Do not neglect what's on your hands: localizing software faults with exception trigger stream

Xihao Zhang
School of Computer Science,
Wuhan University
Wuhan, China
zhangxihao@whu.edu.cn

Yi Song*
School of Computer Science,
Wuhan University
Wuhan, China
yisong@whu.edu.cn

Xiaoyuan Xie†
School of Computer Science,
Wuhan University
Wuhan, China
xxie@whu.edu.cn

Qi Xin
School of Computer Science,
Wuhan University
Hubei Luojia Laboratory
Wuhan, China
qxin@whu.edu.cn

Chenliang Xing
School of Computer Science,
Wuhan University
Wuhan, China
xingchenliang@whu.edu.cn

ABSTRACT

Existing fault localization techniques typically analyze static information and run-time profiles of faulty software programs, and subsequently calculate suspiciousness values for each program entity. Such strategies typically have overbroad information to be analyzed and lead to unsatisfactory results. Exception is a widely-used programming language feature. It is closely related to the execution status during the execution of programs, and thus can be incorporated into automatic fault localization techniques for better effectiveness. Based on this intuition, we propose EXPECT, a novel fault localization technique that makes use of exception information, a valuable source of data for fault localization while being often ignored in previous research. Specifically, EXPECT first constructs exception trigger streams (including exception trigger information and execution traces), and then localizes faults by tracing bifurcation points between different exception trigger streams. Moreover, the tie-breaking problem can be also benefited from the use of exception trigger streams. Experimental results demonstrate the advantages of EXPECT: it achieves as high as 38.26% improvements in localizing faults regarding the Exam metric in comparison to the state-of-the-art fault localization technique, and it reduces the scales of ties in existing FL methods by up to 99.08%.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

*Co-first author.
†Corresponding author.

KEYWORDS

Program debugging, Software testing, Fault localization, Exception handling, Execution trace

ACM Reference Format:
Xihao Zhang, Yi Song, Xiaoyuan Xie, Qi Xin, and Chenliang Xing. 2024. Do not neglect what's on your hands: localizing software faults with exception trigger stream. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695479

1 INTRODUCTION

Software debugging is a pivotal part of software quality assurance [39, 58, 59] and has received much attention from developers. Fault localization (FL) is regarded as the most tedious and painful process in debugging [42, 49], and researchers have proposed different types of automatic fault localization techniques to tackle this challenge [23, 27, 33, 38, 40, 67]. Among these techniques, methods based on dynamic information have been demonstrated to generally deliver higher precision compared with techniques based on static information [24, 61]. Thus, in this work, we also choose to adopt dynamic information.

A typical pipeline of these techniques first gathers dynamic runtime information of software programs. Then, they aim to reveal some potential correlation between the gathered information and software faults. A suspiciousness value is generally assigned to each software element (e.g., a method, a basic block, or a statement) and a ranking list of these elements can then be formed accordingly, which will further serve as a guide for human developers during debugging. Two representative types of dynamic information-based FL techniques are Spectrum-Based Fault Localization (SBFL) [2, 11, 20, 38, 56, 57] and Mutation-Based Fault Localization (MBFL) [30, 37, 40, 68]. Recently, a novel semantics-based method, SmartFL is proposed [67], which introduces a probabilistic approach and has been demonstrated to outperform the state-of-the-art (SOTA) SBFL and MBFL techniques.

However, **the scale of run-time information** gathered in these existing methods **is somewhat overbroad**. More specific information that can help spot the root cause of failures is not fully utilized. For example, **SBFL** generally collects coverage information of passed and failed test cases, constructs the program spectrum, and feeds it into a risk evaluation formula to statistically approximate the correlation between the spectrum and the underlying fault. However, the coverage-based spectrum typically has a large scale. For example, the lengths of the coverage vectors of famous open-source projects Chart, Time, and MyBatis[1] are approximately 40,900, 14,900, and 11,000, respectively. Such overbroad information hinders a precise analysis of root causes and may thus degrades the effectiveness of SBFL. Moreover, it is hard for SBFL techniques to distinguish program elements having the same spectrum information, which, unfortunately, is very common in practice. The mentioned bottlenecks of SBFL have already been reported in previous studies [22, 63, 64, 66]. **MBFL** mutates faulty programs and collects test results before and after mutation. By analyzing whether the test results are changed by mutations on each program statement, MBFL intends to identify the faulty part of the program. The intuition behind this strategy is that if mutations on a statement have a higher possibility to change the test result in the failing tests instead of the passing tests, the corresponding statement may have a higher possibility to contain the fault. However, to assess the suspiciousness value of statements, MBFL needs to collect test results after mutating each statement, the scale of such information is even larger than that of coverage information. Moreover, MBFL is severely limited by high costs in both time and computation, which reduces the practicability of MBFL in real-world debugging [30, 71]. Though **dynamic slicing-based strategies** have been introduced to reduce the run-time profiles to be analyzed in SBFL and MBFL, the scale of dynamic slices is still decided by the program under test and thus could be overbroad [7], and faulty statements are possible to be missed in some methods [33, 69]. Recently, Zeng et al. presented a novel semantics-based FL technique, **SmartFL** [67], which models program semantics as well as information from static analyses and dynamic execution traces, and employs a probabilistic approach to localize faults. However, the scale of semantics information gathered and analyzed in this work is still decided by the scale of execution traces. When there are massive statements executed by the failed test cases, the scale of semantics information may be close to that of the coverage information. Besides, the workflow of SmartFL is more complex and thus has higher learning costs compared with traditional tactics.

Therefore, it is natural to think about whether there is dynamic information having the ability to mine the correlation between observed failures and faults **on a relatively small scale**. *Exception* is a strategy originally designed to capture and report execution status while running programs. In general, developers use "*try*" to monitor some run-time events (e.g., reference to a null pointer or division by zero). The exception handling code checks whether an exception is thrown and further deals with thrown exceptions. By referring to the best practices of exception-handling [54] which suggest that "*handling exceptions as close to the problem as you can*", we found that these "*try*" blocks can be regarded as **critical checkpoints to inspect the status** of program execution. As a consequence, the status at these critical checkpoints has the potential to provide proper dynamic information and thus serve the purpose of fault localization.

To investigate the potential of the above exception information and further integrate it into fault localization, we preliminarily conduct an exploratory study on over 20 popular open-source Java projects[2]. In this exploratory study, we first inspect the exception handling strategies in each project to find a proper type of information that is generally available and can thus make our method widely applicable, after which we determine to utilize the exception trigger information (i.e. whether an exception is triggered or not) in try blocks. Then, we simulate 40 faulty versions of a project that give failed results, and run the failed test cases of each version on both the original version of the project and the faulty version itself. By collecting their corresponding exception trigger information in each covered try block, we form the *exception trigger stream* for each version. We observed that the seeded fault usually lies between the bifurcation point (i.e. the first covered try block having different triggered-or-not information in the passed and failed exception trigger streams) and its previous covered try block (Please see Section 3.1 for the motivation example).

Inspired by the motivation example, we present EXPECT, an **EX**ce**P**tion trigg**E**r stream-based fault lo**C**aliza**T**ion technique. For each failed test case, EXPECT first collects its **exception trigger stream in the failed execution**, and then utilizes history versions, different development branches, or mutants of the program under test as substitutions for correct versions to collect **the exception trigger stream in the passed execution**. Then, this pair of exception trigger streams are compared to identify the bifurcation point, and a vote-based strategy is used to determine suspicious statements based on execution traces. Last, EXPECT applies a tie-breaking strategy to distinguish statements with the same number of votes and deliver the final suspiciousness ranking list. Though there have been FL techniques driven by exception information, they typically target locating specific faults that lead to unhandled exceptions using traditional dynamic information [14, 19, 36, 46]. To the best of our knowledge, there is no previous study that uses exception trigger stream to localize faults.

For the evaluation, we obtain Gson, FastJson, and Jackson, three high-quality projects used in a former work of exception handling location recommendation learning [17], as well as Chart and Time, two commonly-adopted projects in the field of fault localization [21, 23, 27, 41, 47] (Gson and Jackson are also commonly-adopted for fault localization techniques). We inject mutated faults into these projects and generate 600 faulty versions. Experimental results show the competitiveness of EXPECT, it achieves as high as 38.26% improvements in localizing faults regarding the Exam metric in comparison to the SOTA fault localization technique, and it reduces the scales of ties in existing FL methods by up to 99.08%.

