RC24989 (W1005-021) May 4, 2010
Computer Science

IBM Research Report

Directed Test Generation for Improved Fault Localization

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

Directed Test Generation for Improved Fault Localization

Shay Artzi Julian Dolby Frank Tip Marco Pistoia

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

ABSTRACT

Fault-localization techniques that apply statistical analyses to execution data gathered from multiple tests are quite effective when a large test suite is available. However, if no test suite is available, what is the best approach to generate one? This paper investigates the fault-localization effectiveness of test suites generated according to several test-generation techniques based on combined concrete and symbolic (concolic) execution. We evaluate these techniques by applying the *Ochiai* fault-localization technique to generated test suites in order to localize 35 faults in four PHP Web applications. Our results show that the test-generation techniques under consideration produce test suites with similar high fault-localization effectiveness, when given a large time budget. However, a new, "directed" test-generation technique, which aims to maximize the similarity between the path constraints of the generated tests and those of faulty executions, reaches this level of effectiveness with much smaller test suites. On average, when compared to test generation based on standard concolic execution techniques that aims to maximize code coverage, the new directed technique preserves fault-localization effectiveness while reducing test-suite size by 86.1% and test-suite generation time by 88.6%.

1. INTRODUCTION

When a test fails, developers need to find the location of the fault in the source code before they can fix the problem. In recent years, a number of automated techniques have been proposed to assist programmers with this task, which is usually called *fault localization*. Many fault-localization techniques attempt to predict the location of a fault by applying statistical analyses to data obtained from the execution of multiple tests (see, for example, [18, 19, 20, 17, 2, 26]). The basic idea is that a statement [18], control-flow predicate [19], or def-use pair [26] is more *suspicious* (or more likely to be responsible for a test failure) if it correlates strongly with failing tests, and less suspicious if it correlates strongly with passing tests.

For a given application that contains a fault and a given fault-localization algorithm, one way to measure a test suite's suitability for fault localization is the number of statements that must be explored until the fault is found, assuming that statements are explored in order of decreasing suspiciousness. However, as in previous work on fault localization [17, 4], we instead choose to concentrate on the percentage of faults that are *well-localized*—these are faults for which less than 1% of all executed statements need to be examined until the fault is found, assuming that statements are examined in order of decreasing suspiciousness. In this paper, we explore a number of strategies for *generating* test suites and measure their *fault-localization effectiveness*, defined as the percentage of well-localized faults for that suite. Our goal is to determine which test-generation strategy achieves the best tradeoff between size and effectiveness for the test suites that it generates.

The research presented in this paper was conducted in the context of *Apollo* [5, 6, 4], a tool that uses combined *conc*rete and symb*olic* (*concolic*) execution [12, 27, 9, 13, 30] to generate failure-inducing inputs for PHP Web applications. *Apollo* currently targets two kinds of failures: (i) *HTML failures*, in which the application generates malformed HTML, and (ii) *execution failures*, which manifest themselves by a program crash or an obtrusive error message. In recent work [4], we incorporated several variations of the *Tarantula* fault-localization technique [18] in *Apollo*, and demonstrated that these can localize faults quite well using a test suite that was generated using concolic execution. However, in those previous experiments, the test suites were not generated with fault localization in mind, but with the dual objective of maximizing code coverage and finding as many failures as possible. Therefore, it is not clear whether the test suites used in [4] have maximal effectiveness for fault localization. In particular, the question is if even better fault localization can be achieved using test suites that are generated with other test-generation strategies. Furthermore, it is conceivable that the use of other test-generation strategies might enable equally effective fault localization using much smaller test suites.

In this paper, we assume a scenario where a user has just encountered a failure and where no test suite is available. For such situations, we want to answer the following research question: *What is the best strategy for generating a test suite that has maximal effectiveness for fault localization?* Notice that the effectiveness of fault-localization techniques is premised on the availability of a high-quality test suite that provides good coverage of the application's statements. However, even 100% coverage does not guarantee good fault localization.

Consider, for example, a scenario where two statements s_1 and s_2 are executed by exactly the same tests and where *Tarantula* [18] or *Ochiai* [2] is used for fault localization. These fault-localization techniques compute a program construct's suspiciousness rating from the number of passing tests and failing tests that execute it. Hence, if s_1 and s_2 are executed by the same tests, those techniques will report them as being equally suspicious. In such cases, creating an additional test that executes s_1 but not s_2 (or vice versa) may enable better fault localization. In this paper, we present a number of strategies for generating tests that are parameterized by a *similarity criterion*, and evaluate their fault-localization effectiveness. The similarity criterion measures how similar the execution characteristics associated with two tests are, and is used to *direct* concolic execution towards generating tests whose execution characteristics are similar to those of a given failing test.

We implemented the techniques in *Apollo*, an automated tool that detects and localizes faults in PHP Web applications. We evaluated the test-generation techniques by localizing 35 faults in four PHP applications. The results that we present in Section 5 show that a new, directed test-generation technique based on *path-constraint similarity* 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 concolic execution algorithm of [4], which aims to maximize code coverage, our directed technique reduces test-suite size by 86.1% and test-suite generation time by 88.6%.

