statement-based *Tarantula* algorithm. Similar effects can be

TABLE 3
Results of Fault Localization Using the Different Fault Localization Techniques Using the *Tarantula* Algorithm

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
faqforge	*Tar*	53.7(7.7)	23.3	43.3	33.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	210.8(30.1)	70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	30.0
	Tar+SM	15.0(2.1)	70.0	20.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod	45.4(6.5)	46.7	26.7	23.3	3.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod **or** *SM*	25.2(3.6)	93.3	3.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.3
	Tar+Mod+SM	4.1(0.6)	93.3	6.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
webchess	*Tar*	164.9(18.7)	16.7	33.3	0.0	4.2	41.7	0.0	0.0	0.0	0.0	0.0	4.2
	SM	626.5(70.9)	29.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	70.8
	Tar+SM	119.8(13.6)	41.7	25.0	0.0	0.0	29.2	0.0	0.0	0.0	0.0	0.0	4.2
	Tar+Mod	58.1(6.6)	45.8	45.8	0.0	0.0	8.3	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod **or** *SM*	68.4(7.7)	58.3	37.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	Tar+Mod+SM	28.4(3.2)	62.5	37.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
schoolmate	*Tar*	500.4(18.0)	37.5	37.5	5.0	0.0	0.0	0.0	0.0	7.5	0.0	10.0	2.5
	SM	1249.3(45.0)	55.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	45.0
	Tar+SM	80.0(2.9)	85.0	12.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.5
	Tar+Mod	439.4(15.8)	50.0	27.5	5.0	0.0	0.0	0.0	0.0	0.0	7.5	0.0	10.0
	Tar+Mod **or** *SM*	350.2(12.6)	87.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.5
	Tar+Mod+SM	6.6(0.2)	95.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
timeclock	*Tar*	128.3(5.0)	17.6	52.9	29.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	304.7(11.8)	88.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	11.8
	Tar+SM	21.5(0.8)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod	130.6(5.1)	29.4	41.2	29.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod **or** *SM*	157.3(6.1)	88.2	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.9
	Tar+Mod+SM	22.8(0.9)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
phpsysinfo	*Tar*	44.5(1.1)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	3107.5(75.0)	25.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	75.0
	Tar+SM	42.3(1.0)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod	10.3(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Tar+Mod **or** *SM*	1041.3(25.1)	75.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	25.0
	Tar+Mod+SM	6.5(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
aggregated	*Tar*	243.0(10.1)	27.0	40.9	14.8	0.9	8.7	0.0	0.0	2.6	0.0	3.5	1.7
	SM	773.4(46.6)	57.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	42.6
	Tar+SM	61.4(4.1)	71.3	17.4	3.5	0.0	6.1	0.0	0.0	0.0	0.0	0.0	1.7
	Tar+Mod	196.5(6.8)	47.0	32.2	12.2	0.9	1.7	0.0	0.0	0.0	2.6	0.0	3.5
	Tar+Mod **or** *SM*	202.1(11.0)	82.6	9.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.8
	Tar+Mod+SM	12.9(1.0)	87.0	12.2	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

The columns of the table indicate 1) the subject program, 2) the fault localization technique used, 3) the average number of statements to inspect, and average percentage of statements to inspect, and 4)-14) indicate the percentage of faults in each range of percentage of statements to inspect.

observed when modeling techniques are combined with the *Jaccard* algorithm ($37.4\% \rightarrow 53.9\%$) and the *Ochiai* algorithm ($37.4\% \rightarrow 54.8\%$).

- For the combined technique that uses the $Tar{+}Mod$ technique for execution failures and the SM technique for HTML failures, between 82.6 percent (*Tarantula*) and 83.5 percent (*Jaccard*, *Ochiai*) of all faults are well localized.
- The most effective fault-localization technique is obtained by using the variant of each algorithm that combined modeling and the source-mapping techniques ($Alg{+}Mod{+}SM$). Using this technique, 87.0 percent of all faults are well localized for *Tarantula*, and 87.8 percent for *Jaccard* and *Ochiai*, on average over all subjects.

While we have discussed only aggregated data so far, the results appear to be consistent across the five subject applications. It is interesting to note that the effectiveness of the more precise modeling of conditionals and return value depends on whether the subject program contains any faults that consist of missing branches in conditions, or incorrect handling of return values. For one subject (**webchess**), this accounts for an almost 30 percent improvement in the number of well-localized faults over the basic *Tarantula* algorithm ($16.7\% \rightarrow 45.8\%$), whereas for another (**timeclock**), it makes a smaller difference of slightly more than 10 percent. In summary, we found that the $Tar{+}Mod{+}SM$ fault localization technique yields a more than threefold increase in the percentage of well-localized faults, when compared with the unenhanced *Tarantula* algorithm, and that the $Jac{+}Mod{+}SM$ and $Och{+}Mod{+}SM$ techniques yield a more than twofold increase over the unenhanced *Jaccard* and *Ochiai* algorithms. Most of this improvement is due to the use of the source mapping. This is undoubtedly due to the fact that many of the localized faults manifest themselves via malformed HTML output. Our treatment of conditional and return-value expressions accounts for a smaller part of the gains in precision, but is still helpful in the cases where the fault consists of a missing branch in a conditional statement, or a return value is incorrectly handled.

