A CLUSTERING-BASED STRATEGY TO IDENTIFY COINCIDENTAL CORRECTNESS IN FAULT LOCALIZATION

YI MIAO and ZHENYU CHEN
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China

SIHAN LI
Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA

ZHIHONG ZHAO and YUMING ZHOU
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China

Coverage-based fault localization techniques leverage the coverage information to identify the faulty elements of a program. However, these techniques can be adversely affected by coincidental correctness, which occurs when the defect is executed but no failure is revealed. In this paper, we propose a clustering-based strategy to identify coincidental correctness in fault localization. The insight behind this strategy is that tests in the same cluster have similar behaviors. Thus a passed test in a cluster with many failed tests is highly possible to be coincidentally correct because it has the potential to execute the faulty elements as those failed ones do. We evaluated this technique from two aspects: the ability to identify coincidental correctness and the effectiveness to improve fault localization. The experimental results show that our strategy can alleviate the coincidental correctness problem and improve the effectiveness of fault localization.

Keywords: Coincidental correctness; Cluster analysis; Cluster test selection; Fault localization

1. Introduction

Software testing is an important part of software quality assurance. Once failures are detected by testing, we need to find the root cause of the failures. It is known as fault localization.

*A preliminary version of this paper was presented at the 24th International Conference on Software Engineering and Knowledge Engineering [1].

1Corresponding Author: Software Institute, Nanjing University, Nanjing 210093, China. Email: zychen@software.nju.edu.cn
localization. Since fault localization is a time-consuming and tedious work, many automatic techniques have been developed to assist fault localization [2-6]. As an effective automatic fault localization technique, Coverage-Based Fault Localization (CBFL) leverages the execution information of both the failed runs and passed runs to assist the developer in identifying program entities that induce given failures. The program entities can be of different granularities, such as statement, block, predicate, function, etc. The intuition behind these techniques is that entities in a program that are primarily executed by failed tests are more likely to be faulty than those that are primarily executed by passed tests [7].

Despite CBFL has showed promising results in previous studies [2-7], there are still some challenges to further improve its effectiveness. One of the main challenges is the coincidental correctness problem [8]. A test is said to be coincidentally correct if it executes the faulty statements but reveals no failure. The PIE (Propagation, Infection and Execution) model [9] emphasizes that for a failure to be observed, the following three conditions must hold: 1) the defect is executed, 2) the program has transited into an infectious state, and 3) the infection has been propagated to the output. The case is called coincidental correctness, when the program produces the correct output while the defect is executed, i.e. the condition 1) holds. The CBFL techniques always leverage the execution information (executed or not) and the output information (passed or failed) to locate the fault. Therefore, whether the data state of the program is affected or not is out of concern. It has been indicated that coincidental correctness is responsible for reducing the safety of a CBFL technique [10]. That is, when coincidentally correct tests are present, the faulty entity is likely to be ranked as less suspicious than when they are not present.

It is natural to identify coincidental correctness to improve the effectiveness of CBFL techniques. However, the task of identifying coincidental correctness is challenging, because the faulty location cannot be known in advance. In this paper, we propose a cluster-based strategy to identify coincidental correctness. Our strategy is inspired by the success stories of cluster analysis in test selection. Tests with similar behaviors could be grouped into the same subset by clustering their corresponding execution profiles [21-23]. The intuition behind our strategy is that tests in the same cluster have similar behaviors, which means, passed tests in a cluster with some failed tests have the potential to cover the faulty statements as the failed ones do. Therefore, these passed tests within this cluster are very likely to be coincidentally correct.

The main contributions of this paper are as follows.

1. An empirical study is conducted to show that coincidental correctness is common in software testing. The results demonstrate that it is safely reduced for CBFL techniques.
2. We propose a novel strategy to identify coincidental correctness by clustering execution profiles of tests. Two strategies, removing and relabeling, are introduced to deal with coincidentally correct tests.
3. An empirical study is conducted to show the effectiveness of our strategy for three popular CBFL techniques: Ochiai, Tarantula, and Jaccard.

The remainder of this paper is organized as follows. Section 2 demonstrates the motivation of our work. Section 3 describes our approach to identify coincidentally correct tests in detail. Section 4 presents the experimental work and results for the proposed approach. Section 5 introduces some related work on coincidental correctness and cluster analysis on software testing. Finally, Section 6 concludes our work.

2. Motivation

2.1. Adverse Effect of Coincidental Correctness

CBFL techniques assign a score, which represents the suspiciousness value, to a program entity, and produce a diagnosis report that consists of the suspicious program entities ranked by the suspiciousness value in descending order. The score is computed by a numeric function an a ranking metric. One of the assumptions of CBFL is that the program will fail when the faulty entities of the program are executed. However, it is not always true because of coincidental correctness.