The remainder of this paper is organized as follows. Section 2 presents a motivating example. Section 3 provides details on our techniques for concolic execution and fault localization. Section 4 presents the similarity criteria with which our directed test generation algorithms are parameterized. Section 5 presents the implementation and experimental results. Finally, related work is discussed in Section 6, followed by conclusions in Section 7.

2. EXAMPLE

Figure 1 shows a simple PHP script that scales one point to have the same magnitude as another. Object-oriented code in PHP resembles code in C++ or Java. The `point` class in Figure 1 declares two fields, `x` and `y`. Also declared in class `point` are methods `magnitude()`, which computes the distance of a point from the origin, and `scale()`, which scales a point to have the same magnitude as an argument point. Two PHP features of particular note are the `isset()` construct for checking whether a variable has been defined, and the `$_REQUEST` associative array, which is used to access input values that were provided to a PHP script.

The script code that follows the class definition in Figure 1 creates a `point` object and then initializes its `x` and `y` fields with values `x` and `y` that were provided as inputs to the script by accessing $_REQUEST. The program then calls `isset()` to check if input values `scale_x` and `scale_y` have been provided to the script. If so, the program creates and initializes the point `$scale` by which to scale the point `$x`. The intended use of this script is to be given two points, and scale one of them according to the other. There are two issues with this code:

1. This scaling is not well defined for the origin, which causes a division by zero in the `scale()` method in that case. We will assume that the desired fix for this is to leave the origin point unscaled. Note that the division by zero is a warning rather than an error in PHP, and so execution continues using a default value—0 in this case—as the "result" of the division.

2. There is also an inconsistency if the scaling parameters are not provided: In this case, the `scale` point will not be created, but it will be used anyway, resulting in attempting to call `magnitude()` on a non-object. This error aborts the PHP script, and execution ceases at that point. We will assume that, in the absence of the `scale_x` and `scale_y` parameters, the intended behavior is not to scale the point.

Our fault-localization procedure assumes that we have some test—either written by hand or generated by some technique, possibly *Apollo*—that exposes a given failure. There are many tests that could reveal these failures, and so we choose the following ones arbitrarily.

```
<?php
error_reporting( E_ALL );

class point {
 var $x;
 var $y;

 function magnitude() {
   return sqrt($this->x*$this->x+$this->y*$this->y);
 }

 function scale($p) {
   $factor = $p->magnitude() / $this->magnitude();
   $this->x = $this->x * $factor;
   $this->y = $this->y * $factor;
 }
};

$x = new point();
$x->x = $_REQUEST['x'];
$x->y = $_REQUEST['y'];

if (isset($_REQUEST['scale_x']) &&
   isset($_REQUEST['scale_y'])) {
 $scale = new point();
 $scale->x = $_REQUEST['scale_x'];
 $scale->y = $_REQUEST['scale_y'];
}

$x->scale($scale);

print $x->x." ".$x->y." ".$x->magnitude()."\n";
?>
```

Figure 1: Example PHP Program with Bugs

1. The first fault can be revealed by any choice of inputs that defines the `scale_x` and `scale_y` parameters and uses 0 for both `x` and `y`. In this case, the `if` test will succeed, and so the script will call `scale()`. Then the call to `magnitude()` on `$this` in `scale()` will return 0, triggering the divide-by-zero failure.

2. The second fault can be revealed by any input that does not define either `scale_x` or `scale_y` or both. In this case, the `if` test will fail, causing the code that creates and initializes `$scale` not to execute. In this case, the script will try to scale `$x` by the undefined `$scale`, resulting in an error when it tries to access a field of the undefined value.

3. APPROACH

This section describes in detail our solution for directed test generation for fault localization.

3.1 Concolic Testing

We begin by briefly reviewing the combined concrete and symbolic execution algorithm as embodied in *Apollo* [4]. The idea of this algorithm is to execute an application on some initial input

```
parameters: Program $\mathcal{P}$, Seed Input $\mathcal{I}_0$
result    : Tests $\mathcal{T}$;
$\mathcal{T}$ : setOf(⟨input,output⟩)
1  $\mathcal{T} := \emptyset$;
2  toExplore := getConfigs($\mathcal{I}_0$);
3  while toExplore $\neq \emptyset$ && !timeExpired() do
4      input := selectionMethodology.nextInput(toExplore);
5      output := executeConcrete($\mathcal{P}$, input);
6      $\mathcal{T} := \mathcal{T} \cup \{\langle$input,output$\rangle\}$;
7      toExplore := toExplore $\cup$ getConfigs(input);
8  return $\mathcal{B}$;

9  Subroutine getConfigs(input):
10 configs := $\emptyset$;
11 $c_1 \wedge \ldots \wedge c_n$ := executeSymbolic($\mathcal{S}_0$, $\mathcal{P}$, input);
12 foreach $i = 1, \ldots, n$ do
13     newPC := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
14     input := solve(newPC);
15     if input $\neq \bot$ then
16         enqueue(configs, ⟨newPC, input⟩);
17 return configs;
```

Figure 2: A Simplified Test-generation Algorithm

(e.g., an arbitrarily or randomly chosen input), and then on additional inputs obtained by solving constraints derived from exercised control-flow paths that capture the execution's dependency on program input.

