On the Effectiveness of the Tarantula Fault Localization Technique for Different Fault Classes

Aritra Bandyopadhyay, Sudipto Ghosh
Department of Computer Science
Colorado State University
Fort Collins, CO, USA
Email: $\{$*baritra, ghosh*$\}$*@cs.colostate.edu*

***Abstract*—Unlike test generation techniques, spectrum-based fault localization techniques have not been rigorously evaluated for their effectiveness in localizing different classes of faults. In this paper, we evaluate the effectiveness of the Tarantula fault localization technique. We state that the following three properties of a fault affect the effectiveness of localizing it: (1) accessibility, (2) original state failure condition, and (3) impact. Accessibility refers to how easy or hard it is to execute a faulty statement. It is measured by the size of the backward slice of the faulty statement. The original state failure condition is the condition that must be satisfied to create a local failure state upon executing the faulty statement. Impact refers to the fraction of the program that is affected by the execution of the faulty statement, measured by the size of the forward slice of the faulty statement.**

The results of our evaluation with the Siemens benchmark suite show that (1) original state failure condition based fault classes have no relationship with the effectiveness of localization, and (2) faults that are hard to access and have low impact are most effectively localized. These observations are consistent across random and branch coverage based test suites.

***Keywords*-fault localization; fault classes; Tarantula;**

I. Introduction

Spectrum-based fault localization techniques (e.g., [1], [2], [3]) localize some faults more effectively than others. Research literature that report evaluations of these techniques make isolated observations about the differences in effectiveness. For example, Jones et al. [1] observed that faulty initialization statements are assigned low suspiciousness scores because they are executed in both the successful and the failing tests. Gupta et al. [4] showed that certain classes of faults may or may not be detected depending on whether data slices, full slices, or relevant slices are used for localizing the fault. Failure inducing chops, proposed by Gupta et al. [5], fail to contain faulty statements that do not depend on a program's input variables.

Abreu et al. [6] studied how effectiveness is affected by coincidental correctness, which is related to class of the fault. A test case that executes a fault but does not produce a failure is called coincidentally correct. Abreu et al. [6] showed that the presence of coincidental correctness decreases the effectiveness of spectrum-based fault localization. Faults in some classes produce failure states more easily upon execution than faults in some other classes. Thus, the fault class affects the likelihood of coincidental correctness, which, in turn, affects the effectiveness of spectrum-based fault localization.

Notwithstanding the isolated observations on the effectiveness of spectrum-based fault localization approaches and the studies on coincidental correctness, there is a lack of systematic evaluations that investigate the effect of fault classes on the effectiveness of fault localization. In this paper, we report on an empirical evaluation of the Tarantula [1] fault localization technique using fault class as an independent variable.

The question that arises is what fault classes are appropriate in the context of spectrum-based fault localization. We only consider deterministic faults. We state that the following three properties of faulty statements affect the effectiveness of Tarantula: (1) accessibility, (2) original state failure condition, and (3) impact. Accessibility of a fault addresses how hard it is for a test to execute the fault. Original state failure condition is the condition that must be satisfied in order to raise a local failure state upon the execution of the faulty statement. Impact is concerned about the fraction of the program that is affected by the execution of the faulty statement. For the original state failure condition, we use the classification of faults by Richardson et al. [7]. We present our own fault classifications based on accessibility and impact. We also illustrate how these properties affect Tarantula's effectiveness.

Our evaluation demonstrates the following: (1) Among the fault classes based on original state failure condition, such as operator faults, variable definition and variable reference faults, there is no clear variation of effectiveness, and (2) Faults that are hard to reach and have low impact are the ones that are the most effectively localized.

The rest of the paper is organized as follows. Section II summarizes the Tarantula technique. Section III presents an analysis of the three properties of faults that affect Tarantula's effectiveness. Section IV describes techniques to classify faults based on those properties. Section V presents the design of our study and an analysis of the results.

Section VI discusses the threats to validity of our study. Section VII summarizes related work. Section VIII presents our conclusions and plans for future work.

