Which Generated Test Failures Are Fault Revealing? Prioritizing Failures Based on Inferred Precondition Violations using PAF∗

Mijung Kim
The Hong Kong University of Science and Technology
Hong Kong, China
mjkimab@cse.ust.hk

Shing-Chi Cheung
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Sunghun Kim
The Hong Kong University of Science and Technology
Hong Kong, China
hunkim@cse.ust.hk

ABSTRACT
Automated unit testing tools, such as Randoop, have been developed to produce failing tests as means of finding faults. However, these tools often produce false alarms, so are not widely used in practice. The main reason for a false alarm is that the generated failing test violates an implicit precondition of the method under test, such as a field should not be null at the entry of the method. This condition is not explicitly programmed or documented but implicitly assumed by developers. To address this limitation, we propose a technique called Paf to cluster generated test failures due to the same cause and reorder them based on their likelihood of violating an implicit precondition of the method under test. From various test executions, Paf observes their dataflows to the variables whose values are used when the program fails. Based on the dataflow similarity and where these values are originated, Paf clusters failures and determines their likelihood of being fault revealing. We integrated Paf into Randoop. Our empirical results on open-source projects show that Paf effectively clusters fault revealing tests arising from the same fault and successfully prioritizes the fault-revealing ones.

CCS CONCEPTS
• Software and its engineering → Software verification and validation; Software testing and debugging;

KEYWORDS
Automated Test Generation, Test Failure Prioritization, Fault-revealing Test Cases, Fault-inducing Data-flow Analysis

ACM Reference Format:

∗This work is supported by the Hong Kong RGC/GRF grant 16202917, MSRA collaborative research grant, and Nvidia academic program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION
To reduce the manual effort on writing unit tests, researchers have developed automated test generation tools based on various underlying techniques such as random testing [6, 40], search-based testing [14, 56], and dynamic symbolic execution [5, 20, 50].

These automated testing tools provide developers with a set of failing tests that can possibly detect uncovered faults. However, the failures from these tests are often false alarms that do not reveal real faults [16, 18, 23, 39, 51]. This greatly hinders the usefulness of unit test generation tools [2, 15, 16, 23].

The main reason for false alarm failures is that generated failing tests violate the implicit preconditions that developers would be aware of while coding [16, 18, 23, 39]. Rather than indicating the existence of a real fault, a generated test for the method under test (MUT) may fail just because it violates a method’s precondition on the variables whose values are used when it fails. We call these variables crash variables, denoted by cv. Judging the legitimacy of a generated failing test is challenging as these preconditions are not explicitly programmed or documented by developers.

We propose an automated technique called Paf to rank a generated failing test’s legitimacy in the situation of undocumented preconditions implicitly assumed by developers. It is inspired by the following observation.

The red solid arrow in Figure 1(a) shows a failing test exercising a dataflow that assigns a null value to the crash variable cv, causing NullPointerExceptions. As the null value is originated after the entry of MUT, this failure-inducing dataflow is considered local to the MUT computation. It indicates that the dataflow is wholly induced by the MUT’s implementation programmed by its developers. As a result, the chances of its violating an MUT’s precondition is
small, and the failing test is likely fault revealing. In another failure-inducing dataflow given by the red solid arrow in Figure 1(b), the value assigned to cv that causes NullPointerExceptions is originated before entering MUT. The dataflow is considered non-local to MUT computation. It indicates that a portion of the dataflow is not induced by the MUT’s implementation but by the generated test’s logic. There is a chance that this dataflow portion causes violation of some MUT’s preconditions that are implicitly assumed by its implementation.

After the locality of failure-inducing dataflows is computed, our idea is to cluster failures due to a common cause by dataflow similarity and estimate their likelihood of violating preconditions by further examining the dataflows of other related tests. The likelihood increases when more dataflows concerning the crash variable are found in other passing tests. Suppose the test exercising the non-local failure-inducing dataflow above (Figure 1(b)) crashes due to dereferencing the null value of cv at a statement s in Callee1. However, there are two other passing tests (dotted green arrows) with non-local dataflows resulting in the use of the same variable cv at s. Since these dataflows are non-local, they have exercised other def-use relations before entering MUT, causing cv to hold a non-null value at statement s. As such, the existence of these passing dataflows might suggest an implicit MUT preconditions that prevents cv from holding a null value at s. When more such passing dataflows are found, the null pointer exception of the failing test is more likely to occur because of an implicit precondition violation rather than a real fault.

There are existing test generation techniques that mitigate the implicit precondition issue while generating tests using simple heuristics such as exception types and method modifiers [6], dynamic invariants [7, 38], and temporal properties among method invocations [18]. However, the scope of what they consider implicit preconditions is limited to only certain types of exceptions [6], or pairs of method invocations [18]. Also, the performance of many of these techniques [7, 18, 38] heavily relies on the quality of manually-written tests. Therefore, generated failures by these tools may still suffer from precondition violations and produce false alarms.

An effective alternative is to analyze generated failures and prioritize their likelihood of precondition violations. Existing test prioritization techniques aim to execute a test suite for regression testing [10, 11, 25, 48] or mutation testing [34, 62, 63]. They do not consider prioritization of failures found in generated tests.

In this paper, we present a technique called Paf (Prioritization of Automatically-generated Failures) that clusters failing tests arising from the same cause by means of similar dataflows for crash variables. It also classifies likely fault-revealing tests and prioritizes failing tests based on their likelihood of violating the implicit preconditions of the crash variables. Par first analyzes the failing test executions of MUT and determines if they may violate a precondition implicitly assumed by the developers on crash variables. For the failing tests that are not subject to such violation, Paf classifies them as likely fault-revealing and reports them at the top of the list. Paf then sorts the tests in the classified list in a reverse order to their likelihood of violation. The likelihood is estimated by observing other dataflows reaching the same crash variable in the generated passing tests.