Experimental data and code are available in our repository [1], for any intention of replication or reuse.

This paper makes the following contributions:

[1]Chart: https://github.com/jfree/jfreechart
Time: https://github.com/JodaOrg/joda-time
MyBatis: https://github.com/mybatis/mybatis-3

[2]We use Java projects to conduct experiments, because Java is one of the most prevalent programming languages and has a well-designed exception handling mechanism.

- **A novel source of information introduced to fault localization tasks.** The main contribution of this work is to mine the potential of exception information in localizing faults. To the best of our knowledge, this is the first time that the exception information is directly linked with the fault location.
- **A promising fault localization technique based on exception trigger streams.** We utilize exception trigger information to form exception trigger streams, and further propose EXPECT, a novel fault localization technique based on exception information that is already at hand but typically ignored.
- **A comprehensive evaluation.** Evaluated on five popular benchmarks with well-recognized metrics, EXPECT shows significant improvements in both fault localization and tie-breaking tasks compared with the SOTA FL techniques to date.

2 BACKGROUND

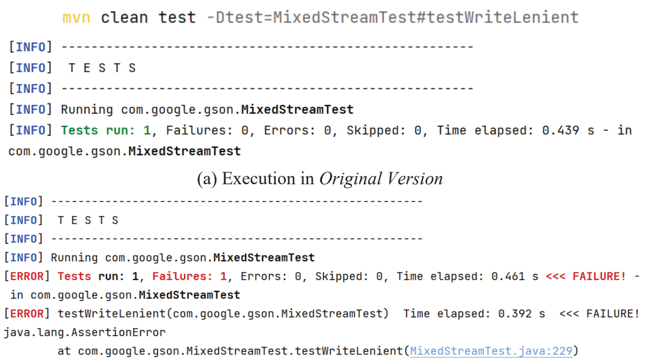
As mentioned above, exception is one kind of strategy typically used to deal with run-time events [9], the program can throw an exception for developers to catch and analyze the cause of that exception. Java exceptions are instances of the Throwable class, including checked exception, run-time exception, and errors [15, 34]. When an exception is thrown, it will be caught and handled by the nearest exception handling code, thus, **some of the best practices suggest developers use the try block in places where they believe abnormal execution status is more likely to be detected** [8, 15, 54], and the design of these critical checkpoints naturally reflects developers' understanding of the program [6, 32, 34]. The exception handling mechanism can help to improve the reliability of software programs, since it aims to report execution status while executing programs. The triggered-or-not behaviors of exception are lightweight messages to inspect the execution status, and it is intuitive to utilize such information for debugging.

3 PRELIMINARIES

3.1 Motivation Example

Seeing that more specific information that can describe the observed failures while having a relatively small scale has not been properly utilized, we propose to use exception information, since exception handling code is originally designed **as critical checkpoints** that inspect the execution status, and hence has good potential of providing hints for localizing faults.

We first carry out an exploratory study involving over 20 popular open-source Java projects to determine the specific type of information in try blocks to be collected and analyzed. We found out that programs generally have very diverse handling strategies in try blocks when an exception is thrown, such as recording logs, further throwing the exception, ignoring the exception, continuing to process logic, and so on. Actually, this finding is consistent with former studies [17, 34, 45]. Since our method must consider the information that can be generally obtained in any project having try blocks, it is obvious that the information extracted from a specific handling strategy is not a suitable candidate. Therefore, we choose to **only consider the triggered-or-not information in each try block** (i.e., whether the entire try block is executed with or without any exception thrown). Such information can be provided by any try block despite the specific exception handling practices, and thus **makes our method generally applicable**. In the following example, we will demonstrate **how to collect** such information, and **how it helps to locate** the fault.



```
mvn clean test -Dtest=MixedStreamTest#testWriteLenient
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.google.gson.MixedStreamTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.439 s - in
com.google.gson.MixedStreamTest
```

(a) Execution in *Original Version*

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.google.gson.MixedStreamTest
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.461 s <<< FAILURE! -
 in com.google.gson.MixedStreamTest
[ERROR] testWriteLenient(com.google.gson.MixedStreamTest)  Time elapsed: 0.392 s  <<< FAILURE!
java.lang.AssertionError
        at com.google.gson.MixedStreamTest.testWriteLenient(MixedStreamTest.java:229)
```

(b) Execution in *Faulty Version*

Figure 1: The execution results of *testWriteLenient*

We employ Gson[3] as the example program, which aims at converting Java objects and JSON strings to each other. Gson has approximately 5,000 executable statements in its core classes and over 1,000 test cases. We create a "*Faulty Version*" by mutating the operator "||" to "&&" in line 447 of the source file "*Gson.java*" (the unmutated version is referred to as "*Original Version*"). The mutated part of the source code can be found in Listing 1.

Listing 1: A *Faulty Version* of Gson

```
    ......
141 public final class Gson {
     ......
446  static void checkValidFloatingPoint(double value) {
       // Fault: "&&" should be "||"
447    if(Double.isNaN(value) && Double.isInfinite(value)){
448      throw new IllegalArgumentException(value+"...");}}
     ......}
```

This fault is revealed by the test case "*testWriteLenient*"[4]. It can be found from Figure 1 that this test case gives unexpected results. The "*Faulty Version*" fails to pass the test case with an assertion error, and the "*Original Version*" passes it.

Let us first consider applying traditional methods to locate the seeded fault, e.g, analyzing widely-used dynamic information such as coverage information or execution traces. According to our investigation, the length of the coverage vector of Gson is nearly 5,000, and the number of lines in the execution traces of "*testWriteLenient*" is nearly 2,000, which is on a large scale thus making the evidence towards the underlying fault somewhat difficult to analyze, and may thus weaken the effectiveness of fault localization.

Now we consider using the triggered-or-not information in try blocks to distinguish the two executions and further localize the fault. Our script found 75 try blocks in Gson (a much smaller scale

[3]The version of Gson we use is archived in our repository [1]. The latest version of Gson can be found in https://github.com/google/gson.
[4]The test case is provided in our repository [1], it could be accessed through https://github.com/mudcarofficial/EXPECT/tree/main/motivationExample&runningExample.

Table 1: Simplified exception trigger streams collected when running *testWriteLenient* in failed and passed executions

Checkpoints	Failed Execution in "*Faulty Version*"	Passed Execution in "*Original Version*"
1	...... Try Block#1 - not triggered	...... Try Block#1 - not triggered ......
2	...... Try Block#2 - not triggered	...... Try Block#2 - not triggered ......
3	...... Try Block#2 - not triggered	...... Try Block#2 - not triggered ......
4	...... Try Block#3 - not triggered	...... Try Block#3 - not triggered ......
5	...... Try Block#2 - not triggered	...... Try Block#2 - not triggered ......
6	...... Try Block#2 - not triggered	...... Try Block#2 - not triggered ......
	`if (Double.isNaN(value) && Double.isInfinite(value))`	`if (Double.isNaN(value) || Double.isInfinite(value))`
7	...... Try Block#3 - not triggered	...... Try Block#3 - triggered ......

than that of coverage information or execution traces). We track each try block a test execution enters and record whether all statements in the try block are successfully executed or not. Take the try block shown in Listing 2 as an example, in which a printing statement is inserted at the end of the try block. If a test execution enters this try block but does not print the "not triggered" statement, we consider that an exception is triggered in this try block, making the execution process immediately enter the catch block or the finally block, and we record the exception in this try block as "triggered". By listing such triggered-or-not information in each try block the current execution has entered, we can obtain **the exception trigger stream**, which partly shows the execution status at the covered critical checkpoints.

Listing 2: Try Block#3 (Line 842 of "*Gson.java*")

