Preprint: This is an accepted version of a peer reviewed manuscript published online first by the Software Quality Journal, March, 2016.

Noname manuscript No.
(will be inserted by the editor)

Fault Localization for Automated Program Repair: Effectiveness, Performance, Repair Correctness

Fatmah Yousef Assiri · James M. Bieman

Received: date / Accepted: date

Abstract Automated program repair (APR) tools apply fault localization (FL) techniques to identify the locations of likely faults to be repaired. The effectiveness, performance, and repair correctness of APR depends in part on the FL method used. If FL does not identify the location of a fault, the application of an APR tool will not be effective– it will fail to repair the fault. If FL assigns the actual faulty statement a low priority for repair, APR performance will be reduced by increasing the time required to find a potential repair. In addition, the correctness of a generated repair will be decreased since APR will modify fault-free statements that are assigned a higher priority for repair than an actual faulty statement. We conducted a controlled experiment to evaluate the impact of ten FL techniques on APR effectiveness, performance, and repair correctness using a brute force APR tool applied to faulty versions of the Siemens Suite and two other large programs: space and sed.

All FL techniques were effective in identifying all faults; however, Wong3 and Ample1 were the least effective FL techniques since they assigned the lowest priority for repair in more than 26% of the trials. We obtained the worst APR performance significantly when Ample1 was used since it generated a large number of variants in 29.11% of the trials, and took the longest time to produce potential repairs. Jaccard FL improved repair correctness by generating more validated repairs–potential repairs that pass a set of regression tests, and generating potential repairs that failed fewer regression tests. Also Jaccard's performance is noteworthy in that it never generated a large number of variants during the repair process compared to the alternatives.

F. Y. Assiri
College of Computing and Information Techonlogy, King AbdulAziz University, Jeddah, K.S.A.
Tel.: +966-54-291-1982
E-mail: fyassiri@gmail.com

J. M. Bieman
CS Department, Colorado State University, Fort Collins, CO 80523-1873, U.S.A.

Keywords automated program repair · brute-force · fault localization technique · effectiveness · performance · repair correctness

1 Introduction

Debugging is an expensive process [22] that includes locating software faults and fixing them. Automated program repair (APR) refers to techniques that locate and fix software faults automatically, and thus promise to dramatically reduce debugging costs. APR techniques apply fault localization (FL) to guide a repair tool towards code segments that are more likely to contain faults. Then an APR tool can modify the code most likely to contain faults. FL techniques compute a suspiciousness score to indicate the likelihood that each statement contains a fault. A list of potentially faulty statements (LPFS), ordered by their suspiciousness, is created for use by the repair tool.

The effectiveness, performance, and repair correctness of APR can be impacted by the selected FL technique. APR effectiveness is the ability to fix faults, while performance is the time or number of steps required to find a potential repair, and repair correctness indicates how well a potentially repaired program retains the required functionality. An ineffective fault localization technique might mislead the repair process by missing the statements where a fault hides, assigning a low score to the actual faulty statement, or identifying too many statements that might contain the faults. Missing faulty statements will lower APR effectiveness by failing to find a potential repair. Assigning a low score to a faulty statement will not decrease APR effectiveness, but it will reduce APR performance and generate an incorrect repairs since the APR will unproductively modify many fault-free statements before reaching an actually faulty statement. On the other hand, identifying too many statements might improve APR effectiveness by increasing the chance of finding a potential repair, but with potentially poor performance and incorrect repair. In the worst case, a fault localization technique can mark all statements in a program as potentially faulty, which can decrease the performance of APR dramatically especially with large programs. Le Goues et al. [27] found that for APR "time is governed" by the number of potentially faulty statements rather than program size. Thus, a fault localization technique that marks fewer statements, and/or places an actually faulty statement at the head of the LPFS will decrease the number of variants generated by an APR technique to find a potential repair, thus improving APR performance. It also will improve repair correctness by guiding the APR to actually faulty statements and reduce the chances of modifying non-faulty statements.

Different FL techniques have been used with APR to locate potential faults. Weimer et al. [21, 39] apply a simple *Weighting Scheme* that assigns weight values to statements based on their execution by passing and failing tests. Higher weights are assigned to statements that are executed only by failing tests, and lower weights are assigned to statements that are executed by both passing and failing tests. They excluded statements that are only executed by passing tests to prevent changing correct statements. Our previous paper [11], which evaluated the correctness of generated repairs, uses the *Weighting Scheme* following the Weimer et al. approach. Nguyen et al. [33] use the *Tarantula* fault localization technique [25,24,23]. Debroy and Wong [18,19] use Tarantula and *Ochiai* fault localization techniques to rank program statements based on their likelihood of containing faults. Using better fault localization techniques can identify actual faulty statements more quickly, thus improving APR effectiveness, performance, and repair correctness.

The effectiveness of different fault localization techniques on automated program repairs has been evaluated by Qi et al. [35]. Their evaluation used GenProg, an APR that implements a genetic algorithm to find the optimal solution from the solution space. Genetic algorithms employ randomness through the selection of program modification operators (PMOs) and a crossover operator; the randomness of the search algorithm might affect the accuracy of the reported results since there is dependency between the accuracy of the fault localization technique and the randomness of the search algorithm. The FL technique might accurately locate the actual faulty statement, but the search algorithm can select a PMO that does not fix the fault. A brute-force algorithm is guaranteed to fix a fault if a repair is feasible. Therefore, a failure to find a potential repair can be related to the selected FL technique. On the other hand, Debroy and Wong evaluated the impact of Tarantula and Ochiai with an APR technique that applies a brute-force search method. The effectiveness of different FL techniques is measured in terms of the number of PMOs applied until a fault is fixed, and they found that Ochiai fixed more faults with fewer PMOs compared to Tarantula.

In this paper we do not propose a novel approach; rather, we designed and conducted a controlled experiment to accurately evaluate the effect of FL on APR effectiveness, performance, and repair correctness. There was a study by Qi et. al. [35] to evaluate the impact of different FL techniques when used for APR effectiveness and performance. However, to the best of our knowledge, we are the first to study the impact of different fault localization techniques within APR on the correctness of generated repairs. We evaluate FL techniques with an APR that applies a brute-force search algorithm to eliminate the randomness of a genetic algorithm that might affect the accuracy of the results reported by Qi et al. [35].

Even though Debroy and Wong applied a brute-force search method to evaluate FL techniques when used for APR, the differences between their evaluation and our evaluation are as follows: 1) Debory and Wong compared the effectiveness of two FL techniques (Tarantula and Ochiai) while we compared the effectiveness of ten of the most recent FL techniques in the literature, 2) Debroy and Wong use a brute-force search method that applies PMOs in a random order; however, we order PMOs so that PMOs with the greater chance of fixing faults are applied earlier, 3) Our study measures APR effectiveness, performance, and the correctness of generated repairs when using different FL techniques, while their approach only measured the effectiveness of FL techniques, and 4) We measured APR effectiveness when different FL techniques are used as the ability of an FL technique to identify actual faulty statements, while they measured effectiveness as the number of PMOs applied until a repair is produced.

We studied the impact of ten well known fault localization techniques: Jaccard, Optimal, OptimalP, Ochiai, Ochiai2, Tarantula, Ample1, Ample2, Wong3, and Zoltar. Our evaluation was conducted on six subject programs that include large programs (more than 12K LOC). Our results show that all ten fault localization techniques successfully identified actually faulty statements. Ample1 decreased APR performance significantly since it assigned faulty statements low priority for repair, which caused the APR process to generate a large number of variants, and took a greater amount of time to produce potential repairs. Jaccard produced more validated repairs–potential repairs that pass a set of regression tests. In addition, Jaccard always assigned faulty statements a high priority for repair. It took an average of 43.28 seconds to produce potential repairs, which was the best performance of the tested FL techniques.

The main contributions of this paper are the following:

– A framework for comparing FL techniques in terms of their impact on the effectiveness, performance, and repair correctness when used for automated program repair. The MUT-APR evaluation framework is built by adapting GenProg to (1) fix operator faults, (2) use a brute-force search, and (3) apply ten FL techniques.

– A determination of the impact of FL techniques on an APR tool's ability to repair a faulty statement. We found that all studied FL techniques identified actually faults statements.

– Our results show that Ample1 decreased APR performance significantly compared to the alternative FL techniques, while differences between the alternatives are not always statistically significant at the 0.95 confidence level.

– Jaccard produced more repairs (%70.88) that were validated by passing all regression tests compared to all alternative FL techniques. Also, Jaccard generated potential repairs that rarely failed regression tests (an average of 2.57%).

2 Background

2.1 Fault Localization Techniques (FL)

Fault localization techniques were introduced in order to guide developers towards the most suspicious statements to check during debugging. Lately, FL techniques are employed by APR to guide search algorithms towards statements that are more likely to hide faults than other statements (step 1 in

Table 1: The dynamic behavior of the faulty program *gcd* when executed against tests in $T1$, ..., $T5$. *Sus. Score* is the suspiciousness score computed using Tarantula.

Stmt ID	Stmt	T_1	T_2	T_3	T_4	T_5	Sus. Score
	gcd (int a, int b) {						
1	if(a < 0) //fault					✓	1.00
2	{ printf("%g \n ", b) ;						0.00
3	return 0 ; }						0.00
4	while(b ! = 0)	✓	✓	✓	✓	✓	0.50
5	if(a > b)	✓	✓		✓	✓	0.57
6	a = a − b ;		✓		✓		0.00
7	else	✓	✓		✓	✓	0.57
8	b = b − a ;	✓	✓		✓	✓	0.57
9	printf("%g \n ", a) ;	✓	✓	✓	✓		0.00
10	return 0 ;	✓	✓	✓	✓		0.00
	}						

Figure 1). Thus, applying FL helps to fix faults faster without breaking other required functionality.