We implement Paf and evaluate it on 10 versions of five popular open-source projects. The results show that Paf effectively clusters fault revealing failing tests that share a common cause. The results also show that Paf accurately classifies fault revealing tests and places fault-revealing tests at higher priority. Paf outperforms the precondition-violation filtering component of existing test generation techniques (such as JCrasher and techniques using dynamic invariants). Among the 5,770 generated failures by Randoop for the subjects, Paf reported 24 fault revealing alarms. The failures clustered by these alarms have a precision of 78.8%. Among these alarms, four are new faults detected by Paf. They were confirmed and fixed by the developers. Paf makes the following contributions:

- A technique that analyzes the dataflows of crash variables, and thereby identifies if a failure can be induced by the synthesized logic of generated tests.
- A technique that groups failures (i.e., failing tests) based on the similarity of their failure-inducing dataflows and prioritizes them using the associated likelihood of violating implicit preconditions of crash variables.
- A prototype implementation that integrates Paf into Randoop, and an experiment of it on 10 versions of five popular open source projects. The experiment results show that Paf achieves high fault detection rate.

In the rest of this paper, we first illustrate our technique with a motivating example. Next, we explain the methodology of our technique and present the experiment. We then discuss the related work, conclusion and future work.

2 MOTIVATING EXAMPLE

Let us consider a motivating example in Figure 2. It shows an automatically generated test suite of two failing and two passing tests for class ProjectEntry. The source code is real world code from our subjects although they are slightly modified for the illustration purpose. The test cases are generated by a state-of-the-art test generation tool, Randoop. The failing tests throw NullPointerExceptions that are caused by dereferencing a null value of variables “e” and “project” at the crash statements, 47 and 59, respectively. As such, they are considered the crash variables of the failing tests.

To investigate whether a given MUT (e.g., index0f for fTest1 and handleInput for fTest2) prescribes a precondition on the crash variables, Paf finds where the null value used by each crash variable is originated. To do that, Paf locates the statement that creates the null value to be received by the crash variable. This statement is referred to the crash origin of the failure-inducing dataflow. Details of identifying the crash origin will be explained in Section 3.1.

Based on the location of an crash origin, Paf determines whether the concerned dataflow is local or non-local. Since the crash occurs during an MUT’s computation, the locality of a dataflow follows the crash point (i.e., crash statement) of its crash origin. Consider fTest1 in the motivating example. The null value received by variable e causing NullPointerExceptions is originated from entries.get(key) at Line 45 during the computation of index0f. So, Line 45 is the crash origin of the failure-inducing dataflow exercised by fTest1. Paf considers the failure-inducing dataflow, which starts at Line 45 and ends at Line 47, local to the MUT index0f’s call graph. In the case for fTest2, the null value received by variable project is originated
Which Generated Test Failures are Fault Revealing?

Four Generated Test Cases for ProjectEntry Class

Failing Tests
1. public void fTest1() {
   2. Project p = new Project();
   3. ProjectEntry pc = new ProjectEntry(p);
   4. pc.put((Object)10.0d, (Object)100.0d);
   5. int i = pc.indexOf("hi!"); // MUT
   6. }

7. }

8. public void fTest2() {
9. ProjectComponent pc = new ProjectComponent();
10. pc.setProject(new Project("hi"));
11. pc.handleInput(new byte[], (1)); // MUT
12. }

Passing Tests
13. public void pTest1() {
14. ProjectComponent pc = new ProjectComponent();
15. pc.setProject(new Project("hi"));
16. pc.handleInput(new byte[], (1));
17. }

18. }

19. public void pTest2() {
20. Project p = new Project();
21. ProjectComponent pc = new ProjectComponent(p);
22. pc.handleInput(new byte[], (5));
23. }

Source Code for ProjectEntry Class
31. private HashMap entries;
32. private Project project;
33. 
34. public ProjectEntry() {
35. }

36. public ProjectEntry(Project project) {
37. this.project = project;
38. entries = new HashMap(10);
39. }

40. public void put(Object key, Object value) {
41. Entry e = new Entry.get(key);
42. entries.put(key, e);
43. }

44. public int indexOf(Object key) {
45. Entry e = (Entry) entries.get(key);
46. int pos = 0;
47. while(e.prev != sentinel) { // NPE 1
48. pos++, e = e.prev;
49. return pos;
50. }
51. }

52. public void setProject(Project project) {
53. this.project = project;
54. }

55. public Project getProject() {
56. return project;
57. }

58. public int handleInput(byte[] buffer, int offset) {
59. return getProject().input(buffer, offset); // NPE 2
60. }

Figure 2: Example of source code and a set of automatically-generated tests.

Table 1 presents the summary of the collected data. In this exam-
ple, the def-use relation of assigning the null value to project at
Line 32 and the use of the project’s value at the crash statement 59
arises from the logic synthesized by the generated failing test. There
are chances that the synthesized logic is inapplicable to the assumed
usages of handleInput, i.e., violating its implicit preconditions.

After determining the locality of a failure-inducing dataflow, Paf
observes whether there are other related dataflows to the concerned
processes, i.e., violating its implicit preconditions.

As such, local failure-inducing dataflows have less chances of
precondition violation as compared with non-local failure-inducing
dataflows. Paf then reorders the failing tests in each group based on
the proportion of the def-use relations reaching the crash statement
exercised in passing tests. This is, the more such passing tests exist,
the likelihood of a precondition violation increases.

Table 1 presents the summary of the collected data. In this exam-
ple, fTest1 is more likely to be fault-revealing than fTest2 because
the failure-inducing dataflow is local. Although the partitioning
based on the locality is sufficient to reorder the failing tests in this
example, if we further investigate the likelihood, it would be $0/1 = 0$
for fTest1 and $1/1 = 1$ for fTest2. Thus, the failing tests are ordered
fTest1 → fTest2. In the real world, fTest1 reveals a real fault that
was reported in the bug repository and fixed. On the other hand,
fTest2 has been confirmed as a false alarm by the developer.

### Table 1: Motivating example for the failing tests in Figure 2.

<table>
<thead>
<tr>
<th>Test</th>
<th>Crash Origin</th>
<th>Locality of failure-inducing dataflows</th>
<th>Def-use reaching crash statement</th>
<th>Passing tests exercising def-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>fTest1</td>
<td>Line 45</td>
<td>local</td>
<td>(45, 47)</td>
<td>None</td>
</tr>
<tr>
<td>fTest2</td>
<td>Line 32</td>
<td>non-local</td>
<td>(56, 59)</td>
<td>pTest1 pTest2</td>
</tr>
</tbody>
</table>

3 OUR APPROACH

In this section, we present the details of Paf. Figure 3(a) shows the
four processing phases of Paf. It takes a target program, a set of
generated failing tests and a set of generated passing tests as inputs,
and outputs a list of reordered failing tests.