II. Tarantula Background

Tarantula [1] uses statement coverage information from multiple failing and successful runs to assign a suspiciousness score to every program statement. Tarantula is based on the heuristic that the statements executed primarily by the successful test cases are the least suspicious, and the ones executed primarily by the failing test cases are the most suspicious. Jones et al. [1] expressed the heuristic in a formula that assigns colors to statements from the color range red to green. An equivalent formula that assigns suspiciousness scores from the range [0..1] is given below:

$$susp(s) = \frac{\%failed(s)}{\%passed(s) + \%failed(s)} \qquad (1)$$

Here $susp(s)$, $\%passed(s)$, and $\%failed(s)$ respectively denote the suspiciousness score assigned to the statement s, the percentage of passing test cases that execute s, and the percentage of failing test cases that execute s.

Two statements may share the same suspiciousness score because the value of the ratio in Equation 1 may be the same for them even though they have different values for $\%passed(s)$ and $\%failed(s)$. Under such situations, Tarantula uses brightness (denoted below by $bright$) to resolve the tie by comparing the absolute values of $\%passed(s)$ or $\%failed(s)$, whichever is larger.

$$bright(s) = max(\%passed(s), \%failed(s)) \qquad (2)$$

There may be statements with the same suspiciousness score and brightness value because they are executed by the same number of passing and failing test cases, but the individual test cases executing the statements are different. Tarantula does not specify how to resolve such ties. If a tester inspects the statements with the same color and brightness value in any random order, half of them would need to be inspected on an average. We assume that average case.

To evaluate Tarantula's effectiveness, we measure the number of statements, $numExamined(s)$, a tester needs to examine on average. We divide it by the number of executable statements, n, in the program. We calculate the percentage of code examined, i.e., $\frac{numExamined(s)}{n} \times 100$ as the measure of effectiveness. A smaller value of the measure denotes higher effectiveness.

III. Analysis of Fault Properties

We state that the following three properties of faults affect the effectiveness of Tarantula: (1) accessibility, (2) original state failure condition, and (3) impact. Below we define these properties and illustrate how their variation can cause a corresponding variation in the effectiveness of Tarantula.

A. Accessibility

Accessibility measures how hard it is for a test execution to access a faulty statement. Harder to access faulty statements will be executed by fewer tests. All failings tests must execute the faulty statement in order to cause failures. However, a harder to access faulty statement will be executed by fewer passing tests, thereby reducing its $\%passed(s)$ value and increasing its suspiciousness. We use the following example to illustrate the effect of accessibility.

Table I
Examples of Faults with Differing Accessibility

Original program	Faulty $Version_1$	Faulty $Version_2$
`int a,b,c;`	`int a,b,c;`	`int a,b,c;`
`int r=0;`	`int r=0;`	`int r=0;`
`read(a,b,c);`	`read(a,b,c);`	`read(a,b,c);`
`if (a >= b){`	`if (a >= b){`	`if (a >= b){`
`  r = a+b;`	**`  r = a-b;`**	`  r = a+b;`
`  if (b > c){`	`  if (b > c){`	`  if (b > c){`
`    r = r+10;`	`    r = r+10;`	**`    r=r-10;`**
`  }`	`  }`	`  }`
`}`	`}`	`}`
`print(r);`	`print(r);`	`print(r);`

Table I shows a program and its two faulty versions. The fault in each version is shown in bold. The fault in $Version_1$ is easier to execute than the one in $Version_2$ because the former is guarded by two conditional statements, while the latter is guarded by only one.

To illustrate the application of Tarantula on the two faulty versions, we consider 6 tuples that provide input values for (a, b, c): t_1 = (1, 2, 3), t_2 = (2, 1, 3), t_3 = (2, 0, 3), t_4 = (4, 3, 0), t_5 = (4, 3, 1), and t_6 = (4, 0, 1) Table II shows the suspiciousness score for each statement in $Version_1$ along with information about which test case executed the statement (denoted by a $\surd$ mark), and the result of each test case (P = Pass, F = Fail). The statements are arranged in decreasing order of the suspiciousness score. Table III shows the same information for $Version_2$.