Figure 2 shows the simplified pseudocode of our test-generation algorithm.[1] The inputs to the algorithm are a program $\mathcal{P}$, and an initial seed input (for simplicity the seed input parameter is singular, but in practice it is possible to supply a set of seed inputs). The output of the algorithm is a set of tests. Each test is a pair in which the first element is an input to the program, and the second element is the corresponding output.

The algorithm uses a set of configurations. Each configuration is a pair of a path constraint and an input. A *path constraint* is a conjunction of conditions on the program's input parameters. The configuration set is initialized with the configurations derived from the seed input (which can be the empty input) (line 2). The program is executed concretely on the input (line 5). Next, the algorithm uses a subroutine, *getConfigs*, to find new configurations. First, the program is executed symbolically on the same input (line 11). The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, which is satisfied by the path that was just executed from entry to exit of the whole program. The subroutine then creates new inputs by solving modified versions of the path constraint (lines 12–16), as follows: For each prefix of the path constraint, the algorithm negates the last conjunct (line 13). A solution to such an alternative path constraint, if it exists, corresponds to an input that will execute the program along a prefix of the original execution path, and then take the opposite branch, presumably covering new code. In other words, in this basic approach test generation is directed towards maximizing branch coverage. The algorithm uses a constraint solver (the *solve* auxiliary function) to find an input satisfying the given path constraint, or returns $\bot$ if no satisfying input exists (line 14).

As can be seen at line 4, the test-generation algorithm is parameterized by a selection methodology, which selects the next configuration to explore during the test generation. We use the selection methodology to direct the test generation for our needs. For instance, if the selection methodology is based on a similarity criterion, the test generation will be directed towards generating similar tests.

For the program in Figure 1, we illustrate test generation starting from an input that exhibits the the first bug, `x=0`, `y=0`, `scale_x=1`, `scale_y=3`. This will execute all statements, and reveal the path constraint `isset(scale_x)` $\wedge$ `isset(scale_y)`. One possible next step is to negate the second path constraint, resulting in a new path constraint `isset(scale_x)` $\wedge \neg$ `isset(scale_y)`. An input that would satisfy these constraints is `x=4`, `y=0`, `scale_x=3`, since this leaves `scale_y` undefined as required. There are other possible choices, clearly, but this is one way to generate a new different input. These inputs will appear later as F and B3 in the fault-localization example in Figure 3.

This mechanism is essentially a search over different inputs that result in different executions embodied in different tests. Thus, each step involves a choice of what execution to try next. Based on these choices, the set of generated tests will be different, and these differences may affect the results of fault localization.

3.2 Fault Localization

We focus on fault-localization techniques that compare the statements executed by passing and failing tests to estimate what statements are likely responsible for faults. A wide range of techniques has been proposed [18, 19, 20, 2, 26], but we focus on the *Ochiai* metric, which defines the *suspiciousness* of a statement j, s_j, as follows:

$$s_j = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}}$$

using the terminology of [2]. Here a_{11} is the number of failing tests that executed statement j, a_{01} is the number of failing tests that did not execute statement j, a_{10} is the number of passing tests that executed statement j. The idea is that statements are to be inspected in order of decreasing suspiciousness.

Based on our experience in prior work [4], we have augmented the notion of statement used in much prior fault-localization work. Previous work has focused on some representation of source location (e.g., line number); we generalize this to a tuple consisting of a line number and possibly one of two other components:

- An abstraction of the return value of the statement, if it is a function call
- The conditional value for `if` and `switch` statements

We use these tuples in exactly the same way as statements are used in prior work; we apply the *Ochiai* formula to the set of tuples from each execution, rather than to the set of statements.

3.3 Localization Example

To illustrate the impact of the test suite on fault localization, consider the two test suites in Figure 3, each with four tests. These two test suites were generated by starting from a failing test F:

```
x=0,y=0,scale_x=1,scale_y=3
```

which exposes the first bug. Since the executions are determined by the inputs, we start by showing the sets of inputs for each test suite.

