Which Generated Test Failures are Fault Revealing?[*]

Mijung Kim, Shing-Chi Cheung, and Sunghun Kim
The Hong Kong University of Science and Technology
Hong Kong, China
{mjkimab, scc, 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 Tests, Fault-inducing Data-flow Analysis

ACM Reference Format:
Mijung Kim, Shing-Chi Cheung, and Sunghun Kim. 2018. Which Generated Test Failures are Fault Revealing?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236058

[*]The authors thank the anonymous reviewers and CASTLE members for their constructive comments. This work is supported by the Hong Kong RGC/GRF grant 16202917, MSRA collaborative research grant, and Nvidia academic program.

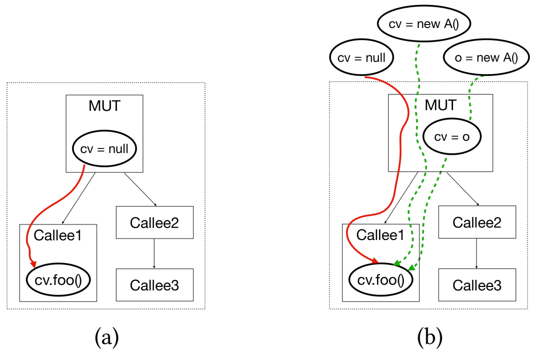


Figure 1: Call graphs to illustrate (a) local and (b) non-local dataflows regarding the crash variable (cv).

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 those 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 made implicitly by developers are rarely documented.

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 *Callee*1. 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 precondition 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 likeliness 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. Paf 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 possible 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., `indexOf` 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 `indexOf`. 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 `indexOf`'s call graph. In the case for `fTest2`, the `null` value received by variable `project` is originated from the default field initializer at Line 32 before `handleInput` is invoked. So, the failure-inducing dataflow exercised by `fTest2` is considered non-local to the MUT `handleInput`'s call graph. Note that 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