```java
try {
  adapter.write(writer, src);
  // The instrumented printing statement
  System.out.println("not triggered")
  } catch (IOException e) {
   throw new JsonIOException(e);
  } catch (AssertionError e) {
   throw new AssertionError("AssertionError(GSON " +
    GsonBuildConfig.VERSION + "): " + e.getMessage(),e);
  } finally { ...... }
```

The exception trigger streams of the test case "*testWriteLenient*" is given in Table 1, in which the second column is for the failed execution in "*Faulty Version*" and the third column is for the passed execution in "*Original Version*". The entire exception trigger streams of the test case "*testWriteLenient*" only involves three try blocks (i.e., Try Block#1, Try Block#2, and Try Block#3. Try Block#3 is shown in Listing 2, Try Block#1 and Try Block#2 are in our repository [1] due to the space limit), and seven critical checkpoints (a try block may be executed for multiple times, for each time the execution enters a try block, it is regarded as entering a unique checkpoint). At each checkpoint, we record the triggered-or-not information. For example, the information in Checkpoint 1 collected in passed execution tells that "*testWriteLenient*" first enters Try Block#1, and does not trigger any exception in this critical checkpoint. But Checkpoint 7 in the same execution tells that Try Block#3 is the seventh entered checkpoint and an exception is triggered in it.

We compare the detailed information of the two exception trigger streams. It can be noticed that a difference exists in Checkpoint 7: Try Block#3 catches an exception when the test case is running in the passed execution[5], while no exception is triggered by the same test case running in the failed execution. We refer to **the first checkpoint** having different triggered-or-not information in two exception trigger streams **as the bifurcation point.**

[5]Note that a thrown exception is not necessarily related to faults or failed executions in practice.

The bifurcation point reveals a different execution status captured in the corresponding critical checkpoint. Following the famous Propagation, Infection, and Execution (PIE) model [50], it is not difficult to conjecture that such a different status may be due to the execution of different statements prior to the bifurcation point. By further inspecting the statements executed before the bifurcation point, we find that the seeded fault in this example lies between the bifurcation point and the previous checkpoint (as shown in the line between Checkpoints 6 and 7 in Table 1). In our exploratory study, we generate another 40 faulty versions of Gson and extract the bifurcation points in each version using the same strategy, finding that in 80% of the versions, the faulty statements can be found between the bifurcation point and the closest checkpoint before the bifurcation point (relevant data of the exploratory study is available in our repository [1]). Thus we draw our inspiration to localizing faults with exception trigger streams: **In an execution trace, a faulty statement is conjectured to lie between the bifurcation point and its previous checkpoint,** because when a faulty statement is executed and triggers an abnormal status, the bifurcation point is the first checkpoint to observe such an error.

3.2 Challenges

Inspired by the above observation, we design EXPECT, an **EX**ce**P**tion triggE**r** stream-based fault lo**C**aliza**T**ion technique, to utilize the exception triggering information. Specifically, EXPECT first focuses on locating the bifurcation points between exception trigger streams collected in failed and passed executions, which will then be used to localize the faulty statement based on execution traces. To make EXPECT a feasible method in practice, there are still some challenges to be addressed. The first challenge is about "**what to compare**" in identifying the bifurcation point. In the above example, we generate mutants to simulate faults and compare the exception trigger streams on these mutants with those on the original version (i.e. the ground truth) to locate the faults. However, in real practices, such "ground truth" is missing, we can only collect the exception trigger stream in failed executions. So we need to address the challenge of "what to compare" to make the identification of the bifurcation point feasible.

Secondly, even if we have some solutions to provide versions for comparison, we still need to address the challenge of "**how to compare**" to identify the bifurcation point. Exception trigger streams of the two executions in the comparison do not always have the identical checkpoint sequence. We need to determine the strategy for identifying the bifurcation point in different scenarios.

Last but not least, the solution for the challenge of "**how to locate**" in assessing suspiciousness values for each statement needs to be well designed. There may be massive statements executed between checkpoints, and we may find multiple bifurcation points when there is more than one failed test case, so we need to design a method to get the final suspiciousness ranking based on bifurcation points and execution traces.

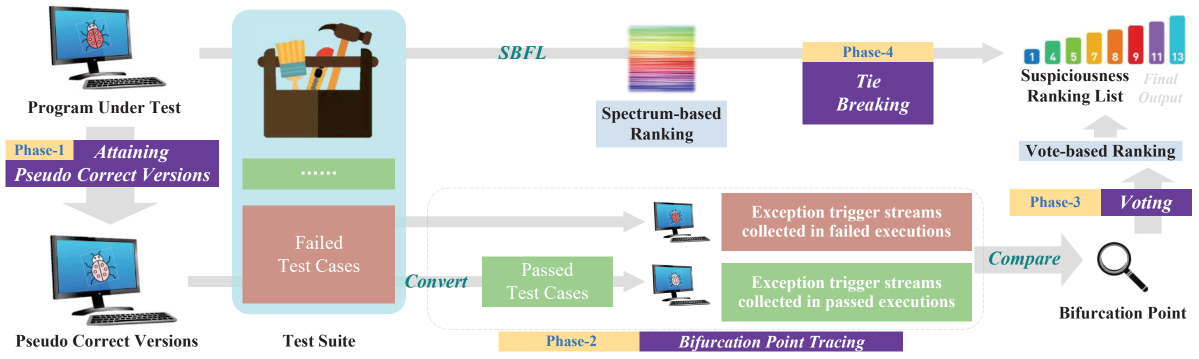

Program Under Test
Phase-1 Attaining Pseudo Correct Versions
Pseudo Correct Versions
Test Suite
Failed Test Cases
Convert
Passed Test Cases
SBFL
Spectrum-based Ranking
Phase-4 Tie Breaking
Suspiciousness Ranking List
Final Output
Vote-based Ranking
Phase-3 Voting
Exception trigger streams collected in failed executions
Exception trigger streams collected in passed executions
Compare
Bifurcation Point
Phase-2 Bifurcation Point Tracing

Figure 2: The overview of EXPECT

4 APPROACH

4.1 Overview

To tackle the first challenge of "what to compare", we get the pseudo correct versions of the program under test by utilizing history versions, different development branches, or mutants of the program, and collect the exception trigger streams while running the originally failed test cases on these versions. Details of this solution are elaborated in Section 4.2. For the second challenge that needs to locate the bifurcation points of each pair of exception trigger streams collected in failed and passed executions, we compare the exception trigger information in different situations. This is elaborated in Section 4.3. For the third challenge, to assess the suspiciousness of program statements, we propose a vote-based strategy combined with a tie-breaking tool. Please refer to Section 4.4 and Section 4.5 for the details. The overall flow of EXPECT is summarized in Figure 2, which consists of four main phases:

- **Phase-1:** EXPECT executes the test suite against the program under test, for each failed test case, attaining a pseudo correct version of the program that can turn the result of the test case into "passed". Such a version can be generated by different ways, and is used to collect the exception trigger stream of the corresponding test case in passed execution.
- **Phase-2:** EXPECT collects pairs of exception trigger streams while executing each failed test case on the program under test and its corresponding pseudo correct version, then comparing each pair of exception trigger streams and locating the bifurcation points.
- **Phase-3:** According to the bifurcation points, EXPECT employs a vote-based strategy to calculate a suspiciousness score for each program statement, for getting a vote-based ranking.
- **Phase-4:** EXPECT applies a tie-breaking tool on the vote-based ranking, deriving the final suspiciousness ranking.

As a reminder, EXPECT does not care whether an exception is triggered at a particular checkpoint **in an individual execution**. Instead, the difference between a passed and failed execution is the key information for locating the fault. This is because the mission of exception is mainly to deal with run-time events, rather than directly indicate static faults (e.g., even a fault-free version may also have exceptions being triggered). But the positions of try blocks are critical checkpoints to observe the execution status, and the difference between the execution status of a passed and a failed execution shall be helpful in locating the fault.

4.2 Attaining Pseudo Correct Versions

To address the first challenge of "what to compare" in Section 3.2, we propose to use "*Pseudo Correct Version*" as the substitute for the correct version, on which the test result of the failed test case is turned to "passed". As such, the exception trigger stream of the same test case collected in this "*Pseudo Correct Version*" can be used as the passed reference to locate the bifurcation point.

We first define the concept and justify the use of "*Pseudo Correct Versions*". Revisit the example in Section 3.1, in which the "*Original Version*" as the ground truth is generally unavailable in practice. In other words, we do not have a version to provide an exception trigger stream as a passed reference to compare with. Therefore, we create a "*Substitute Version*" by seeding a different fault than the fault in Listing 1, in the original "*Gson.java*" file. We require that the test case "*testWriteLenient*" passes on this "*Substitute Version*". An interesting finding is that the exception trigger stream on this "*Substitute Version*" **is identical** to the one on the "*Original Version*". In other words, though this "*Substitute Version*" is not a real correct version (because we have no way to obtain such a correct version in real practice), at least it does not contain the previous target fault (i.e. the fault revealed by the failed test case). In contrast, for the previous target fault, the "*Substitute Version*" approximates the real correct version, in terms of both the testing result and the exception trigger stream.