In Phase 1, Paf first identifies the crash variable for each fail-
ing test. It then finds the crash origin, a statement where the value
assigned to the crash variable is originated in the failing test.
This statement becomes the starting point of the failure-inducing
dataflow. In Phase 2, Paf groups the failing tests into different sets,
called test flow-sets, based on the similarity of their failure-inducing
dataflows, and then further partitions these sets (i.e., test flow-sets)
into two categories (local or non-local) based on the location of
their crash origins. In Phase 3, Paf determines the likelihood of pot-
tential precondition violations by examining the related dataflows
exercised by other passing tests. Finally, in Phase 4, Paf reorders
the classified groups of test flow-sets based on their locality and

3https://issues.apache.org/jira/browse/COMLECTIONS-28
4https://bz.apache.org/bugzilla/show_bug.cgi?id=49400
likelihood obtained in Phases 2 and 3. Figure 3(b) shows an example of a reordered list returned by Paf.

3.1 Phase 1: Find Crash Variable and Its Origin

In Phase 1, Paf investigates how an incorrect value propagates to a crash variable in each failing test.

Identify Crash Variable Paf identifies the crash variable cv that is used at the crash statement for each failing test. The information about the crash statement (e.g., file name, method name, and line number) is provided by the top frame of a stack trace. However, identifying cv from the top stack frame is not always straightforward because a statement can involve multiple variables.

To cope with this challenge, Paf uses dynamic analysis by instrumenting the program and monitoring all variables used at the top stack frame. Paf considers different types of exceptions to choose candidate variables to monitor because a crash variable plays different roles in different exceptions. Suppose that the statement of the top frame is var = array[i+j].foo(). When a test throws an Array-OutOfBoundsException, candidates to monitor are those variables used as an array index, which in this case is i+j only. Paf analysis is done on the intermediate representation of the program (e.g., Jimple for Soot [58]). Therefore, when more than one variable is involved in an array index like i+j, array[i+j] would be represented like r = i+j; array[r]. Thus, in our analysis, the crash variable candidate would be r. When a test throws a NullPointerException, however, the candidates are array and array[i+j]. Paf then runs the failing tests on the instrumented program to identify the exact crash variable, which is the one accessed right before the failure.

Note that Paf also supports multiple crash variables in a failing test. For example, if an IllegalArgumentException occurs under a branch condition involving multiple arguments, Paf identifies all the argument variables used in the branch condition as the crash variables. Paf performs the analysis for the rest of Phase 1 and Phase 2 for each crash variable individually.

Derive Crash Origin and Failure-Inducing Dataflow Paf derives a crash origin of the crash variable’s value (v) used at the crash statement in the failing test under consideration. The crash origin is a statement where the value v is assigned in the failing execution. This value is subsequently received by the crash variable, resulting in a program failure at the crash statement.

Given a failing execution and a crash variable, Paf identifies the crash origin by transitively tracing backward from the the crash statement the interprocedural def-use associations (DUA)\(^4\) [26] given the execution trace of a failing test.

Paf performs the backward tracing only along copy statements [1] that either copy a variable’s value to another variable (e.g., a=b) or assign a method’s return value to a variable (e.g., a=getB()). This is because the value assigned at the crash origin should reach the crash variable without being modified. As such, the backward tracing stops when it hits a non-copy statement. For example, consider a non-copy statement s in an intermediate representation, \(r = a+b\); where \(r\) is assigned with 0 and causes a DividedByZeroException. In this case, Paf stops the backward tracing at s and returns it as the crash origin rather than continuing tracing further for a and b.

The set of def-use associations collected during the backward tracing form a chain that describes a failure-inducing dataflow starting with the value v’s creation at the crash origin and ending with the use of v through the crash variable at the crash statement. The dataflow traverses a def-clear path of the crash variable from its assignment of value v to the use of its value at the crash statement.

Revisit fTest2 in Figure 2 for illustration. The backward tracing is done starting from the crash statement 59 until it hits Line 32 where project is initialized with a default value, null. The failure-inducing dataflow is formed with a chain of interprocedural DUs \{(32, 56, project), (56, 59, returnVar)\} where returnVar refers to a return variable that copies the return value of getProject(). Paf returns 32 as the crash origin, which is the definition statement of the first DU in the failure-inducing dataflow.

3.2 Phase 2: Group and Partition Failing Tests

In Phase 2, Paf groups failing tests in two steps. In the first step, failing tests sharing the same crash origin, crash variable and crash statement are clustered into the same set, which we call a test flow-set. The clustering is motivated by an observation that generated failing tests exercising close failure-inducing dataflows are mostly redundant to each other sharing the same failure cause. It helps alleviate developers’ debugging effort from the need to inspect all failing tests in the same flow-set.

In the second step, Paf partitions the failure-inducing dataflow of a flow-set into two categories based on its crash origin’s location (local or non-local). Note that all tests in the same test flow-set share the same failure-inducing dataflow. Paf considers a dataflow local if its crash origin is executed after the entry of MUT; otherwise non-local. Paf makes a special treatment of field initializers since they do not belong to any methods. As field initializers are executed

\(^{4}\text{A def-use association represents a data-flow relationship in a triple } (d, u, v) \text{ such that a variable } v \text{ which has been defined in Statement } d \text{ reaches and is used in Statement } u \text{ without being redefined along a control flow path that connects } d \text{ to } u.\)
Which Generated Test Failures are Fault Revealing?

spontaneously when a constructor is called, Pf bundles them with the constructor and considers their execution after the entry of the constructor. Note that the dataflow similarity and origin locality are orthogonal features. Therefore, the order of clustering and partitioning does not matter.