Four Generated Test Cases for ProjectEnry 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((Object)"hi!"); // MUT
6   }
7
8   public void fTest2() {
9     ProjectEntry pc = new ProjectEntry();
10    byte[] byte_array = new byte[] {};
11    pc.handleInput(byte_array, (-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 ProjectEnry Class

```
31  private HashMap entries;
32  private Project project;
33
34  public ProjectEntry(){}
35  public ProjectEntry(Project project){
36    this.project = project;
37    entries = new HashMap(10);
38  }
39
40  public void put(Object key, Object value) {
41    Entry e = new Entry(key, value);
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: Motivating example for the failing tests in Figure 2.

Test	Crash Origin	Locality of failure-inducing dataflows	Def-use reaching crash statement	Passing tests exercising def-use
fTest1	Line 45	local	(45, 47)	None
fTest2	Line 32	non-local	(56, 59)	pTest1 pTest2

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 crash variable exercised by passing tests. For fTest1, there are no other def-use relation reaching the crash statement. For fTest2, there is one def-use relation exercised by passing tests reaching the crash statement, and it is defined at Line 56.

Paf then partitions the generated failing tests into two groups: local and non-local. Local failing tests are ranked before non-local failing tests because a local failure-inducing dataflow arises wholly from the logic of the MUT and its callees. In most cases, the MUT developers should be aware of the underlying 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 example, 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[2]. On the other hand, fTest2 has been confirmed as a false alarm by the developer[3].

3 OUR APPROACH

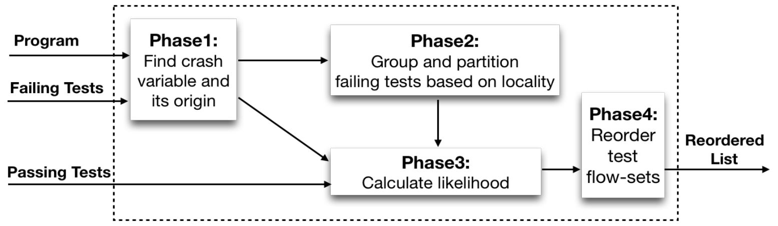
In this section, we present the details of Paf. Figure 3(a) shows the four processing phases of Paf. It takes a target program, a set of generated failing tests and a set of generated passing tests as inputs, and outputs a list of reordered failing tests.

In Phase 1, Paf first identifies the crash variable for each failing 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 potential 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 likelihood obtained in Phases 2 and 3. Figure 3(b) shows an example of a reordered list returned by Paf.

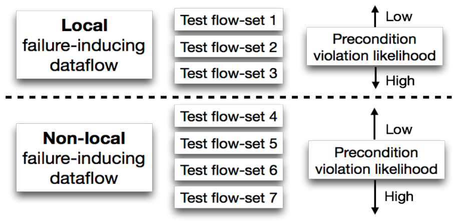
3.1 Phase 1: Find Crash Variable and its Origin

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

[2]https://issues.apache.org/jira/browse/COLLECTIONS-28
[3]https://bz.apache.org/bugzilla/show_bug.cgi?id=49400


Program
Failing Tests
Passing Tests
Phase1: Find crash variable and its origin
Phase2: Group and partition failing tests based on locality
Phase3: Calculate likelihood
Phase4: Reorder test flow-sets
Reordered List

(a) Overview


Local failure-inducing dataflow
Test flow-set 1
Test flow-set 2
Test flow-set 3
Low
Precondition violation likelihood
High
Non-local failure-inducing dataflow
Test flow-set 4
Test flow-set 5
Test flow-set 6
Test flow-set 7
Low
Precondition violation likelihood
High

(b) Output diagram

Figure 3: Overview of our approach (a) and an output example of the reordered list returned by PAF (b).

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 ArrayOutOfBoundException, 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 DivideByZeroException. 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 DUAs {(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 DUA 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 spontaneously when a constructor is called, PAF 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, PAF measures the likelihood of a potential precondition violation

[4] A def-use association represents a data-flow relationship in a triple (d, u, v) such that a variable v which has been defined in Statement d reaches and is used in Statement u without being redefined along a control flow path that connects d to u.

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, PAF calculates the likelihood according to the DUA 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, PAF extracts a set of DUAs with respect to the crash variable cv at the crash statement u. Each extracted DUA (d, u, cv) satisfies three criteria: (1) d refers to a statement where cv 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, PAF measures its likelihood based on the number of DUAs in θ covered by at least one passing test as follows:

$$\text{Likelihood} = \frac{\text{\# DUAs in } \theta \text{ covered by passing tests}}{\text{\# DUAs in } \theta} \quad (1)$$

PAF 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 FS_1 and FS_2, respectively. FS_1 assumes the θ for `fTest1`, which is {(45, 47, e)}. Since there is no passing test that covers this DUA, FS_1'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. FS_2 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 FS_2 is, therefore, 1/1 = 1.

3.4 Phase 4: Reorder Failing Tests

In Phase 4, PAF 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 would look like Figure 3(b).

Recall that PAF can handle multiple crash variables involved in a single failing test (as discussed in Phase 1). PAF performs the analysis for each crash variable individually. That is, PAF computes the likelihoods for each flow-sets corresponding to the multiple crash variables, and uses the largest likelihood for reordering the test flow-set. The largest likelihood is used because a failing test would be a false alarm if any of its failure-inducing dataflows to one of the crash variables violates implicit preconditions.