Denmat et al. [12] pointed out the limitation of Tarantula and argued that the effectiveness of this technique largely depended on the hypothesis that executing the faulty statements leads most of the time to a failure. Abreu et al. [13] investigated the diagnostic accuracy of CBFL as a function of several parameters, including “observation quality” and “observation quantity”. The observation quality was measured by the percentage of failed runs executing the defect (faulty statement) with respect to the total tests executing it. The observation quantity was related to the number of passed and failed tests. Their results showed that the accuracy of CBFL increased slightly as the observation quality improved. They also observed that adding failed tests may improve accuracy whereas adding passed tests had unpredictable effects, because of the coincidental correctness in passed tests.

In this paper, we use two strategies, cleansing strategy and relabeling strategy, to deal with the coincidental correct tests (see details in Section 3). The two strategies will lead to a reduction of the number of passed tests, which will further lead to an increase of the ratio of the failed tests. As a result, the accuracy of the CBFL technique could be improved.

To explain the motivation clearly, we introduce a simple example to show the negative impact of coincidental correctness. In the following, we use Ochiai [18] to measure the suspiciousness metric of statements.

\[
M(s) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{nf})(a_{ef} + a_{ep})}}
\]  

(1)

In Equation 1, \(s\) is a statement; \(a_{ef}\) is the number of failed tests that execute \(s\); \(a_{nf}\) is the number of failed tests that do not execute \(s\); and \(a_{ep}\) is the number of passed tests that execute \(s\).
As shown in Table 1, there are three tests $t_1, t_2, t_3$, in which $t_1$ is failed, $t_2$ and $t_3$ are passed, and $t_3$ is coincidentally correct. There are four statements $s_1, s_2, s_3, s_4$, in which $s_2$ is the faulty statement. The coverage information $a_{ef}, a_{nf}$ and $a_{ep}$ is shown in Table 1. The Ochiai values of $s_1, s_2, s_3, s_4$ are calculated by Equation 1. It is not difficult to see that $s_4$ the most suspicious one and the faulty one $s_2$ is masked. If we can identify that $t_3$ is coincidentally correct and then clean $t_3$ in fault localization. Then $a_{ep}$ values are changed to $a_{ep}^*$ values, Ochiai values are changed to Ochiai$^*$ values, by removing $t_3$. The faulty statement $s_2$ could be popped up properly in the rank of fault localization.

### 2.2. Prevalence of Coincidental Correctness

In [10], Masri et al. demonstrated that coincidental correctness was prevalent in many cases. To show the prevalence of the scenario under study, we conducted an experiment on the Siemens programs. The Siemens set contains seven C programs, and all of them can be downloaded from the SIR repository [26]. Each of the programs has a correct version, a number of faulty versions seeded with a single fault, and a corresponding test suite. We took the following steps to assess the frequency of coincidental correctness:

1. Compare the correct version with each of the seeded versions using a diff command, and get the faulty locations of the faulty versions. These faulty locations are audited manually further.
2. Execute the test suite on the original and seeded versions, collect their corresponding output results of both versions and the execution profiles of the faulty versions.
3. A failure is found if the output of the correct version and those of the corresponding seeded version are different.
4. According to the execution profile, if a faulty element is executed during a test, but no observable failure is detected, we categorize the test as coincidental correctness.

Our study takes into account 115 seeded versions, and excludes the other versions because they contain code-missing errors or the faulty statements are not executable. These faulty versions are excluded because CBFL is not applicable to
them and these versions will never cause a coincidental correctness problem. The results indicate that the exhibited level of coincidental correctness is significant. The main observations are:

1. 15% faulty versions exhibit a low level, in the range [0%, 10%).
2. 50% faulty versions exhibit a medium level, in the range [10%, 60%).
3. 18% faulty versions exhibit a high level, in the range [60%, 90%).
4. 17% faulty versions exhibit an ultra-high level, in the range [90%, 100%).
5. The average level of coincidental correctness over all the versions is 50.94%.

The above results indicate that coincidental correctness is common in software testing and fault localization.

3. Our Approach

3.1. Framework

The symbols we use throughout the rest of the paper are explained as follows:

- $T$: the test suite used for a given program.
- $T_p$: the set of passed tests.
- $T_f$: the set of failed tests.
- $T_{cc}$: the set of coincidentally correct tests.
- $T_{icc}$: the set of identified coincidentally correct tests.

In Section 2.1, we show that not accounting for $n$ (the number of coincidental correct tests) will make the Ochiai metric misleading. More specifically, the suspiciousness of $e$ (the faulty element of the program) will be underestimated. Note that not only the Ochiai metric will be adversely affected by coincidental correctness, so are other CBFL techniques. An empirical study will be presented in Section 4.3.2 to demonstrate that, coincidental correctness problem happens in many of the CBFL techniques, as long as they share the same input data and principle to locate the fault.