Table II
Suspiciousness Scores of Statements in $Version_1$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	$susp$
r=r+10				$\surd$	$\surd$		1
r=a-b		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.6
if(b>c)		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.6
int a,b,c	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.5
int r=0	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.5
read(a,b,c)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.5
if(a>=b)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.5
print(r)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	0.5
Result	P	F	P	F	F	P	

Table III
SUSPICIOUSNESS SCORES OF STATEMENTS IN $Version_2$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	$susp$
r=r-10				√	√		1
r=a+b		√	√	√	√	√	0.57
if(b>c)		√	√	√	√	√	0.57
int a,b,c	√	√	√	√	√	√	0.5
int r=0	√	√	√	√	√	√	0.5
read(a,b,c)	√	√	√	√	√	√	0.5
if(a>=b)	√	√	√	√	√	√	0.5
print(r)	√	√	√	√	√	√	0.5
Result	P	P	P	F	F	P	

The suspiciousness score assigned to the faulty statement in $Version_1$ is 0.6, while the score of the faulty statement in $Version_2$ is 1. Locating the fault in $Version_1$ requires us to examine 2 statements, while locating the fault in $Version_2$ requires us to examine only one statement. Because the fault in $Version_1$ is more easily accessible than that in $Version_2$, the former is executed by more passing tests, which causes the $Version_1$ fault to have a lower suspiciousness score than the $Version_2$ fault. This results in a lower rank for the $Version_1$ fault because the relative suspiciousness of the remaining statements of the program is same for both the versions.

In general, for any two faulty versions v_1 and v_2, if the fault in v_1 is easily accessible than the one in v_2, the probability that an arbitrary test input executes the fault in v_1 is greater than that for v_2. Thus, for an arbitrary test suite, the fault in v_1 is more likely to be executed by more passing tests than the fault in v_2, to result in a lower suspiciousness score of the fault in v_1 than that of the fault in $v2$. If the relative suspiciousness scores of the non-faulty statements remain the same in both versions, the fault in v_1 will rank lower than the fault in v_2.

B. Original State Failure Condition

Richardson et al. [7] defined the original state failure condition as the condition to be satisfied for an execution of a faulty statement to result in a local failure. To cause a failure at the output, a test input must satisfy the original state failure condition and propagate the local failure to the output. Richardson et al. proposed the RELAY model of fault detection, which provides a framework for analyzing how a fault causes a failure.

In the following example, we illustrate how the effectiveness of Tarantula can vary with the original state failure condition. Table IV shows the original program from Table I, its faulty version $Version_1$ and a new faulty version $Version_3$. To illustrate the application of Tarantula on these two versions we again consider the same test cases described in section III-A. Table V shows which test case executes which statement, the suspiciousness scores, and the results of the test cases in $Version_3$. The statements are arranged in decreasing order of the suspiciousness score.

Table IV
EXAMPLES OF FAULTS WITH DIFFERENT ORIGINAL STATE FAILURE CONDITIONS

Original Program	Faulty $Version_1$	Faulty $Version_3$
`int a,b,c;`	`int a,b,c;`	`int a,b,c;`
`int r=0;`	`int r=0;`	`int r=0;`
`read(a,b,c);`	`read(a,b,c);`	`read(a,b,c);`
`if (a >= b){`	`if (a >= b){`	`if (a >= b){`
`  r = a+b;`	**`  r = a-b;`**	**`  r = 100;`**
`  if (b > c){`	`  if (b > c){`	`  if (b > c){`
`    r = r+10;`	`    r = r+10;`	`    r = r+10;`
`  }`	`  }`	`  }`
`}`	`}`	`}`
`print(r);`	`print(r);`	`print(r);`

Table V
SUSPICIOUSNESS SCORES OF STATEMENTS IN $Version_3$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	$susp$
r=100		√	√	√	√	√	1
if(b>c)		√	√	√	√	√	1
r=r+10				√	√		1
int a,b,c	√	√	√	√	√	√	0.5
int r=0	√	√	√	√	√	√	0.5
read(a,b,c)	√	√	√	√	√	√	0.5
if(a>=b)	√	√	√	√	√	√	0.5
print(r)	√	√	√	√	√	√	0.5
Result	P	F	F	F	F	F	

The fault in $Version_3$ is assigned a suspiciousness score of 1, while the fault in $Version_1$ is assigned a suspiciousness score of 0.6. In $Version_3$, we need to examine one statement before finding the fault. The statement `r = r+10` has the same suspiciousness as the fault but has a lower $bright$ value. In $Version_2$, one needs to examine two statements before finding the fault.

The accessibility of the faults in the two versions are the same but their original state failure conditions are different. The original state failure condition of the fault in $Version_1$ is $a + b \neq a - b$, while the original state failure condition of the fault in $Version_1$ is $a + b \neq 100$. The input values for a, b, and c in the test cases are below 10. With such input values, the original state failure condition of the fault in $Version_3$ is harder to satisfy than that of the fault in $Version_1$. Thus, in $Version_3$, whenever the fault is executed, the program fails. Thus, the faulty statement never executes in a passing run and the faulty statement obtains a suspiciousness score of 1.