Inspired by this phenomenon, we further conduct the same attempt on 40 randomly generated faulty versions and find that for 95.65% of the failed test cases, we can easily obtain such a "*Substitute Version*", on which the failed test cases become passed, and more importantly, **the corresponding exception trigger streams are identical to that in the "*Original Version*" (i.e. the ground truth)**. This finding justifies that such a "*Substitute Version*" is promising to replace the unavailable correct version. In the following discussion, **we refer to a "*Substitute Version*" as a "*Pseudo Correct Version*",** which is formally defined as: a different version of the program under test that is capable of converting the failed test case to a passed one. Relevant data of this section can be found in our repository [1].

To get pseudo correct versions, we suggest utilizing history versions or different development branches of the program in which the execution results of originally failed test cases are "passed". History versions of a program are usually preserved during the development process for potential needs of analysis and rollback, these versions may not contain the fault caused by later code changes. Different development branches are typically produced when different developers begin to change functions based on the same program version while needing to avoid interference, the fault introduced by one branch may not exist in other ones. History versions and different development branches are usually not difficult to attain in real-world development environments (e.g., Gson has more than 2,000 history versions, FastJson has more than 4,000 history versions and 10 active branches) and need no extra costs to generate. For pseudo correct versions that can not be attained through the aforementioned ways, they can be generated by applying a mutation-based technique and generating mutants of the program until the mutant that reverses the result of the failed test case shows up, such a mutant can serve as the pseudo correct version.

Failed test cases in a program are denoted as f_i (i = 1, 2, ..., n), where n is the number of failed test cases. For each failed test case f_i, EXPECT generates an pseudo correct version that can convert the execution result of f_i to pass, we refer to this version as pcv_i. The execution of f_i on the original program under test and pcv_i are denoted as e_i and pe_i, respectively. After obtaining pcv_i of each f_i, EXPECT collects exception trigger streams of e_i and pe_i, which can be denoted as ets_fail_i and ets_pass_i, respectively. The collection of exception trigger streams is conducted based on source code instrumentation.

Note that it is not reasonable to locate the fault by simply comparing traditional dynamic information (such as coverage information and dynamic slices) between the program under test and the pseudo correct version, because such information in pseudo correct versions has significant differences compared with that in correct versions, and thus can not be used to replace information in the correct version. This issue is further discussed in Section 7.2.

4.3 Bifurcation Point Tracing

In this section, we first give definitions of exception trigger streams in EXPECT. A whole exception trigger stream is gathered during the execution of a test case and **consists of two parts of information: the triggered-or-not status in try blocks and the execution traces of program statements.** The execution traces are only used to vote for statements in Section 4.4. When running a test case, we collect each statement it executes as well as the exception information in each checkpoint it enters, these two types of information are organized in the order of execution and form the exception trigger stream, as shown in Table 2. We use the try block in Listing 2 as an example to assist in the elaboration of symbol definitions.

Formula 1 shows the structure of a checkpoint. *Begin* marks the entrance of a checkpoint, while *End* marks the exit of it. In Listing 2, *Begin* marks the execution of code line 2, *End* marks the execution of code lines 4, 6, or 8. The same try block may be executed more than once, we use $Identifier$ to describe a unique checkpoint. For information begins with *End*, it contains $Triggered$ telling whether an exception is triggered in the corresponding checkpoint (0 for not triggered, 1 for triggered). The $Identifier$ field and the $Triggered$ field together represent the execution status at each checkpoint. In addition to exception trigger information, a whole exception trigger stream also includes execution traces of statements. Formula 2 shows the structure of execution traces. Each group of traces contains the statements executed between the bounds of checkpoints (the *Begin* and the *End* of a checkpoint are regarded as its bounds). $Begin/End : Identifier$ represents that before which bound is the group of traces collected. For example, in line 3 of Table 2, $Statements\_Begin : 2$ tells that these statements are executed just before the entrance of a checkpoint, and the $Identifier$ field of that checkpoint is 2. Note that the last line of an exception trigger stream is specifically defined as $Statements\_Ending$, since no try block is entered after executing these statements. Assuming that the try block in Listing 2 is entered once by a test case and the exit of this checkpoint is represented as $End : 1\_0$, which means that no exception is triggered in this checkpoint, then the execution of code line 2 would be recorded and contained in $Statements\_End : 1$, since it happens just before the exit of the checkpoint.

Table 2: The structure of exception trigger stream

	Exception trigger stream
1	Statements_Begin_1
2	Begin:1
3	Statements_Begin_2
4	Begin:2
5	Statements_End_2
6	End:2_Triggered
7	Statements_End_1
8	End:1_Triggered
	......
n	Statements_Ending

$$Begin/End : Identifier(\_Triggered) \tag{1}$$

$$Statements\_Begin/End : Identifier \tag{2}$$

We denote the bifurcation point of f_i as bp_i, which is determined by analyzing ets_fail_i (obtained while executing f_i on the original program under test) and ets_pass_i (obtained while executing f_i on the pseudo correct version), the process of which begins with comparing the execution status at the gathered checkpoints. As introduced above, checkpoints are gathered when executing f_i and thus organized in the order of execution in each exception trigger stream. When comparing the execution status at checkpoints between ets_fail_i and ets_pass_i, we align them according to their order in the exception trigger stream and compare them in pairs. The execution status at the n^{th} gathered checkpoint in ets_fail_i would be compared with that at the n^{th} gathered checkpoint in ets_pass_i.

To compare the execution status at a pair of checkpoints, we focus on the information collected at the exit of them (i.e., information starts with *End* according to Formula 1). EXPECT judges the existence of different execution status by comparing the $Identifier$ and $Triggered$ fields of Formula 1. If the $Identifier$ field is not exactly the same, the current pair of checkpoints are gathered in different try blocks, and if the $Triggered$ field is not exactly the same, the triggered-or-not behaviors in the two checkpoints are different. If any of these two fields show a difference, the execution status at corresponding checkpoints will be regarded as different.

After comparing execution status at checkpoints in ets_fail_i and ets_pass_i in pairs, there are two different situations. If EXPECT finds a pair of checkpoints that have different execution status, we have Situation-1 (Checkpoints with different execution status captured). Otherwise, we have Situation-2 (Checkpoints with different execution status not captured).

Situation-1: Checkpoints with different execution status captured. Among all pairs of checkpoints that have different execution status, the pair that is earliest gathered when executing f_i is determined as the bifurcation point. For example, if the first gathered checkpoints and the second gathered checkpoints both have different execution status, the first gathered checkpoints would be determined as the bifurcation point, since they are gathered earlier when executing f_i on the original program under test and pcv_i. Considering that the numbers of checkpoints in the two exception trigger streams may be different, we only compare the execution status at the first $min\_checkpoint_i$ pairs of checkpoints, in which $min\_checkpoint_i$ stands for the smaller number of checkpoints in ets_fail_i and ets_pass_i.

If EXPECT can not find any difference in the execution status at the first $min\_checkpoint_i$ pairs of checkpoints, we seek to determine the bifurcation point according to **the numbers of checkpoints** in ets_fail_i and ets_pass_i, which will be described in Situation-2.

Situation-2: Checkpoints with different execution status not captured. There are five possible sub-situations according to the number of checkpoints in ets_fail_i and ets_pass_i, we discuss each of them below.