### 3.3 Phase 3: Calculate Violation Likelihood

After partitioning test flow-sets into local and non-local groups, Pf measures the likelihood of a potential precondition violation within each group. Intuitively, the likelihood of a failing test to violate preconditions increases when its crash statement can be reached by more passing tests. That is, a failing test is more likely a false alarm when its crash statement is reached 10 times by 10 different passing dataflows than when it is reached 100 times by the same passing dataflow. Therefore, counting just the number of passing tests reaching the crash statement alone might cause a bias towards or against specific dataflows.

To address this challenge, Pf calculates the likelihood according to the dataflow coverage by passing tests reaching the crash statement. Passing tests that cover different DUAs are considered distinct. Specifically, it works as follows. For each failing test flow-set, Pf extracts a set of DUAs with respect to the crash variable cc at the crash statement u. Each extracted DUA (d, u, cc) satisfies three criteria: (1) d refers to a statement where cc is defined, (2) u refers to the crash statement, and (3) there exists a def-clear path from d to u in the program’s control flow graph.

We denote the set of extracted DUAs as θ. Using the extracted DUAs for each failing test flow-set, Pf measures its likelihood based on the number of DUAs in θ covered by at least one passing test as follows:

\[
\text{Likelihood} = \frac{\# \text{DUAs in } \theta \text{ covered by passing tests}}{\# \text{DUAs in } \theta}
\]

Pf orders failing test flow-sets according to the likelihoods within their locality group. Suppose fTest1 and fTest2 in Figure 2 are two failing tests belonging to two different test flow-sets FS₁ and FS₂, respectively. FS₁ assumes the θ for fTest1, which is {45, 47, 4}. Since there is no passing test that covers this DUA, FS₁’s likelihood is 0. Note that all failing tests in the same test flow-set assume the same likelihood because the set of extracted DUAs (i.e., θ) would be identical for all tests in the same test flow-set. FS₂ assumes the θ for fTest2, which is {56, 59, returnVar}. This DUA is covered by two other passing tests pTest1 and pTest2. The likelihood of FS₂ is, therefore, 1/1 = 1.

### 3.4 Phase 4: Reorder Failing Tests

In Phase 4, Pf reorders the set of failing test flow-sets in each group (i.e., local and non-local) according to the computed likelihood. A failing flow-set with a lower likelihood is ranked before that with a higher one. Test flow-sets ended up with a tie in likelihood values are reordered among themselves in a random order.

After the reordering is done, the final output of a reordered list of test flow-sets.. We selected Randoop because of its popularity and adoption by the industry [39] and academia for automated test generation.

We integrated Pf into Randoop(v3.1.0) [40] so that Pf outputs the ordered list of test flow-sets.. We selected Randoop because of its popularity and adoption by the industry [39] and academia for automated test generation.

We implemented the Pf tool on top of the Soot framework [58]. Pf analyzes and instruments the bytecode of a program by utilizing the Jimple intermediate representation of Soot. To compute def-use associations (DUAs) and monitor def-use coverage, we used a fine-grained data-dependence analysis tool, DUA-Forensics [49] that provides standard interprocedural data-flow algorithms for object-oriented languages [26, 27, 41] on top of Soot.

Our implementation supports four types of runtime exceptions that are the most common types generated from Randoop. Those types include (1) NullPointerExceptions (NPEs), (2) IllegalArgumentExceptions, (3) IllegalStateExceptions, and (4) ArrayOutOfBoundExceptions. Pf treats an application specific exception thrown under a branch condition as an IllegalStatementException. As shown

---

### Table 2: Generated failures and their types from Randoop

<table>
<thead>
<tr>
<th>Subject</th>
<th>Total Fails</th>
<th>NPE</th>
<th>Assertion Errors</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>1274</td>
<td>93.1%</td>
<td>4.4%</td>
<td>2.5%</td>
</tr>
<tr>
<td>Collections</td>
<td>329</td>
<td>72.2%</td>
<td>24.5%</td>
<td>3.3%</td>
</tr>
<tr>
<td>Ivy</td>
<td>463</td>
<td>60.8%</td>
<td>9.7%</td>
<td>29.5%</td>
</tr>
<tr>
<td>Math</td>
<td>45</td>
<td>91.2%</td>
<td>8.8%</td>
<td>0%</td>
</tr>
<tr>
<td>Rhino</td>
<td>531</td>
<td>77.4%</td>
<td>18.8%</td>
<td>3.8%</td>
</tr>
<tr>
<td>Average</td>
<td>528.4</td>
<td>79.0%</td>
<td>13.2%</td>
<td>7.8%</td>
</tr>
</tbody>
</table>
Table 3: Subjects used in the empirical studies.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Label</th>
<th>LOC</th>
<th>Class under Test</th>
<th>Class w/ Failing Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>1.6.5</td>
<td>Ant1</td>
<td>86K</td>
<td>572</td>
</tr>
<tr>
<td></td>
<td>1.8.1</td>
<td>Ant2</td>
<td>102K</td>
<td>873</td>
</tr>
<tr>
<td>Collections</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>Coll1</td>
<td>7K</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>Coll2</td>
<td>11K</td>
<td>204</td>
</tr>
<tr>
<td>Ivy</td>
<td>2.2.0</td>
<td>Ivy1</td>
<td>50K</td>
<td>489</td>
</tr>
<tr>
<td></td>
<td>2.4.0</td>
<td>Ivy2</td>
<td>51K</td>
<td>495</td>
</tr>
<tr>
<td>Math</td>
<td>2.2</td>
<td>Math1</td>
<td>49K</td>
<td>784</td>
</tr>
<tr>
<td>Rhino</td>
<td>1.7.R2</td>
<td>Rhn1</td>
<td>43K</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>1.7.R3</td>
<td>Rhn2</td>
<td>55K</td>
<td>392</td>
</tr>
<tr>
<td></td>
<td>1.7.R5</td>
<td>Rhn3</td>
<td>57K</td>
<td>411</td>
</tr>
</tbody>
</table>