It is worth noting that, as defined in [14], a failure is an event that occurs when the delivered service deviates from the correct service. In our context, the output of the correct version is used as an oracle. If the output of the seeded version is different from that of the correct one, a failure is detected.

Given a test suite $T$, which is comprised of $T_p$ and $T_f$, our goal is to identify $T_{cc}$ from $T_p$. Let $T_{icc}$ be the set of identified coincidentally correct tests. Consequently, each element of $T_{icc}$ is a potential candidate of the members of $T_{cc}$.

In this paper, we propose a clustering-based strategy to identify $T_{icc}$. Using execution profiles as the features fed into a clustering algorithm, we assume that tests having similar behaviors will be clustered together. Furthermore, tests which execute the faulty element and have similar execution paths with the failed tests are likely to be clustered together. In this sense, if a cluster consists of both failed
runs and passed runs, the passed runs within this cluster are very likely to be coincidentally correct.

Figure 1 is the framework of our approach. It shows how a developer takes the advantage of our strategy to improve the effectiveness of the diagnosis. First, a set of tests is executed on the given program. As a result, each test is labeled “passed” or “failed” according to the output result. Execution profiles which reveal the coverage information are collected at the same time. Then, clustering is conducted on the execution profiles. The next step is to identify coincidentally correct tests using the method mentioned above, and the identified tests are added to $T_{icc}$. After that, we use two strategies described in Section 3.5 to deal with the tests belong to $T_{icc}$. Finally, a CBFL technique is applied to the refined test suite.

![Fig. 1. Framework of our approach](image)

### 3.2. Execution Profiles

CBFL utilizes program elements of various granularity. In general the elements could be individual statements, basic blocks, branches or functions. In our study, as Ochiai is adopted as the fault localization technique, statement-level execution traces are collected.

We use gcov (GNU call-coverage profiler) [27] to obtain the statement coverage information. A ".gcov" file is generated after a test $t_i \in T$ is executed, which records the number of times each line of code has been executed. For test $t_i$, its statement coverage profile $t_i = <s_1, s_2, \ldots, s_n>$, where $n$ represents for the number of lines of the given program. If the $i$th line of code is executed, $s_i = 1$; otherwise, $s_i = 0$.

### 3.3. Clustering Analysis

The profiles of the test suite $T$ are collected to form the input for clustering analysis. The profile of each test is regarded as an object to be clustered. The number of objects is equal to the number of the tests in $T$. In our context, $n$-dimensional Euclidean distance [28] is used as the distance function. This function is easily calculated and widely used. Given two test profiles $t = <s_1, s_2, \ldots, s_n>$ and $t' = <$
$s'_1, s'_2, \ldots, s'_n >$, the distance of the two tests is:

$$D(t, t') = \sqrt{\sum_{i=1}^{n} (s_i - s'_i)^2} \quad (2)$$

Our study employs the simple K-means as the clustering algorithm, as it is simple and fast. In addition, it performs reasonably well in our previous experiments [15-17]. The simple K-means takes the number of clusters as a parameter. In our experiment, this number is set according to the size of $T$. Let $CN$ denote the number of clusters, $CN = |T| * p$, where $|T|$ is the size of $T$ and $0 < p < 1$. The previous studies worked well in the cases of $p = 2\%$ to $p = 5\%$. Hence, we set $p = 1\%, 2\%, 4\%$ and $6\%$ in our experiment, respectively. More details about how to decide the value of $p$ will be discussed in Section 4.3.1.

### 3.4. Identifying Strategy

For test selection and failure prediction [16][17], it is desirable to group passed tests and failed tests into different clusters. However, the ideal result is difficult to obtain in the real applications. Passed tests and failed tests may be mixed in the same cluster. One of the main reasons is coincidental correctness.

A failed test must execute the faulty statements, but not vice versa. Some tests executing the faulty statements may not raise a failure, i.e. they are passed and called coincidental correctness. In clustering analysis, it is assumed that tests with similar execution profiles will be clustered together. The passed tests which are put into the same cluster with the failed ones are much likely to execute the faulty statements, i.e. to be coincidentally correct. Therefore, it is reasonable to identify these passed tests in “mixed” clusters as coincidental correctness and add them to $T_{icc}$, the set of coincidentally correct tests.

Please note that our identifying strategy is neither sound nor complete. Some non-coincidentally correct tests may be added to $T_{icc}$ and some coincidentally correct tests may be missed. We will study false positives and false negatives with respect to different $p$ (in the previous subsection) in our experiment.

### 3.5. Two Dealing Strategies

After we identify the set coincidentally correct tests $T_{icc}$, we should deal with these tests to improve the effectiveness of CBFL techniques. Here, we propose two strategies to deal with $T_{icc}$, such that the suspiciousness values of the faulty statements increase.