FL techniques locate potentially faulty statements in the source code by computing a *suspiciousness score* for each statement that indicates its likelihood of containing a fault. Then, statements are ordered based on their suspiciousness. Developers can use the suspiciousness score to order their search for a fault to debug. Spectrum-based fault localization (SBFL) [7,12,16,24, 36,5,15] is a common FL approach; it compares the program behavior of a passing execution to that of a failing execution. SBFL collects information on the dynamic behavior of program statements when they are executed against each test in a test suite. SBFL methods record the number of passing and failing tests executed for each statement, and compute a suspiciousness score for each statement. Statements that are executed more often during a failing run are considered to be more likely to contain faults, thus are assigned a higher suspiciousness score than other statements in the program. Many heuristics have been proposed to compute statement suspiciousness scores [7,25,24,36, 5,5,32].

To illustrate how FL techniques rank program statements using the computed suspiciousness scores, we used the C program adapted from an example used by Weimer et al. [39] (Table 1), which computes Eculid's greatest common divisor. FL techniques count the number of passing and failing tests for each statement. Each FL technique uses a formula to compute suspiciousness scores. This example uses five test inputs $TS = T_1, T_2, T_3, T_4, T_5$ in which T_1, T_2, T_3, and T_4 are passing tests, and T_5 is a failing test, and Tarantula computes suspiciousness score for each statement using equation from Table 2. A list of potentially faulty statement (LPFS), which consists of statement IDs and their suspiciousness scores is created and sorted (Table 3). The LPFS contains all statements with a suspiciousness score greater than zero, and will be used by the APR tool to locate the fault.

Table 2: Fault Localization Heuristics

FL	Formula
Jaccard	$\frac{Tests_{ef}(s)}{Tests_{ep}(s)+Tests_{totf}}$
Optimal	$\begin{cases} -1 & \text{if } Tests_{nf}(s) > 0 \\ Tests_{np}(s) & \text{otherwise} \end{cases}$
OptimalP	$Tests_{ef}(s) - \frac{Tests_{ep}(s)}{Tests_{totp}+1}$
Ochiai	$\frac{Tests_{ef}(s)}{\sqrt{Tests_{totf} \times (Tests_{ep}(s)+Tests_{ef}(s))}}$
Ochiai2	$\frac{Tests_{ef}(s)*Tests_{np}(s)}{\sqrt{(Tests_{ef}(s)+Tests_{ep}(s))*(Tests_{np}(s)+Tests_{nf}(s))*(Tests_{totf})*(Tests_{totp})}}$
Tarantula	$\frac{\frac{Tests_{ef}(s)}{Tests_{totf}}}{\frac{Tests_{ep}(s)}{Tests_{totp}}+\frac{Tests_{ef}(s)}{Tests_{totf}}}$
Ample	$\lvert\frac{Tests_{ef}(s)}{Tests_{totf}} - \frac{Tests_{ep}(s)}{Tests_{totp}}\rvert$
Ample2	$\frac{Tests_{ef}(s)}{Tests_{totf}} - \frac{Tests_{ep}(s)}{Tests_{totp}}$
Wong3	$Tests_{ef} - h, \quad where \quad h = \begin{cases} Tests_{ep} & \text{if } Tests_{ep} \leq 2 \\ 2 + 0.1 * (Tests_{ep} - 2) & \text{if } 2 \leq Tests_{ep} \leq 10 \\ 2.8 + 0.001 * (Tests_{ep} - 10) & \text{if } Tests_{ep} > 10 \end{cases}$
Zoltar	$\frac{Tests_{ef}(s)}{Tests_{totf}+Tests_{ep}(s)+\frac{10000*Tests_{nf}(s))*Tests_{ep}(s)}{Tests_{ef}(s)}}$

Table 3: List of Potentially Faulty Statements (LPFS) in format used by the APR tool

Statement ID	Suspiciousness score
1	1.00
5	0.57
7	0.57
8	0.57
4	0.50

2.2 Automated Program Repair (APR)

Automated program repair (APR) techniques locate and fix faults. APR techniques take a faulty program and a set of repair tests, and produce a repaired program. APR techniques consist of three main steps: fault localization (Step 1), variant creation (Step 2), and variant validation (Step 3). Figure 1 describes the overall organization and activities of APR techniques.

First, an APR technique locates faults (Step 1 in Figure 1) by applying a fault localization technique creating the LPFS as explained in the previous section. Then, an APR technique fixes faults (Step 2 in Figure 1) by modifying a faulty program using a set of program modification operators (PMOs)

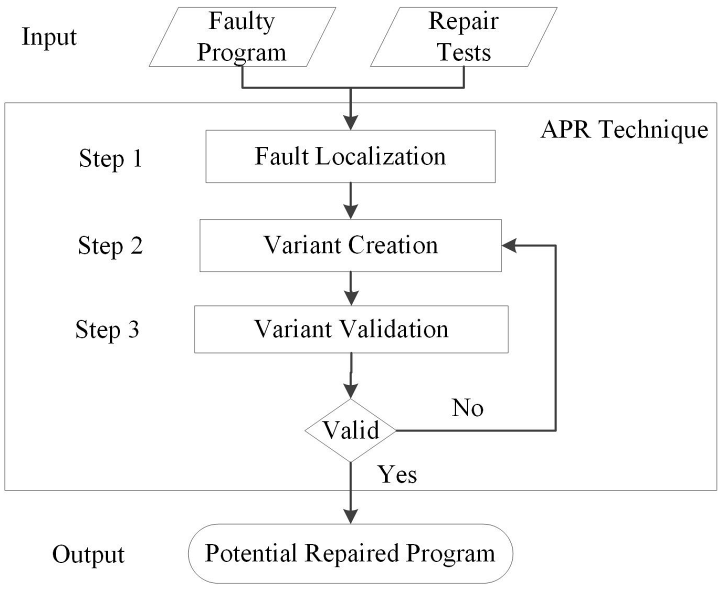


Fig. 1: Overall Automated Program Repair (APR) Process

that change the code in the faulty statement to generate a new version of the faulty program, which is called a *variant*. An APR technique applies a search algorithm to create variants by selecting and applying a PMO; some search algorithms run for multiple iterations and in some cases APR techniques generate a variant from a variant produced in prior iterations. The variant is validated (Step 3 in Figure 1) by executing it against a set of repair tests. The variant is called a *potential repair* or *potential repaired program* if it passes all of the repair tests. The repair process stops when it finds a potentially repaired program, or when the number of iterations have reached a limit. A potential repair is considered a *validated repair* when it passes a set of tests (often regression tests) that were not included in repair tests.

3 Automated Program repair (APR) Framework

To fix faults, we used our MUT-APR repair tool [11]. MUT-APR was built by adapting the GenProg version 1 framework [21,39,38,27]. Unlike GenProg which makes use of existing code in the subject program to repair faults, MUT-APR applies a set of PMOs that construct new operators to replace faulty ones creating new variants within a genetic algorithm. For this paper, we replaced the genetic algorithm in MUT-APR with a brute-force search algorithm, and added support for the use of different FL techniques applying the changes developed by Qi et al. [35].

Table 4: Program Modification Operators (PMOs) Supported by MUT-APR.

PMOs	Description
ROR	Relational Operator Replacement
AOR	Arithmetic Operator Replacement
BWOR	BitWise Operator Replacement
SOR	Shift Operator Replacement

3.1 Program Modification Operators (PMOs)

MUT-APR makes use of a mutation-based technique adapted from Debroy and Wong [18,19] to fix simple faults. Simple operator faults are common mistakes made by developers. For example, relational operator faults, such as the use of > instead of >=, can produce off-by-one errors, and many operator faults are security vulnerabilities [1]. Many faults can be fixed by one-line modifications [34,17]. In addition, our previous study shows that the use of simple PMOs improves the correctness of generated repairs by generating more validated repairs and generated repairs are similar to repairs done by humans [11].

Faults are fixed in source code by constructing new operators. PMOs are applied to change a faulty operator into a set of alternatives until finding the correct operator. For example, MUT-APR can generate five different variants when changing a mutable statement (a mutable statement is a program construct that can be changed by one of the supported PMOs) that contains the > operator. We apply PMOs to fix faults in all statements that contain relational operators, arithmetic operators, bitwise operators, and shift operators (Table 4). The PMOs are as follows: (1) change relational operators in *if* statements, *return* statements, *assignments*, and *loops*, (2) change arithmetic operators, bitwise operators and shift operators in *return* statements, *assignments*, *if* bodies, and *loop* bodies.

Although MUT-APR supports PMOs to fix faults in binary operators, MUT-APR does not apply PMOs to fix faults in Logical Operators due to MUT-APR's use of CIL [32]. CIL is an intermediate language for C programs to manipulate source code and generate a simplified version of source code. It transforms logical operators into *if-then* and *if-else* blocks. Therefore, in order to fix faults in logical operators w.r.t. CIL would require making changes to the generated *if* blocks rather than the operators. We left this for future work.

3.1.1 PMO Algorithm

Algorithm 1 is used in the implementation of our PMOs. MUT-APR selects the first potentially faulty statement from the LPFS. If the statement contains an operator (Line 4), the operator is checked (line 5). Then, a PMO, that is one of the alternatives of the potential faulty operator, is selected (line 6)(e.g., *stmti* contains >, and PMO that is change > operator into an alternative is selected), the statement type is checked (line 7), and a new statement *stmtj* is created (line 8). Then *stmti* is substituted by *stmtj* (line 9) creating a new variant.