Table 4: Our test dataset. FR denotes fault revealing.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Distinct Faults</th>
<th>Randoop Failing Tests</th>
<th>PAF Test Flow-sets</th>
<th>Passing Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant1</td>
<td>6</td>
<td>1086</td>
<td>548</td>
<td>74</td>
</tr>
<tr>
<td>Ant2</td>
<td>6</td>
<td>1462</td>
<td>77</td>
<td>124</td>
</tr>
<tr>
<td>Coll1</td>
<td>1</td>
<td>402</td>
<td>38</td>
<td>34</td>
</tr>
<tr>
<td>Coll2</td>
<td>1</td>
<td>256</td>
<td>17</td>
<td>33</td>
</tr>
<tr>
<td>Ivy1</td>
<td>1</td>
<td>360</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>Ivy2</td>
<td>0</td>
<td>565</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>Math1</td>
<td>2</td>
<td>45</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>Rhn1</td>
<td>1</td>
<td>258</td>
<td>41</td>
<td>46</td>
</tr>
<tr>
<td>Rhn2</td>
<td>1</td>
<td>676</td>
<td>37</td>
<td>152</td>
</tr>
<tr>
<td>Rhn3</td>
<td>1</td>
<td>660</td>
<td>44</td>
<td>121</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>5770</td>
<td>811</td>
<td>735</td>
</tr>
</tbody>
</table>

In Table 2, PAF can support over 80% of failures from Randoop with these four error types.

Note that PAF does not handle assertion failures triggered from test cases. Since assertions written in tests check violations of post-conditions (rather than preconditions) after MUT calls are already returned, it is difficult to automatically infer precondition violations in such situations.

### 4.2 Subjects and Experiment Design

Table 3 shows the five open-source Java projects that we used for our experiment subjects. To select our subjects, we identified open source projects used for evaluation in earlier related work using Randoop [40], such as Palus [64], OCAT [32], and Palulu [3]. We the exclusions of those projects that (1) were not actively maintained for at least past 18 months in Github, (2) Randoop did not produce any failing tests for, or (3) PAF found no local test flow-sets.

To conduct our experiments for each subject, we ran PAF on the deployed jar that bundles the entire set of classes (counted in the fifth column of Table 3), and acquired the sets of failing tests for the classes counted in the last column of Table 3. To avoid possible biases caused, we used the default command line settings for Randoop. Particularly, we used default settings for null inputs (i.e., no direct null passing as an argument). That is, NullPointerExceptions caused with null arguments are filtered out. Thus, Column 3 in Table 2 lists NPEs occurred only with non-null argument inputs.

We validated whether a generated failing test is fault revealing using the following criteria. A failing test of a program P is considered fault revealing if the test passes in a subsequent version of P, indicating that the relevant fault in P has been fixed. For those failing tests exercising local failure-inducing dataflows, when we could not find the relevant patch making the test pass, we reported it in the subject’s bug repository. As a result, four bug reports filed by us have been confirmed and fixed (two for Ant, one for Math, and one for Ivy). If a failing test still fails in the latest code, we consider the test non-fault-revealing. The rationale behind this criterion is based on an earlier study by Ray et al. [46]. It reports that “if a bug is introduced to code, the bug will be fixed within few months”. All the subjects that we have selected have been actively maintained and released for at least 18 months. As such, we consider faults of an actively maintained program are mostly fixed within 18 months. This implies that if there has been no changes to make the test pass for last 18 months, the test is likely to be non-fault-revealing. To supplement this criterion on non-fault-revealing tests, we ensure the test fails in the latest version with no changes made to the MUT’s execution. For those tests that failed in the latest version with some changes, we manually inspected and checked whether changes have been made to the evaluation of crash variables. If not, it is considered non-fault-revealing. If so, we filed an issue report seeking developers’ confirmation. We found two such cases and reported. The developer confirmed all of them as non-fault-revealing, which conforms to our criterion.

### 5 EXPERIMENTAL RESULTS

In this section, we present and discuss our experimental results on the research questions. The tool and dataset are available at https://github.com/PAMSE/PAFAnalysis.

Table 4 presents the information of the generated failing tests returned by Randoop and PAF. Column 2 shows the distinct number of faults that can be revealed by Randoop tests. We consider each patch identifies a distinct fault. Column 3 lists the number of failing tests generated by Randoop. Among these tests, Column 4 shows the number of confirmed fault revealing tests based on the criteria described in Section 4.2. Column 5 lists among Randoop’s tests in Column 2, the number of test flow-sets sharing the same crash origin, crash statement, and crash variable. Column 6 represents the number of fault revealing test flow-sets. Although tests in the same test flow-sets may not execute the same path, all tests in a same set are either all fault revealing or all not fault revealing. Our validation of the fault revealing flow-sets found that the patches committed in a subsequent version to fix the faults made all tests in the same test flow-sets pass. This further indicates that all tests in a same fault revealing flow-set reveal the same fault. Finally, the last column lists the number of passing tests generated by Randoop. We used these passing tests to determine the likelihood of precondition violations.

Note that Ivy2 does not have any confirmed fault revealing tests. For this reason, Ivy2 is not involved in the study of RQ1 and RQ3.
Table 5: Results for RQ1. Clustering accuracy

<table>
<thead>
<tr>
<th>Subject</th>
<th>F-Measure</th>
<th># of Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PAF</td>
<td>ReB</td>
</tr>
<tr>
<td>Ant1</td>
<td>1</td>
<td>0.708</td>
</tr>
<tr>
<td>Ant2</td>
<td>0.987</td>
<td>0.979</td>
</tr>
<tr>
<td>Coll1</td>
<td>1</td>
<td>0.593</td>
</tr>
<tr>
<td>Coll2</td>
<td>1</td>
<td>0.64</td>
</tr>
<tr>
<td>Ivy1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ivy2*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Math1</td>
<td>0.925</td>
<td>1</td>
</tr>
<tr>
<td>Rhino1</td>
<td>1</td>
<td>0.048</td>
</tr>
<tr>
<td>Rhino2</td>
<td>1</td>
<td>0.053</td>
</tr>
<tr>
<td>Rhino3</td>
<td>1</td>
<td>0.044</td>
</tr>
</tbody>
</table>

Additionally, for Ant1, the reason why the number of fault revealing tests is significantly larger than other subjects is that 512 out of 548 tests are in the same test flow-set. Because the crash statement is located in a superclass of many subclasses, it is often executed and failed whenever the subclasses are tested.