On the contrary, in $Version_1$, the faulty statement is executed in two passing runs. This reduces the suspiciousness score of the faulty statement to 0.6. Because the relative suspiciousness of the remaining statements are the same, the fault in $Version_2$ ranks higher than that in $Version_1$.

C. Impact

In this paper, impact measures what fraction of the program statements is affected by the execution of the faulty statement. Below, we illustrate how impact affects the effectiveness of Tarantula. We consider an original program and its two faulty versions as shown in Table VI. The faulty statements are shown in bold.

Table VI
EXAMPLES OF FAULTS WITH DIFFERENT IMPACTS

Original Program	Faulty $Version_1$	Faulty $Version_2$
`1 int a,b,c;`	`1 int a,b,c;`	`1 int a,b,c;`
`2 int r;`	`2 int r;`	`2 int r;`
`3 read(a,b,c);`	`3 read(a,b,c);`	`3 read(a,b,c);`
`4 if (a >= b)`	`4` **`if (a <= b)`**	`4 if (a >= b)`
`5  r = a + b;`	`5  r = a + b;`	`5  r=a + b;`
`6 if (b >= c)`	`6 if (b >= c)`	`6` **`if (b <= c)`**
`7  r+=foo(b,c);`	`7  r+=foo(b,c);`	`7  r+=foo(b,c);`
`8 print(r);`	`8 print(r);`	`8 print(r);`

The faulty statements in $Version_1$ and $Version_2$ have the same accessibility because neither is nested within any conditional statement. They also have the same original state failure condition because both replace the $>=$ operator in the correct statements with the $<=$ operator. However, the faulty statements have different impacts. The value of the faulty condition in $Version_1$ determines whether line 5 will be executed or not. The value of the faulty condition at in $Version_2$ determines whether the function `foo` will be called or not. The function `foo` may have many statements in it. Thus, the evaluation of the faulty condition in $Version_2$ controls the execution of more statements than that for $Version_1$.

The faulty statements in both $Version_1$ and $Version_2$ will obtain a suspiciousness score of 0.5 with any arbitrary test suite. This is because any test case executes both the faulty statements. In $Version_1$, if failing test cases always evaluate the faulty condition to $true$, line 5 always gets executed in the failing runs. Thus, line 5 obtains a higher suspiciousness score than the faulty statement. In $Version_2$, if the failing test cases always evaluate the faulty condition to $true$, all the statements in `foo` get executed in the failing test cases, and obtain a higher suspiciousness value than the faulty statement. This lowers the rank of the faulty statement in $Version_2$. However, if the failing test cases execute the faulty condition in $Version_2$ to false, the statements in `foo` do not execute in the failing runs and are ranked lower than the faulty statement. This improves the rank of the faulty statement.

We state that if many statements are directly or indirectly control dependent on the fault, the fault may have a lower or higher rank depending on how it affects the execution of the control dependent statements. If the control dependent statements are executed during failing runs, then they obtain higher suspiciousness scores than the faulty statement. If failure involves non-execution of the control dependent statements, then the fault is likely to obtain a higher rank.

IV. FAULT CLASSIFICATION

In this section we describe the classifications of faults based on accessiblity, original state failure condition, and impact. We use the original state failure based classification of faults by Richardson et al. [7]. We present a new fault classification based on accessiblity and impact.

A. Accessiblity-based Classification

To measure accessibility, we state that if more conditional statements guard a faulty statement, then the faulty statement is less accessible. The backward static slice (calculated using only control dependencies) of the faulty statement contains all such guarding conditional statements. Therefore, we measure accessibility by the size of the backward slice of the faulty statement as a percentage of the program size. However, this measure does not address the strength of the conditions in the conditional statements. Based on the sizes of backward slices, we define an ordinal classification of the faults.

We apply the k-means clustering algorithm with $k = 2$, to divide the slice sizes into two clusters. K-means is the most commonly used clustering algorithm when the number of clusters, k, is known and the nature of the distribution of the underlying data is not known. K-means clustering divides a set of data points into k clusters such that the sum of squares of distances between the data points and the corresponding cluster centroids is minimal. Table VII shows the accessibility-based fault clusters with the high and low mean along with their corresponding mean values, ranges and number of faults.