Algorithm 1 PMOs Pseudocode to Modify Simple Operators

```
1: Inputs: Program P and List of Potentially Faulty Statements LPFS
2: Output: mutated program
3: for all statements stmti in the LPFS do
4:    if stmti contains an operator then
5:       let stmtOp = checkOperator(stmti)
6:       let pmo = choose(ChangestmtOpToOp),
7:       let stmtType = checkStmtType(stmti)
8:       let stmtj= apply(stmti,pmo)
9:       substitute stmti with stmtj in program P
10:   end if
11: end for
12: return P with stmti substituted by stmtj
```

3.1.2 Illustrated Example

A faulty Euclid's greatest common divisor adapted from an example used by Weimer et al. [39] illustrates our approach. The original fault was a missing statement (line 3), which cannot be fixed by MUT-APR's PMOs. In order to demonstrate how MUT-APR fixes binary operator faults, we inserted the missing statement and seeded a fault in the *if* statement in line 1 as shown in Table 1.

The *gcd* program in Table 1 has three relational operators and two arithmetic operators: line 1: *if(a <0)*, line 4: *while(b !=0)*, line 5: *if(a>b)*, line 6: *a = a - b*, and line 8: *b = b - a*. The faulty operator is in the first *if* statement (line 1). In order to fix the fault, the operator in line 1 must be switched to ==.

We assume that all three relational operator statements and the two arithmetic operators are identified as potentially faulty statements plus two other statements, and the statement in line 1 is the first statement in the LPFS, thus it is selected first for modification. MUT-APR checks the operator in the statement and selects one of its alternatives (based on the predefined order). The faulty operator in statement 1 is <. If the tool selects the PMO that changes < to > , a new variant is created by constructing a new operator for line 1. The variants are compiled and executed against repair tests. This variant fails two test inputs: (0,55) and (55,0). Since the variant did not pass all repair tests, it is not a potential repair and the process continues. Another statement and PMO are selected. MUT-APR will successfully repair the fault in line 1 when it selects the correct PMO.

3.2 Search Algorithm

APR applies a search algorithm to select a program modification operator (PMO) from a pool of PMOs to modify a suspicious statement (step 2 in Figure 1). PMOs can be selected in a predefined order as done in a brute-force search method, or randomly as done in stochastic search. Brute-force requires an exhaustive search that applies all possible changes to the program until a potential repair is found, which is inefficient and can be infeasible with large programs. In contrast, a genetic algorithm applies mutation and crossover operators to modify a faulty program. A genetic algorithm randomly selects PMOs from a pool of operators, and a crossover operator combines changes from two parent variants to generate a new child variant. A genetic algorithm does not guarantee a potential repair due to its randomness. In this study, we apply a brute-force APR process to eliminate the randomness of a genetic algorithm, and to guarantee a potential repair when the FL technique identifies the faulty statements and the repair is supported by the set of PMOs.

A brute-force algorithm applies all possible PMOs for each mutable statement from the LPFS. PMOs are applied in a predefined order; we order relational PMOs to apply operators with a greater chance to fix the faults before other operators; thus, decreasing the number of PMOs applied in the exhaustive search to find potential repairs (the ordering mechanism is beyond the scope of this paper, for more details look at the dissertation manuscript [10]).

3.2.1 Brute-force Algorithm

Algorithm 2 describes how the brute-force algorithm fixes faults within MUT-APR. The algorithm modifies statements sequentially. It takes the first statement in the LPFS, and checks if it contains an operator (line 4). If the operator is mutable, it applies all possible alternatives (line 5-10). Each change creates a variant (line 7). The fitness value is computed for each generated variant (line 8) by executing it against the repair tests (one of the inputs to the repair process in Figure 1). If a variant that passes all repair tests is found (in other words, the variant has a fitness value equal to the maximum fitness value), a potential repair is found. If not, the process continues with the next statement in the LPFS until a potential repair is found (line 9) or the algorithm reaches the last statement in the LPFS without a potential repair.

3.3 Fitness Function

Fitness values are computed for each generated variant to determine its goodness. The fitness function, adapted from GenProg [39], is given in Equation 1. $Tests_{pass}$ and $Tests_{fail}$ are the number of passing and failing tests respectively, and W_{pass} and W_{fail} are positive constants that represent the weights of passing and failing tests respectively. Failing tests are assigned a weight of 10 and passing tests are assigned a weight of 1 following the approach by

Algorithm 2 Brute-Force Pseudocode

```
1: **Inputs**: Program P , List of Potentially Faulty Statements LPFS, and maximum fitness
   value
2: **Output**: Variant
3: for i=0 to length(LPFS)-1 do
4:    let stmtOp = checkOp(stmti)
5:    for all program modification operators pmo for stmtOp do
6:       repeat
7:          let variant= apply(stmti, stmtOp, pmo)
8:          let variant_fitness= computeFitness(variant)
9:       until variant_fitness= maximum fitness || mOp is the last PMO for stmti
10:   end for
11:   if i != last index in the LPFS then
12:      i++
13:   end if
14: end for
15: return variant
```

Weimer et al. [39]. A variant (v1) that passes failing tests will have a higher fitness value than another variant (v2) that passes all passing tests but not failing tests. Thus, v1 will be more likely to be used for the next generation than v2. If a variant that maximizes the fitness function (equal to *max*, which is one input to the algorithm) is found, a potential repair is identified and the process stops. Different weights can be assigned to repair tests. Weimer et al. [28] studied the impact of assigning different weights to passing and failing tests on APR effectiveness; however, this is beyond the scope of this paper.

$$fitness = |Tests_{pass}| * W_{pass} + |Tests_{fail}| * W_{fail} \tag{1}$$

4 Fault Localization Techniques (FL)

We used ten well known fault localization techniques proposed in the literature: Jaccard, Optimal, OptimalP, Ochiai, Ochiai2, Tarantula, Ample1, Ample2, Wong3, and Zoltar. We selected these ten FL techniques for the following reasons: (1) they have been proposed recently (between 2002 and 2007), (2) Ochiai was identified as a highly effective FL technique from a developer point of view [5,30], and Debory and Wong's [18,19] results showed that more faults were fixed with fewer PMOs using the Ochiai FL technique with their APR approach, (3) Jaccard was identified as the most effective FL technique with GenProg [35], (4) Tarantula was used in prior work with APR techniques [18,33], and (5) Optimal and OptimalP were found to be more effective than Ochiai and Jaccard from a developer point of view [32].

We implemented the selected fault localization techniques using about 649 LOC of OCaml code. Each FL technique applies a different heuristic; Table 2 shows the formula to compute the suspiciousness score for each fault localization technique; where $Tests_{ep}(s)$ is the number of passing tests that are executed for statement s, $Tests_{ef}(s)$ is the number of failing tests that are executed for statement s, $Tests_{np}(s)$ is the number of passing tests that are not executed for statement s, $Tests_{nf}(s)$ is the number of failing tests that are not executed for statement s, and $Tests_{totp}$ and $Tests_{totf}$ are the total number of passing and failing tests respectively in the test suite.

5 Study Method

In this section we describe research questions and metrics used to answer each question, study data, and the evaluation design to study the impact of different FL techniques on APR effectiveness, performance, and repair correctness.

5.1 Research Questions and Evaluation Metrics

Our evaluation study is designed to answer the following research questions:

RQ1: What is the relative APR effectiveness when different FL techniques are employed?

APR effectiveness is the ability to fix faults. An FL technique that successfully determines the actual faulty statements improves APR effectiveness. On the other hand, an FL technique that fails to identify actual faulty statements limits APR effectiveness.

RQ2: Which FL technique give faulty statements the highest priority for repair?

We are concerned with the accuracy of FL techniques in identifying actual faulty statements from an automated program repair point of view. APR selects statements sequentially from the LPFS, thus their position in the LPFS is essential to the APR. A fault localization technique that places the actual faulty statement at the head of the list prevents unwanted modifications by changing statements that hold program functionality. Therefore, we measure the priority of a statement by its position in the LPFS produced by the FL technique. The position of a statement in the LPFS is its LPFS rank. Statements with higher suspiciousness scores are placed near the head of the list, thus have a lower LPFS rank (higher priority for repair) compared to other statements. For example, in Table 3 statement ID 1 has the lowest (best) LPFS rank (LPFS rank =1) since it has the highest suspiciousness score. On the other hand, statement ID 4 has the highest (worst) LPFS rank (LPFS rank = 5).

We compared the LPFS rank of the actual faulty statement using the LPFS that is created by each FL technique. An FL technique that assigns a higher suspiciousness score to the actual faulty statement, placing it near the head of the LPFS (lower LPFS rank), improves APR performance and repair correctness compared to another FL technique that places the actual faulty statement far from the head of the LPFS (higher LPFS rank).

RQ3: How does the use of different FL techniques affect the number of generated variants (NGV) until a potential repair is found?

In order to repair faults, APR tools apply PMOs to modify a location in the faulty program to generate variants. Modifying a non-faulty statement generates an invalid variant (a variant that does not pass all repair tests). If the actual faulty statement is placed earlier in the LPFS, then the APR will change the faulty statement and produce fewer invalid variants. NGV, defined by Qi et al. [35], measures the number of generated variants until a potential repair (a variant that passes all repair tests) is found. We compared NGV when applying a brute-force APR using different FL techniques (lower NGV is better). An FL technique that assigns a low LPFS rank to actual faulty statement requires fewer statements to be modified, thus creating fewer variants (reducing NGV).