Situation-2.1: Same number of checkpoints. If the numbers of checkpoints in ets_fail_i and ets_pass_i are also equivalent, we think that exception information of f_i collected in passed and failed executions is insufficient to find the different execution status. No bifurcation point of f_i is found in this situation, and f_i will not be used to localize the fault in later phases.

Situation-2.2: Fewer checkpoints in the failed execution. If there are fewer checkpoints in ets_fail_i, the last checkpoint in ets_fail_i is regarded as the bifurcation point. Though there is no explicit difference in the execution status at the bifurcation point, it is conjectured that e_i shows a different execution status after meeting this checkpoint, as pe_i gathers more checkpoints after the bifurcation point.

Situation-2.3: More checkpoints in the failed execution. If there are more checkpoints in ets_fail_i, the last pair of checkpoints compared in the former process (i.e., the $min\_checkpoint_i^{\text{th}}$ gathered checkpoints) are regarded as the bifurcation point. Similar to the previous situation, we conjecture that e_i shows a different execution status by entering another checkpoint after meeting the bifurcation point, while there are no more checkpoints after the bifurcation point in ets_pass_i.

Situation-2.4: No checkpoint in the failed execution. If only ets_pass_i contains checkpoints, EXPECT can not determine a checkpoint in ets_fail_i as the bifurcation point. However, the difference indeed exists and we will handle this sub-situation in Section 4.4.

Situation-2.5: No checkpoint in the passed execution. If only ets_fail_i contains checkpoints, we do not regard any checkpoint as the bifurcation point, for no checkpoint is the exact location that ets_fail_i and ets_pass_i have different execution status. The checkpoints in ets_fail_i are entered while e_i is already having a different execution status compared with pe_i, thus, these checkpoints should not be seen as the bifurcation point. This sub-situation will be handled in Section 4.4.

4.4 Voting

In this phase, each f_i votes for suspicious statements according to its corresponding bifurcation point bp_i and the situation it belongs to. The voting strategy of each situation (except for situation-2.1 because it is not utilized in the voting phase as discussed above) is described below. To facilitate comprehension, Table 3 gives examples the voting strategies, in which the bifurcation points and the range of statements to be voted for are highlighted.

Situation-1: Checkpoints with different execution status captured. Statements executed between bp_i and the previous checkpoint are voted, for they are more likely to contain the fault that causes the different execution status captured by bp_i.

Situation-2: Checkpoints with different execution status not captured. This situation consists of the following sub-situations.

Situation-2.2: Fewer checkpoints in the failed execution. Besides voting for statements executed between bp_i and the previous checkpoint, considering that there is a possibility that the fault lies after bp_i, to make our method more effective, we also vote for statements executed after bp_i.

Situation-2.3: More checkpoints in the failed execution. The strategy is mostly the same as that of Sub-Situation-2.2, the only difference lies in the way of voting for statements executed after bp_i. In this situation, ets_fail_i contains more checkpoints, and we only let f_i vote for statements executed between bp_i and the checkpoint immediately following it.

Situation-2.4: No checkpoint in the failed execution. If only ets_fail_i contains no checkpoint, it is conjectured that the execution of e_i has a different execution status before entering a checkpoint, thus, all statements executed in e_i are voted.

Situation-2.5: No checkpoint in the passed execution. If only ets_fail_i contains checkpoints, we infer that e_i has a different execution status before entering the first checkpoint, thus, statements executed before the first checkpoint are voted.

Note that try blocks may have nested relationships, more formally, giving two try blocks A and B, if the execution enters B after entering A and exits B before exiting A, we regard the try block B as being nested within the try block A. When the bifurcation point is found, any statement inside the corresponding try block may be the root cause of the captured different execution status, and we vote for all statements inside the try block no matter whether there exist other nested try blocks. Refer to the example in Table 2, if the bifurcation point is found in line 8, all statements in lines 3, 5, and 7 will be voted. This strategy is necessary and intuitive, it helps to vote for all statements that have a relatively high possibility to be the source of the bug.

Each f_i casts one vote for each unique statement within its voting range. The number of votes each statement gets is calculated by adding up the number of failed test cases that vote for it, as shown in Formula 3 and Formula 4, in which s represents a unique statement. We rank all statements of the program under test according to the number of votes they receive, statements with more votes are

Table 3: Voting strategies in different situations

	Situation-1		Sub-Situation-2.2		Sub-Situation-2.3		Sub-Situation-2.4		Sub-Situation-2.5	
	ets_pass$_i$	*ets_fail$_i$*	*ets_pass$_i$*	*ets_fail$_i$*	*ets_pass$_i$*	*ets_fail$_i$*	*ets_pass$_i$*	*ets_fail$_i$*	*ets_pass$_i$*	*ets_fail$_i$*
1	Statements_Begin_1	Statements_Begin_1	Statements_Begin_1	Statements_Begin_1	Statements_Begin_1	Statements_Begin_1	Statements_Begin_1	Statements_Ending	Statements_Ending	Statements_Begin_1
2	Begin:1	Begin:1	Begin:1	Begin:1	Begin:1	Begin:1	Begin:1			Begin:1
3	Statements_End_1	Statements_End_1	Statements_End_1	Statements_End_1	Statements_End_1	Statements_End_1	Statements_End_1			Statements_End_1
4	End:1_0	End:1_0	End:1_0	End:1_0	End:1_0	End:1_0	End:1_0			End:1_0
5	Statements_Begin_2	Statements_Begin_2	Statements_Begin_2	Statements_Ending	Statements_Ending	Statements_Begin_2	Statements_Ending			Statements_Ending
6	Begin:2	Begin:2	Begin:2			Begin:2				
7	Statements_End_2	Statements_End_2	Statements_End_2			Statements_End_2				
8	End:2_0	End:2_1	End:2_0			End:2_0				
9	Statements_Ending	Statements_Ending	Statements_Ending			Statements_Ending				

regarded as more suspicious. The output of this phase is a vote-based ranking, as depicted in Figure 2.

$$NumOfVotes_s = \sum_{i=1}^{n} Vote_s^i \tag{3}$$

$$Vote_s^i = \begin{cases} 0 & if\ f_i\ not\ voting\ for\ Statement_s \\ 1 & if\ f_i\ is\ voting\ for\ Statement_s \end{cases} \tag{4}$$

4.5 Tie-breaking

In previous phases, the same group of statements executed between two checkpoints may not be further distinguished, resulting in a considerable tie. Identifying the faulty statement as one of the most suspicious while having plenty of statements equally ranked is not acceptable, since the outputs of fault localization techniques are usually provided for developers as guidelines, a big tie can seriously weaken the effectiveness of our method. To solve this problem, we employ a suspiciousness evaluation method as the tie-breaking tool for EXPECT, any fault localization method that can evaluate the suspiciousness of statements can be employed. Here we simply apply the widely used SBFL techniques, which separately calculate the suspiciousness values for each program statement. Though not promised to deliver outputs with high effectiveness, these techniques can serve as tools for breaking ties introduced by previous phases in EXPECT.

We apply SBFL techniques to calculate the suspiciousness values for each statement. In the final suspiciousness ranking, statements are first ranked by the number of votes they receive, then, for statements indistinguishable according to the votes, they are further ranked by the calculated suspiciousness scores. This tie-breaking process combines the advantages of EXPECT and other traditional techniques, and can deliver highly effective and fine-grained results.

For smooth understanding, we give a running example comprising the mentioned steps of EXPECT in our repository [1].

5 EXPERIMENTAL SETUP

5.1 Research Questions

- **RQ1: Competitiveness of EXPECT.** Does EXPECT outperform the state-of-the-art technique SmartFL in terms of fault localization effectiveness?
- **RQ2: The performance of EXPECT in breaking ties.** Fault localization techniques may bring ties when estimating the suspiciousness of program elements. How much does EXPECT contribute to resolving these ties?
- **RQ3: Ablation Analysis.** The voting phase and the tie-breaking phase are two key components of EXPECT. How does each of them impact the fault localization effectiveness of EXPECT?
- **RQ4: Impact of different voting strategies.** In principle, there can be various voting strategies. Which one is more effective in voting for the faulty statement precisely?

5.2 Parameter Setting

In our experiment, we apply Crosstab [57], Ochiai [2], DStar [56], and Naish2 [38] (four of the most effective SBFL techniques) to EXPECT's tie-breaking component, to form four different configurations, namely, $EXPECT_C$, $EXPECT_O$, $EXPECT_D$, and $EXPECT_N$, respectively. To gather exception trigger streams in the passed executions, we find pcv_i for each f_i from other faulty versions, which can be regarded as different development branches and mutations. Such a choice is flexible and can be adjusted depending on specific situations.