Because the first test suite defines the two scale parameters for all four tests, all tests will execute all of the code; tests F and A1 will expose the divide-by-zero failure. And because we record return values as well, differences are detectable in the executions of the tests. In particular, the `magnitude()` method returns a value, so

[1]The full algorithm can be found in [6], Figure 6.

test	args
test suite 1	
F	`x=0,y=0,scale_x=1,scale_y=3`
A1	`x=0,y=0,scale_x=3,scale_y=2`
A2	`x=5,y=0,scale_x=1,scale_y=0`
A3	`x=3,y=2,scale_x=2,scale_y=4`
test suite 2	
F	`x=0,y=0,scale_x=1,scale_y=3`
B1	`x=3`
B2	`x=5,y=4`
B3	`x=4,y=0,scale_x=3`

Figure 3: Example test suites

tests generate different tuples for that method when it is called on points with different magnitudes. Note also that 22 is the line of the `if` statement, so its statements are augmented with the conditional outcome. This is illustrated in Figure 4.

test	*statements executed*
F	18, 19, 20, <22, true>, 23, 24, 25, 26, 29, 12, <8,3.1>, <8, 0>, 13, 14, 31, <8, 0>
A1	18, 19, 20, <22, true>, 23, 24, 25, 26, 29, 12, <8,3.6>, <8, 0>, 13, 14, 31, <8, 0>
A2	18, 19, 20, <22, true>, 23, 24, 25, 26, 29, 12, <8,1>, <8, 8.6>, 13, 14, 31, <8, 1>
A3	18, 19, 20, <22, true>, 23, 24, 25, 26, 29, 12, <8,3.6>, <8, 3.6>, 13, 14, 31, <8, 3.6>

Figure 4: Example test suite 1 executions

Consider the calls to `p.magnitude()` in Figure 4; recall that we are recording both the statement itself and its return value. There are two of the four tests that execute this statement and get a result of 0. Looking at the *Ochiai* formula, we see that it is of suspiciousness 1, since $a_{11} = 1$ and $a_{10} = a_{01} = 0$. Indeed, one way to fix this issue is to handle the case when this call returns 0. This result and the lower suspiciousness for all other statements is shown in Figure 5.

On the other hand, for the second test suite, only the given test F exhibits the bug. And also only F executes the call to `scale()`, so we see many more statements that are correlated with the bug. This is shown in Figure 6 and Figure 7, where statements 23, 24, 25, 26, 12, 13, 14 and 31 all correspond exactly to the failing test.

Thus, we observe that, especially for small test suites, the choice of tests can make a big difference. Our work focuses on selection strategies that allow us to focus fault localization quickly.

4. SIMILARITY METRICS

This paper evaluates various strategies for automatic test generation for the purpose of fault localization. Given a failing execution, the general intuition behind our techniques is that localizing the corresponding fault is more effective 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.

For this to be more precise, we need to formalize the concept of "similarity" between two executions. This leads us to introducing a similarity criterion, which is a function that takes as an input two

statement	*executions*	*suspiciousness*
12	F, A1, A2, A3	.71
13	F, A1, A2, A3	.71
14	F, A1, A2, A3	.71
18	F, A1, A2, A3	.71
19	F, A1, A2, A3	.71
20	F, A1, A2, A3	.71
23	F, A1, A2, A3	.71
24	F, A1, A2, A3	.71
25	F, A1, A2, A3	.71
26	F, A1, A2, A3	.71
29	F, A1, A2, A3	.71
31	F, A1, A2, A3	.71
<22,true>	F, A1, A2, A3	.71
<8,0>	F, A1	1
<8,1>	A3	0
<8,3.1>	F	.71
<8,3.6>	F, A2	.5
<8,8.6>	A2	0

Figure 5: Example test suite 1 suspiciousness

test	*statements executed*
F	18, 19, 20, <22, true>, 23, 24, 25, 26, 29, 12, <8,3.1>, <8, 0>, 13, 14, 31, <8, 0>
B1	18, 19, 20, <22, false>, 29, 12
B2	18, 19, 20, <22, false>, 29, 12
B3	18, 19, 20, <22, false>, 29, 12

Figure 6: Example test suite 2 executions

executions, and produces as output a percentage index that indicates how similar the two executions are. More formally, if E is the set of all the executions of a program, a *similarity criterion* is a function $\sigma_\alpha : E \times E \rightarrow [0, 100]$, where α is itself a function that abstracts executions. Specifically, α maps each execution $e \in E$ to a set of characteristics of e that depend on the particular similarity metric under consideration. There can be multiple similarity criteria, each based on what characteristics are considered when measuring similarity and, consequently, what abstraction function α is being considered.

A similarity criterion σ_α can be extended to a function $\sigma'_\alpha : E \times 2^E \rightarrow [0, 100]$, defined as follows:

$$\sigma'_\alpha(e, S) := \max_{e' \in F}\{\sigma_\alpha e'\}, \forall e \in E, \forall F \subseteq E$$

which can be used to compare a passing execution with a set of failing executions.

In order to guide our test generation technique (Section 3.1) towards generating similar executions, a similarity function is used as the selection methodology (Figure 2, line 4). The selection methodology is responsible for selecting the next input to explore, thus directing the generation to explore similar executions.

In this paper, we consider two different similarity metrics: path constraints and inputs. These two approaches and the relevant similarity criteria are described in the next subsections.