Consider List 2 from Table 5b, NGV is the sum of possible changes for each statement from the head of the list until producing a potential repair. A potential repair is found by changing the operator in the faulty statement (statement ID = 1) into a correct operator. Thus, for this particular example, NGV is sum of the possible changes for statements ID 5, statement ID 8, and statement ID 1. Statement ID 5 contains the $>$ operator, which has five possible changes, statement ID 8 contains the $-$ operators, which has four different possible changes, and statement ID 1 contains the $<$ operator, which has 5 different changes. However, changes will be applied to statement ID 1 until a repair is found; thus possible changes to statement ID 1 to produce a potential repair can be a value between 1 and 5. The NGV for this example NGV can be a minimum of 10 and a maximum of 14.

The difference between the LPFS rank metric (used for RQ2) and NGV is that the LPFS rank metric is totally dependent on the FL technique; however, NGV can be influenced by the number of mutable statements with lower LPFS ranks than the actual faulty statement. For example, consider the use of two FL techniques (FL1 and FL2) to identify the actual faulty statement in the *gcd* program. FL1 creates List1 (Table 5a), and FL2 creates List2 (Table 5b). Both techniques assign the same LPFS rank for the actual faulty statement (the actual faulty statement, statement ID 1, in both lists has an LPFS rank = 3). However, NGV depends on the number of mutable statements prior to the actual faulty statement. List1 consists of two statements prior to the actual faulty statement (statement ID 2 and 3) but neither can be mutated by MUT-APR PMOs; thus NGV can be equal to any value between 1 and 5 (depending on the order of the application of alternative PMOs that transform faulty operator $<$ into the correct one $==$). On the other hand, List2 consists of two mutable statements prior to the actual faulty statement (statement 5 and 8), thus NGV can be equal to any value between 2 and 10.

RQ4: Does the use of different FL techniques affect the total time required to find a potential repair?

We computed the total time required to find a potential repair. Total time is the sum of the time needed to generate a new variant, compile and execute

Table 5: List of Potentially Faulty Statements (LPFS) for *gcd* created by two FL techniques: *LPFS1* is created by FL1 and *LPFS2* is created by FL2

(a) LPFS1

Statement ID	Suspiciousness score	LPFS rank
2	0.96	1
3	0.91	2
1	0.80	3
5	0.71	4
7	0.68	5
8	0.57	6
4	0.5	7

(b) LPFS2

Statement ID	Suspiciousness score	LPFS rank
5	0.96	1
8	0.91	2
1	0.80	3
2	0.72	4
3	0.6	5
4	0.5	6
10	0.5	7

each generated variant on the repair tests, and compute its fitness values for all variants until producing a potential repair. We compared the total time, measured in seconds, when each FL technique is used.

RQ5: Does the use of different FL techniques affect the correctness of generated repairs?

Repair correctness concerns how well a potential repair retains the required functionality. To measure repair correctness, we defined PFR, which is the percentage of failed potential repairs, and PFT, which is the percentage of failed regression tests for each potential repair. PFR measured the percent of potential repairs that failed at least one regression test (Equation 2). In other words, it measured the percent of validated repairs (validated repair is a potential repair that passes a set of regression tests), which is $100 - PFR$, and PFT measures how far each failing potential repair from being a validated repair which is computed by Equation 3.

$$PFR_{FLTechnique} = \frac{PotentialRepairs_{failed}}{TotalPotentialRepairs} * 100 \tag{2}$$

$$PFT_{PotentialRepair} = \frac{RegressionTests_{failed}}{TotalRegressionTests} * 100 \tag{3}$$

Figure 2 describes steps that we use to study repair correctness. APR requires a set of repair tests generating one or more potential repairs, then we execute potential repairs on a set of regression tests, and compute PFR and PFT.

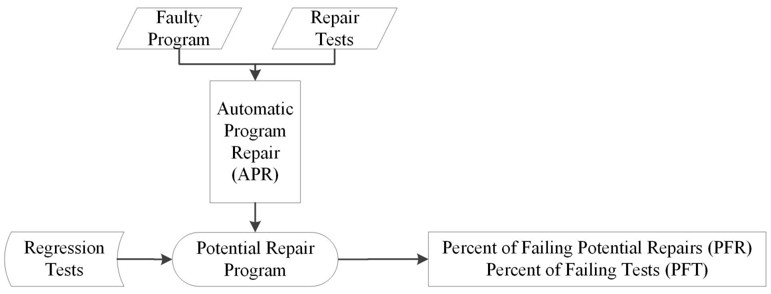

Faulty Program
Repair Tests
Automatic Program Repair (APR)
Regression Tests
Potential Repair Program
Percent of Failing Potential Repairs (PFR)
Percent of Failing Tests (PFT)

Fig. 2: Steps to study repair correctness.

Table 6: Benchmark programs. Each *Program* is an original program from the SIR [2]. *LOC* is the number of lines of codes. *#Faulty Versions* is the number of faulty versions. *Average # Repair Tests* is the average number of repair tests for each faulty version.

Program	LOC	# Faulty Versions	Average # Repair Tests
tcas	173	4	6.4
replace	564	5	19
schedule2	374	2	9
tot_info	406	4	8
space	6195	2	38.4
sed	12062	2	28
Total	19774	19	108.9

5.2 Study Data

5.2.1 Subject Programs

To evaluate our approach, we used six C programs from the Software artifacts Infrastructure Repository (SIR) [2] along with a comprehensive set of test inputs. We used the Siemens Suites: tcas, replace, schedule2, and tot_info. We also used two larger programs: space and sed. We found different versions of each program with the fault of interest.

Even though many benchmarks are available in the SIR, we excluded faulty versions which have no tests to execute the fault. We excluded other faulty versions with faults other than operator faults (print-tokens, print-tokens2, and schedule) since the ability to fix faults depends on the set of PMOs that are supported by the approach. For example we excluded print-tokens from our study because the seeded fault can be fixed by inserting or deleting a statement and it cannot be fixed by our PMOs. We also excluded grep and gzip because they could not be compiled within our framework.

Subject programs have sizes ranging from 173 to 12K lines of code; each program is seeded with a single fault. We used multiple faulty versions for each subject program. Faulty versions are taken from the SIR. We also used Proteum/IM 2.0 [20], which is a C mutation tool, to create additional faulty versions that are seeded with single operator faults. We seeded subject programs with only single operator faults to guarantee finding a potential repair w.r.t to the selected PMOs if the FL technique localizes a fault. This approach controls for other factors that can impact the effectiveness of MUT-APR.

Our study includes nineteen faulty versions with a total of 19,774 lines of code are used in our evaluation. Table 6 identifies the subject programs along with their size, the number of faulty versions (after removing equivalent mutants), and the average size of repair tests.

5.2.2 Repair Tests

One of the inputs to an APR tool is a set of repair tests. We selected a set of repair tests that have at least one failing test, and one passing test. Failing tests execute faults, and passing tests protect program functionality. Repair tests for the Siemens Suites are taken from the SIR repository. Test suites for the large programs provided by the SIR contain too many test inputs, which will slow the APR process, since repair tests are used to validate each generated variant. We created repair tests for each large program containing at least one failing test and 20 passing tests following Qi et al.[35]. In addition, a study by Abreu et al. [5] found that fault localization techniques give a stable behavior when no less than 20 test inputs are used. To validate our results, we repeated the study for each faulty version using five different repair tests that are selected/created randomly using test data provided by the SIR except for two versions of tcas program (tcas-v5 and tcas-25), which used three and one repair tests, respectively, since there are no other test suites that execute faults. We only used five different test suites due to the unavailability of different branch repair tests that include at least one passing and one failing tests for each faulty version in the SIR repository. The average number of repair tests can be found in the last column of Table 6.

5.3 Evaluation Design

For each faulty version, we used each FL technique to create an LPFS. Then the list is used by MUT-APR to find potential repairs. We executed each FL technique five times on each faulty version with five different repair tests except for tcas. On one version of tcas we used three different repair tests, and on the other version of tcas we used one repair test. We excluded the two versions of the replace program in which the actual faulty statements were not identified by all FL techniques when analyzing the results for RQ2, RQ3, and RQ4 (only 17 faulty versions were used). In total, our evaluation includes 79 trials (faulty versions $\times$ number of test suites for each faulty version). We compute the average and median values of our measurements for each faulty version across the multiple test suites, and the average value across all trials. We excluded the trials in which at least one FL technique did not identify actual faulty statement. To measure PFT, we exclude trials in which a potential repair passes all regression tests. In other words, we excluded trials with $PFR = 0$.

We compared the distributions of the measurements for the raw data and the transformed data using the *sqrt* function to make the data more normal. Then, to measure the statistical difference we applied Mixed Model [13], which is preferable over ANOVA, to study the effect of many measurements using the same subject program. We studied the difference at the 0.95 confidence level. Our experiments were conducted on a fedora Linux machine with an Intel Xeon R 2.67GHz CPU and 7.7 GB memory size. MUT-APR source code and experiments data can be found at http://fyassiri.wix.com/mutapr

6 Results

6.1 RQ1: Relative APR Effectiveness

To compare APR effectiveness when different FL techniques are used, we studied the ability of FL techniques to identify actual faulty statements. If an FL technique fails to identity the actual faulty statement, APR will fail to repair the fault. All ten FL techniques successfully identified actual faulty statements for all faulty versions except for two faulty versions of replace (version 25 and version 31) in which all FL techniques failed to identify faulty statements.

We checked these two versions of *replace* to investigate why FL techniques failed to identify actual faulty statements. In both versions, faults were in an *if* statement nested inside a *switch* statement. When the program is executed against the failing tests, the *if* statement is checked and it returns false. Thus, the execution jumps to the *default* statement, which caused failures. However, the faulty *if* statements were not recorded as one of the executed statements (when the if statement evaluated to false, the coverage tool did not add it to the list of executed statements). This problem is due to the coverage code that creates the instrumented version of faulty programs which records coverage information. The coverage code is adapted from GenProg, and modifying the code is left for the future.