It is interesting to note that, since the source-mapping technique 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*, *Jaccard*, or *Ochiai*, with one based on source mapping. One could consider using different techniques for different kinds of faults (e.g., use *Tarantula* for execution failures, and the oracle-based technique for HTML failures). 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. For example, the combined technique $Tar{+}Mod$ **or** SM is less effective (82.6 percent of all statements being well localized) than the combined technique $Tar{+}Mod{+}SM$ (87.0 percent), and

TABLE 4
Results of Fault Localization Using the Different Fault Localization Techniques Using the *Jaccard* Algorithm

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
faqforge	*Jac*	45.4(6.5)	23.3	53.3	23.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	210.8(30.1)	70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	30.0
	Jac+SM	14.5(2.1)	70.0	20.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod	34.7(5.0)	50.0	36.7	13.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	24.7(3.5)	96.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.3
	Jac+Mod+SM	3.0(0.4)	96.7	3.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
webchess	*Jac*	130.3(14.7)	16.7	37.5	0.0	41.7	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	SM	626.5(70.9)	29.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	70.8
	Jac+SM	101.4(11.5)	41.7	25.0	0.0	29.2	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	Jac+Mod	59.6(6.7)	45.8	41.7	0.0	4.2	8.3	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	69.7(7.9)	58.3	33.3	0.0	0.0	4.2	0.0	0.0	0.0	0.0	0.0	4.2
	Jac+Mod+SM	44.1(5.0)	62.5	29.2	0.0	0.0	8.3	0.0	0.0	0.0	0.0	0.0	0.0
schoolmate	*Jac*	98.2(3.5)	62.5	35.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.5
	SM	1249.3(45.0)	55.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	45.0
	Jac+SM	79.6(2.9)	85.0	12.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.5
	Jac+Mod	39.3(1.4)	62.5	35.0	0.0	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	349.5(12.6)	87.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.5
	Jac+Mod+SM	5.5(0.2)	95.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
timeclock	*Jac*	56.6(2.2)	29.4	64.7	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	304.7(11.8)	88.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	11.8
	Jac+SM	19.6(0.8)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod	51.8(2.0)	41.2	52.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	155.7(6.0)	88.2	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.9
	Jac+Mod+SM	19.6(0.8)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
phpsysinfo	*Jac*	44.5(1.1)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	3107.5(75.0)	25.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	75.0
	Jac+SM	42.3(1.0)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod	10.3(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	1041.3(25.1)	75.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	25.0
	Jac+Mod+SM	6.5(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
aggregated	*Jac*	83.1(5.6)	37.4	45.2	7.0	8.7	0.0	0.0	0.0	0.0	0.0	0.0	1.7
	SM	773.4(46.6)	57.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	42.6
	Jac+SM	57.0(3.6)	71.3	17.4	3.5	6.1	0.0	0.0	0.0	0.0	0.0	0.0	1.7
	Jac+Mod	43.2(3.1)	53.9	38.3	4.3	1.7	1.7	0.0	0.0	0.0	0.0	0.0	0.0
	Jac+Mod **or** *SM*	201.8(11.0)	83.5	7.8	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.0	7.8
	Jac+Mod+SM	15.0(1.3)	87.8	9.6	0.9	0.0	1.7	0.0	0.0	0.0	0.0	0.0	0.0

The columns of the table indicate 1) the subject program, 2) the fault localization technique used, 3) the average number of statements to inspect, and average percentage of statements to inspect, and 4)-14) indicate the percentage of faults in each range of percentage of statements to inspect.

the results for the corresponding variations of *Jaccard* and *Ochiai* look similar.

Figs. 6, 7, and 8 show graphs depicting the aggregated data of the corresponding tables. 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 $Alg{+}Mod{+}SM$ technique outperforms all other techniques (for each of the basic algorithms). In particular, note that, for *any* percentage n between 0 and 100 percent, $Alg{+}Mod{+}SM$ localizes more faults than any of the other algorithms when up to n percent of all statements are examined in decreasing order of suspiciousness.

6.5 RQ3—Test-Generation Effectiveness

In the remainder of the evaluation section, we will only report results based on the $Och + Mod + SM$ algorithm because we found this algorithm to be the most effective one in Section 6.4, and because the results for the other algorithms are similar. We first discuss the "maximal" fault-localization effectiveness of the test suites generated by the four test-generation strategies above, as measured by the percentage of well-localized faults, assuming each strategy is given an infinite amount of time to construct a test suite. In practice, we found that it sufficed to have each strategy generate 100 tests for each fault, with the exception of **schoolmate**, which required 252 tests to reach a plateau. Generating more tests beyond this point resulted in larger test suites, but not in an increased number of well-localized faults.[15] Table 6 shows three columns for each subject program and each technique. These columns show, from left to right: 1) on average, for each subject program, the percentage of faults that is well localized, 2) on average, the absolute number of statements that need to be inspected to localize each fault, and 3) on average, the percentage of executed statements that need to be inspected to localize each fault. For example, for **faqforge**, both the *Base* and *PCS* techniques eventually localize 100 percent of the faults to within 1 percent of all executed statements. Furthermore, on average, each of these faults is localized by these techniques to 4.6 statements, which corresponds to 0.6 percent of all executed statements. The *Coverage* and *IS* generation techniques also reach 100 percent well-localized faults on **faqforge** eventually, albeit a slightly higher plateau of 5 and 5.1 statements, respectively, that need to be inspected, which corresponds to 0.7 percent of all executed statements.

In summary, the test-generation strategies are capable of generating test suites with nearly identical maximal fault-localization effectiveness when given an infinite amount of time. In particular, for **faqforge**, **schoolmate**, and **phpsysinfo**, 100 percent of all faults were eventually well localized by each technique. However, for **webchess**, only 77 percent of all faults were eventually well localized by each technique.

15. It is theoretically possible that further gains in fault-localization effectiveness could be achieved by generating additional tests, but we consider this to be very unlikely.

TABLE 5
Results of Fault Localization Using the Different Fault Localization Techniques Using the *Ochiai* Algorithm