4 EXPERIMENT SETUP

To assess our approach, we implemented PAF and evaluated it on 10 versions of five real-world Java programs.

Our evaluation investigates three research questions:

Table 2: Generated failures and their types from Randoop

Subject	Total Fails	NPE	Assertion Errors	Other
Ant	1274	93.1%	4.4%	2.5%
Collections	329	72.2%	24.5%	3.3%
Ivy	463	60.8%	9.7%	29.5%
Math	45	91.2%	8.8%	0%
Rhino	531	77.4%	18.8%	3.8%
Average	528.4	79.0%	13.2%	7.8%

RQ1: How accurately does our approach cluster fault revealing tests with the same failure cause?
This study aims to evaluate whether test flow-sets sharing the same crash origin, crash statement, and crash variable can accurately cluster fault revealing tests with the same failure cause.

RQ2: How likely is a failing test fault-revealing if it exercises a local failure-inducing dataflow?
This study aims to evaluate the usefulness of partitioning failing tests based on the locality of the dataflows.

RQ3: Can our approach improve the rate of the fault detection of generated failing tests?
This study aims to demonstrate the effectiveness of our prioritization technique by considering whether our approach can detect faults faster than other approaches.

4.1 Implementation

We integrated PAF into Randoop(v3.1.0) [40] so that PAF outputs only those passing and failing tests exercising the distinct failure-inducing dataflows along with the list of the reordered failing tests. We selected Randoop because of its popularity and adoption by the industry [39] and academia for automated test generation.

We implemented the PAF tool on top of the Soot framework [58]. PAF 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 coverages, 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. PAF treats an application specific exception thrown under a branch condition as an IllegalStatementException. As shown in Table 2[5], 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.

[5] Assertion errors shown in Table 2 include ones triggered from the tests.

Table 3: Subjects used in the empirical studies.

Subject		Label	LOC	Class under Test	Class w/ Failing Tests
Ant	1.6.5	Ant1	86K	572	12
	1.8.1	Ant2	102K	873	16
Collections	2.0	Coll1	7K	52	6
	2.1	Coll2	11K	204	8
Ivy	2.2.0	Ivy1	50K	489	10
	2.4.0	Ivy2	51K	495	18
Math	2.2	Math1	49K	784	10
Rhino	1.7.R2	Rhn1	43K	153	7
	1.7.R3	Rhn2	55K	392	25
	1.7.R5	Rhn3	57K	411	25

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

		Randoop		PAF		
Subject	Distinct Faults	Failing Tests	FR Tests	Test Flow-sets	FR Test Flow-sets	Passing Tests
Ant1	6	1086	548	74	6	25,587
Ant2	6	1462	77	124	7	36,106
Coll1	1	402	38	34	1	12,972
Coll2	1	256	17	33	1	10,761
Ivy1	1	360	1	65	1	7,047
Ivy2*	0	565	0	64	0	13,232
Math1	2	45	8	22	3	6,204
Rhn1	1	258	41	46	1	13,626
Rhn2	1	676	37	152	1	15,691
Rhn3	1	660	44	121	1	15,132
Total	20	5770	811	735	22	156,358

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 excluded those that (1) were not actively maintained for at least past 18 months in Github, (2) Randoop could not generate tests due to complex external library dependencies, (3) Randoop did not produce any failing tests, and (4) 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. 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.

Table 5: Results for RQ1. Clustering accuracy

	F-Measure			# of Clusters			
Subject	PAF	ReB	MSe	PAF	ReB	MSe	Opt
Ant1	1	0.708	0.708	5	14	7	5
Ant2	0.987	0.979	0.711	7	7	4	6
Coll1	1	0.593	0.733	1	23	2	1
Coll2	1	0.64	0.786	1	10	2	1
Ivy1	1	1	1	1	1	1	1
Ivy2*	-	-	-	-	-	-	-
Math1	0.925	1	0.583	3	2	3	2
Rhino1	1	0.048	0.988	1	41	2	1
Rhino2	1	0.053	0.857	1	37	3	1
Rhino3	1	0.044	0.842	1	44	2	1

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 failure 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:

$$\text{Precision}(C_i, O_j) = \frac{|C_i \cap O_j|}{|C_i|}, \text{Recall}(C_i, O_j) = \frac{|C_i \cap O_j|}{|O_j|}$$

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

$$\text{F}(C_i, O_j) = \frac{2 * \text{Recall}(C_i, O_j) * \text{Precision}(C_i, O_j)}{\text{Recall}(C_i, O_j) + \text{Precision}(C_i, O_j)}$$

$$\text{F-measure}(C) = \sum_i \frac{|C_i|}{N} * max_j\{\text{F}(C_i, O_j)\}$$

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-promixity 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 ReBucekt 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 EClat [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

Table 6: Precision (Pre.) and Recall (Rec.) results for RQ2.

Subject	Test Flow-Sets				Individual Tests									
	FR Flow Set	PAF			FR Test	PAF			JCrasher			Daikon		
		True / Alarm	Pre.	Rec.		True / Alarm	Pre.	Rec.	True / Alarm	Pre.	Rec.	True / Alarm	Pre.	Rec.
Ant1	6	2/2	100	33.3	548	20/20	100	3.6	334/392	85.2	60.9	21/174	12.1	3.8
Ant2	7	1/1	100	14.3	77	10/10	100	13.0	13/50	26	16.9	75/330	22.7	97.4
Coll1	1	1/1	100	100	38	38/38	100	100	16/87	18.4	42.1	38/90	42.2	100
Coll2	1	1/1	100	100	17	17/17	100	100	8/77	10.4	47.1	17/34	50	100
Ivy1	1	1/8	12.5	100	1	1/36	2.8	100	0/169	0	0	1/238	0.4	100
Ivy2	0	0/2	0	-	0	0/15	0	-	0/265	0	-	0/147	0	-
Math1	3	3/3	100	100	8	8/8	100	100	5/25	20	62.5	8/45	17.8	100
Rhino1	1	1/2	50	100	41	41/44	93.2	100	0/48	0	0	41/166	24.7	100
Rhino2	1	1/2	50	100	37	37/40	92.5	100	0/138	0	0	37/424	8.7	100
Rhino3	1	1/2	50	100	44	44/46	95.7	100	0/259	0	0	44/495	8.9	100
Total	22	12/24	50	54.5	811	216/274	78.8	26.6	376/1510	24.9	46.4	282/2143	13.2	34.8

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.