5.3 Datasets

EXPECT has a requirement of exception handling code that a program can provide, if there are almost no try blocks, the effectiveness of EXPECT would be weakened. To better evaluate the competitiveness of EXPECT, we obtain Gson, FastJson, and Jackson that are used for exception handling code recommendation learning in a former work [17] and have approximately 75, 745, and 56 try blocks, respectively. We also select Chart and Time from Defects4J which are commonly used for evaluating fault localization techniques [21, 23, 27, 41, 47] and have 42 and 75 try blocks, respectively (Gson and Jackson are also projects in Defects4J). More details of project selection are provided in Section 7.1.

Though Defects4J provides real-world bugs, the versions of the buggy programs are rather old, each requiring different versions of dependencies and JDK (the latest JDK version supported is 1.8, while recently released versions of selected projects require at least JDK 11). Considering that we need to configure the programs and collect the exception trigger information (which is not originally provided in the pre-configured framework of Defects4J) in the framework of EXPECT, reproducing these bugs would need massive human efforts to handle the configuration and compatibility issues in each version. Thus, we randomly inject mutated operators into recently released versions of selected projects and create 600 faulty versions as the datasets for our experiment[6].

[6]Mutation-based faults have been demonstrated to be capable of simulating real-world faults and thus providing reliable and trustworthy results for testing and debugging experiments, they have been widely used in former studies [4, 5, 12, 13, 31, 48]. The script to create mutations can be found in our repository [1].

Table 4: Comparison with the SOTA technique in effectiveness

Techniques	Exam_Best		Exam_Average		Exam_Worst		MRR		WSR_NB	WSR_B
$SmartFL$	28.88		28.88		28.88		0.2705		╲	╲
$EXPECT_C$	18.31	**36.60%↑**	21.40	**25.90%↑**	24.48	**15.24%↑**	0.7167	**164.95%↑**	**1.18E-108**	**1.20E-98**
$EXPECT_O$	18.19	**37.02%↑**	21.32	**26.18%↑**	24.45	**15.34%↑**	0.7173	**165.18%↑**	**7.12E-109**	**7.29E-99**
$EXPECT_D$	17.83	**38.26%↑**	20.94	**27.49%↑**	24.06	**16.69%↑**	0.7171	**165.10%↑**	**1.18E-108**	**1.20E-98**
$EXPECT_N$	18.06	**37.47%↑**	21.14	**26.80%↑**	24.23	**16.10%↑**	0.7033	**160.00%↑**	**9.60E-110**	**1.62E-99**

5.4 Metrics and Environments

We adopt three metrics widely used in previous studies to evaluate the performance of EXPECT, including Exam [23, 26, 33, 70], Mean Reciprocal Rank (MRR) [10, 29, 35, 52], and Wilcoxon Signed-Rank tests (WSR) [26, 56, 62].

Exam calculates the number of statements to be examined before finding the fault. We consider three types of the Exam score: Best (Exam_Best), Average (Exam_Avg), and Worst (Exam_Worst). In Exam_Best and Exam_Worst, The faulty statement is assumed to be checked before/after all other statements with the same suspiciousness value, while in Exam_Avg, the rank of the fault is the average rank of statements equally suspicious. Note that a lower Exam score represents a better performance.

The MRR metric [51] is defined as the mean of the reciprocal position at which the first relevant element is found. In our experiment, it is calculated by Formula 5, in which K represents the number of faulty versions and $rank_i$ is the rank of the fault in version i. We apply the Best strategy for the MRR metric. A higher MRR score represents a better performance.

$$MRR = \frac{1}{K} \sum_{i=1}^{K} \frac{1}{rank_i} \tag{5}$$

WSR [53] is used to evaluate the statistical significance of our experimental results, i.e., whether EXPECT outperforms existing techniques significantly. We design two one-tailed alternative hypothesis, NB (Not Bad) and B (Better). The NB is that the baseline techniques require equal or greater number of statements to be checked than EXPECT before localizing the fault, and the B is that the baselines need a strictly greater number of statements to be checked. We also use the Best strategy for this metric.

Our experiment is based on Ubuntu 16.04.1 LTS with JDK 17.

6 RESULT AND ANALYSIS

6.1 RQ1: Competitiveness of EXPECT

To evaluate the competitiveness of EXPECT, **we compare its fault localization effectiveness with the state-of-the-art technique, SmartFL [67]**, which has been demonstrated to outperform the existing SBFL and MBFL techniques.

The experimental results are shown in Table 4, it can be seen that EXPECT configured with different suspiciouness evaluation methods all outperform the baseline significantly. Table 4 gives the improvement in different metrics made by EXPECT. Specifically, for the Exam metric, EXPECT improves at least 15.24% in all conditions, and it achieves over 20% improvement on average in the Average and Worst conditions. For the MRR metric, EXPECT improves at least 160%. Besides, EXPECT is more lightweight compared with SmartFL, which suffers from high learning costs, further showing the superiority of our method.

Table 5: EXPECT's performance in breaking ties

M	Crosstab	Ochiai	DStar	Naish2
Original FL methods M	90.75	680.21	680.08	90.75
$EXPECT_M$	6.17	6.26	6.23	6.17
Improvement	**93.20%**	**99.08%**	**99.08%**	**93.20%**

For the WSR tests, Table 4 shows the p-values of the NB and B alternative hypotheses. The results support us in rejecting the two corresponding null hypotheses, and concluding that EXPECT is more effective than the baseline from the perspective of the statistics.

To summarize, EXPECT can obviously outperform the SOTA technique in the tasks of ranking the faulty statement at a high position. Our experiment shows the competitiveness of EXPECT, and it also highlights the feasibility and importance of utilizing exception information in fault localization tasks.

6.2 RQ2: The Performance of EXPECT in Breaking Ties

Tie-breaking is an essential job in fault localization, because the identical risk values of the faulty statement and innocent statements could largely degrade the effectiveness of localization. In order to evaluate how well EXPECT can alleviate the problem of tie in existing fault localization techniques with such a problem, we compare $EXPECT_C$, $EXPECT_O$, $EXPECT_D$, and $EXPECT_N$ with the original FL methods, namely Crosstab, Ochiai, DStar, and Naish2, respectively.

The second row of Table 5 is the average number of statements having the same suspiciousness value as the faulty statement given by original FL methods M (M takes Crosstab, Ochiai, DStar, and Naish2). It can be seen that all the original FL methods suffer from ties seriously. The average numbers of statements having the same suspiciousness value as the faulty statement given by $EXPECT_M$ (configure each technique M as the tie-breaking tool for EXPECT) are in the third row of Table 5. The results show promising performance of EXPECT in breaking ties: compared with the scales of ties in these original FL methods M, the scales of ties in $EXPECT_M$ after the tie-breaking phase is rather low, the improvement reaches up to 99.08%. This phenomenon shows that the strategy of conducting fault localization based on exception information and introducing a tie-breaking phase is capable of effectively alleviating the problem of ties.

As a reminder, even though SmartFL gives fine-grained fault localization results with almost no tie, EXPECT still outperforms it in the Exam_Worst metric (as demonstrated in RQ1). That is to say, developers need to check fewer statements before locating the fault using EXPECT than using SmartFL even if the impact of

Table 6: Improvement with the voting phase

Techniques	Exam_Best		Exam_Average		Exam_Worst		MRR		WSR_NB	WSR_B
$NoVote_C$	309.15	**94.08%**↑	354.53	**93.96%**↑	399.90	**93.88%**↑	0.6786	**5.62%**↑	**1.90E-128**	**1.92E-35**
$EXPECT_C$	18.31		21.40		24.48		0.7167			
$NoVote_O$	66.94	**72.82%**↑	407.05	**94.76%**↑	747.15	**96.73%**↑	0.6793	**5.60%**↑	**1.90E-128**	**1.92E-35**
$EXPECT_O$	18.19		21.32		24.45		0.7173			
$NoVote_D$	70.01	**74.53%**↑	410.05	**94.89%**↑	750.09	**96.79%**↑	0.6832	**4.97%**↑	**1.90E-128**	**5.24E-35**
$EXPECT_D$	17.83		20.94		24.06		0.7171			
$NoVote_N$	339.36	**94.68%**↑	384.74	**94.50%**↑	430.11	**94.37%**↑	0.6788	**3.61%**↑	**3.10E-128**	**2.94E-33**
$EXPECT_N$	18.06		21.14		24.23		0.7033			

Table 7: Improvement with the tie-breaking phase

Techniques	Exam_Best		Exam_Average			Exam_Worst		
NoTie-Breaking	1.34		1604.48			3207.62		
$EXPECT_C$	18.31	↓	21.40	**98.67%**	↑	24.48	**99.24%**	↑
$EXPECT_O$	18.19	↓	21.32	**98.67%**	↑	24.45	**99.24%**	↑
$EXPECT_D$	17.83	↓	20.94	**98.67%**	↑	24.06	**99.25%**	↑
$EXPECT_N$	18.06	↓	21.14	**98.67%**	↑	24.23	**99.24%**	↑

Table 8: Comparison of different voting strategies

Voting strategies	Exam_Best	Exam_Average	Exam_Worst
"Original"	1.34	1604.48	3207.62
Vote_3	1.18	1656.72	3312.26
Vote_5	**0.70**	1610.97	3221.24
Vote_distance	1.35	**1529.18**	**3057.01**
Vote_interval	98.17	1646.77	3195.36

ties is considered in the Worst condition, which further shows the advantage of EXPECT.

6.3 RQ3: Ablation Analysis

We perform an ablation analysis to investigate the impact of each of the two essential components, i.e., the voting phase and the tie-breaking phase, on EXPECT's fault localization capability.