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
faqforge	*Och*	45.2(6.5)	23.3	53.3	23.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	210.8(30.1)	70.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	30.0
	Och+SM	14.4(2.1)	70.0	20.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod	34.6(4.9)	50.0	33.3	16.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	24.7(3.5)	96.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.3
	Och+Mod+SM	3.0(0.4)	96.7	3.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
webchess	*Och*	130.3(14.7)	16.7	37.5	0.0	41.7	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	SM	626.5(70.9)	29.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	70.8
	Och+SM	101.4(11.5)	41.7	25.0	0.0	29.2	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	Och+Mod	34.2(3.9)	45.8	50.0	0.0	4.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	57.2(6.5)	58.3	37.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.2
	Och+Mod+SM	18.7(2.1)	62.5	37.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
schoolmate	*Och*	98.2(3.5)	62.5	35.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.5
	SM	1249.3(45.0)	55.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	45.0
	Och+SM	79.6(2.9)	85.0	12.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.5
	Och+Mod	23.1(0.8)	65.0	35.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	349.5(12.6)	87.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.5
	Och+Mod+SM	5.5(0.2)	95.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
timeclock	*Och*	56.5(2.2)	29.4	64.7	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	304.7(11.8)	88.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	11.8
	Och+SM	19.5(0.8)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod	51.7(2.0)	41.2	52.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	155.6(6.0)	88.2	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.9
	Och+Mod+SM	19.5(0.8)	88.2	5.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
phpsysinfo	*Och*	44.0(1.1)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SM	3107.5(75.0)	25.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	75.0
	Och+SM	41.8(1.0)	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod	10.3(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	1041.3(25.1)	75.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	25.0
	Och+Mod+SM	6.5(0.2)	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
aggregated	*Och*	83.1(5.6)	37.4	45.2	7.0	8.7	0.0	0.0	0.0	0.0	0.0	0.0	1.7
	SM	773.4(46.6)	57.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	42.6
	Och+SM	57.0(3.6)	71.3	17.4	3.5	6.1	0.0	0.0	0.0	0.0	0.0	0.0	1.7
	Och+Mod	32.2(2.4)	54.8	39.1	5.2	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Och+Mod **or** *SM*	199.2(10.8)	83.5	8.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.8
	Och+Mod+SM	9.7(0.7)	87.8	11.3	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

The columns of the table indicate 1) the subject program, 2) the fault localization technique used, 3) the average number of statements to inspect, and average percentage of statements to inspect, and 4)-14) indicate the percentage of faults in each range of percentage of statements to inspect.

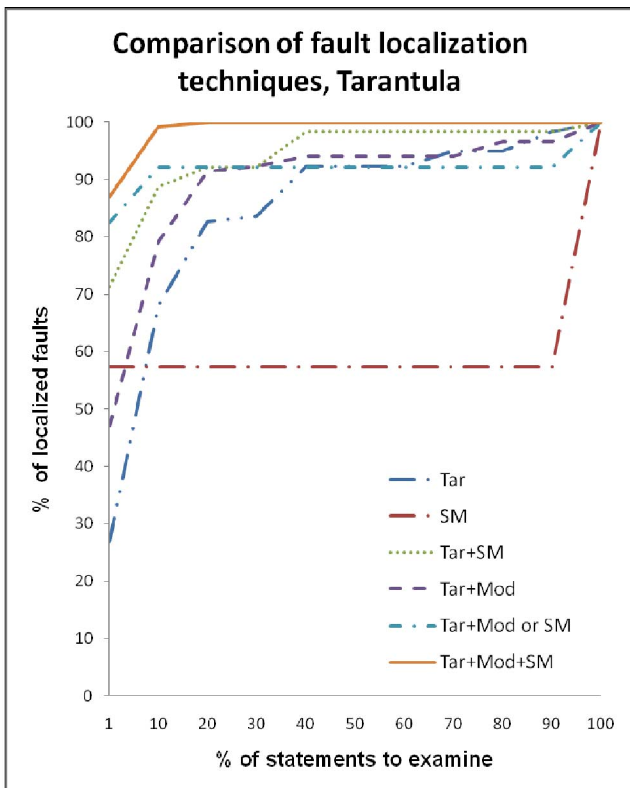


Fig. 6. Effectiveness comparison of different fault-localization techniques, using the *Tarantula* algorithm. X-axis: Percentage of statements that need to be inspected. Y-axis: Percentage of faults.

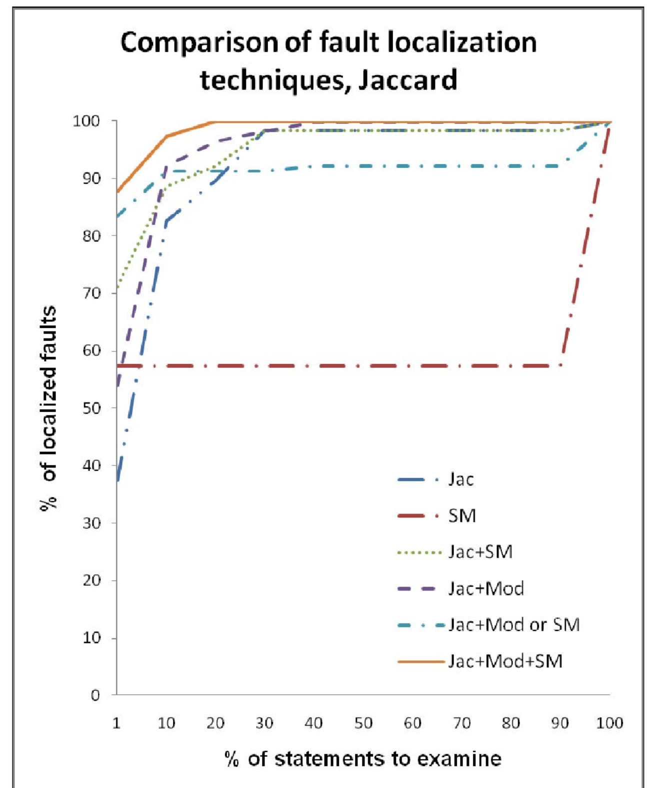


Fig. 7. Effectiveness comparison of different fault-localization techniques, using the Jaccard algorithm. X-axis: Percentage of statements that need to be inspected. Y-axis: Percentage of faults.