Table VII
CLUSTERS OF FAULTS BASED ON ACCESSIBLITY

Cluster	Cluster mean (%Bslice)	Cluster range (%Bslice)	#faults
High	4.67%	3.56% - 8.55%	57
Low	2.28%	0.6% - 3.42%	73

The above classification is ordinal. This is because there is an ordering relation of accessiblity between the faults that belong to one cluster and the faults that belong to the other.

B. Classification Based on Original State Failure Condition

We use the fault classification proposed by Richardson et al. [7], which is based on the original state failure condition. The classification contains six fault classes: (1) constant reference faults, (2) variable reference faults, (3) variable definition faults, (4) conditional operator faults, (5) relational operator faults, and (6) arithmetic operator faults.

This classification is nominal because the original state failure condition of one fault class cannot be compared with that of another fault class. The original state failure conditions for different classes are derived in different ways.

On inspecting the faulty versions in the Siemens benchmark suite, we found the following additional fault classes: (7) missing/ added statement faults, (8) missing/added branch faults, (9) missing/added conditional clause faults. The above fault classes need to be considered because they have characteristic ways of deriving their original state failure conditions. This classification is nominal because the original state failure condition of one fault class cannot be compared with that of another fault class.

C. *Impact-based Classification*

We measure the impact of a faulty statement by the size of its forward static slice (as percentage of the program size) using both data and control dependencies. The forward static slice with respect to a variable at a statement is the set of statements that are affected by the value of the variable at that statement through data and control-dependencies. We calculate the forward static slice of a faulty statement by the union of the forward slices of all variables appearing in the faulty statement.

Based on the sizes of forward slices, we define an ordinal classification of the faults by dividing the slice sizes into two clusters using the k-mean clustering. Table VIII shows the impact based clusters

Table VIII
CLUSTERS OF FAULTS BASED ON IMPACT

Cluster	Cluster mean (%Fslice)	Cluster range (%Fslice)	#faults
High	87.3%	63.5% - 99.0%	47
Low	33.7%	2.14% - 54.68%	83

D. *Factorial Design*

The classifications based on accessiblity and impact are both ordinal. We use a factorial design by crossing the classes from the two classifications to obtain four treatments as shown in Table IX.

Table IX
FACTORIAL DESIGN

Treatment	**Acronym**
Hard to access and high impact	HH
Hard to access and low impact	HL
Easy to access and low impact	LL
Easy to access and high impact	LH

V. EVALUATION

The goal of our evaluation is to investigate the effectiveness of Tarantula in localizing faults of different classes. Our independent variable is fault class, described in Section IV. Our dependent variable is the effectiveness of fault localization. We described the measure effectiveness in Section II.

We consider the nature of the test suite to be a confounding variable in our study. Previous work (e.g., [8]) on selection of test suites has demonstrated that the choice of test suites affects the effectiveness. To reduce the effect of the confounding variable, we perform our study with two types of test suites: (1) Branch coverage based test suite, and (2) Random test suite.

We use the Siemens benchmark suite for this study. The benchmark suite contains 7 benchmark programs `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, and `tot_info`. The sizes of the programs range from 141 to 512 lines of code. There are 130 faulty versions of the programs. We use 20 branch coverage based test suites and 20 random test suites for each benchmark

A. *Results and Analysis*

In Section V-A1, we present the results for fault classes based on the original state failure condition. In Section V-A2, we present the results for the accessibility and impact based fault classes.

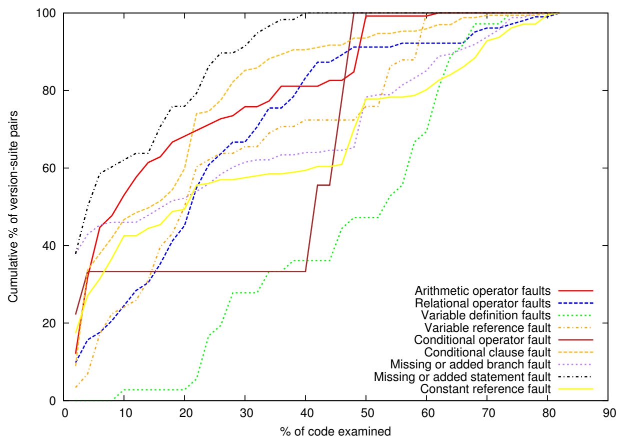
1) Original State Failure Condition Based Classes: To collect data on the effectiveness of the approaches, we execute each test suite on each faulty version. If at least one test case in a test suite fails for a faulty version, we obtain the Tarantula rank of the faulty statement for the (faulty version(F), test suite(T)) pair. If there is no failing run obtained for a (faulty version(F), test suite(T)) pair, then we do not have a data point for that pair. Table X shows the number of faults and number of (F, T) pairs of each original state failure condition based fault class.