First, we ablate the voting component of EXPECT and then compare this variant (referred to as NoVote) with the original EXPECT. The result is given in Table 6, it can be seen that the original EXPECT significantly outperforms NoVote, it improves at least 72.82% in the Exam_Best metric, as well as at least 96.79% in the Exam_Worst metric. Considering the WSR tests, it is also reasonable for us to reject the two corresponding null hypotheses and conclude that the original EXPECT is more effective than the variant in which the voting phase is ablated. These data show that the voting phase makes a critical contribution.

Then, we ablate the tie-breaking component of EXPECT and compare this variant (referred to as NoTie-Breaking) with the original EXPECT. We use the Exam metric to compare these methods in different conditions (i.e., Best, Average, and Worst). The result is given in Table 7, it can be seen that the original EXPECT improves significantly in the Average and Worst conditions. Note that the Best condition ignores the influence of ties, it may not properly evaluate the performance of FL methods when there are serious ties, for example, when all statements are ranked equally, the score of the Exam_Best metric would be 0, while such a ranking list is meaningless. Thus, the original EXPECT is still more effective in practice with alleviated ties. These data show that the tie-breaking phase helps to deliver finer-grained results.

In conclusion, the voting phase of EXPECT plays a key role in effectively localizing faults, while the tie-breaking phase contributes to giving fine-grained results.

6.4 RQ4: Impact of Different Voting Strategies

We design four alternative voting strategies to be compared with the original strategy used in EXPECT (referred to as "Original"): Vote_3, Vote_5, Vote_distance, and Vote_interval. Vote_3 and Vote_5 expand the selection of statements to be voted and consider three and five checkpoints above the bifurcation point instead of only one. Vote_distance and Vote_interval are based on Vote_5. Specifically, Vote_distance adjusts the number of votes for each statement, the more checkpoints there are between the statement and the bifurcation point, the fewer votes it receives. Vote_interval adjusts the number of votes according to the statement's frequency of occurrence in different intervals between pairs of neighbor checkpoints. In this research question, we only consider the vote-based ranking before the tie-breaking phase for a more proper comparison.

The results are given in Table 8. It can be seen that serious ties exist in all strategies, which is mainly due to the statistical impact introduced by faulty versions in which the bifurcation point is not found. For example, 71,396 executable statements are gathered in FastJson, if all statements receive 0 votes, the tie would be 71,395, which is anomalously huge. A similar problem also exists in SBFL techniques, by utilizing SBFL techniques as the tie-breaking tools, ties in most versions are effectively resolved, as discussed in Section 6.2.

Compared with "Original", Vote_5 is more effective in the Best situation, which may be attributed to its expanded voting range. Vote_distance performs better in the Average and the Worst situations, which may be attributed to its complicated voting strategy. To alleviate the statistical impact mentioned above, we further compare them in 300 randomly chosen faulty versions excluding those with ties of more than 10,000 statements, the result is that Vote_5 gets 240.05 and 478.75 Exam scores in the Average and the Worst situations, while "Original" and Vote_distance get 190.11 and 378.6, as well as 174.54 and 347.57 scores, respectively. Considering that Vote_5 and Vote_distance are more complicated and bring extra costs, "Original" is better for its balanced performance in different metrics as well as relatively low time costs and learning costs.

7 DISCUSSION

7.1 Eligibility of Programs for Using EXPECT

The success of EXPECT depends on exception trigger information collected in exception handling code, which highlights the importance of try blocks. If programs have few try blocks, the effectiveness of EXPECT on them could be weakened, because it could be hard to distinguish exception trigger streams collected in passed and failed executions, and further find the bifurcation point in such a scenario. In our experiments, we first observe the number of try blocks in alternative projects, programs with too few try blocks are filtered in this process (for example, the source files in the core package of Math have 12,363 lines of code with no try block, thus we do not use Math for experiment).

But in our opinion, the need for try blocks will not remarkably hinder EXPECT's practicability. This is because **well-designed exception handling code is demanded for high-quality software** [8, 15, 54], and more and more developers are dedicating their effort to inserting proper exception handling code into programs. This trend supports the smooth running of EXPECT and provides it with a promising implementation environment in the future.

7.2 Revisit the Usage of Pseudo Correct Versions

In Section 4.2, we use pseudo correct versions as substitutions for correct versions to collect the exception trigger streams in passed executions. It may seem intuitive to consider comparing some widely used dynamic information collected in the program under test and pseudo correct versions to pinpoint the fault. However, such a strategy is not feasible. Unlike the exception trigger information in the pseudo correct versions and the correct version, traditional information such as coverage and execution traces can be very different between pseudo correct versions and the correct version. We conjecture that this phenomenon may be due to the characteristics of these types of traditional dynamic information, for they can be easily changed by any code modifications. On the contrary, exception information is a good inspector for execution status, it may not be easily disturbed by code changes unless the code change brings an abnormal execution status.

As an example, refer to the complete exception trigger streams of "*Substitute Version*" and "*Original Version*" provided in our repository [1], though the exception trigger information in two versions is identical, the execution traces are different between the two versions, in this scenario, comparing the execution traces in "*Faulty Version*" and "*Substitute Version*" can not localize the fault effectively. Similar phenomena can be found in many versions of our exploratory study, which is also provided in our repository [1].

8 THREATS TO VALIDITY

The first threat to validity is about the open-source Java projects chosen as our experimental datasets. Though projects used in our experiments are representative and popular, we have to filter projects with too little exception handling code. When there are too few try blocks in the program under test, EXPECT may fail to find the bifurcation point and vote for any statement. In such a scenario, the effectiveness of EXPECT would be reduced to that of the tie-breaking tool. The second threat to validity is about the metrics we use to evaluate the performance of EXPECT. We apply three widely-used metrics, i.e., Exam, MRR, and WSR tests, but there may be more metrics that are capable of evaluating EXPECT from different perspectives. In the future, we plan to use more metrics in our experiment for a more robust evaluation.

9 RELATED WORK

There are a large number of automatic fault localization approaches using dynamic information in the literature. Among them, SBFL is regarded as one of the most representative techniques. It collects the coverage information of test cases, followed by analyzing the differences in program spectra and allocating suspiciousness values for each program element [16, 44]. Different risk evaluation formulas responsible for this process are proposed [2, 11, 20, 38, 56, 57], some of which are generated by machine learning techniques [65]. Considering not only the original faulty program, MBFL techniques generate mutants of the program to obtain more available information by observing the behaviors of mutants [37, 40, 68]. Apart from the coverage information and test results, there are more types of information introduced to the fault localization tasks. For example, the values of variables are typically used to find out bug-related variables for finer-grained fault localization [18, 23], the information provided by stack traces is collected for their abilities to indicate the currently active function calls and the point where the crash or bug occurred [55, 60], and dynamic program slicing is employed to focus on a particular program execution and reduce the scale of suspicious statements [3, 43]. Nowadays, machine learning algorithms are also widely applied to combine outputs of traditional fault localization techniques and different sources of information [25, 27, 28, 70]. Among the techniques based on dynamic information, SmartFL published in 2022 has been proven to be one of the most advanced techniques so far [67]. In this work, we propose EXPECT, which is demonstrated to significantly outperform this SOTA technique in fault localization.

10 CONCLUSION