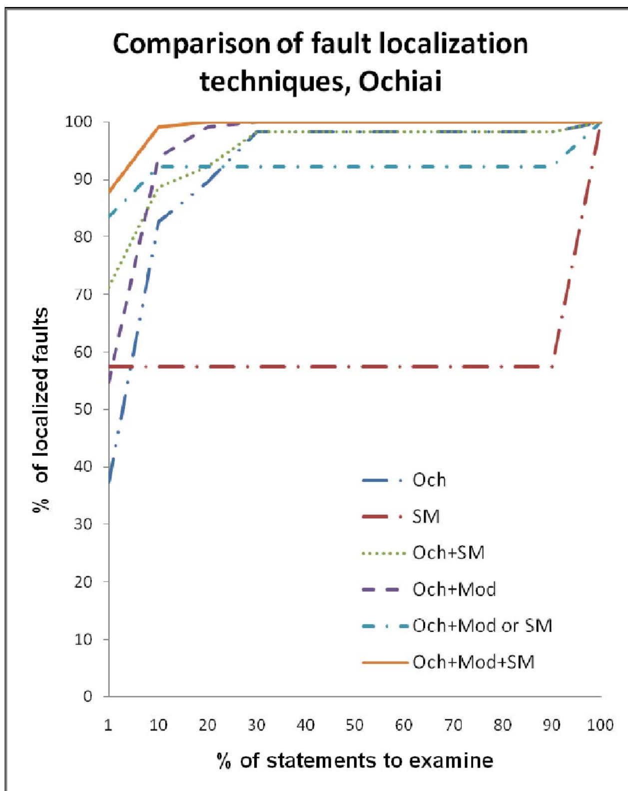


Fig. 8. Effectiveness comparison of different fault-localization techniques using the Ochiai algorithm. X-axis: Percentage of statements that need to be inspected. Y-axis: Percentage of faults.

6.6 RQ4—Test-Suite Size

As we have seen, the different test-generation techniques eventually achieve very similar effectiveness. However, the question remains to what extent the test-generation techniques require a different number of tests to reach this plateau. Table 7 shows two columns for each subject program and each test-generation technique. These columns show, from left to right: 1) the number of tests that is needed to reach the maximal percentage of well-localized faults as reported in Table 6, and 2) the time required to generate these tests. Here it should be noted that the time reported in point 2 is an average over all faults for the *Coverage*, *PCS*, and *IS* techniques. For the *Base* technique, there is just one test suite that is used for all faults, and the time reported is the time needed to generate that test suite.

As can be seen in Table 7, there is significant difference in how quickly the different test-generation techniques converge on the optimal result. For **faqforge**, the *Base* test-generation technique that we used in [7] requires 60 tests to reach the maximal percentage of well-localized faults, whereas the *PCS* technique requires only five tests. The amount of time required to generate a test suite differs similarly, with 63.6 seconds for the *Base* technique and only 7.3 seconds for the *PCS* technique. The graphs in Fig. 9 provide some more detail on how quickly the test-generation strategies converge toward their maximal effectiveness. Each graph shows the percentage of well-localized faults plotted against the number of generated tests, for each of the generation techniques. By examining the graphs, we can observe that the directed strategies (*IS* and *PCS*) converge much faster than the undirected strategies (*Coverage* and *Base*). In three of the four subject programs (**webchess**, **faqforge**, and **phpsysinfo**), the *PCS* strategy is superior. In the case of **schoolmate**, however, the *IS* strategy (seven tests) is slightly better than *PCS* (11 tests).

On the whole, we conclude that the *PCS* strategy is the preferred technique. On average, *PCS* requires only 6.5 tests to achieve the optimal number of well-localized faults, versus 46.8 tests for the *Base* strategy. This can be viewed as an improvement of $((46.8 - 6.5) * 100)/46.8 = 86.1\%$. Similarly, we notice that, on average, the *Base* strategy takes 131.2 seconds for test generation, compared to only 14.9 seconds required by *PCS*, for an improvement of 88.6 percent.

TABLE 6
Summary, for Each Test-Generation Technique and Subject Program, of the Percentage of Faults That Is Well Localized and the Absolute Number (# stmts) and Percentage (Percent stmts) of Executed Statements That Need to Be Inspected on Average Until the Fault Is Localized

program	*Base*			*Coverage*			*PCS*			*IS*		
	% wl	# stmts	% stmts	% wl	# stmts	% stmts	% wl	# stmts	% stmts	% wl	# stmts	% stmts
webchess	77	11.3	1.3	77	16.4	1.9	77	16.9	1.9	77	16.8	1.9
faqforge	100	4.6	0.6	100	5	0.7	100	4.6	0.6	100	5.1	0.7
schoolmate	100	11.4	0.4	100	6.8	0.2	100	4.6	0.2	100	4.6	0.2
phpsysinfo	100	17	0.7	100	17	0.7	100	14	0.6	100	14	0.6
Average	100	11.1	0.75	100	11.3	0.9	100	10	0.83	100	10.1	0.85

TABLE 7
Summary, for Each Test-Generation Technique and Subject Program, of the Time (Time(s) for *Base* and Avg Time(s) for *Coverage*, *PCS*, and *IS*) and Number of Tests (# Tests) Required to Achieve the Maximal Percentage of Well-Localized Faults as Reported in Table 6

program	*Base*		*Coverage*		*PCS*		*IS*	
	# tests	time(s)	# tests	avg time(s)	# tests	avg time(s)	# tests	avg time(s)
webchess	69	78.9	20	22.7	7	12.3	11	15.3
faqforge	60	63.6	37	47.5	5	7.3	22	28.9
schoolmate	18	20.1	253	386.2	11	14.9	7	9.4
phpsysinfo	40	362.3	87	818	3	25.2	8	68.4
Average	46.8	131.2	99.2	318.6	6.5	14.9	12	30.7