```
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 dataflows 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 into 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

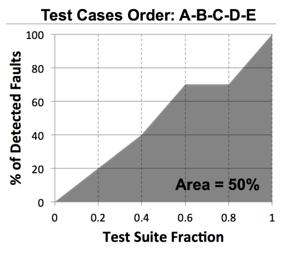
The goal of this study is to evaluate how fast PAF detects the faults using a PAF's ranked list. To do this, we measure the fault detection rate of prioritized test suites in this study. We use a widely-used metric called APFD (Average Percentage Faults Detected) [10, 11, 48]. APFD measures the weighted 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:

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n} \quad (2)$$

n and m indicate the numbers of failing tests and faults, respectively. TF_i means the rank of the *first* test case that reveals fault i in the reordered test set.

	Fault									
Test	1	2	3	4	5	6	7	8	9	10
A	X				X					
B						X	X			
C	X	X	X	X	X	X	X			
D										
E								X	X	X

(a)



(b)

Figure 4: Example of the APFD (Average Percentage Faults Detected) measure.

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 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 Ant1 and 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

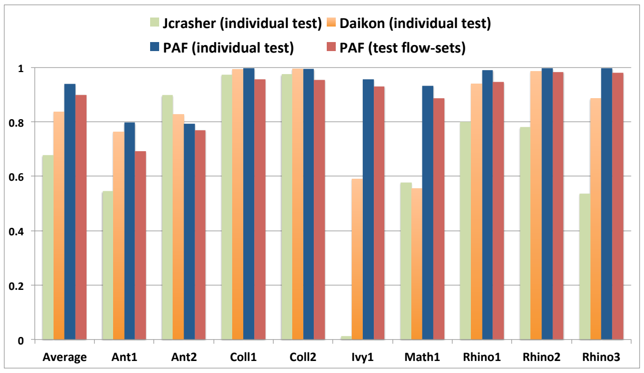


Figure 5: Results for RQ3 showing APFD scores.

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 inadequate 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 [57]. 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]. ZoomIn [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 technique 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 the set of 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 technique to human-written tests and investigate their fault-detection capability.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (01 Aug 2018), 1959–1981. https://doi.org/10.1007/s10664-017-9570-9

[3] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. 2006. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS).* Portland, OR.