6.2 RQ2: Which FL Technique Gave Actual Faulty Statements the highest priority (lowest LPFS rank) for Repair?

We evaluated the accuracy of FL techniques in identifying actual faulty statements by comparing the LPFS rank of the actual faulty statement produced by each FL technique. We compared the LPFS rank of faulty statements using seventeen faulty versions (79 trials).

First, we compared the LPFS rank of actual faulty statements across all of the trials. We found that in one out of 79 (1.27%) of the trials Ochiai2 and

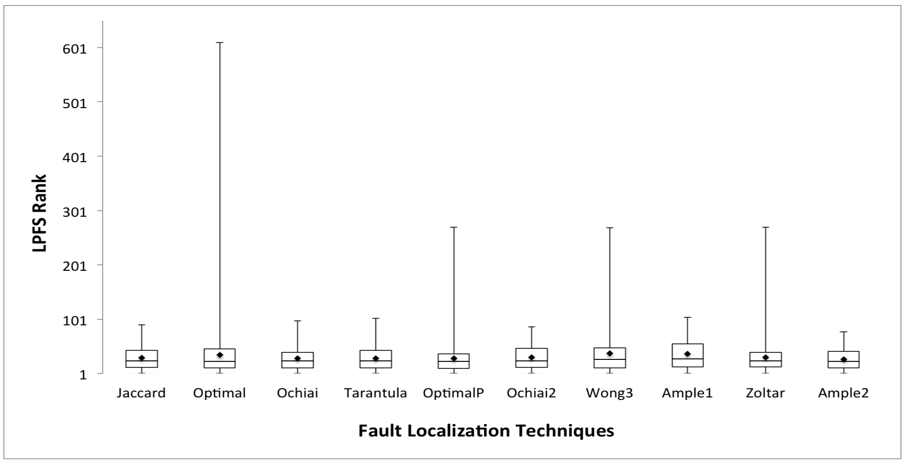
Table 7: LPFS Rank, NGV, and Total Time of different FL techniques.

	LPFS Rank			NGV			Total time		
FL	Mean	Median	Std	Mean	Median	Std	Mean	Median	Std
Jaccard	29.26	24.00	22.59	38.34	26.50	37.09	43.28	10.33	78.03
Optimal	35.72	23.50	70.18	45.47	28.00	78.29	40.38	10.28	110.6
Ochiai	28.87	24.00	22.84	38.07	26.50	37.77	42.61	10.20	75.81
Tarantula	28.75	24.00	23.12	38.27	26.50	38.49	42.74	10.35	75.82
OptimalP	28.59	23.00	34.35	40.83	25.50	61.05	46.64	10.31	90.06
Ochiai2	30.38	24.50	22.71	40.21	33.00	36.52	42.22	11.61	74.41
Wong3	38.14	27.00	50.13	48.21	29.50	78.35	50.82	10.42	108.9
Ample1	36.43	28.00	27.69	49.59	36.00	44.05	71.205	15.19	117.1
Zoltar	30.82	24.00	35.07	41.75	26.50	60.87	46.45	10.23	89.55
Ample2	27.24	23.50	21.08	38.50	26.50	36.23	36.01	10.20	63.50

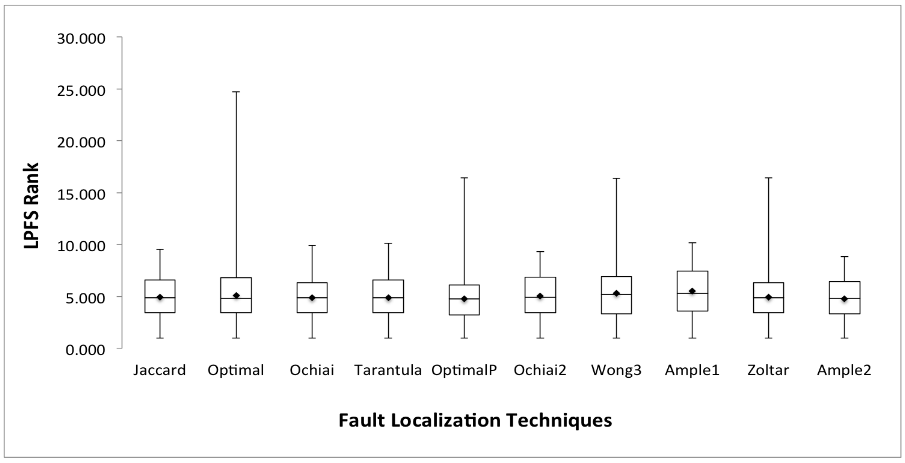
Ample2 assigned the lowest priority for repair (highest LPFS rank) to actual faulty statements, in two out of 79 (2.53%) of the trials Jaccard assigned the lowest priority for repair to actual faulty statements, and in four out of the 79 (5.06%) of the trials Optimal assigned the lowest priority for repair to actual faulty statements. On the other hand, Wong3 and Ample1 assigned a lowest priority for repair (highest LPFS rank) to actual faulty statements in 21 and 26 trials out of the 79 trials, respectively. In other words, Wongs3 assigned lowest priority for repair to actual faulty statements in 26.59% of the trials, and Ample1 assigned lowest priority for repair to actual faulty statements in 32.91% of the trials. However, Wong3 and Ample2 assigned acutal faulty statements the highest priority for repair in on trial, Ample1 assigned actual faulty statements the highest priority for repair in two trials, and Optimal assigned actual faulty statement the highest priority for repair in four trials.

We compared the average LPFS rank for all FL techniques (Table 7). On average, Ample2 assigned the highest priority for repair (an average LPFS rank of 27.24) to actual faulty statements, followed by Ochiai and Tarantula which assigned an average of 28.87 and 28.75 LPFS ranks to actual faulty statements. Wong3 assigned the lowest priority for repair to actual faulty statements (an average LPFS rank of 38.14), followed by Ample1 (an average LPFS rank of 36.43). Then, we compared the medians. We found that Ample1 assigned the lowest priority to actual faulty statements compared to all other FL techniques (28 LPFS rank).

Figure 3a shows the distributions of raw LPFS rank data, and Figure 3b shows the distributions of transformed LPFS rank data. Boxes represent the distribution of the LPFS data when each FL technique was used, the middle line represents medians, and the dots represent mean values. The top whisker represents the maximum value, and bottom whisker represents the minimum value. Box plots show that most of the FL techniques have similar distributions with slight differences, except Wong3 and Ample1 in which most LPFS ranks data are distributed at higher values (lowest priority for repair). In addition, Optimal assigned very high LPFS ranks to some faulty statements, which is shown by the top whisker.



(a) Raw LPFS Rank Data



(b) Transformed LPFS Rank Data

Fig. 3: LPFS rank for faulty statements for each FL technique. Lower LPFS rank is better.

We studied the statistical difference between Wong3, Ample1 and all other FL techniques, and we found that the different is significant between Ample1 and Ample2 (p-value = 0.04) and between Wong3 and Ample2 (p-value = 0.01), Jaccard (p-value = 0.03), Ochiai (p-value = 0.03), OptimalP (p-value = 0.03), and Tarantula (p-value = 0.03) at the 0.95 confidence level.

One thing to notice here is the large average and standard deviation values, as shown in Table 7. These values relate to the size of programs used in our evaluations. The data set includes large programs that required checking more than one thousand statements before reaching faulty statements, thus a very high LPFS rank was assigned to actual faulty statements which skew the average and the standard deviation values of LPFS rank, NGV, and TotalTime.

In summary, all FL techniques assigned the same LPFS ranks to actual faulty statements except Wong3 and Ample1 which assigned the lowest priority for repair (highest LPFS rank) to actual faulty statement in 26.59% and 32.91% of trials, respectively. The difference is statistically significant between Ample1 and Ample2, and between Wong3 and Ample2, Jaccard, Ochiai, OptimalP, and Tarantula at the 0.95 confidence level.

6.3 RQ3: Number of Generated Variants (NGV)

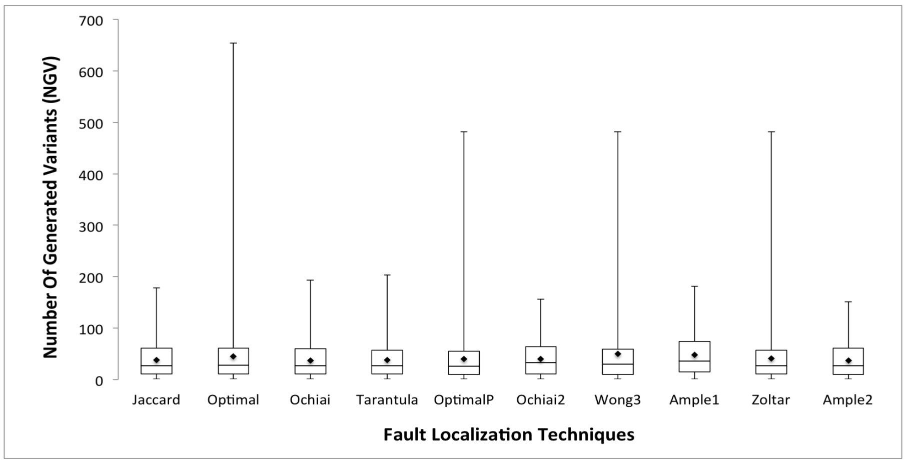
We compared the number of generated variants (NGV) until a potential repair is found to determine the most effective FL technique for use in APR. An FL technique that gives lower NGV is better since it decreases the chances of producing invalid variants before finding the correct one. We compared the NGV for all FL techniques across all trials; in one out of 79 (1.27%) of the trials Ochiai, Ochiai2, and Ample2 generated the largest number of variants until producing potential repairs compared to other FL techniques, in four out of 79 (5.06%) of the trials Optimal has the largest NGV, in six trials out of 79 (7.59%) of the trials Wong3 produced the largest NGV until a potential repair is found, and in 23 trials out of 79 (29.11%) of the trials Ample1 generated the largest number of variants until a potential repair is found.