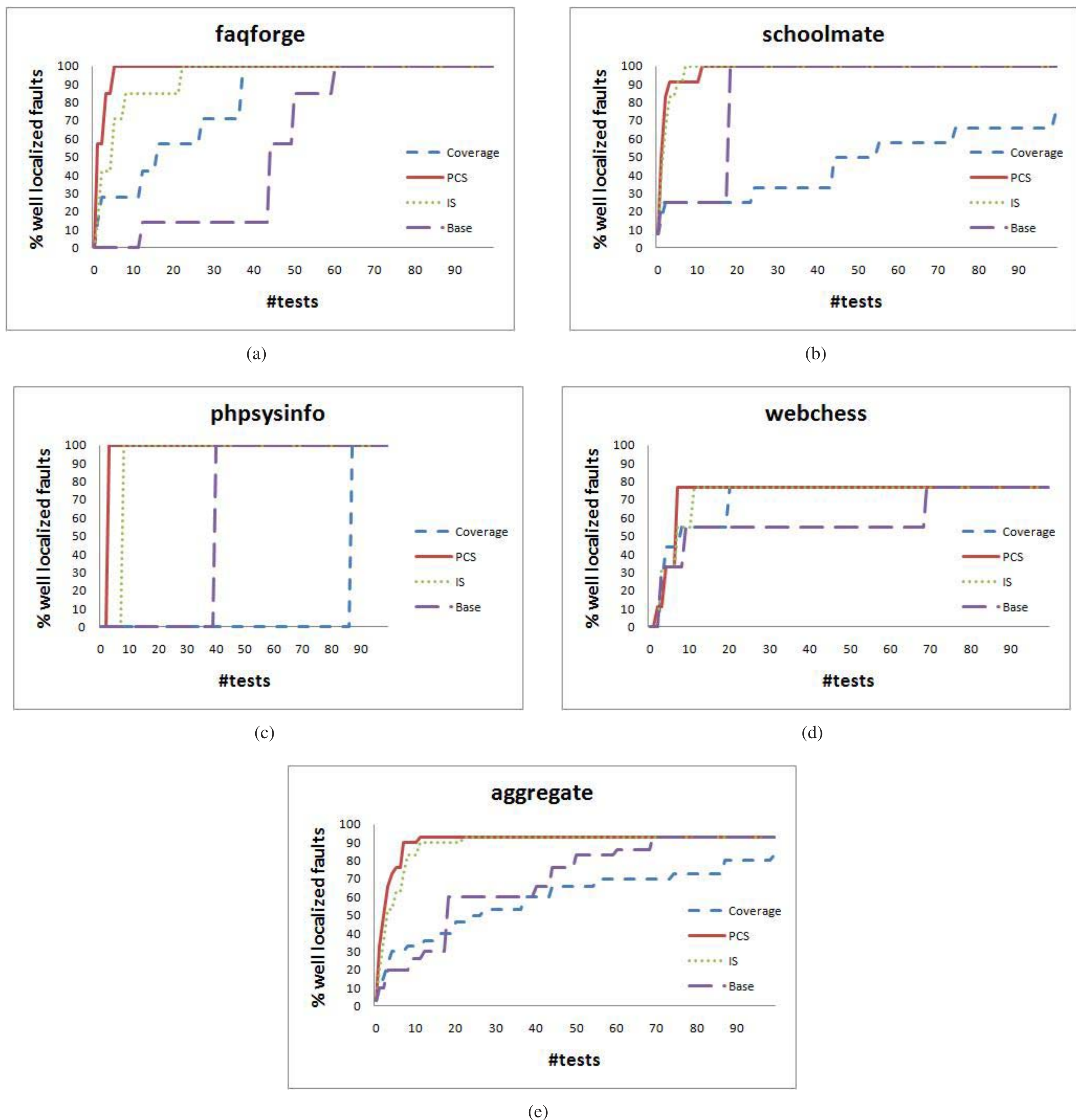


Fig. 9. Average number of statements to inspect for all the execution failure found in each subject program. (a) faqforge, (b) schoolmate, (c) phpsysinfo, (d) webchess, and (e) aggregated.

6.7 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 and are relatively small. 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 Table 1. The relatively small size of the programs allows us to understand them enough to be able to fix the bugs that we found, and our technique does not rely on small size per se, in that nothing in them scales with program size. Furthermore, the same subject programs were also used as subject programs by Minamide [34]. 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. 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., [29], [28], [39]).

Renieris and Reiss [38] describe an evaluation approach in which goodness of a report is based on the distance in a program dependence graph (PDG) from a report to an actual fault. The idea is that a programmer would follow program logic from a report, so distance in the program logic is a good metric. It seems clear that programmers would indeed follow their notion of the program logic, but it is not so clear that a formal PDG captures the intuitive notion of the programmer. Thus, while our assumption that the programmer would follow a linear order is perhaps oversimplified, it is a simple metric and it is not clear how to improve it.

7 Related Work

This section reviews the literature on fault localization, focusing primarily on fault localization techniques that predict the location of faults based on the analysis of data from multiple tests or executions. In addition, we discuss research that explores the impact of test-suite composition on fault-localization effectiveness.

7.1 Fault Localization

7.1.1 Early Work

Early work on fault localization relied on the use of program slicing [42]. Lyle and Weiser [33] 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 were later explored by Pan and Spafford [35] and Agrawal et al. [4].

7.1.2 Trace Comparisons

Renieris and Reiss [38] use *set-union*, *set-intersection*, and *nearest neighbor* methods for fault localization; these all work by comparing execution traces of passing and failing program runs.

- **set-union.** It 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) [22].
- **set-intersection.** It 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.
- **nearest neighbors.** It selects the passing test case whose execution spectrum most closely resembles that of the failing test case according to one of two distance criteria,[16] 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 a ranking technique in which additional statements are considered based on their distance to previously reported statements along edges in the System Dependence Graph [22].

Nearest Neighbor was evaluated on the Siemens suite [23], and was found to be superior to the *set-union* and *set-intersection* techniques.

7.1.3 Tarantula, Ochiai, and Jaccard

Tarantula [29], *Ochiai* [3], and *Jaccard* [16], [24] are different similarity coefficients that have been used for fault localization, by computing for each program construct a *suspiciousness rating* that reflects the likelihood for that construct to contribute to a failure. This suspiciousness rating of a program construct is computed as a function of the ratio of passing and failing executions that exercise it. In this paper, we apply the *Tarantula*, *Ochiai*, and *Jaccard* fault-localization algorithms in a new domain: fault-localization for web applications written in PHP. Previous evaluations of these algorithms 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 fault localization, 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.