4.1 Path-Constraint Similarity

In general, any execution is generated by a set of inputs to the program. This defines a function $f : 2^I \rightarrow E$, where I is the set of inputs to the program. Function f maps any set of program inputs to one program execution. Furthermore, given a particular set

statement	*executions*	*suspiciousness*
12	F, B1, B2, B3	.5
13	F	1
14	F	1
18	F, B1, B2, B3	.5
19	F, B1, B2, B3	.5
20	F, B1, B2, B3	.5
23	F	1
24	F	1
25	F	1
26	F	1
29	F, B1, B2, B3	.5
31	F	1
<22,true>	F	1
<22,false>	B1, B2, B3	0
<8,0>	F	1
<8,3.1>	F	1

Figure 7: Example test suite 2 suspiciousness

of program inputs $\iota \in I$, a heuristic function can compute an input from a given path constraint. This defines a function $g : 2^P \rightarrow 2^I$, where P is the set of path constraints that can arise during the execution of the program. The composition function $f \circ g : 2^P \rightarrow E$ can be used to base a similarity criterion on path constraints instead of actual executions, which is a very useful property in test generation. Specifically, given a set π of path constraints and the corresponding execution $e = f(g(\pi))$, we define $\alpha(e) = \pi$, and we use the resulting function $\alpha : E \rightarrow 2^P$ to parameterize the similarity criterion σ.

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 cardinality of the largest 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, $\sigma_\alpha(e_1, e_2) = 83.3\%$ if the similarity criterion is based on subset comparison, and $\sigma_\alpha(e_1, e_2) = 50\%$ 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.

4.2 Input Similarity

With this approach, we compare the inputs to different executions. Each execution $e \in E$ is reduced to only its inputs, as follows. Given a set π of path constraints, we consider the corresponding set of execution inputs $g(\pi)$, and we define $\alpha(e) = g(\pi)$. We then use the resulting function $\alpha : E \rightarrow 2^I$ to parameterize the similarity criterion σ.

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, $\sigma_\alpha(e_1, e_2) = 83.3\%$.

Example.

The example in Figure 1 motivates the use of different similarity criteria for fault-localization-oriented test generation. Since the sample program only exhibits three different paths, a test-generation technique based on input similarity is more effective for fault localization.

Let us assume that the program in Figure 1 fails with inputs given in F, where F is defined as in Figure 3. By looking at test suites A and B in Figure 3, we observe that test suite A, generated with the input-similarity technique, allows for quick fault localization. This is due to the fact that the faulty statement in the program is executed by multiple failing tests in suite A, namely F and A1. Even more importantly, suite A has the advantage of presenting a passing test, A2, that is similar to the failing execution F.

If all the tests in the A and B suites are available, A1 will be the first test to be selected with the input-similarity strategy since it is the most similar to the faulty execution F that is given as input to the algorithm. That will be followed by test A2, which is the second test to be the most similar to the faulty execution. A2 will be followed by A3. Notice that both A2 and A3 are passing tests. This makes it possible to localize and isolate the faulty statement with a test-suite size of only 4. In contrast, the coverage strategy would get full coverage with F, and then would select random tests, with a potential test-suite size of 7, before making the fault localizable.

5. IMPLEMENTATION AND EVALUATION

We implemented several test-generation strategies in *Apollo* [5, 4], a tool for automatically finding and localizing faults in PHP web applications. This section reports on the implementation, and on experiments that measure the effectiveness of the different test-generation strategies.

5.1 Implementation

For the purpose of test generation, we use *Apollo* [5, 6], which employs a *shadow interpreter* based on the Zend PHP Interpreter V5.2.2[2]. *Apollo* 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`[3] constraint solver to solve path constraints during the concolic generation. The process of concolic execution is orchestrated by a standard Apache[4] Web server that uses the instrumented PHP interpreter.

Our fault-localization technique performs conditional [4] and return-value modeling. These two enhancements where implemented on top of *Apollo*'s shadow interpreter. For the conditional modeling, *Apollo* records all comparisons in the executed PHP script. For each comparison, the shadow interpreter stores the statement's line number and the relevant boolean result. For a `switch` statement, the shadow interpreter stores the line number of the switch and the set of results for all executed `case` blocks. For return-value modeling, 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 non-zero `int` and `double` values, `true` and `false` boolean values, constant and non-constant values, as well as empty and non-empty arrays, strings, and resources.

[2]`http://www.php.net/`
[3]`http://choco-solver.net`
[4]`http://www.apache.org/`

program	version	#files	PHP LOC	#downloads
faqforge	1.3.2	19	734	14,164
webchess	0.9.0	24	2,226	32,352
schoolmate	1.5.4	63	4,263	4,466
phpsysinfo	2.5.3	73	7,745	492,217

Table 1: Characteristics of subject programs. The #files column lists the number of `.php` and `.inc` files in the program. The PHP LOC column lists the number of lines that contain executable PHP code. The #downloads column lists the number of downloads from `http://sourceforge.net`.

subject	# faults
webchess	10
faqforge	7
schoolmate	16
phpsysinfo	2
total	35

Figure 8: Number of faults used in the localization experiments. Each fault manifested itself as an execution failure.