- Cleansing strategy: Tests in $T_{icc}$ are removed from the original test suite $T$.
- Relabeling strategy: Tests in $T_{icc}$ are relabeled from “passed” to “failed”.

Assume that there are $k$ tests that execute the faulty statement but do not raise a failure. Two strategies can be applied on these tests to improve the effectiveness of
CBFL techniques. We use Ochiai [19] to explain the impact of two dealing strategies on the suspiciousness.

The cleansing strategy is to remove these tests from the test suite, that is, to subtract \( k \) from \( a_{ep} \). Consequently, the suspiciousness metric of the faulty statement \( s \) will be:

\[
M'(s) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{nf})(a_{ef} + a_{ep} - k)}}
\]  

(3)

It is not difficult to see that \( M(s) \leq M'(s) \), because \((a_{ef} + a_{ep}) \geq (a_{ef} + a_{ep} - k) \geq 0\).

The relabeling strategy is to relabel those tests from “passed” to “failed”, i.e., to subtract \( k \) from \( a_{ep} \) and add \( k \) to \( a_{ef} \), the suspiciousness metric of the faulty statement \( s \) will be:

\[
M''(s) = \frac{a_{ef} + k}{\sqrt{(a_{ef} + a_{nf} + k)(a_{ef} + a_{ep})}}
\]  

(4)

Note that \( M(s) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{nf})(a_{ef} + a_{ep})}} \) is monotonically increasing with respect to \( a_{ef} \) [19]. Therefore \( M''(s) \geq \frac{a_{ef} + k}{\sqrt{(a_{ef} + k + a_{ef})(a_{ef} + k + a_{ep})}} \geq M(s) \).

The relabeling strategy seems more aggressive than the removing strategy. However, please remember that the identifying strategy is neither sound nor complete. The effectiveness and limitations of the two dealing strategies will be studied empirically in section 4.3.

3.6. Fault Localization

In this study, we select Ochiai, Tarantula and Jaccard as the CBFL techniques for evaluation. Due to the limitation of space, compared to Tarantula and Ochiai, more details about results of Ochiai are given. Ochiai is a recently proposed technique and the metrics is more effective than Tarantula in locating faults. Although the three techniques share the basic principle of CBFL, and they operate on exactly the same input data, as demonstrated in [24], the Ochiai similarity coefficient can improve diagnostic accuracy over other coefficients, including those used by the Pinpoint and Tarantula tools. As a result, it will be more convincing that if our approach can improve a well-performed CBFL technique. The experimental results of Tarantula and Jaccard are auxiliary to demonstrate that our approach is applicable across many CBFL techniques.

Note that the objective of this study is not to compare various fault localizers, but rather to develop a strategy that will improve the CBFL across multiple fault localizers. Although existing CBFL techniques use different metrics for the coverage entities, most of them share the same basic principle to locate the defect. In other words, they have the same hypothesis and share similar input data. Therefore, we believe that if our approach works on Ochiai, Tarantula and Jaccard, it will perform reasonably well on other fault localizers. In the future work, we will conduct experiments to investigate this conjecture.
4. Empirical Study

4.1. Subject Programs

The set of Siemens programs is a widely used benchmark for evaluating software testing and fault localization methods. Our study uses six programs with the test suite as the subject programs. The detailed information on these programs is listed in Table 2. The second column of the table shows the number of faulty versions of each program used in our experiment. The third column shows the lines of code (the number of executable statements in the parentheses) of each program.

<table>
<thead>
<tr>
<th>Program</th>
<th># Versions</th>
<th>LOC</th>
<th># Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>UN_F</td>
<td>UN_T</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>printtokens</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>printtokens2</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>schedule2</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>totinfo</td>
<td>23</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>91</td>
<td>17</td>
<td>8</td>
</tr>
</tbody>
</table>

The program *tcas* has only 173 lines of code, of which 54 are executable statements. Many tests may have the same execution profiles. Therefore, it is not included because it is not suitable for clustering analysis. Similar to the experimental setup in [7], our study uses executable statements rather than LOC. As a result, we ignore the modifications in the header files, mutants in variable declaration statements, or modifications in a macro statements started with `#define`. We exclude these versions containing code-missing errors, because *gcov* is applied for the statement-hit spectrum collection. These two types of versions are denoted by UN_F, as shown in Table 2. An other reason of versions not used in our experiment is the unreasonable tests: all are passed or no coincidental correctness. It is denoted by UN_T, as shown in Table 2. Finally 66 versions of programs were used in our experiment.

4.2. Evaluation Metrics

In order to evaluate the ability of our approach to identify coincidental correctness, we quantify the generated false negatives and false positives [7]. Also, to assess the impact of our approach on the effectiveness of CBFL, we use the T-score reduction as the evaluation metric.
1. Metric of false negative:

\[ \frac{|T_{cc} - T_{icc}|}{|T_{cc}|} \]

This measure assesses whether we have successfully identified all the coincidentally correct tests. The lower the measure value is, the better the identifying accuracy is.