7.1.4 Empirical Evaluations

Jones and Harrold [28] conducted a detailed empirical evaluation in which they apply *Tarantula* to faulty versions of the Siemens suite [23], and compare its effectiveness to that of several other fault-localization techniques. In the fault-localization literature, the effectiveness of a fault-localization technique is customarily measured 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 [18], [3], [38], [28]. Santelices et al. [39] 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 their comparison is that the branch-based algorithm is more precise than the statement-based one, and that def-use-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. We also

16. 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.

explore special treatment of branches, but with a rather different goal. Santelices et al. use branch information to impute suspiciousness to specific statements; we are using branches combined with branch outcomes to impute blame to specific control-flow paths to approximate where missing code ought to be.

Recent papers by Jones and Harrold [28] and Abreu et al. [3] present empirical evaluations of fault localization techniques, including several of the techniques discussed above, using the Siemens suite. The emerging consensus appears to be that the *Ochiai* similarity metric advocated by Abreu et al. [3] is more effective than *Tarantula* [39], [2] and *Jaccard* [16], [24]. Our experiments appear to confirm this.

7.1.5 Statistical Debugging

Liblit et al. [30] present a fault-localization technique that analyzes the correlation between executed branches with failures using regularized logistic regression. As the focus of Liblit et al.'s work is on fault localization in deployed software, where runtime overhead needs to be kept at a minimum, the approach is sampling based. Liblit et al. show that effective bug isolation can even be done with low sampling frequencies such as 1/1,000. In later work, Liblit et al. extended sampling-based statistical debugging to: 1) handle scenarios with multiple faults [31], 2) cover complex Boolean formulas rather than the simple success/failure of predicates [10], and 3) isolate concurrency-related bugs [27]. Recently, Arumuga Nainar and Liblit [11] showed how the runtime overhead of sampling-based statistical debugging can be reduced by adaptive reinstrumentation of deployed code.

7.1.6 Other Techniques

Dallmeier et al. [19] 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 [18], [48], Zhang et al. [49], and Jeffrey et al. [25] 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 [31], [32] or path profiles [17] and failures, time spectra [45], and correlations between changes made by programmers and test failures [41], [37]. In recent work by Zhang et al. [50], suspiciousness scores are associated with basic blocks and control-flow edges, and computed by solving a set of equations (using, e.g., Gaussian elimination) that reflect control flow between basic blocks. Park et al. [36] recently described an approach for fault localization in concurrent Java programs in which occurrences of nonserializable access patterns are correlated with failures using the *Jaccard* formula.

Baah et al. [12] recently applied causal-inference techniques to the problem of fault localization in order to control the confounding bias caused by unknown variables. They propose variations on a number of fault-localization algorithms, including *Tarantula* and *Ochiai*, that correct for this bias, and show that this can improve fault localization.

7.2 The Impact of Test-Suite Composition on Fault-Localization Effectiveness

Our work on directed test generation is primarily aimed at scenarios where a failure occurs but where no test suite is available to apply statistical fault localization techniques. To address this scenario, we use a variation on combined concrete and symbolic execution [20], [40], [15], [21], [44] that is guided by various similarity metrics to generate test suites composed of tests with execution characteristics that are similar to those of the failing test. This is the same spirit as the Nearest Neighbors algorithm [38], but instead of *selecting* tests that are similar to a given failing test, we are *generating* such tests. Below, we discuss several categories of related research.

7.2.1 Generating Passing Tests That Are Similar to Failing Tests

Wang and Roychoudhury [43] present a fault localization technique that generates a similar successful run given a failing run when that is possible. Here, the basic idea is to find a program execution by attempting to invert specific conditionals. They start at the end of the execution to get the most similar successful execution. Their mechanism is similar to ours at the level of inverting conditions and using a constraint solver to derive inputs that yield that outcome. However, they focus on generating a specific successful run similar to a specific failing run, and do not employ statistical techniques that might help generalize across multiple successes and failures. Also, in general, their technique requires that "checking whether [a given similar run] was successful has to be done manually." This contrasts with our use of an oracle to determine success or failure.

Zeller introduced *Delta Debugging* [47] which minimizes some aspect of a test that induces failure. For instance, given a large source file that causes a compiler to crash, this technique may be able to isolate the particular problematic portion of the file. This technique isolates the important differences between a succeeding and failing execution by systematically reexecuting the program on inputs "in between" these executions. Unlike our work on directed test generation, Zeller's work is primarily focused on minimizing a failing input rather than systematically generating inputs to look for failures. Also, as with Wang and Roychoudhury, this work focuses on changing specific tests rather than applying statistical analysis to a collection of tests.

7.2.2 The Impact of Test-Suite Composition on Fault Localization

Several other projects have explored the relationship between the composition of a test suite and its effectiveness for fault localization. Baudry et al. [13] study how the fault-localization effectiveness of a test suite can be improved by adding tests. They propose the notion of a *dynamic basic block*, which is a set of statements that is covered by the same tests, and a related testing criterion that aims to maximize the number of dynamic basic blocks. Baudry et al. use a genetic algorithm for deriving new tests from existing ones by a series of mutation operations. Our research also aims to improve fault localization effectiveness by creating tests, but our starting point is a situation where no test suite

is available. In such cases, it is not clear how mutation-based approaches, which generate new tests from existing ones, could be applied.

Other researchers have focused on the opposite problem: determining how *reducing* the size of a test suite impacts fault localization effectiveness. Yu et al. [46] study the impact of several test-suite reduction strategies on fault localization. They conclude that statement-based reduction approaches negatively affect fault localization effectiveness, but that vector-based reduction approaches, which aim to preserve the set of statement vectors exercised by a test suite, have negligible effects on effectiveness. Jiang et al. [26] also study the impact of test-suite reduction strategies on fault-localization effectiveness. One of the strategies they consider (AS) prefers those test cases that maximally increase the number of additional statements covered. They report that reducing a test suite to half its original size according to this strategy only has minimal impact on fault-localization effectiveness.

7.2.3 Other Directed Symbolic Execution

In [14], the authors investigate three strategies to direct combined concrete and symbolic execution to improve coverage. Heuristics based on the Control-Flow Graphs (CFGs) of functions are evaluated in the context of real C programs. The most effective CFG-based heuristic is able to improve branch coverage substantially (by more than a factor of 2) for their largest program. The mechanism of using some explicit metric to direct the search resembles ours, but the goal is very different since they are trying to increase coverage by generating new different inputs whereas we are trying to improve fault-localization effectiveness by generating new similar inputs.