5.2 Research Questions

For each of the test-generation strategies under consideration, we are interested in determining the maximal fault-localization effectiveness that can be achieved using test suites generated according to that strategy. As in previous work on fault localization, we will concentrate on the percentage of well-localized faults for which less than 1% of all executed statements need to be examined until the fault is found, assuming that statements are examined in order of decreasing suspiciousness. It is reasonable to expect that a limited amount of time will be available for test generation. Therefore, we are also interested in determining how quickly each of the test-generation strategies under consideration converges towards its maximal effectiveness. This leads us to formulate the following research questions:

RQ1. 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?

RQ2. How many tests need to be generated by each test-generation strategy in order to reach its maximal fault-localization effectiveness?

5.3 Subject Programs

For the evaluation, we selected four open-source PHP programs from `http://sourceforge.net`:

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

Figure 1 presents some characteristics of these programs.

5.4 Methodology

In order to answer our research questions, we needed localized faults. We used actual faults that were discovered by *Apollo* [5, 6]. In our previous work, *Apollo* was used to discover two types of failures: HTML failures that occur when malformed HTML is generated, and execution failures when an input causes a crash or obtrusive error message. In this paper, we restrict our attention to execution failures for which the location of the fault is not immediately obvious from an error message. The reason for restricting our attention to these cases is that, in our opinion, these are the situations where the use of an automated fault localization is most warranted.

We manually localized all faults. For each fault, we devised a patch and ensured that applying this patch fixed the problem, by running the tests again, and making sure that the associated failures had been corrected. Figure 8 summarizes the number of faults for each subject program.

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

Base Test generation using the concolic execution algorithm of [6], 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 concolic execution algorithm of [6], but starting test generation from the failing test.

PCS (Path Constraint Similarity) Test generation using the subset-based path-constraint similarity metric that was described in Section 4.1

IS (Input Similarity) Test generation using the input similarity that was described in Section 4.2.

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 according to the *Ochiai* technique [2] with the improvements described in Section 3.2. Similar to previous fault-localization studies [18, 11, 17, 26], we measured the effectiveness of a fault localization algorithm as the minimal number of statements that need to be inspected until the first faulty line is detected, assuming that statements are examined in order of decreasing suspiciousness. We then computed the number of statements to be inspected as a percentage of the number of executed statements. Finally, we computed the percentage of faults that are "well-localized", meaning that they require the inspection of less than 1% of all executed statements.

5.5 RQ1

We first discuss the "maximal" fault-localization effectiveness of the test suites generated by the four test-generation techniques above, as measured by the percentage of well-localized faults, assuming each technique is given a infinite amount of time to construct a test suite. In practice, we found that it sufficed to have each technique 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[5]. Table 2 shows three columns for each subject program and each technique. These columns show, from left to right: (i) on average, for each subject program, the percentage of faults that is well-localized, (ii) on average, the absolute number of statements that needs to be inspected to localize each fault, and (iii) on average, the percentage of executed

[5]It is theoretically possible that some minor further gains could be achieved by generating many additional tests, but we consider this to be very unlikely.

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 2: 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 (% stmts) of executed statements that need to be inspected on average until the fault is localized.

statements that needs to be inspected to localize each fault. For example, for **faqforge**, both the *Base* and *PCS* techniques eventually localize 100% of the faults to within 1% 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% of all executed statements. The *Coverage* and *IS* generation techniques also reach 100% 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% 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% of all faults was eventually well-localized by each technique. However, for **webchess**, only 77% of all faults was eventually well-localized by each technique.

5.6 RQ2

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 3 shows two columns for each subject program and each test generation technique. These columns show, from left to right: (i) the number of tests that is needed to reach the maximal percentage of well-localized faults as reported in Table 2, and (ii) the time required to generate these tests. Here, it should be noted that the time reported in (ii) 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 3, there are 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 [4] requires 60 tests to reach the maximal percentage of well-localized faults, whereas the *PCS* technique requires only 5 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 Figure 9 provide some more detail on how quickly the test generation strategies converge towards 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 (7 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 that we used in our previous work [4]. This can be viewed an 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%.