2. Metric of false positive:

\[ \frac{|(T_p - T_{cc}) \cap T_{icc}|}{|T_p - T_{cc}|} \]

This measure assesses whether we have mistakenly categorized tests as coincidentally correct. Similarly, the lower the measure value is, the better the identifying accuracy is.

3. Metric of effectiveness improvement:

\[ \Delta TS = TS - TS' \]

TS and TS' represent the T-score before and after applying our approach respectively. T-score is widely used in evaluating fault localization techniques [5, 7, 19]. It measures the percentage of code that has been examined in order to find the defect, and is defined as

\[ TS = \frac{|V_{examined}|}{|V|} \times 100\% \]

where |V| refers to the size of the program (lines of the executable statements), and |V_{examined}| refers to the number of statements examined by the programmer in order to find the defect. The smaller the T-score is, the more effective the method will be. Therefore, a larger \( \Delta TS \) implies a greater improvement.

4.3. Experimental Results

4.3.1. Identifying Accuracy

Figure 2 shows the ability of our approach to identify coincidental correctness for 66 versions. It takes \( p \) (the ratio of the number of clusters) as a parameter, and \( p = 1\%, 2\%, 4\% \) and 6\% for Figure 2(a) to 2(d), respectively. “FN” and “FP” are short for “False Negative” and “False Positive”. The horizontal axis stands false negative (FN) and the vertical axis stands false positive (FP).

The results are similar in Figure 2(a)-2(d): a larger number of clusters yield a higher rate of false negatives but a lower rate of false positives. It is reasonable because the purity of a cluster increases as the number of clusters (i.e. \( p \)) increases. As a result, some of the coincidentally passed tests, once put into a cluster with some failed ones, are spread to another cluster full of passed tests. Therefore, these coincidentally correct tests will be missed. Similarly, some non-coincidentally passed tests will be spread to another cluster full of passed tests so that these tests will not be mistaken for coincidental correctness.

It is worth mentioning that the false positive rate is negatively correlated to the false negative rate. Because the higher the false negative rate, the fewer tests
will be considered as coincidentally correct. Therefore, the probability to incorrectly identify a test as coincidental correctness is lower.

In general, there is no a fixed ratio of clustering (i.e. the number of clusters) suitable for all cases. It mainly depends on the size of the test suite, and how much risk the developers are willing to take in order to identify the coincidental correct tests. If the size of test suite is large, a relatively low value of $p$ can be chosen to keep the level of the false negatives. If the developers have a high demand for accuracy of the recognition of coincidental correctness, a relatively high value of $p$ can be chosen to keep the level of false positives.

4.3.2. Effectiveness Improvement

To evaluate the impact of our strategy on the effectiveness of fault localization, we calculate the T-score of each version before and after applying our strategy. Thus, T-score reduction is computed for statistical analysis. Note that, during evaluation, only T-scores less than 20% are taken into account. More specifically, after applying
our strategy, only those versions that have T-scores less than 20% are used for statistical analysis and comparison. The reason is that it is not common to ask programmers to examine more than 20% of the code in practice [19].

Three CBFL techniques, Ochiai, Tarantula and Jaccard, are used in our experiment. Among these CBFL techniques, Ochiai consistently outperforms several other coefficients used in fault localization and data clustering. This is because compared to the other coefficients, Ochiai is much more sensitive to presence in failed runs than to presence in passed runs. This is well-suited to fault diagnosis because the execution of faulty code does not necessarily lead to a failure, while failures always involve a fault [24].

Therefore, we will analyze the result of Ochiai in detail. Due to space limitations, we cannot list all the results of Tarantula and Jaccard. But the experimental data implies that the conclusion is consistent over the three CBFL techniques.

We use box plot to depict the overall experimental results. Each box plot represents the statistics of the T-score reduction for each subject program. The bottom and top of the box is 25th and 75th percentile, respectively. The line within the box denotes the median of the values in the box and the point denotes the mean. The ends of the whiskers represent the minimum and maximum of all the data.

Figure 3 and 4 illustrates the impact of our approach on the effectiveness of CBFL. Note that, CBFL, here refers to Ochiai specifically. It takes \( p \) as a parameter. Figure 3 depicts the results applying the cleaning strategy, and Figure 4 depict the results applying the relabeling strategy. The ideal situation, where all the coincidentally correct tests are picked out, with 0% false negatives and 0% false positives, is also shown on the figure as a comparison. The bold dot presents the average T-score reduction for each program under the ideal situation.

In average, using different values of \( p \), 2 out of 6 programs have been improved by the cleaning strategy, and the T-score reduction can reach up to 23.57%. 3 to 4 out of 6 programs have been improved by the relabeling strategy, and the T-score reduction can reach up to 32.52%, which is quite impressive.