Table 7 shows the average and median NGV for each FL techniques across all trials. Ample1 generated an average of 49.59 variants until a potential repair is found, which is higher than all other FL techniques. In addition, the median value for Ample1 is the highest (36 variants) compared to all other FL techniques. Figures 4a and 4b show the distributions of the raw NGV data and transformed NGV data for each FL technique. NGV are distributed at higher values with slightly higher median and mean values when Ample1 was used compared to all alternatives. The top whisker shows the maximum value of NGV, and Optimal have the highest maximum value compared to all other FL techniques. The difference is significant between Ample1 and Ample2, Jaccard, Ochiai, Ochiai2, and Tarantula at the 0.95 confidence level (p-values between 0.01 and 0.05)

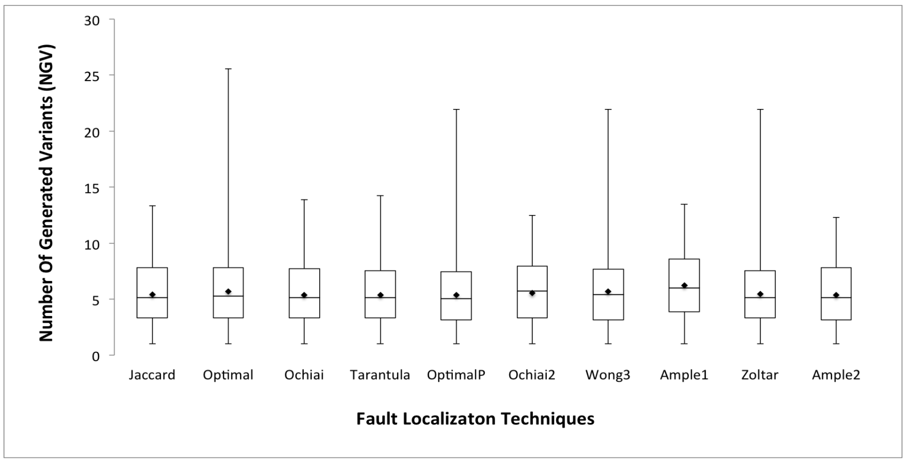
In summary, Ample1 reduced APR performance since it generates a large number of variants in 29.11% of the trials, and the difference between Ample1 and most of alternative FL techniques is significant.

6.4 RQ4: Total Time

We compared the total time required until a potential repair is found when each FL technique is used to generate the LPFS. First, we compared FL techniques across all trials (79 trials). Jaccard, Ochiai, OptimalP, Zoltar, and Ample2 took the longest time to produce potential repairs for one trial out of 79 (1.27%



(a) Raw NGV Data

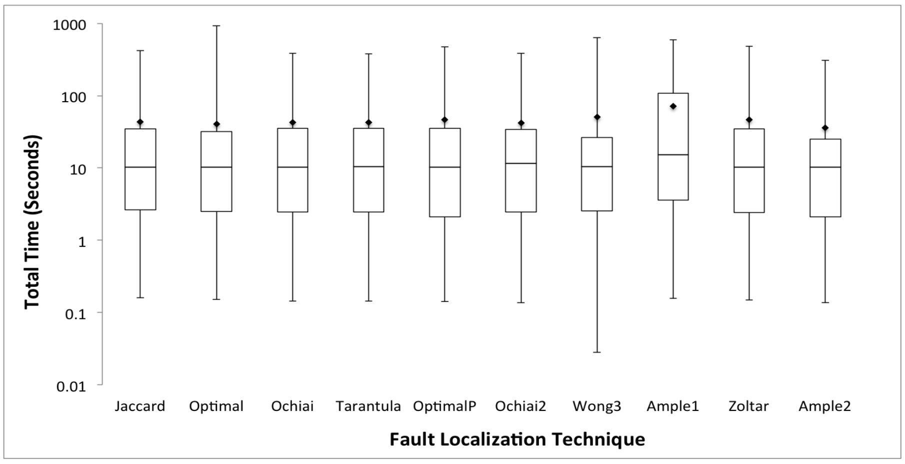


(b) Transformed NGV Data

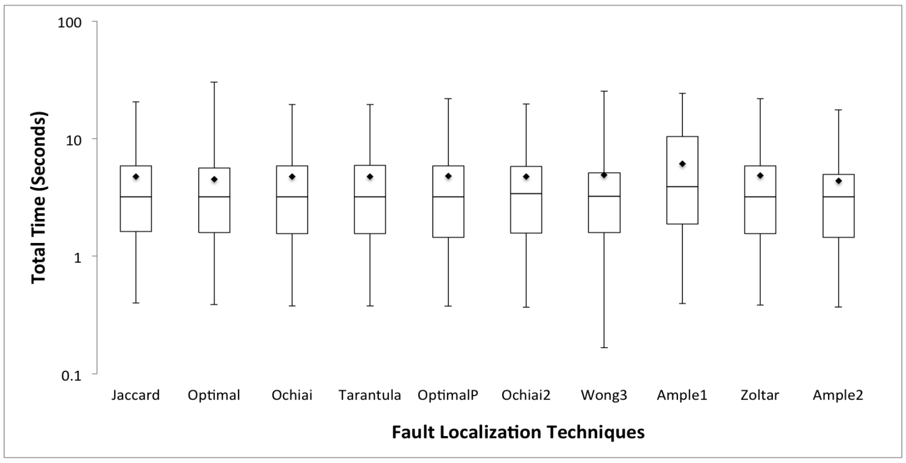
Fig. 4: NGV required to find potential repairs for each FL technique. Lower NGV is better.

of the trials), Ochiai2 took the longest time in four out of 79 (5.06%) of the trials, Wong3 and Ample1 took the longest time for 13 and 26 trials out of 79 (16.46% and 32.9%) of the trials; respectively.

Then, we compared the average and the median total time for each FL technique. Table 7 shows that Ample1 required the longest average and median time (71.79 seconds and 15.19 seconds, respectively) to repair faults, followed by Wong3 which required an average of 50.87 seconds. All other FL techniques required similar times. Figure 5a shows the distribution of the raw total time data required to repair faults and Figure 5b shows the distributions of the transformed total time data when each FL technique was used to identify



(a) Raw Total Time Data



(b) Transformed Total Time Data

Fig. 5: Total time required to find potential repairs for each FL technique. Lower NGV is better.

actual faulty statements. Ample1 took the longest time to produce potential repairs for most of the trials as shown by the box plots, with the largest median and mean values. Wong3 has a slightly higher mean value; however, its median value is similar to the other FL techniques. We found that the difference is statistically significant between Ample1 and all alternative FL techniques, except Wong3, at the 0.95 confidence level (p-values between 0.00 and 0.05) .

To conclude, Ample1 decreases APR performance since it required the longest total time (an average of 71.02 seconds) to produce potential repairs

Table 8: Repair correctness of different FL techniques. *PFR* is the percentage of failing potential repairs, and *PFT* is the percent of failing regression tests for each potential repair.

FL Technique	PFR (%)	Validated Repairs (%)	Average PFT (%)	Median PFT (%)
Jaccard	29.11	70.88	2.57	0.09
Optimal	34.18	65.82	9.68	0.09
Ochiai	32.91	67.08	2.65	0.09
Tarantula	32.91	67.08	2.55	0.09
OptimalP	31.65	68.35	2.65	0.09
Ochiai2	32.91	67.08	2.65	0.09
Wong3	31.65	68.35	2.48	0.09
Ample1	32.91	67.08	2.62	0.09
Zoltar	32.91	67.08	2.65	0.09
Ample2	31.65	68.35	2.61	0.09

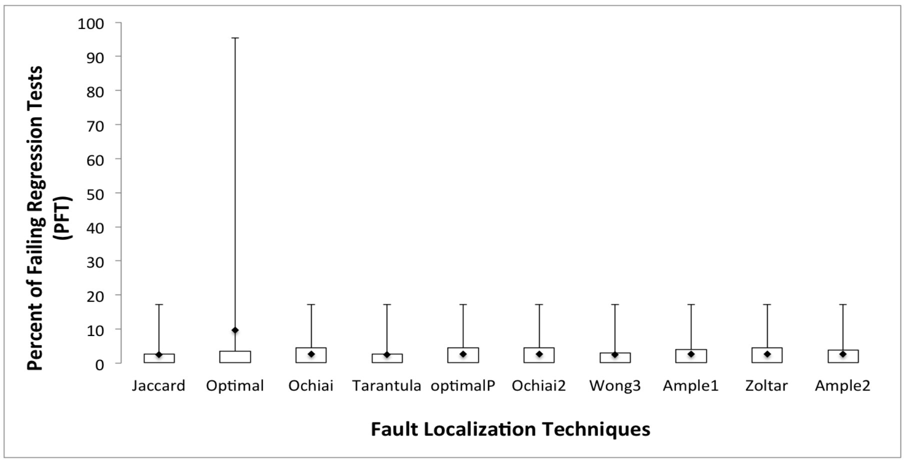
compared to all alternative FL techniques. The difference between all FL alternatives, except Wong3, is significant at the 0.95 confidence level.