Table X
FAULT DATA FOR ORIGINAL STATE FAILURE CONDITION BASED FAULT CLASSES

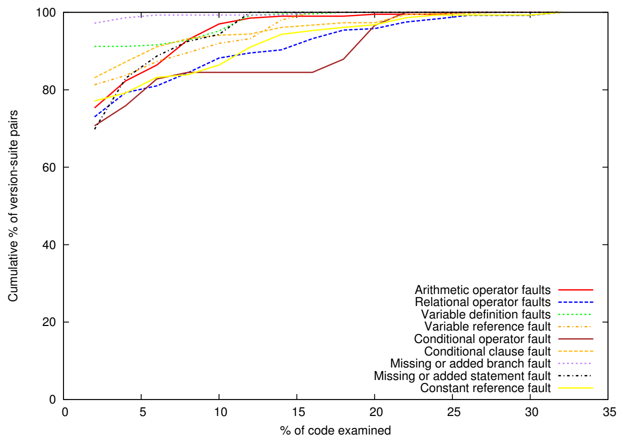
		#(F, T)	
Fault class	**#faults**	**Branch**	**Random**
Arith. op. faults	12	132	114
Rel. op. faults	18	102	114
Cond. op. faults	6	9	22
Var. def. faults	16	36	32
Var. ref. faults	16	58	73
Const. ref. faults	21	207	153
Missing/Added cond. clause	23	169	146
Missing/Added stmt.	6	58	30
Missing/Added branch	15	161	82

To obtain the graphical representation, we divide the entire range of percentage of code examined (0-100%) into small ranges. We choose ranges of length 2%. For each range, we calculate the percentage of (faulty version, test suite) pairs for which the percentage of code examined belongs to that range. We calculate the cumulative frequency distribution from this data. We plot the cumulative frequency distribution in a graph. The x-axis of the graphs shows the percentage of code examined. The y-axis shows the cumulative percentage of (faulty version, test suite) pairs. Thus, a point (10, 20) in the graph denotes that for 20% of the (faulty version, test suite) pairs one needs to examine at most 10% of the code.

Figures 1(a) and 1(b) respectively show the plots for different classes of faults based on original state failure condition, with branch coverage based test suite and random test suites.



(a) Branch Coverage Based Test Suite



(b) Random Test Suite

Figure 1. Effectiveness of Tarantula for Original State Failure Condition Based Classes

As shown in Figure 1(a), when the branch coverage test suite is used, there is no complete ordering between the fault classes in terms of how effectively they are localized. The curves show that different fault classes are more frequently localized at different ranges of % of code examined. For example, missing/added branch faults are more frequently localized within the range (0-25%) than constant reference faults. However, constant reference faults are more frequently localized within the range (25-50%) than missing/added branch faults.

We can observe some partial orders of effectiveness among the fault classes. Arithmetic operator faults, conditional clause faults, and missing/added statement faults are clearly more effectively localized than relational operator faults, missing/added branch faults, and variable reference faults.

The relative effectiveness changes significantly when we use random test suites, as shown in Figure 1(b). We can arrange the following fault classes in the order of how effectively they are localized (from high effectiveness to low): (1) missing/added branch faults, (2) variable definition faults, (3) relational operator faults, (4) conditional operator faults. These orders do not hold for branch coverage based test suites.

In summary, the observations are as follows: (1) Only some partial order of effectiveness is found between the fault classes, (2) The relative effectiveness vary significantly with the type of test suites used. This leads us to conclude that there is no clear relationship between original state failure condition based fault classes and the effectiveness of fault localization.

2) Accessibility and Impact Based Fault Classes: Table XI shows the number of faults for each of the four combinations of accessibility and impact based fault classes. It also shows the number of (F, T) pairs for each fault class and for each type of test suite.