It can be noticed that, using the same value of \( p \), the height of the box in Figure 4(a) - 4(d) is larger than that in Figure 3, which reveals the fact that relabeling strategy may bring greater improvement on the effectiveness of fault localization, but it also brings the risks that some versions of the programs will deteriorate drastically.

Table 3 summarizes the results of paired-samples t-test on the differences between the T-scores before and after applying our strategy. These results are based on Ochiai, Tarantula and Jaccard. Suppose the T-score before using our strategy is A, and the T-score after using our strategy is B. \( H_0 : A \leq B \), \( H_1 : A > B \). Taking Ochiai for example, from this table, we can observe that:

- Cleansing strategy. The improvement is significant at the 0.05 level under the cluster ratio of 4% and 6%. The improvement is significant at the 0.10 level under all of the cluster ratios except 1%. 
Clustering-based Strategy to Identify Coincidental Correctness

Table 3. P-values of Paired-samples t-test on the T-score improvement

<table>
<thead>
<tr>
<th>CBFL</th>
<th>Cluster ratio</th>
<th>Cleansing</th>
<th>Relabeling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MeanA(%)</td>
<td>MeanB(%)</td>
</tr>
<tr>
<td>Ochiai</td>
<td>1%</td>
<td>7.23</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>2%</td>
<td>6.30</td>
<td>5.76</td>
</tr>
<tr>
<td></td>
<td>4%</td>
<td>6.84</td>
<td>5.98</td>
</tr>
<tr>
<td></td>
<td>6%</td>
<td>6.84</td>
<td>6.05</td>
</tr>
<tr>
<td></td>
<td>ideal</td>
<td>10.21</td>
<td>3.85</td>
</tr>
<tr>
<td>Tarantula</td>
<td>1%</td>
<td>8.36</td>
<td>5.86</td>
</tr>
<tr>
<td></td>
<td>2%</td>
<td>7.54</td>
<td>6.14</td>
</tr>
<tr>
<td></td>
<td>4%</td>
<td>7.21</td>
<td>5.98</td>
</tr>
<tr>
<td></td>
<td>6%</td>
<td>7.21</td>
<td>5.83</td>
</tr>
<tr>
<td></td>
<td>ideal</td>
<td>10.82</td>
<td>4.16</td>
</tr>
<tr>
<td>Jaccard</td>
<td>1%</td>
<td>8.49</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>2%</td>
<td>6.92</td>
<td>5.47</td>
</tr>
<tr>
<td></td>
<td>4%</td>
<td>6.58</td>
<td>5.23</td>
</tr>
<tr>
<td></td>
<td>6%</td>
<td>6.82</td>
<td>5.50</td>
</tr>
<tr>
<td></td>
<td>ideal</td>
<td>13.23</td>
<td>3.85</td>
</tr>
</tbody>
</table>

Fig. 3. Effectiveness Improvement of Cleansing Strategy
• Relabeling strategy. The improvement is significant at the 0.05 level under all of the cluster ratios except 4%. The improvement is significant at the 0.10 level under all of the cluster ratios.

Statistical test is also conducted on the results of the ideal situation. Under the ideal situation, using cleansing and relabeling strategy, both the p-values are < 0.01, and indicate a very significant improvement. These results indicate that both cleansing strategy and relabeling strategy could lead to a significant improvement in T-score.

Similar conclusions can be made from the experimental results of Tarantula and Jaccard. Also, it can be seen from Table 3 that, the experimental result of Ochiai is not as good as Tarantula and Jaccard. Moreover, for Jaccard, the improvement is significant at the 0.05 level under all of the cluster ratios, no matter using cleansing or relabeling strategy. It is expected because the limitation of the latter two CBFL techniques. Tarantula and Jaccard, while simple and well known, is somewhat outdated. So, there is much space for improvement. Many other fault
localization techniques have been proposed in the recent past that have been shown to be more effective, like Ochiai, for example. They are used in our experiment for evaluation merely because they share the same input data with Ochiai and it can be helpful to prove that the proposed strategy can be applied to any CBFL technique.

The above result shows that it is promising to improve the effectiveness of CBFL by dealing with coincidental correct tests. And using our approach, the preliminary experimental result is encouraging and convincing. These figures also reveal some issues: (1) For some programs, the impact of our approach is negligible. (2) Using the relabeling strategy, the effectiveness of fault localization decreases drastically for some versions of the programs. In summary, the cleansing strategy is relatively a safe method. The reason is that if $p$ is set to a reasonable value. We hence recommend that the developers use the cleansing strategy as the primary technique to improve the effectiveness of fault localization and use the relabeling strategy as the auxiliary technique.