6.5 RQ4: Repair Correctness

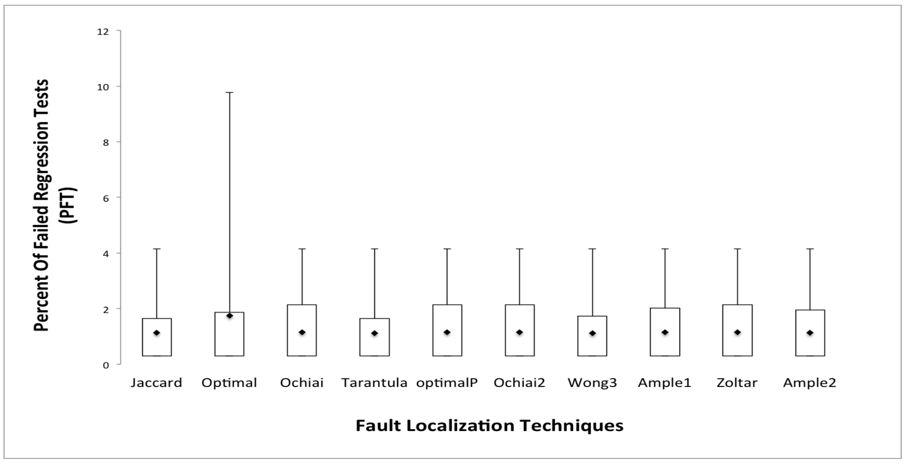
To measure the impact of different FL technique on repair correctness, we measured the percent of failing potential repairs (PFR) when each FL technique is used. Jaccard produced more validated repairs (70.88% of potential repairs are validated) compared to all alternative FL techniques, which produced between 68.35% and 67.08% validated repairs (Table 8).

Then, we compared the PFT for all trials with failing potential repair; we excluded trials with potential repairs that pass all regression tests ($PFR = 0$). Thus 25 trials are included for *replace*, *tcas*, and *sed* programs. The results showed that PFT is similar across all trials for all FL techniques except Jaccard, which produced potential repairs that failed more regression tests compared to all alternative FL techniques in one trial of *sed* program, and Optimal which generated repair that failed more regression tests compared to all FL techniques in three trials of *sed* program.

We compared the mean and the median of PFT. We found that Optimal reduced repair corretness since it generated potential repairs that failed an average of 9.68% of regression tests. All other FL techniques generated potential repairs that failed an average of 2.48% to 2.65% of regression tests. However, the median value of PFT is similar for all FL techniques. Figure 6a and 6b show the distributions for raw PFT data and transformed PFT data, respectively. Optimal has the largest PFT values as shown by the top whisker, and the largest mean but similar median compared to alternatives. We analyzed the statistical difference using the Mixed Model, we we found the difference between Optimal and all alternative FL techniques is statistically significant (p-values between 0.02 and 0.03) at the 0.95 confidence level.



(a) Raw PFT Data



(b) Transformed PFT Data

Fig. 6: Total time required to find potential repairs for each FL technique. Lower NGV is better.

In summary, Jaccard produced a higher proportion of correct potential repairs with APR since it generated more validated repairs (validate repairs = 70.88%) and generated potential repairs failed few regression tests ($PFT = 2.57\%$). On the other hand, Optimal produced the lowest proportion of correct potential repairs since it generated potential repairs that failed an average of 9.68% of regression tests compared to all alternative FL techniques.

Table 9: Correlation results between performance metrics: LPFS Rank, Number of Generated Variants (NGV), and Time of different FL Techniques.

FL Technique	LPFS Rank - NGV	LPFS Rank - Total Time	NGV - Total time
Jaccard	0.78	0.13	0.19
Optimal	0.96	0.91	0.91
Ochiai	0.78	0.15	0.22
Tarantula	0.79	0.17	0.24
OptimalP	0.95	0.53	0.54
Ochiai2	0.78	0.11	0.17
Wong3	0.95	0.75	0.74
Ample1	0.86	0.08	0.09
Zoltar	0.91	0.49	0.54
Ample2	0.87	0.23	0.26

7 Discussion

7.1 Correlation between Performance Metrics

We evaluated the impact of different FL localization techniques on the effectiveness, performance, and repair correctness of APR techniques. We compared APR performance when different FL techniques are used in terms of three metrics: LPFS rank, NGV, and time. We studied the relationship between these metrics. There is a high correlation between the LPFS rank of actual faulty statements and the NGV until a potential repair is found (Table 9). FL techniques that assigned lower LPFS ranks to actual faulty statements decreased the NGV until a potential repair is found. Decreasing the NGV improved APR performance in terms of time.

The correlation between time and other two metrics: LPFS rank and NGV is weak. This might be due to the influence of other factors that affect the time to find a potential repair such as the number of repair tests, and the compilation and execution time of variants. In addition, time can be affected by the resources used to run the experiments.

7.2 Fixing Multiple faults or single faults on multiple lines

Fixing multiple faults or single faults that are spread across multiple lines requires applying more than one PMOs for each variant. Our current APR approach cannot fix multiple faults, but it can be extended to target these type of faults. Fixing multiple faults can be implemented by applying all possible combinations of single PMO in which each combination fixes a single fault as proposed by Debroy and Wong [19]. However, this approach can be infeasible since the number of combinations can increase exponentially as the number of PMOs increase. Another approach to fixing multiple faults is to apply PMOs with a search algorithm that runs for multiple iterations, or a search algorithm that applies a crossover operator to combine the changes from two variants into one. In addition, properties of repair tests suites need to be investigated when fixing multiple faults. For example, repair test suites must contain at least one failing test that executes each fault, and the passing and failing tests should not overlap. We demonstrated this idea using the tcas subject program. In a preliminary study, we injected two operator faults, and we created repair test suites following the provided criteria. MUT-APR successfully fixed both faults. More studies are needed to fix multiple faults with APR techniques.

8 Threats to Validity

Internal Validity: Our evaluation studied the impact of different FL techniques on the performance, effectiveness, and repair correctness of automated program repair. Qi et al. [35] evaluated APR effectiveness and performance using different FL techniques with GenProg. We argue that the randomness of the genetic algorithm used by GenProg might affect the accuracy of the reported results. Even if an FL technique accurately locates an actual faulty statement, a search algorithm can randomly select PMOs that do not fix the fault. In this case, faults are not fixed due to the randomness of the search algorithm, not the fault localization technique. To mitigate this threat to internal validity, we applied a brute-force APR to remove the dependency between FL technique accuracy and the randomness of search algorithms.

External Validity: This relates to the ability to generalize the results. MUT-APR fixed single operator faults that are related to the supported PMOs. In addition, MUT-APR is limited to fixing faults requiring a one line modification, and to repair faults in relational operators, arithmetic operators, shift operators, and bitwise operators in different statement types. Therefore, our results might not generalize to other fault types, programs from other domains, or programs with real-world faults. To mitigate external threats, our evaluation consists of programs of different sizes including two large C programs (more than 12K LOC). However, the results of our experiments are impacted by the length of the LPFS and the position of the actual faulty statements in the LPFS, not the program size. Thus, the use of mostly small programs does not affect the accuracy of our results. Jaccard was found to be the best FL with GenProg [39] and MUT-APR [11], However, these results might not generalize to other APR techniques. Thus, the experiment needs to be replicated using other APR techniques.

Construct Validity: The accuracy of FL techniques depends in part on the test inputs used to identify actual faulty statements, which is a threat to construct validity. To mitigate this threat we used five different sets of repair tests (test inputs). For the small programs, we selected test suites randomly from the set of suites provided by the SIR. For the large programs, we created independent repair tests for each faulty version with no less than 20 passing test cases to achieve the best accuracy as reported by Abreu et al. [5].

In addition, properties of repair tests determine the correctness of generated potential repairs [11]. We could not mitigate this threat due to the unavailability of five branch coverage test suites that execute the faults for each faulty version in the SIR repository. In addition, the accuracy of PFR and PFR depends on the quality of regression tests. We used regression tests that satisfy branch coverage; however, different regression tests might produce different results. Thus, PFR and PFT are only estimates and do not directly measure the number of failures caused by the introduction of new faults. A single fault can cause multiple tests to fail.

Total time was used to measure APR performance, but time can be affected by other factors such as the number of repair tests, and the execution time for individual variants. Therefore, total time is another threat to construct validity. To mitigate this threat we ran the experiments using the same machine.

Conclusion validity: This relates to the statistical significant of the results. We applied statistical tests to measure the consistency of the results across all faulty versions, and to measure the statistical difference. Program size variations can be a threat to conclusion validity. To mitigate this threat we transformed our data using the *sqrt function* to make the data more normal. To limit other threats to conclusion validity, we ensured randomness in the experimental setting when selecting/creating tests inputs, and we used the same trials (combination of faulty versions and repair tests) with all FL techniques.

9 Related Work

Fault localization tools are introduced to decrease the cost of finding faults and improve software quality. These tools are based on spectrum-based fault localization (SBFL) techniques to identify actual faulty statements in source code. The Tarantula tool [23] was developed to locate faults in C programs, and AMPLE [16] is an Eclipse plug-in for object-oriented software. Ochiai is used in the molecular biology domain [31], and Jaccard is used in the Pinpoint tool [14].

FL techniques have been evaluated in terms of their accuracy to locate faults. Abreu et al. [4] evaluated the effectiveness of four FL techniques (Pinpoint, Ochiai, Tarantula and AMPLE) in terms of the LPFS rank of the actual faulty statement. Experiments conducted on the Siemens Suites and space program showed that the Ochiai FL technique is the most effective of those evaluated. Abreu et al. [5,3] studied the impact of the quality and the quantity of passing and failing tests on the accuracy of FL techniques. They found that Ochiai, Jaccard, and Tarantula provide accurate results with low quality tests that include only 1% of failing tests that propagate faults to the outputs. They studied the impact of the number of passing and failing tests on the accuracy of FL techniques, and found that adding more failing tests will always improve the accuracy of FL techniques. Adding more than 6 failing tests has a minimal affect but it does not lower the accuracy of fault diagnosis. On the other hand, adding more than 20 passing tests decreased the accuracy of FL techniques. Lee et al. [29] and Naish et al. [32] conducted a more comprehensive study to compare the accuracy of the formulas used to locate faults. Naish et al. also proposed two *Optimal* metrics to locate faults, which outperform other metrics. The latest study by Xie et al. [40] performed a theoretical analysis to evaluate FL techniques, and found that the *Optimal* metrics by Naish et al. have similar behaviors.