In this paper, we propose EXPECT, a novel method that combines exception trigger information and execution traces to localize faults. EXPECT first collects exception trigger streams in passed and failed executions of the program under test, then votes for suspicious statements according to bifurcation points that capture the different execution status between passed and failed executions. Finally, a tie-breaking tool is applied to the statements with the identical risk value. The experiments demonstrate the competitiveness of EXPECT: Compared with the state-of-the-art technique, EXPECT achieves as high as 38.26% improvements in localizing faults regarding the Exam metric. Besides, it reduces the scales of ties in existing FL methods by up to 99.08%.

In the future, we plan to conduct research on open-source projects to better exploit the exception information. Besides, we are also interested in automatically recommending try blocks to get more exception information for analysis.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62250610224 and 62202344.

REFERENCES

[1] 2024. Repository of EXPECT. https://github.com/mudcarofficial/EXPECT

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing*. 39–46.

[3] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* 143–151.

[4] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th International Conference on Software Engineering.* 402–411.

[5] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[6] Eiji Adachi Barbosa and Alessandro Garcia. 2018. Global-aware recommendations for repairing violations in exception handling. In *Proceedings of the 40th International Conference on Software Engineering.* 858–858.

[7] Nazanin Bayati Chaleshtari and Saeed Parsa. 2020. SMBFL: slice-based cost reduction of mutation-based fault localization. *Empirical Software Engineering* 25 (2020), 4282–4314.

[8] Joshua Bloch. 2008. *Effective java.* Addison-Wesley Professional.

[9] Bruno Cabral and Paulo Marques. 2007. Exception handling: A field study in java and. net. In *ECOOP 2007–Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30-August 3, 2007. Proceedings 21.* Springer, 151–175.

[10] An Ran Chen, Tse-Hsun Chen, and Junjie Chen. 2022. How Useful is Code Change Information for Fault Localization in Continuous Integration?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 1–12.

[11] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks.* 595–604.

[12] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

[13] Ruizhi Gao and W Eric Wong. 2018. MSeer: an advanced technique for locating multiple bugs in parallel. In *Proceedings of the 40th International Conference on Software Engineering.* 1064–1064.

[14] Davide Ginelli, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2021. Exception-Driven Fault Localization for Automated Program Repair. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security.* 598–607. https://doi.org/10.1109/QRS54544.2021.00070

[15] James Gosling. 2000. *The Java language specification.* Addison-Wesley Professional.

[16] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10, 3 (2000), 171–194.

[17] Xiangyang Jia, Songqiang Chen, Xingqi Zhou, Xintong Li, Run Yu, Xu Chen, and Jifeng Xuan. 2021. Where to handle an exception? Recommending exception handling locations from a global perspective. In *2021 IEEE/ACM 29th International Conference on Program Comprehension.* 369–380.

[18] Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-Based Fault Localization via Enhanced Decision Tree. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–32.

[19] Shujuan Jiang, Wei Li, Haiyang Li, Yanmei Zhang, Hongchang Zhang, and Yingqi Liu. 2012. Fault localization for null pointer exception based on stack trace and program slicing. In *2012 12th International Conference on Quality Software.* 9–12.

[20] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering.* 467–477.

[21] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 437–440.

[22] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security.* 114–125.

[23] Jeongho Kim, Jindae Kim, and Eunseok Lee. 2019. Vfl: Variable-based fault localization. *Information and Software Technology* 107 (2019), 179–191.

[24] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 1–34.

[25] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering.* 649–660.

[26] Yan Lei, Huan Xie, Tao Zhang, Meng Yan, Zhou Xu, and Chengnian Sun. 2022. Feature-fl: Feature-based fault localization. *IEEE Transactions on Reliability* 71, 1 (2022), 264–283.

[27] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 169–180.

[28] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering.* 661–673.

[29] Zheng Li, Xue Bai, Haifeng Wang, and Yong Liu. 2020. IRBFL: an information retrieval based fault localization approach. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference.* 991–996.

[30] Zheng Li, Haifeng Wang, and Yong Liu. 2020. Hmer: A hybrid mutation execution reduction approach for mutation-based fault localization. *Journal of Systems and Software* 168 (2020), 110661.

[31] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering* 32, 10 (2006), 831–848.

[32] Dino Mandrioli, Bertrand Meyer, et al. 1992. *Advances in object oriented software engineering.* Prentice Hall.

[33] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[34] Hugo Melo, Roberta Coelho, and Christoph Treude. 2019. Unveiling exception handling guidelines adopted by java developers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering.* 128–139.

[35] Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. 2021. Globug: using global data in fault localization. *Journal of Systems and Software* 177 (2021), 110961.

[36] Hamed Mirzaei and Abbas Heydarnoori. 2015. Exception fault localization in android applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems.* 156–157.

[37] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation.* 153–162.

[38] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 1–32.

[39] Ganesh J Pai and Joanne Bechta Dugan. 2007. Empirical analysis of software fault content and fault proneness using Bayesian methods. *IEEE Transactions on Software Engineering* 33, 10 (2007), 675–686.

[40] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[41] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering.* 609–620.

[42] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* 1 (2002).

[43] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 30–39.

[44] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering.* 432–449.

[45] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories.* 212–222.

[46] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault localization and repair for Java runtime exceptions. In *Proceedings of the eighteenth International Symposium on Software Testing and Analysis.* 153–164.

[47] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 273–283.

[48] Yi Song, Xihao Zhang, Xiaoyuan Xie, Quanming Liu, Ruizhi Gao, and Chenliang Xing. 2024. ReClues: Representing and indexing failures in parallel debugging with program variables. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 1–13.

[49] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[50] Jeffrey M. Voas. 1992. PIE: A dynamic failure-based technique. *IEEE Transactions on software Engineering* 18, 8 (1992), 717.

[51] Ellen M Voorhees et al. 1999. The trec-8 question answering track report.. In *Trec*, Vol. 99. 77–82.

[52] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.

[53] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution.* Springer, 196–202.

[54] Rebecca J Wirfs-Brock. 2006. Toward exception-handling best practices and patterns. *IEEE software* 23, 5 (2006), 11–13.

[55] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 181–190.

[56] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[57] W Eric Wong, Vidroha Debroy, and Dianxiang Xu. 2011. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 3 (2011), 378–396.

[58] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[59] Craig S Wright and Tanveer A Zia. 2011. A quantitative analysis into the economics of correcting software bugs. In *Computational Intelligence in Security for Information Systems: 4th International Conference, CISIS 2011, Held at IWANN 2011, Torremolinos-Málaga, Spain, June 8-10, 2011. Proceedings*. 198–205.

[60] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214.

[61] Xi Xiao, Yuqing Pan, Bin Zhang, Guangwu Hu, Qing Li, and Runiu Lu. 2021. ALBFL: A novel neural ranking model for software fault localization via combining static and dynamic features. *Information and Software Technology* 139 (2021), 106653.

[62] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A universal data augmentation approach for fault localization. In *Proceedings of the 44th International Conference on Software Engineering*. 48–60.

[63] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology* 22, 4 (2013), 1–40.

[64] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 06 (2011), 803–827.

[65] Shin Yoo. 2012. Evolving human competitive spectra-based fault localisation techniques. In *Search Based Software Engineering: 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings 4*. 244–258.

[66] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology* 26, 1 (2017), 1–30.

[67] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering*. 958–969.

[68] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices* 48, 10 (2013), 765–784.

[69] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2007. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering* 12 (2007), 143–160.

[70] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. 445–455. https://doi.org/10.1109/SANER.2019.8668002

[71] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.