8 Conclusions

Until now, statistical fault-localization techniques that analyze execution data from multiple tests [29], [30], [31], [3], [39] have been applied primarily in the context of traditional imperative programming languages such as C and Java. In this paper, we have shown how such fault-localization techniques can be made effective in the domain of PHP web applications. We have presented two enhancements to the existing *Tarantula*, *Jaccard*, and *Ochiai* fault-localization techniques that greatly increase their effectiveness:

- The use of an extended domain of (statement, runtime value) pairs for conditional statements and function calls.
- The use of source mapping that correlates statements in the PHP application with the fragments of output that they produce at runtime.

The former helps with the localization of certain common kinds of failures such as missing branches in conditional statements and the return of unexpected values by functions due to corner cases that are not handled properly. The latter increases the precision of fault localization by leveraging the fact that PHP applications are expected to produce syntactically valid HTML output, and that the location of errors in these generated pages can often be determined quite precisely.

A key limitation of traditional fault-localization techniques has been that they require the existence of a suite of passing and failing tests. This renders these techniques powerless in the all-too-common scenario where a programmer is confronted with a failure, but where no test suite is available. To address this case, we have presented an approach for directed test generation, based on combined concrete and symbolic execution, that can be used to generate test suites with excellent fault-localization characteristics. This approach involves parameterizing the symbolic execution framework with a similarity criterion, which measures the similarity between two executions and directs the generation of tests to give preference to the creation of tests with maximal similarity to a given failing test.

We implemented these techniques in *Apollo*, and evaluated the techniques on several open-source PHP applications for which we had previously detected many failures [9]. We determined experimentally that a variant of the *Ochiai* algorithm that includes all our enhancements was the most effective, by localizing faults to within 0.7 percent of all executed statements, on average, over all faults in all subject programs, which is a significant improvement from the 5.6 percent for the unenhanced *Ochiai* algorithm. Moreover, our enhanced version of *Ochiai* localized 87.8 percent of all faults to within 1 percent of all executed statements, compared to only 37.4 percent for the unenhanced *Ochiai* algorithm. We obtained similar, though slightly worse results for the *Tarantula* and *Jaccard* algorithms.

We also implemented the directed test-generation strategies in *Apollo* as variations of the combined concrete and symbolic execution framework in [9]. We experimentally determined that all test-generation strategies that we considered are capable of generating test suites with maximal fault-localization effectiveness, when given an infinite time budget for test generation. However, on average, with a directed strategy based on path-constraint similarity, this maximal effectiveness was achieved after generating only 6.5 tests, compared to 46.8 tests for an undirected test-generation strategy. Accordingly, our directed technique reduces test-suite size by 86.1 percent and test-suite generation time by 88.6 percent when compared to a traditional undirected test-generation strategy.

As part of future work, we plan to experiment with additional similarity metrics that can be used for directed test generation, and evaluate how their effectiveness compares to the path-constraint and input similarity metrics presented in this paper. In addition, we plan to develop techniques for synthesizing fixes for the HTML errors detected by Apollo. We are also interested in developing test generation and fault-localization techniques for client-side JavaScript code that is executed in a browser. A recent paper [5] presents a first step in this direction.

Acknowledgments

Parts of the research presented in this paper were previously presented in two conference papers by the same authors [7], [6].

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," *Proc. 12th Pacific Rim Int'l Symp. Dependable Computing,* pp. 39-46, 2006.

[2] R. Abreu, P. Zoeteweij, and A.J. van Gemund, "On the Accuracy of Spectrum-Based Fault Localization," *Proc. Testing: Academic and Industry Conf. Practice and Research Techniques,* pp. 89-98, Sept. 2007.

[3] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," *Proc. 12th Pacific Rim Int'l Symp. Dependable Computing,* pp. 39-46, 2006.

[4] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong, "Fault Localization Using Execution Slices and Dataflow Tests," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 143-151, 1995.

[5] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip, "A Framework for Automated Testing of Javascript Web Applications," *Proc. Int'l Conf. Software Eng.,* 2011.

[6] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed Test Generation for Effective Fault Localization," *Proc. 19th Int'l Symp. Software Testing and Analysis,* pp. 49-60, 2010.

[7] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical Fault Localization for Dynamic Web Applications," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.,* vol. 1, pp. 265-274, 2010.

[8] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding Bugs in Dynamic Web Applications," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 261-272, 2008.

[9] 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 Trans. Software Eng.,* vol. 36, no. 4 pp. 474-494, July/Aug. 2010.

[10] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical Debugging Using Compound Boolean Predicates," *Proc. Int'l Symp. Software Testing and Analysis,* S. Elbaum, ed., July 2007.

[11] P. Arumuga Nainar and B. Liblit, "Adaptive Bug Isolation," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.,* pp. 255-264, 2010.

[12] G.K. Baah, A. Podgurski, and M.J. Harrold, "Causal Inference for Statistical Fault Localization," *Proc. 19th Int'l Symp. Software Testing and Analysis,* pp. 73-84, 2010.

[13] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," *Proc. 28th Int'l Conf. Software Eng.,* L.J. Osterweil, H.D. Rombach, and M.L. Soffa, eds., pp. 82-91, 2006.

[14] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 443-446, 2008.

[15] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically Generating Inputs of Death," *Proc. Conf. Computer and Comm. Security,* 2006.

[16] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 595-604, 2002.

[17] T.M. Chilimbi, B. Liblit, K.K. Mehra, A.V. Nori, and K. Vaswani, "Holmes: Effective Statistical Debugging via Efficient Path Profiling," *Proc. 31st Int'l Conf. Software Eng.,* pp. 34-44, May 2009.

[18] H. Cleve and A. Zeller, "Locating Causes of Program Failures," *Proc. Int'l Conf. Software Eng.,* pp. 342-351, May 2005.