[4] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques (SysML'08).* USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1855895.1855896

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08).* USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[6] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[7] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 8 (May 2008), 37 pages. https://doi.org/10.1145/1348250.1348254

[8] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 1084–1093. http://dl.acm.org/citation.cfm?id=2337223.2337364

[9] William Dickinson, David Leon, and Andy Podgurski. 2001. Finding Failures by Cluster Analysis of Execution Profiles. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01).* IEEE Computer Society, Washington, DC, USA, 339–348. http://dl.acm.org/citation.cfm?id=381473.381509

[10] Hyunsook Do and Gregg Rothermel. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Softw. Eng.* 32, 9 (Sept. 2006), 733–752. https://doi.org/10.1109/TSE.2006.92

[11] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00).* ACM, New York, NY, USA, 102–112. https://doi.org/10.1145/347324.348910

[12] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[13] P. G. Frankl and S. N. Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Trans. Softw. Eng.* 19, 8 (Aug. 1993), 774–787. https://doi.org/10.1109/32.238581

[14] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (feb. 2013), 276 –291. https://doi.org/10.1109/TSE.2012.14

[15] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2013. Does Automated White-box Test Generation Really Help Software Testers? *(ISSTA 2013).* ACM, New York, NY, USA, 291–301. https://doi.org/10.1145/2483760.2483774

[16] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2014. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2014).

[17] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven Generation of Unit Tests and Oracles *(ISSTA '10).* ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/1831708.1831728

[18] G. Fraser and A. Zeller. 2011. Exploiting Common Object Usage in Test Case Generation. In *ICST'11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation.* IEEE Computer Society, 80–89.

[19] R. Gao and W. E. Wong. 2017. MSeer-An Advanced Technique for Locating Multiple Bugs in Parallel. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2776912

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing *(PLDI '05).* ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[21] Alberto Goffi. 2014. Automatic Generation of Cost-effective Test Oracles. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014).* ACM, New York, NY, USA, 678–681. https://doi.org/10.1145/2591062.2591078

[22] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis.* Saarbrücken, Genmany, 213–224.

[23] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based System Testing: High Coverage, No False Alarms *(ISSTA 2012).* ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2338965.2336762

[24] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency *(ISSTA 2015).* ACM, New York, NY, USA, 223–233. https://doi.org/10.1145/2771783.2771793

[25] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. 2014. A Unified Test Case Prioritization Approach. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 10 (Dec. 2014), 31 pages. https://doi.org/10.1145/2685614

[26] Mary Jean Harrold and Gregg Rothermel. 1994. Performing Data Flow Testing on Classes. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94).* ACM, New York, NY, USA, 154–163. https://doi.org/10.1145/193173.195402

[27] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient Computation of Interprocedural Definition-use Chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (March 1994), 175–204. https://doi.org/10.1145/174662.174663

[28] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *Companion Proceedings of the 37th International Conference on Software Engineering.* IEEE – Institute of Electrical and Electronics Engineers. http://research.microsoft.com/apps/pubs/default.aspx?id=238351

[29] Chen Huo and James Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* ACM, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

[30] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94).* IEEE Computer Society Press, Los Alamitos, CA, USA, 191–200. http://dl.acm.org/citation.cfm?id=257734.257766

[31] Nehul Jain, Saikat Dutta, Ansuman Banerjee, Anil K. Ghosh, Lihua Xu, and Huibiao Zhu. 2013. Using Daikon to Prioritize and Group Unit Bugs. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers.* 215–233. https://doi.org/10.1007/978-3-319-07602-7_14

[32] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10).* ACM, New York, NY, USA, 159–170. https://doi.org/10.1145/1831708.1831729

[33] James A. Jones, James F. Bowring, and Mary Jean Harrold. 2007. Debugging in Parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07).* ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/1273463.1273468

[34] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12).* IEEE Computer Society, Washington, DC, USA, 11–20. https://doi.org/10.1109/ISSRE.2012.31

[35] Chao Liu and Jiawei Han. 2006. Failure Proximity: A Fault Localization-based Approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14).* ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1181775.1181782

[36] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. 2007. Automatically Identifying Known Software Problems. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW '07).* IEEE Computer Society, Washington, DC, USA, 433–441. https://doi.org/10.1109/ICDEW.2007.4401026

[37] Akbar Siami Namin and James H. Andrews. 2009. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09).* ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/1572272.1572280

[38] Carlos Pacheco and Michael D. Ernst. 2005. Eclat: Automatic Generation and Classification of Test Inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05).* Springer-Verlag, Berlin, Heidelberg, 504–527. https://doi.org/10.1007/11531142_22

[39] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding Errors in .Net with Feedback-directed Random Testing *(ISSTA '08).* ACM, New York, NY, USA, 87–96. https://doi.org/10.1145/1390630.1390643

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering.* IEEE Computer Society, Minneapolis, MN, USA.