5.7 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. To minimize this issue, we selected open-source PHP applications that are widely used, as is evidenced by the significant number of downloads reported in Figure 1. Second, it could be the case that the faults we exposed and localized are not representative. However, these bugs represent a range of execution errors, 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 simple one-line code changes, for which we attempted to come up with the simplest possible solution. And we verified that our fixes indeed resolved the issues, so to that extent our fixes are correct. 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. However, we ourselves made use of the fault localization results while debugging. Also, this approach to measuring the effectiveness of fault localization methods has been used in previous research in the area (e.g., [18, 17, 26]), which makes our results more comparable with related work

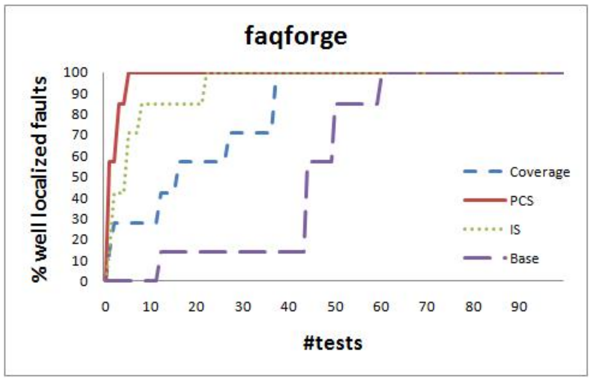
6. RELATED WORK

Fault localization algorithms have been studied extensively in recent years. In this section, we concentrate on fault localization techniques that predict the location of faults based on the analysis of data from multiple tests or executions. A discussion of other fault localization techniques can be found in [4].

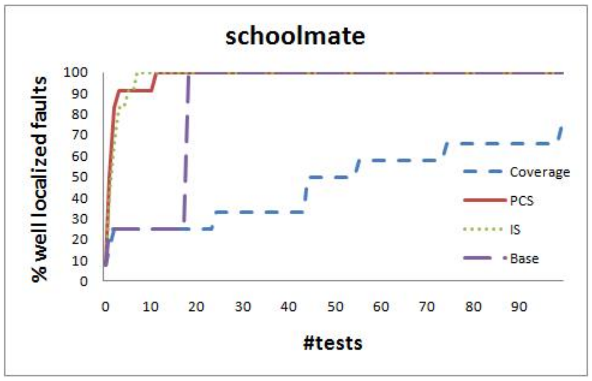
Early work. Early work on fault localization relied on the use of program slicing [29]. Lyle and Weiser [22] introduce *program dicing*, a method for combining the information of different program slices. The basic idea is that, when a program computes a correct value for variable x and an incorrect value for variable y, the fault is likely to be found in statements that are in the slice w.r.t. y, but not in the slice w.r.t. x. Variations on this idea technique were later explored by Pan and Spafford [23], and by Agrawal, *et al.* [3]. In the spirit of this early work, Renieris and Reiss [25] use *set-union* and *set-intersection* methods for fault localization. The

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

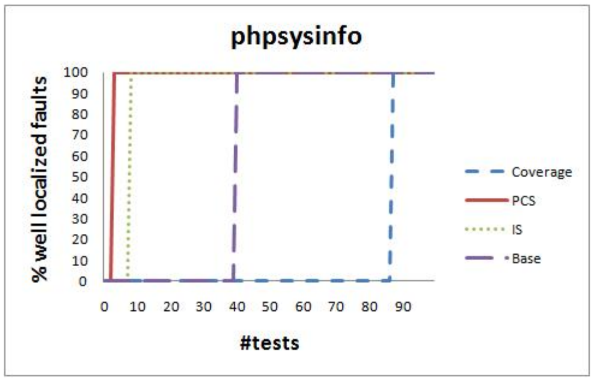
Table 3: Summary, for each test generation technique and subject program, of the time (time(s) for *Base* and avg time(s) for *Coverage*, *PCS* and *IS*) and number of tests (# tests) required to achieve the maximal percentage of well-localized faults, as reported in Table 2.



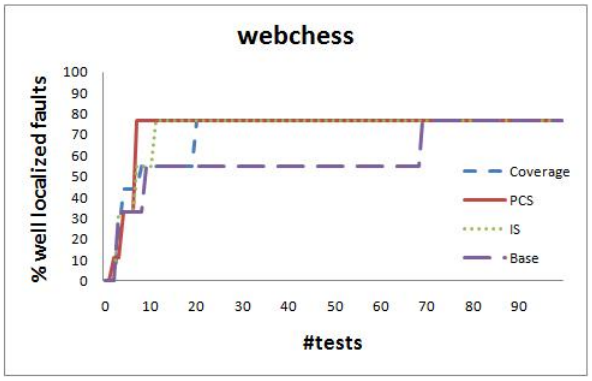
(a)



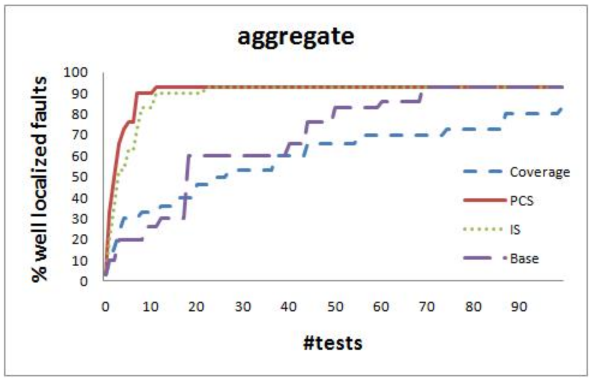
(b)



(c)



(d)



(e)

Figure 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 (e) aggregated

set-union technique computes the union of all statements executed by passing test cases and subtracts these from the set of statements executed by a failing test case. The resulting set contains the suspicious statements that the programmer should explore first. The set-intersection technique identifies statements that are executed by all passing test cases, but not by the failing test case, and attempts to address errors of omission, where the failing test case neglects to execute a statement.