Suppose the report generated using cleansing strategy is $R_1$, and the report generated using relabeling strategy is $R_2$. It is reasonable to set an upper limit $k$. The developer first go through the top $k$ statements of $R_1$, if no faulty statement is observed, then he can turn to go through the top $k$ statements (statements which are examined during the first round are excluded) of $R_2$. If still no faulty elements observed, then increase the value of $k$ and continue with the steps above.

The key factors that influence the improvement of CBFL technique are the rate of false negatives and false positives, which are heavily depended on the clustering results. We plan to improve the methodology to reduce these two metrics in the future work.

4.4. Discussion

The experimental results above indicate that our approach can identify coincidental correctness effectively and consequently improve CBFL techniques. Some practical issues should be considered to make our approach work well on real-world programs. We will discuss two practical issues, scalability and multiple faults in this section.

4.4.1. Scalability

The scalability of an approach is important in practice. It is expected that our approach can work well on large programs. We conducted an experiment on a larger program Space. Space is an interpreter for an array definition language, which can also be downloaded from SIR [26]. Space consists of 6199 lines of code, 136 functions, and 38 faulty versions. The number of tests is 13585.

For a large program, it is reasonable to locate faults in a coarse-grained scale and then refine it. Therefore, we locate faults at the function-level granularity in the experiment. We cluster function call profiles instead of statement coverage profiles of tests. Notice that fault localization and coincidental correctness identification may not be precise at the function-level granularity, but it can be refined within the
faulty function at statement-level granularity later. We instrument each function in the program and run tests one by one to collect the functional profiles. 7 out of 38 versions are not used, due to similar reasons in Section 4.1 (e.g. faults in non-executable code, all tests were passed, etc.). For the remained 31 versions, we do cluster analysis, identify coincidental correctness and then apply the cleansing and relabeling strategies, respectively. The experimental results are encouraging. (1) Three methods, original CBFL, CBFL with cleansing, and CBFL with relabeling, are tied for the best in 13 out of 31 versions. That is, $\Delta TS=0$ in these versions. (2) Figure 5 shows the effectiveness improvement ($\Delta TS$) of the remained 18 versions. The effectiveness improvement on some versions (13, 17, 24, 26) are significant. All results are positive except the cleansing strategy on version 29.

The preliminary results reveal the good scalability of our approach in terms of program size. We will conduct more experiments to demonstrate it. Moreover, the effectiveness improvement on Space is more significant than that on small programs. Overall, we believe that our approach can work well on large programs.

![Fig. 5. Effectiveness Improvement on Space Program](image)

4.4.2. Multiple Faults

The programs used to evaluate our approach have exactly one fault for each version. However, almost all real-world programs contain more than one fault, i.e. multiple faults. This issue should be considered to judge whether our approach can work well in practice.

- **Impact on clustering:** The previous study shows that single-label learning techniques (such as K-means used in this paper) may not be suitable for programs with multiple faults [29]. The purity of clusters may decrease, because a test may reveal multiple faults, i.e., it should be in multiple clusters. To what extent this impact on identifying coincidental correctness should be studied in the future.
Moreover, we should use multi-label learning techniques instead of single-label learning techniques for programs with multiple faults. It is not difficult to replace the clustering techniques because it is independent of the cleansing strategy and the relabeling strategy.

- **Impact on fault localization**: Almost all CBFL techniques are initially proposed for programs with single fault. A previous study indicates that the influence of multiple faults is not as great as expected in fault localization [30]. Still, some researchers improve their CBFL techniques for multiple faults in recent years. The basic idea is to cluster tests before applying CBFL techniques [31]. It is assumed that tests revealing the same fault will be grouped into the same cluster. In this way, the problem of fault localization on multiple faults is reduced to the problem of fault localization on single fault, and then CBFL techniques can work well. This is the main reason to believe that our approach can work well on programs with multiple faults.

- **Fault interaction** is defined as the change in the program behavior due to multiple faults working together. It has been observed in some previous studies [32][33], but these studies show some contradictory. The phenomenon of fault interaction is shown to be frequent in [32]. But the results in [33] show that some types of fault interaction are not frequent. However, there is no evidence on how fault interaction affects fault localization so far. More experiments need to be conducted in the future.

In summary, although we need more experiments on multiple faults, but the existing results and conclusions are not against our approach.

4.5. **Threats to Validity**

**External validity**: The first threat is that our conclusion cannot be generalized to larger programs, because the Siemens programs used in our study are all small C programs. To minimize the threat, we conduct a preliminary study on a larger program *Space* and the results are encouraging. We believe that to some extent, it is already representative of the programs because it has been widely used in the software engineering community [7, 10, 18]. The second threat is that our conclusion cannot be generalized to a program with multiple faults. However, cluster analysis is a natural method for parallel debugging for multiple faults [31].