5.1 RQ1: Clustering Performance

The goal of this study is to evaluate how accurately test flow-sets derived by Paf can cluster fault revealing tests (not all failing tests) with the same fault cause. We obtained the ground truth by considering multiple failing tests induced by the same fault if they failed before a patch was applied and passed after the patch was applied. Ideally, the number of failing test clusters is the same as the number of distinct faults, which is given in Column 2 of Table 4.

As an evaluation metric, we used the F-measure [53], which is widely used for evaluating clustering techniques [8, 42, 53]. To calculate the F-measure, we denote C as the set of all clusters grouped by Paf (i.e., fault revealing test flow-sets) and O as the set of ground truth clusters for the optimal clusters. More specifically, we denote C_i as the i_th cluster, O_j as the j_th cluster, and N as the total number of fault revealing tests in Column 4 of Table 4. We calculate the precision and recall as follows:

\[ \text{Precision}(C_i, O_j) = \frac{|C_i \cap O_j|}{|C_i|} \quad \text{Recall}(C_i, O_j) = \frac{|C_i \cap O_j|}{|O_j|} \]

F-measure computes the weighted average of maximal F-measure for each clusters as follows:

\[ \text{F-measure}(C) = \frac{2 \cdot \text{Recall}(C_i, O_j) \cdot \text{Precision}(C_i, O_j)}{\text{Recall}(C_i, O_j) + \text{Precision}(C_i, O_j)} \]

We compared our results with two baselines, ReBucket [8] and MSeer [19]. The two baselines are state-of-the-art representatives of clustering approaches for program traces. ReBucket groups failing stack traces based on their similarity. MSeer leverages rank-proximity to group failing tests based on the suspiciousness rankings of statements by analyzing execution traces of failing and passing tests. To obtain the suspiciousness rankings, we used a state-of-the-art spectrum-based fault localization tool, called GZoltar [47].

The data in Table 5 show that Paf outperforms the two baselines ReBucket (ReB) and MSeer (MSe) in most cases. The last column (Opt) refers to the ideal case of optimal clustering where all failing tests due to the same fault are clustered in the same set. Paf achieves 100% F-measure for most subjects and at least 92% for all subjects. The reason of preventing Paf from attaining 100% F-measure for Ant2 and Math1 is that a crash origin flows to two different crash statements, and the fix was made at the crash origin.

For ReBucket, if the call stack is not deep and identical to that of other tests, it can also achieve high accuracy like Ant2 and Math1. However, it poorly performs for some subjects (Rhino). This is because ReBucket trains hyper parameters by itself rather than requiring the users to tune them. For Rhino, the parameters are poorly trained and ReBucket yields very low F-Measure.

For MSeer, since failing tests arising from the same fault may traverse different execution paths, the suspiciousness scores of the same statement may vary across the tests, causing them to be partitioned into different clusters.

For Ivy1, the F-measure is 100% for all approaches because there is only one fault revealing test as shown in Table 4. For Ivy2, the results are inapplicable because there is no fault revealing tests as discussed earlier.

The results suggest that test flow-sets grouped by the same crash origins, crash variables and crash statements can effectively cluster failing tests subject to the same fault (or cause). Therefore, Paf enables developers to examine significantly less failing tests because one test in a test flow-set can represent the whole group.

5.2 RQ2: Accuracy of Locality Based Partitioning

In Paf phase 2, it partitions the failure-inducing dataflows induced by failing tests into local and non-local. The goal of this study is to evaluate the accuracy of using this partitioning result to identify if a failing test is fault revealing. To do this, we measured the precision and recall of the partitioning. Since our test flow-sets share the same failure-inducing dataflows, we measure the accuracy in terms of test flow-sets rather than individual failing tests.

We also compared the results with two baselines, other false alarm exception-filtering approaches using JCrasher’s heuristics [6] and Daikon’s dynamic invariants [12]. JCrasher’s heuristics take into account exception types and access modifiers (e.g., public or non-public) of exception-throwing methods to classify whether a given failing test is likely to violate preconditions. For another comparison, we used Daikon’s dynamic invariants [12] mined from generated passing tests from Randoop as counted in Table 4. We extracted relevant invariants with respect to the crash variable at the entry of MUT. If such invariants exist, we classified the corresponding failing tests to the crash variable as false alarms, otherwise as fault-revealing. Existing test generation tools such as DSD-Crasher [7], which is an extended version of JCrasher, and ECat [38] also leverage Daikon’s invariants to filter illegal inputs that violate invariants at the MUT’s entry.

Table 6 presents the comparison results. Since JCrasher and Daikon classify individual tests without grouping, the precision and recall of Paf are measured for both test flow-sets and individual tests. “FR Flow Set” and “FR Test” stand for the number of fault revealing flow-sets and the number of fault revealing failing tests for each subject in our ground truth. “Alarm” and “True” stand for the number of
alarms selected as fault revealing by each individual tool, and the number of true alarms among these alarms, respectively.

The table shows the number of test alarms made by PAF is less than that of other approaches in most cases. PAF achieves significantly higher precision than JCrasher and Daikon for all subjects. PAF also achieves higher or equal recall for all subjects except Ant1 and Ant2. Moreover, for all subjects, PAF achieves at least 100% precision or 100% recall.

For those subjects that could not achieve 100% precision (Ivy1 and three Rhinos), the reason for imprecision is from one common pattern shown in the following code snippet.

```java
public void MUT() {
    m1(0).foo(); } // NPE because m1(0) is null
public Object m1(int i){
    return i>0 ? field : null; } // null is returned
```

The crash origin is local to the MUT’s call graph because `m1` explicitly returns null. Therefore, this NPE occurred due to an incorrect behavior of MUT, rather than the incorrect test input. We reported this issue, but our reports are pending. However, we believe this issue is a bug because we found this fault pattern was properly handled to avoid an exception in other subjects (Ant and Coll). Nevertheless, since there has been no changes, the relevant failing tests cannot be confirmed as fault revealing based on our criterion.