Nearest Neighbors. The *nearest neighbors* fault localization technique by Renieris and Reiss [25] assumes the existence of a failing test case and many passing test cases. The technique selects the passing test case whose execution spectrum most closely resembles that of the failing test case according to one of two distance criteria, 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 (SDG) [14]. *Nearest Neighbor* was evaluated on the Siemens suite [15], and was found to be superior to the *set-union* and *set-intersection* techniques.

Tarantula and Ochiai. *Tarantula* [18] is a fault-localization technique that associates with each statement a *suspiciousness rating* that indicates the likelihood for that statement to contribute to a failure. Jones and Harrold [17] conducted a detailed empirical evaluation in which they apply *Tarantula* to faulty versions of the Siemens suite [15], 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 [11, 2, 25, 17]. Santelices, *et al.* [26] investigate the tradeoffs of applying the *Tarantula* algorithm to different types of program entities: statements, branches, and def-use pairs. The outcome of this study is that the branch-based algorithm is more precise than the statement-based one, and that the def-use-based variant is more precise still.

Recent papers by Jones and Harrold [17] and by Abreu, *et al.* [2] present empirical evaluations of several 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. [2] is more effective than *Tarantula* [26, 1]. Our own experiments with *Apollo* confirm this, which is why the work presented in this paper uses the *Ochiai* technique.

Other Statistical Techniques. Other fault localization techniques analyze statistical correlations between control flow predicates [20, 21] or path profiles [10] and failures, time spectra [31], and correlations between changes made by programmers and test failures [28, 24].

Fault Localization using Generated Test Suites Our work differs from the previous approaches for fault localization described above in that we rely on dynamic test generation techniques to create test suites. In a previous paper [4], we relied on concolic execution [12, 27, 9, 13, 30] to generate test suites. In that work, the generation of tests was guided by the objective of maximizing code coverage, by directing the solver to solutions that cover additional branches of condition statements. We found that, given a time budget of 20 minutes, 87.7% of the faults under consideration are localized to within 1% of all executed statements. The present paper uses the same technique for test generation, but the generation of tests is now guided by various similarity metrics in order to produce tests that are similar to a given failing execution. This is the same spirit as the Nearest Neighbors algorithm [25], but instead of *selecting* tests that are similar to a given failing test, we are attempting to *generate* such tests.

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. [7] 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. [32] 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. [16] 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.

Other Directed Concolic Testing. In [8], the authors investigate 3 strategies to direct concolic testing 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, more than a factor of 2 for their largest program. The mechanism of using some explicit metric to direct concolic 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 localization by generating new similar inputs.

7. CONCLUSIONS

Statistical fault-localization techniques that analyze execution data from multiple tests [18, 19, 20, 2, 26] have been shown to be quite effective when a sufficiently large test suite is available. In this paper, we address the scenario where such a test suite is not available, and the user relies on test-generation techniques to create one. In particular, we consider several test-generation techniques based on combined concrete and symbolic (concolic) execution and compare the fault-localization effectiveness of the generated test suites. The techniques under consideration include a standard, undirected concolic execution algorithm, used in our previous work [4], which attempts to maximize branch coverage. Furthermore, we consider several variants of concolic execution that aim to maximize similarity between the execution characteristics of the generated tests and those of a given fault-exposing execution, according to a number of similarity criteria.

We implemented the techniques in *Apollo*, an automated tool that detects and localizes faults in PHP Web applications. We evaluated the test generation techniques by localizing 35 faults in four PHP applications. The results that we presented in this paper show that a new, directed test-generation technique based on path-constraint similarity 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 concolic execution algorithm of [4], which aims to maximize code coverage, our directed technique reduces test-suite size by 86.1% and test-suite generation time by 88.6%.

As part of future work, we plan to explore additional similarity metrics, and understand how they compare to the path-constraint and input similarity metrics presented in this paper.

8. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of Testing: Academia and Industry Conference - Practice and Research Techniques (TAIC PART'07)*, pages 89–98, September 2007.

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

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

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, editors. *Practical Fault Localization for Dynamic Web Applications*, 2010. To appear.

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

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

[7] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 82–91. ACM, 2006.

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446. IEEE, 2008.

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

[10] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 34–44, Vancouver, Canada, May 2009.

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

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

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

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

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

[16] B. Jiang, Z. Zhang, T. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, July 2009.

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

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

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

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

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

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

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

[24] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In D. S. Rosenblum and S. G. Elbaum, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 239–249. ACM, 2007.

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

[26] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 56–66, Vancouver, Canada, May 2009.

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

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

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

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

[31] C. Yilmaz, A. M. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 81–90. ACM, 2008.

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