**Internal validity**: The main threat is that the tools used to generate execution profiles and do the cluster analysis may influence our conclusion. In our study, we use gcov to record coverage information and rely on the data-mining tool Weka to cluster the execution profiles. Both of them are mature and widely used in previous studies. Therefore, this threat has been minimized.

**Construct validity**: The threat to construct validity is our measurements for the experiment. To minimize the threat, we introduce widely used measurements in fault localization. \( \Delta TS \) is popular in many previous studies on fault localization. False negative (FN) and false positive (FP) are widely used in statistical analysis.
5. Related Work

Voas [9] introduced the PIE model, which emphasizes that for a failure to be observed, the following three conditions must be satisfied: Execution, Infection, and Propagation. The case is termed as weak coincidental correctness, when the program produces the correct output while only the condition Execution is known to be satisfied. W. Masri et al. [10] have proved that coincidental correctness is responsible for reducing the safety of CBFL. That is, when coincidentally correct tests are present, the defect will likely be ranked as less suspicious than when they are not present.

As shown in the previous studies [8, 20], the efficiency and accuracy of CBFL can be improved by cleansing the coincidentally correct tests. However, it is challenging to identify coincidental correctness because we do not know the location of fault beforehand. X. Wang et al. [8] have proposed the concept of context pattern to help coverage refinement so that the correlation between program failures and the coverage of faulty statements can be strengthened. W. Masri et al. [20] have presented variations of a technique that identify the subset of passed tests that are likely to be coincidentally correct. This technique first identifies program elements \( cc_e \) that are likely to be correlated with coincidentally correct tests. Then it categorizes tests that induce some \( cc_e \)s as coincidental correctness. The set of coincidental correct tests would be partitioned into two clusters further. A more suspicious subset will be cleaned to improve the effectiveness of fault localization. The experimental result is promising, however, although it used the same subject program (the Siemens test suite) as ours in their experiment, it is applicable to only 18 versions of the 132 versions, which has a smaller application scope than our approach (applicable to 66 out of 132 versions).

Previous empirical observations have shown that, by cluster analysis, tests with similar behaviors could be grouped into the same clusters. Therefore, cluster analysis has been introduced for test selection. Vangala et al. [21] used execution profiles and static execution to compare tests and applied cluster analysis on them, identifying redundant tests with high accuracy. Dickinson et al. introduced cluster-filtering technique [22, 23]. It groups similar execution profiles into the same clusters and then selects a subset of tests from each cluster based on a certain sampling strategy. Since tests in the same cluster have similar behaviors, the subsets are representative for the test suite so that it is able to find most faults by using the selected subsets instead of the whole test suite.

Inspired by the successful stories, we proposed a cluster-based strategy to identify the subset of tests that are possible to be coincidentally correct. The intuition behind this strategy is that a passed test in a cluster with some failed tests has the potential to cover the faulty elements as the failed ones do. By cleansing or relabeling these tests, adverse effects of coincidental correctness can be reduced, which leads to an increase of the efficiency and accuracy of fault localization.
6. Conclusion and Future Work

In this paper, we propose a clustering-based strategy to identify coincidental correctness from the set of passed tests. To alleviate the adverse effect induced by the coincidental correctness on the effectiveness of CBFL, removing and relabeling strategies are applied on the coincidentally correct tests. We conduct experiments to examine the usefulness of our approach, and analyze the experimental results, which are promising. As a conclusion, the cleansing strategy is relatively a safe method while the relabeling strategy can perform very well only when the false negative and false positive is quite ideal. So it is recommended that the developers use the cleansing strategy as the primary technique and use the relabeling strategy as the auxiliary technique.

There is no doubt that we should conduct more experiments on large programs and adjust our approach to adapt programs with multiple faults in the future. In our study, we use the simple K-means because it is simple and effective. However, it is worth noting that having the failed tests clustered either too centralized or too scattered will have adverse effect on the result. More specifically, they will lead to high rates of false negatives and false positives, respectively. This inspires us to introduce more effective clustering methods [29] to improve fault localization in the future.

We suggest that the developers use CBFL iteratively. After the failure report is generated, the developers first examine the suspicious statement from the top one by one until a fault is found. Then, after the faulty statement is corrected, the developers may run tests again and relabel the tests. This step can be repeated to find more faults. In this context, it is possible that failed tests will be clustered into different clusters, as they are caused by different faults. Therefore, during one iteration, the false positive will be affected and have a higher value than the single-fault case. The extent of this influence on the effectiveness of our proposed approach needs to be investigated in the future work.

Acknowledgements

The work described in this article was partially supported by the National Natural Science Foundation of China (90818027, 61003024, 61073029 and 61170067).

References


Clustering-based Strategy to Identify Coincidental Correctness


Software-artifact Infrastructure Repository. http://sir.unl.edu/, University of Nebraska.

http://gcc.gnu.org/onlinedocs/gcc/Gcov.html