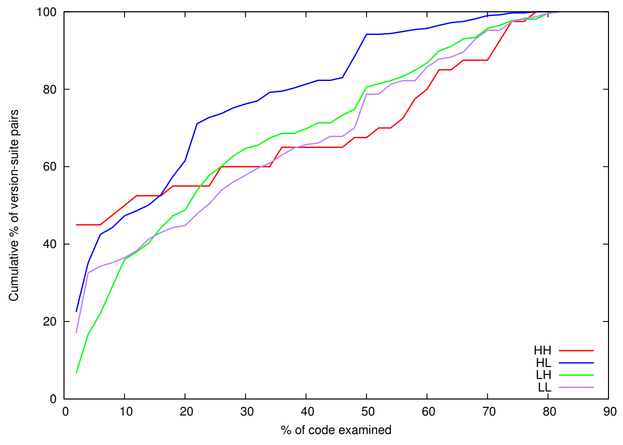
Table XI
Fault Data for Accessibility and Impact Based Fault Classes

Fault class	**#faults**	**#(F, T)**	
		Branch	**Random**
HH	4	40	5
HL	53	395	342
LL	30	230	210
LH	43	258	201

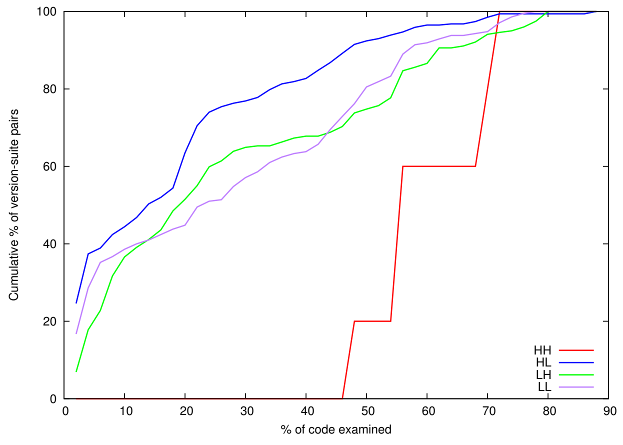
Figures 2(a) and 2(b) show the plots for effectiveness obtained for different fault classes based on accessibility and impact, using branch coverage based test suites and random test suites, respectively.

As Figure 2(a) shows, when branch coverage based test suites are used, HL faults are localized most effectively. This supports our earlier analysis that an HL fault is likely to be most effectively localized because (1) fewer successful tests execute it thereby decreasing its suspiciousness score, and (2) fewer statements may have higher suspiciousness scores than the faulty statement because the execution of fewer statements depends on it.

Although, overall the HL faults are more effectively localized than HH faults, in the range 0-20% of code examined,



(a) Branch Coverage Based Test Suite



(b) Random Test Suite

Figure 2. Effectiveness of Tarantula for Accessibility and Impact Based Fault Classes

HH faults are more frequently localized than HL faults. Each of these HH faults causes failure by not executing the statements that are dependent on it. This results in higher ranks for such a fault.

Both LH and LL faults are less effectively localized than HL faults. This again supports our analysis that faults that are hard to execute are localized more effectively. The overall effectiveness for LL and LH faults are comparable. This also suggests that if a fault is easy to access, then the impact of the fault does not appreciably affect the effectiveness of its localization.

The observations are consistent when we use random test suites, as shown in Figure 2(b). HL faults are most effectively localized than all the other types. The effectiveness for LL and LH faults are comparable. We had 5 data points for HH faults with the random test suites. In all these cases, the HH faults caused failures by executing the statements that are control dependent on them. Therefore, many statements obtained higher suspiciousness scores than the fault statements, thereby decreasing the rank of the faults.

VI. Threats to Validity

The backward static slice of a faulty statement may not always accurately measure the accessiblity of the faulty statement. A backward static slice is the union of all the statements that affect a given statement through control dependencies, along all program paths. When a test executes, the faulty statement is accessed by a particular path. A faulty statement that can be accessed by many easily executable paths may have a large backward static slice but still be easily executable by a single test case.

We use the slice size as a measure of accessiblity. However, the number of statements on which a given statement is control dependent does not always accurately measure the accessiblity of the given statement. Consider a statement s_1 that is nested inside two conditional statements, and another statement s_2 that is nested within a single conditional statement. Thus the accessibility measure of s_1 is higher than that of s_2, and s_1 may be categorized as a hard to access fault, while s_2 an easy to access fault. However, s_2 may be harder to access than s_1 because the condition guarding s_2 may be harder to satisfy than the conjunction of the two conditions that guard s_1.