[41] H. D. Pande, W. A. Landi, and B. G. Ryder. 1994. Interprocedural Def-Use Associations for C Systems with Single Level Pointers. *IEEE Trans. Softw. Eng.* 20, 5 (May 1994), 385–403. https://doi.org/10.1109/32.286418

[42] Sangmin Park, Mary Jean Harrold, and Richard Vuduc. 2013. Griffin: Grouping Suspicious Memory-access Patterns to Improve Understanding of Concurrency Bugs. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 134–144. https://doi.org/10.1145/2483760.2483792

[43] Fabrizio Pastore and Leonardo Mariani. 2015. ZoomIn: Discovering Failures by Detecting Wrong Assertions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. 66–76. https://doi.org/10.1109/ICSE.2015.29

[44] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated Support for Classifying Software Failure Reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 465–475. http://dl.acm.org/citation.cfm?id=776816.776872

[45] Michael Pradel and Thomas R. Gross. 2012. Leveraging Test Generation and Specification Mining for Automated Bug Detection Without False Positives *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 288–298. http://dl.acm.org/citation.cfm?id=2337223.2337258

[46] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2015. The Uniqueness of Changes: Characteristics and Applications *(MSR '15)*. ACM.

[47] André Riboira and Rui Abreu. 2010. The GZoltar Project: A Graphical Debugger Interface. In *Testing – Practice and Research Techniques*, Leonardo Bottaci and Gordon Fraser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–218.

[48] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Softw. Eng.* 27, 10 (Oct. 2001), 929–948. https://doi.org/10.1109/32.962562

[49] Raul Santelices, Yiji Zhang, Haipeng Cai, and Siyuan Jiang. 2013. DUA-forensics: A Fine-grained Dependence Analysis and Instrumentation Framework Based on Soot. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP '13)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2487568.2487574

[50] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C *(ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[51] Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[52] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. 2012. Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 870–880. http://dl.acm.org/citation.cfm?id=2337223.2337326

[53] Michael Steinbach, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. In *In KDD Workshop on Text Mining*.

[54] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. 2010. DyGen: Automatic Generation of High-coverage Tests via Mining Gigabytes of Dynamic Traces. In *Proceedings of the 4th International Conference on Tests and Proofs (TAP'10)*. Springer-Verlag, Berlin, Heidelberg, 77–93. http://dl.acm.org/citation.cfm?id=1894403.1894415

[55] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-oriented Unit-test Generation via Mining Source Code *(ESEC/FSE '09)*. ACM, New York, NY, USA, 193–202. https://doi.org/10.1145/1595696.1595725

[56] Paolo Tonella. 2004. Evolutionary Testing of Classes *(ISSTA '04)*. ACM, New York, NY, USA, 119–128. https://doi.org/10.1145/1007512.1007528

[57] A. Vahabzadeh, A. M. Fard, and A. Mesbah. 2015. An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. 101–110. https://doi.org/10.1109/ICSM.2015.7332456

[58] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[59] M. Waterloo, S. Person, and S. Elbaum. 2015. Test Analysis: Searching for Faults in Tests (N). In *Automated Software Engineering (ASE), 2015*. 149–154. https://doi.org/10.1109/ASE.2015.37

[60] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts *(ICSE '11)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/1985793.1985820

[61] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer. 2011. Stateful testing: Finding more errors in code and contracts. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. 440–443. https://doi.org/10.1109/ASE.2011.6100094

[62] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2483760.2483782

[63] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression Mutation Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 331–341. https://doi.org/10.1145/2338965.2336793

[64] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 353–363. https://doi.org/10.1145/2001420.2001463