[19] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," *Proc. European Conf. Object-Oriented Programming,* pp. 528-550, 2005.

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2005.

[21] P. Godefroid, M.Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," *Proc. Symp. Network and Distributed System Security,* 2008.

[22] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 1 pp. 26-60, 1990.

[23] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. Int'l Conf. Software Eng.,* pp. 191-200, 1994.

[24] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data.* Prentice-Hall, Inc., 1988.

[25] D. Jeffrey, N. Gupta, and R. Gupta, "Fault Localization Using Value Replacement," *Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis,* B.G. Ryder and A. Zeller, eds., pp. 167-178, 2008.

[26] B. Jiang, Z. Zhang, T. Tse, and T.Y. Chen, "How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization," *Proc. 33rd Ann. IEEE Int'l Computer Software and Applications Conf.,* July 2009.

[27] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation," *Proc. 25th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* M. Rinard, ed., Oct. 2010.

[28] J.A. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 273-282, 2005.

[29] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," *Proc. Int'l Conf. Software Eng.,* pp. 467-477, 2002.

[30] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. Conf. Programming Language Design and Implementation,* pp. 141-154, 2003.

[31] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation," *Proc. Conf. Programming Language Design and Implementation,* pp. 15-26, 2005.

[32] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff, "Sober: Statistical Model-Based Bug Localization," *Proc. European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 286-295, 2005.

[33] J. Lyle and M. Weiser, "Automatic Bug Location by Program Slicing," *Proc. Second Int'l Conf. Computers and Applications,* pp. 877-883, 1987.

[34] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," *Proc. Int'l Conf. World Wide Web,* 2005.

[35] H. Pan and E.H. Spafford, "Heuristics for Automatic Localization of Software Faults," Technical Report SERC-TR-116-P, Purdue Univ., July 1992.

[36] S. Park, R.W. Vuduc, and M.J. Harrold, "Falcon: Fault Localization in Concurrent Programs," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.,* pp. 245-254, 2010.

[37] X. Ren and B.G. Ryder, "Heuristic Ranking of Java Program Edits for Fault Localization," *Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis,* D.S. Rosenblum and S.G. Elbaum, eds., pp. 239-249, 2007.

[38] M. Renieris and S.P. Reiss, "Fault Localization with Nearest Neighbor Queries," *Proc. IEEE Int'l Conf. Automated Software Eng.,* pp. 30-39, 2003.

[39] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold, "Lightweight Fault-Localization Using Multiple Coverage Types," *Proc. 31st Int'l Conf. Software Eng.,* pp. 56-66, May 2009.

[40] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proc. European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* 2005.

[41] M. Stoerzer, B.G. Ryder, X. Ren, and F. Tip, "Finding Failure-Inducing Changes in Java Programs Using Change Classification," *Proc. SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 57-68, Nov. 2006.

[42] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages,* vol. 3, no. 3 pp. 121-189, 1995.

[43] T. Wang and A. Roychoudhury, "Automated Path Generation for Software Fault Localization," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 347-351, 2005.

[44] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic Test Input Generation for Web Applications," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 249-260, 2008.

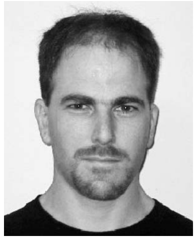
[45] C. Yilmaz, A.M. Paradkar, and C. Williams, "Time Will Tell: Fault Localization Using Time Spectra," *Proc. 30th Int'l Conf. Software Eng.,* W. Schäfer, M.B. Dwyer, and V. Gruhn, eds., pp. 81-90, 2008.

[46] Y. Yu, J.A. Jones, and M.J. Harrold, "An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization," *Proc. Int'l Conf. Software Eng.,* pp. 201-210, 2008.

[47] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *SIGSOFT Software Eng. Notes,* vol. 24, no. 6 pp. 253-267, 1999.

[48] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 1-10, Nov. 2002.

[49] X. Zhang, N. Gupta, and R. Gupta, “Locating Faults through Automated Predicate Switching,” *Proc. Int'l Conf. Software Eng.*, pp. 272-281, 2006.
[50] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, “Capturing Propagation of Infected Program States,” *Proc. Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 43-52, 2009.



Shay Artzi received the BS and MS degrees in computer science from the Technion (Israel Institute of Technology), and the PhD degree in computer science from MIT with a thesis entitled “Dynamically Fighting Bugs: Detection, Prevention, and Elimination.” He is a researcher at the IBM T.J. Watson Research Center in Hawthorne, New York. He has published numerous conference papers and journal articles on various aspects of improving software quality.

Julian Dolby was educated at the University of Wisconsin-Madison as an undergraduate, and at the University of Illinois at Urbana-Champaign as a graduate student, where he worked with Professor Andrew Chien on programming systems for massively parallel machines. He has been a research staff member at the IBM Thomas J. Watson Research Center since 2000. He works on a range of topics, including static program analysis and software testing. His program analysis work has recently been focused on scripting languages like JavaScript and on security analysis of web applications; this work has been included in IBM products, most notably Rational AppScan Standard Edition 8.0, and he is one of the primary authors of the publicly available Watson Libraries for Analysis (WALA) program analysis infrastructure. His testing work has been primarily focused on web applications in the Apollo project, and on finding concurrency bugs using both dynamic execution and model checking.

Frank Tip received the PhD degree from the University of Amsterdam in 1995. Since then, he has been with IBM Research, where he is currently managing the Program Analysis and Transformation Group. His current research interests include refactoring, test generation and fault localization for web applications, data-centric synchronization and declarative object identity for object-oriented programming languages, and change impact analysis.

Marco Pistoia received the PhD degree in mathematics from the Polytechnic Institute of New York University in May 2005. He is a manager, research staff member, and master inventor at the IBM Thomas J. Watson Research Center in Hawthorne, New York, where he manages the Mobile and Collaboration Solutions Group. He has written 10 books and published numerous papers and journal articles on various aspects of program analysis and language-based security. During his career, he has been the recipient of several awards, including two ACM SIGSOFT Distinguished Paper Awards.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.