PAF achieved low recall because of two reasons for Ant1 and Ant2. First, four test flow-sets of both Ant1 and Ant2 reveal faults on fields involving concurrency and logging. Preconditions on these fields should be better handled in the code to reliably support method calls at any time. We confirmed that the fixes had been applied in a subsequent version. Second, five test flow-sets of Ant2 fails when MUT is equals method that overrides a Java library method. Although the null input option is disabled as default, equals directly passes null in the tests because Randoop uses it for its contract checks (e.g. o.equals(null) == false). This is an exceptional case where MUT overrides such Java library methods that strictly require explicit preconditions in the code.

For Ivy2, since no fault revealing tests were found, the precision is 0 and the recall is not applicable. However, the number of selected alarms from PAF is significantly lower than other tools even for individual tests.

The recall of PAF (individual tests) for Ant1 produces outliers that make the average recall dramatically drop. This is because one test-flow set containing 512 tests (mentioned in Table 4’s description) is not selected.

Although PAF outperforms other tools in all aspects, the recall of PAF (individual tests) for Ant2 yields average recall lower than Daikon. However, PAF generates significantly less number of highly precise test alarms for all subjects (for Ant2, 1 vs 330). Additionally, our likelihood measurement successfully addresses this weakness by placing fault revealing tests in higher ranks.

Overall, the results suggest the locality of failure-inducing data-flows offers a nice criterion to select fault revealing failing tests in two aspects. First, four of the local failure-inducing dataflows (local flows) were newly detected by PAF. All of them were confirmed and fixed by the developers. Although three more bug reports associated with other local flows are still pending, none of our reports have been rejected. Second, Randoop generated 5770 failures (failing tests) for 10 versions of 5 popular open source subjects. PAF identified 274 of them able to exhibit local flows. It clustered these 274 failures for 24 test flow-set alarms based on their common causes so that users need only to inspect one failure per flow-set. 12 test flow-sets were found to be truly fault revealing. These flow-sets revealed 216 fault revealing failures. As a result, PAF enables developers to inspect only 24 alarms to confirm 216 fault revealing failures out of its identified 274 failures, yielding a precision of 78.8%.

### 5.3 RQ3: Prioritization Performance

The goal of this study is to evaluate how fast PAF detects the faults using a PAF’s ranked list. To do this, we measure the fault detection rate of prioritized test suites in this study. We use a widely-used metric called APFD (Average Percentage Faults Detected) \[10, 11, 48\]. APFD measures the weighted average of the percentage of detected faults over the life of a test suite. An APFD value ranges from 0 to 1, the higher value means the better (i.e., faster) fault detection rate. APFD can be measured in an equation as follows:

\[
APFD = 1 - \frac{\sum_{i=1}^{m} T F_i + T F_2 + \ldots + T F_m}{n m} + \frac{1}{2n}
\]

\[(2)\]
n and m indicate the numbers of failing tests and faults, respectively. 

\( T_{Pi} \) means the rank of the first test case that reveals fault \( i \) in the reordered test set.

Figure 4 gives an illustrative example borrowed from the prior literature on test prioritization [10, 11, 48]. Figure 4(a) shows the information of a test suite of five tests (A through E), which is able to detect 10 faults. Suppose the test suite is prioritized in order of A-B-C-D-E. Figure 4(b) presents the percentage of detected faults when the fraction of test suite is reached. The area under the curve represents the weighted average of the percentage of detected faults over the life of a test suite. Thus, APFD is 50% in this example.

We compare this APFD score with that of existing techniques, JCrasher and Daikon. Since they do not further rank tests after partitioning as described in RQ2, we placed the group of tests classified with “no potential violation” (“Alarm” in Table 6) at the top, the group of tests classified with “potential violation” at the bottom. We then ranked the tests in each group randomly. As discussed in RQ2, since JCrasher and Daikon do not handle clustering, we measured APFD scores of Paf with and without clustering (i.e., test flow-sets and individual tests).

Figure 5 presents the results. The data in the graph show that on average (the leftmost in the graph), the fault detection rate of our prioritization approach outperforms JCrasher and Daikon. Paf (individual tests) outperforms other techniques in most cases. This demonstrates the usefulness of ranking failing tests using the violation likelihood. Although Paf (test flow-sets) performs slightly worse that Paf (individual tests), the use of test flow-sets reduces the number of test cases to examine significantly by 91.2% (=1-24/274) (Table 4). The results for individual tests are better because APFD scores are calculated with regards to the rank of the first test that reveals the same fault. Since there are multiple tests (sometimes tens or even hundreds) in the same test flow-set, the APFD value can be pushed up by the top ranked fault revealing test. On the other hand, Paf measures the likelihood of a test flow-set based on the test with the highest likelihood, i.e., the lowest ranked fault-revealing test in the set.

Paf consistently achieves at least 90% APFD score for all subjects except Ant2 while JCrasher greatly fluctuates across different subjects. The main reason is that the precondition violation filtering of JCrasher’s heuristics depends solely on exception types and method modifiers. These two types of information may not be effective in inferring a violation of a precondition.

Daikon performs better than JCrasher in most cases although it performs poorly for some subjects (Ivy and Math). Most importantly, however, Paf (individual test) outperforms Daikon in all subjects except Ant2. This implies that considering only invariant violations is insufficient for effective prioritization because the quality of dynamic invariants mined varies by profiles and coverage of passing executions. Paf’s performance may also be affected by this issue. However, unlike Daikon, Paf bases its analysis on failure-inducing dataflows. The analysis can better identify the likelihood of precondition violations and reduce the ambiguities that may arise from the inaccurate invariants mined.

In summary, this study shows that the use of generated passing tests is useful for ranking. The proportion of exercised DUAs by passing tests (rather than the raw number of passing tests) can affect the ranking. This percentage represents the chances that the crash variable is reached in various passing dataflows. This information helps improve the ranking effectiveness when recall is not 100%.

5.4 Threats to Validity

We consider several threats to validity of our experiments.

A potential threat in our experimentation is that the set of fault revealing tests used for the experiments may not be 100% accurate. In general, there is no ground truth that a failing test is not fault revealing. To address this, we adopted well-defined and objective criteria to label fault revealing tests as discussed in Section 4.2.