Forward static slices may not always accurately measure the impact. A forward static slice contains all the statements dependent on a given statement along any program path. A test case executes only one path among them, which may result in a small impact. However, the union of all potentially impacted statements, which makes up the forward static slice, may be large.

The percentage of code examined may not correctly represent the effort to locate the faulty statement. This is because as a tester examines each statement, he determines if the statement is faulty or not. Determining the correctness of each statement requires an amount of effort, which may be different for different statements. Percentage of code examined does not address these differences in effort.

Threats to external validity stem from the nature of benchmarks used. The programs are relatively small, and contain only one fault each.

VII. Related Work

Many existing empirical evaluations of fault localization techniques studied the effect of using different models and spectra. Renieris et al. [2] evaluated their nearest neighbor model with two types of spectra: (1) binary coverage spectrum that contains coverage of basic blocks, and (2) permutation spectrum that contains execution counts of the basic blocks. They showed that the use permutation spectrum results in better effectiveness. Santelices et al. [3] evaluated the Ochiai model with three different types of spectra: (1) statement coverage, (2) branch coverage, and (3) du-pair coverage. The results showed that du-pair coverage is more effective than branch coverage, which is more effective than statement coverage.

Jones et al. [9] performed an empirical study to compare the effectiveness of five fault localization techniques: (1) Tarantula, (2) Set intersection, (3) Set union, (4) Nearest neighbor [2], and (5) Cause transition. The results showed that Tarantula was the most effective in localizing faults.

Some existing empirical evaluations studied the effect of using different types of test suites. Yu et al. [10] investigated the impact of various test suite reduction strategies on the effectiveness of fault localization. They used two types of test suite reduction strategies: (1) statement-based reduction, which selects from a test suite a reduced test suite that covers the same statements as the original suite, and (2) vector-based reduction where the reduced test suite covers the same set of statement vectors (set of statements executed by one test case) as the original test suite. Their results showed that statement based reduction significantly reduces the effectiveness, while vector-based reduction has negligible effect.

Abreu et al. [6] showed that effectiveness improves if there are fewer instances of coincidental correctness. They also studied the effect of the number of passing and failing runs on the effectiveness. Their results show that (1) the effectiveness improves if more failing test cases are used, and (2) the effect of including more passing runs is unpredictable.

None of these empirical evaluations studied the effect of faults classes on the effectiveness. In our paper, we have used fault class as an independent variable.

VIII. Conclusions and Future Work

Our results show that faults that are hard to access and have low impact are the most effectively localized. No conclusions can be drawn about the effect of the original state failure condition on the effectiveness on Tarantula.

In the future, we plan to use large scale studies to investigate the effect of the three properties. The studies will use larger benchmark programs, a higher number of faulty versions, and more test cases. To address the confounding effect of test suites even better, we will use test suites satisfying other adequacy criteria, such as data-flow coverage criteria. We also plan to evaluate the the effectiveness of spectrum based fault localization techniques for non-deterministic faults such as concurrency faults.

Since forward and backward slices may not be the most accurate measures of accessibility and impact, we will use other measures that estimate accessibility and impact more accurately.

Finally, we want to use the results of the evaluation to improve the effectiveness of spectrum-based fault localization. An example would be to adjust the suspiciousness scores of statements that are hard to access but that obtained higher suspiciousness scores because of their location in the program.

Acknowledgments

We thank GrammaTech for providing us with an academic license of the full version of CodeSurfer, which we used to the calculate forward and backward slices.

References

[1] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, 2002, pp. 467–477.

[2] M. Renieris and S. Reiss, "Fault localization with nearest neighbour queries," in *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, Montreal, Canada, 2003, pp. 30–39.

[3] R. Santelices, J. A. Jones, Y. Yanbing, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, Vancouver, Canada, 2009, pp. 56–66.

[4] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Softw. Engg.*, vol. 12, no. 2, pp. 143–160, 2007.

[5] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005, pp. 263–272.

[6] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[7] D. Richardson and M. Thompson, "An analysis of test data selection criteria using the relay model of fault detection," *IEEE Transactions on Software Engineering*, vol. 19, pp. 533–553, 1993.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *ISSTA '10: Proceedings of the 19th International Symposium on Software Testing and Analysis*, Trento, Italy, 2010, pp. 49–60.

[9] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005, pp. 273–282.

[10] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 201–210.