Lately, research has been directed toward automated program repair (APR) techniques to reduce debugging costs. GenProg is a well known APR technique developed by Weimer and his colleagues [21,39,38,27]. It uses a genetic programming algorithm to fix faults automatically in C programs. GenProg can fix a variety of faults including segmentation faults and infinite loops. Arcuri [8, 9] proposed an approach and tool, called JAFF, that use genetic programming for automatic bug fixing for Java programs. Ackling et al. [6] developed *py*EDB tool to automate repairs of Python software. SemFix [33] is a tool for fixing faults through semantic analysis; Debroy and Wong [18,19] applied a brute-force search method to repair faults using first order progam modification operators. Wei et al. [37] developed a tool, called *AutoFix-E*, to automate fault fixing in Eiffel programs equipped with contracts. Kim et al. [26] describe the Pattern-based Automatic program Repair tool (*PAR*), which repairs faults by generating patches using fix patterns. Ten patterns are created based on patches commonly written by humans.

APR techniques must locate faults in order to fix them. Existing FL techniques are employed by APR tools to locate and fix faults automatically. Nguyen et al. [33] use Tarantula heuristics to identify actual faulty statements. They compared the impact of Ochiai on SemFix, and found that Ochiai only fixed two more faulty versions of the *tcas* program. Debroy and Wong [18,19] compared APR efficiency using Tarantula and Ochiai heuristics, and found that Ochiai is a better FL technique that allowed more faults to be fixed with fewer PMOs. However, they used a large number of test inputs to evaluate the effectiveness of FL heuristics; using a large number of test inputs might affect the accuracy of their results since Abreu et al. [5,3] found that using more than 20 passing tests has a negative impact on FL techniques.

GenProg uses a basic Weighting Scheme to locate faults, and some APR techniques [11,26] apply the Weighting Scheme following the GenProg approach. Only one prior study evaluates the effectiveness of different FL techniques employed by APR techniques [35]. Qi et al. found that APR had the best performance when applying the Jaccard heuristic. They investigated why Jaccard produced the best results, and found that Jaccard assigned very low scores (high LPFS rank) to non-faulty statements and very high scores (low LPFS rank) to actual faulty statements, which decreased the chances of producing invalid potential repairs, even though some other FL techniques assigned higher scores (lower LPFS rank) to actual faulty statements. However, this work evaluates the effectiveness of FL techniques using the GenProg random search algorithm, which can affect the accuracy of the results.

Our results show that all studied fault localization techniques have similar impact on APR, except Jaccard, which improved repair correctness compare to alternatives. The FL technique is not the only factor that can impact APR effectiveness, performance, and repair correctness. To ensure the accuracy of our results, we control all other factors, such as repair test suites and PMOs that are used in the repair process. In addition, we used an exhaustive search to remove the dependency between the search algorithms and fault localization techniques.

10 Conclusion and Future work

Fault localization (FL) techniques are employed by APR tools to reduce the number of potentially faulty statements to be modified in order to find a potential repair. FL techniques that identify faulty statements will improve APR effectiveness. An FL technique that places actual faulty statements at the head of the list of potentially faulty statements (LPFS), will improve APR performance and repair correctness since fewer statements will be modified until a potential repair is found, thus decreasing the time required to fix faults and preventing unwanted modification to non-faulty statements.

Our evaluation shows that all ten FL localization techniques identified actual faulty statements. Wong3 and Ample1 assigned the lowest priority for repair (highest LPFS rank) to actual faulty statements in 26.59% and 32.91% of the trials, and Ample1 generated the largest number of variants (49.59 variants) until producing potential repairs in 29% of the trials. In addition, Ample1 took the longest time to generate a repair (an average of 71.02 second). Ample1 decreased APR performance because it assigned the lowest priority for repair to actual faulty statements, which increased the NGV and total time required to find potential repairs. On the other hand, Jaccard produced more correct repairs, and generated potential repairs that failed fewer regression tests.

Our results contribute towards improving effectiveness, performance, and repair correctness of APR techniques. We provide a framework for evaluating alternative FL techniques along with an evaluation of these techniques. We plan to combine the results from this study to evaluate the impact of different search algorithms to improve the performance of APR techniques without decreasing their effectiveness and repair correctness.

Acknowledgements This project was supported by the Ministry of Higher Education, Saudi Arabia. The authors would like to thank Westley Weimer and his research group for sharing GenProg, and Yuhua Qi et al. [35] for sharing the GenProg-FL tool.

References

1. National vulnerability database. `http://nvd.nist.org`

2. Software-artifact infrastructure repository. `http://sir.unl.edu/php/previewfiles.php`
3. Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. Journal of Systems and Software **82**(11), 1780–1792 (2009)
4. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: An evaluation of similarity coefficients for software fault localization. In: Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on, pp. 39–46. IEEE (2006)
5. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, pp. 89–98. IEEE (2007)
6. Ackling, T., Alexander, B., Grunert, I.: Evolving patches for software repair. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation, GECCO '11, pp. 1427–1434. ACM, New York, NY, USA (2011)
7. Agrawal, H., Horgan, J.R., London, S., Wong, W.E.: Fault localization using execution slices and dataflow tests. In: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on, pp. 143–151. IEEE (1995)
8. Arcuri, A.: On the automation of fixing software bugs. In: Companion of the 30th international conference on Software engineering, ICSE Companion '08, pp. 1003–1006. ACM, New York, NY, USA (2008)
9. Arcuri, A.: Evolutionary repair of faulty software. Applied Soft Computing **11**(4), 3494 – 3514 (2011)
10. Assiri, F.Y.: Assessment and improvement of automated program repair mechanisms and components. Ph.D. dissertation, Colorado State University (2015)
11. Assiri, F.Y., Bieman, J.M.: An assessment of the quality of automated program operator repair. In: Proceedings of the 2014 ICST Conference, ICST '14 (2014)
12. Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. Software Engineering, IEEE Transactions on **36**(4), 528–545 (2010)
13. Brown, H., Prescott, R.: Applied mixed models in medicine. John Wiley & Sons (2006)
14. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: Problem determination in large, dynamic Internet services. In: Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pp. 595–604. IEEE (2002)
15. Chilimbi, T.M., Liblit, B., Mehra, K., Nori, A.V., Vaswani, K.: Holmes: Effective statistical debugging via efficient path profiling. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pp. 34–44. IEEE (2009)
16. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight defect localization for Java. In: ECOOP 2005-Object-Oriented Programming, pp. 528–550. Springer (2005)
17. Dallmeier, V., Zimmermann, T.: Extraction of bug localization benchmarks from history. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 433–436. ACM (2007)
18. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pp. 65 –74 (2010)
19. Debroy, V., Wong, W.E.: Combining mutation and fault localization for automated program debugging. Journal of Systems and Software **90**, 45–60 (2014)
20. Delamaro, M.E., Maldonado, J.C.: Proteum/IM 2.0: An integrated mutation testing environment. In: Mutation testing for the new century, pp. 91–101. Kluwer Academic Publishers, Norwell, MA, USA (2001)
21. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, GECCO '09, pp. 947–954. ACM, New York, NY, USA (2009)
22. Hailpern, B., Santhanam, P.: Software debugging, testing, and verification. IBM Systems Journal **41**(1), 4–12 (2002)
23. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, pp. 273–282. ACM, New York, NY, USA (2005)

24. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on Software engineering, pp. 467–477. ACM (2002)
25. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization for fault localization. In: Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada, pp. 71–75. Citeseer (2001)
26. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 802–811. IEEE Press (2013)
27. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. Software Engineering, IEEE Transactions on **38**(1), 54 –72 (2012)
28. Le Goues, C., Weimer, W., Forrest, S.: Representations and operators for improving evolutionary software repair. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, GECCO '12, pp. 959–966. ACM, New York, NY, USA (2012)
29. Lee, H.J., Naish, L., Ramamohanarao, K.: Study of the relationship of bug consistency with respect to performance of spectra metrics. In: Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on, pp. 501–508 (2009)
30. Lo, D., Jiang, L., Budi, A.: Comprehensive evaluation of association measures for fault localization. In: Software Maintenance (ICSM), 2010 IEEE International Conference on, pp. 1–10. IEEE (2010)
31. Meyer, A.d.S., Garcia, A.A.F., Souza, A.P.d., Souza Jr, C.L.d.: Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). Genetics and Molecular Biology **27**(1), 83–91 (2004)
32. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3), 11:1–11:32 (2011)
33. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 772–781. IEEE Press (2013)
34. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. Software Engineering, IEEE Transactions on **31**, 511–526 (2005)
35. Qi, Y., Mao, X., Lei, Y., Wang, C.: Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pp. 191–201. ACM, New York, NY, USA (2013)
36. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Automated Software Engineering, 2003. Proceedings of the 18th IEEE International Conference on, pp. 30–39. IEEE (2003)
37. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: Proceedings of the 19th international symposium on Software testing and analysis, ISSTA '10, pp. 61–72. ACM, New York, NY, USA (2010)
38. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010)
39. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 364–374. IEEE Computer Society, Washington, DC, USA (2009)
40. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. **22**(4), 31:1–31:40 (2013)