Threats also arise when the results from the experiment are not generalizable to other environments, such as in other testing tools. To mitigate these threats, we evaluated our experiment using tests generated from Randoop, which is a popular automatic test generation tools used both in academia and industry. In future, we plan to investigate Paf’s results on other test generation tools. Since Paf analysis considers Java runtime exceptions described in Section 4.1. Paf can be integrated into other test generation tools in the same way as Randoop.

To alleviate the generalizability threat, we selected five popular open-source projects with different sizes and application types (e.g., commandline tools and libraries) for experiments. However, our results may not be generalizable to commercial projects, GUI projects, and projects written in other languages.

6 RELATED WORK

Our contributions relate to the work that clusters failing tests with the same failure cause, improves the input quality of generated tests, and assesses the fault-detection ability of test suites.
For the area of clustering failing tests, existing techniques mainly leverage similarities among stack traces [4, 8, 36] or execution traces [9, 19, 33, 35, 42, 44]. For the execution trace based approach, there are two types, rank-proximity and trace-proximity. The rank-proximity uses distances between pairwise rankings returned from spectrum-based fault localization [19, 33, 35, 42]. The trace-proximity clusters tests based on the coverage or profile similarity of their execution traces [9, 33, 44]. It was reported that rank-proximity is more advanced and outperforms trace-proximity [35]. Our technique not only clusters failing executions with the same causes, but also prioritizes them based on the likelihood of precondition violations.

To improve the input quality of generated tests, some existing techniques address the implicit precondition issue. As discussed in Introduction, JCrasher [6] leverages exception types and access modifiers, DSD-Crasher [7] and Eclat [38] use dynamic invariants of Daikon [12] mined from sample executions. In addition, Fraser et al. [18] uses temporal properties among method invocations. Other work addresses the false alarm issue using specification mining of API protocols [45] and search-based approach at the GUI level [23]. Our approach is orthogonal to these techniques because they focus on generating fault-revealing tests while ours prioritizes generated tests for early fault detection.

Jain et al. [31] proposed a technique that determines the feasibility of argument inputs generated by dynamic symbolic executions using Daikon’s invariants, and ranks the generated inputs based on the confidence values of invariants. However, this approach targets argument inputs generated for a single method rather than a sequence of methods that our approach targets.

Various techniques [3, 32, 38, 54, 55, 61, 64] were proposed to enhance the feasibility of generated test inputs by leveraging dynamic information from sample executions and learning desirable object states. Other techniques were proposed to generate feasible test oracles based on the realistic specifications of expected program behaviors. These techniques use program invariants [38, 60], sequences of program execution [21], seeded defects [17, 52], and JavaDoc comments [22]. However, these techniques were developed to improve code coverage or fault detection ability rather than to reduce the false alarm rate of generated failing tests like Paf.

There has been some research that assesses the effectiveness of a test suite in detecting faults based on several test adequacy criteria. Some studies [13, 30] showed that the dataflow adequacy criteria can influence the test suite effectiveness in the fault-detecting ability. This finding supports the underlying analysis of Paf, which uses data-dependences chains to identify fault-revealing tests. An earlier study [37] showed that the size of a test suite can also influence the fault-detection ability. It found that after a certain point, the fault detection rate barely increases even though the size of a test suite keeps increasing. This finding also supports our results where Paf assigns high ranks for most fault-revealing test flow-sets.

Additionally, to improve the fault detection ability of test suites, attention was paid to the problem of faulty test code, which produces false alarms and reduces the quality of test suites [37]. Herzig and Nagappan [28] developed a false alarm detection approach by mining association rules using the false alarm history in the past. A test analysis technique developed by Waterloo et al. [59] categorizes test patterns to find faulty tests.

Our approach differs from these techniques in that it targets automatically-generated tests and uses program analysis rather than machine learning. It does not assume the availability of a false alarm history. Moreover, these techniques target faulty or obsolete tests written by developers. Thus, the root causes of these faults differ from ours, which is related to implicit precondition violations.

A number of techniques were developed to improve test suite maintenance. Some of them aimed to assess the quality of human-written tests using dynamic tainting analysis [29] and test dependency analysis [24]. Zoomln [43] is a technique developed to help improve oracles of automatically-generated tests using dynamic invariants from human-written tests. The main goal of these techniques is different from ours because they try to improve the quality of test suites [24, 29] and test oracles [43] for test suite maintenance rather than for failure inspection.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a technique called Paf that clusters generated failing tests into test flow-sets due to the same fault and prioritizes these tests in a reverse order to their likelihood of violating an implicit precondition. We introduce the concepts of crash variables and crash origins.

Paf performs the analysis in three steps. First, Paf derives a failure-inducing dataflow concerning the crash variable for each failing test. Tests inducing similar failure-inducing dataflows are considered to share a common failing cause and clustered into the same test flow-set. Second, Paf checks whether such dataflows are local or non-local to the execution of the method under test (MUT) based on the location of their crash origins. Local dataflows are given higher priority than non-local ones because a local dataflow indicates the failure is wholly induced by the MUT’s implementation programmed by its developers. As a result, the chances of its violating an MUT’s precondition is small. Third, Paf examines the dataflows exercised by other related passing tests to estimate the likelihood of potential precondition violations. The likelihood increases when more dataflows concerning the crash variable are found in other passing tests. The likelihood is measured by calculating the proportion of other dataflows exercised by passing tests. Paf finally reorders failing tests first by placing the group of tests exercising local dataflows prior to that exercising non-local ones. Paf further sorts tests in each group by the violation likelihood.

We conducted experiments based on five popular open-source projects with tests generated by Randoop. The experimental results show that test flow-sets can effectively cluster fault revealing tests arising from a common cause. The results also show that locality analysis can accurately classify fault-revealing tests, and the likelihood calculation is effective in prioritizing the fault-revealing tests. Comparing with Rebucket [8] and MSeer [19], Paf can more accurately cluster fault revealing tests. It also outperforms existing techniques that filter potential precondition violations using JCrasher’s heuristics [6] and Daikon [12]’s invariants [7, 38].

For future work, we plan to extend our empirical studies with other testing tools such as EvoSuite. We also plan to adapt our approach to human-written tests and investigate their fault-detection capability.