Trimming Test Suites with Coincidentally Correct Test Cases for Enhancing Fault Localizations

Xiaozhen Xue  
Department of Computer Science  
Texas Tech University  
Lubbock, Texas, USA  
Email: xiaozhen.xue@ttu.edu

Yulei Pang  
Department of Mathematics and Statistics  
Texas Tech University  
Lubbock, Texas, USA  
Email: yulei.pang@ttu.edu

Akbar Siami Namin  
Department of Computer Science  
Texas Tech University  
Lubbock, Texas, USA  
Email: akbar.namin@ttu.edu

Abstract—Although empirical studies have demonstrated the usefulness of statistical fault localizations based on code coverage, the effectiveness of these techniques may be degraded due to the presence of some undesired circumstances such as the existence of coincidental correctness where one or more passing test cases exercise a faulty statement and thus causing some confusion to decide whether the underlying exercised statement is faulty or not. The presence of coincidental correctness hinders the effectiveness of coverage-based fault localizations. The fault localization based on coverage can be improved if all possible instances of coincidental correctness are identified and proper strategies should be employed to either 1) avoid these troublesome test cases or 2) flip their test status.

We propose a technique to effectively identify coincidentally correct test cases. The proposed technique combines support vector machines and ensemble learning to detect mislabeled cases, i.e. coincidentally correct test cases. The ensemble-based support vector machine then can be used to trim a test suite or flip the test status of the coincidental correctness test cases and thus improving the effectiveness of fault localizations. We evaluate the proposed technique through a number of experiments. The results show that the proposed technique can effectively identify coincidental correctness with high accuracy, precision and recall.

Keywords—coincidental correctness, support vector machine, ensemble learning, coverage based faults localization

I. INTRODUCTION

Testing and debugging are the important but time consuming activities in the software development life cycle. Despite the effort spent on developing effective software testing techniques, most of software products are released with severe bugs. A common practice for testing includes designing, developing, and executing required test cases and observing the behavior of the program under test. Once a failure is observed through execution of a test case, the test practitioner employs appropriate debugging techniques and tools to identify the cause of the failure. Automated debugging tools and techniques can greatly enhance the bug catching processes.

Developing effective debugging techniques and tools is challenging and still an open grand research problem. Coverage-based faults localizations are statistical techniques that assist testers to spot the suspicion locations of possible faults automatically. These techniques use the information captured during program’s execution to rank the program’s statements in decreasing order of suspiciousness.

Although it is widely reported that coverage-based faults localization techniques can improve the process of debugging significantly [1], [2], there exist some factors that degrade their effectiveness. One of the major influential factors is the presence of coincidentally correct test cases [3], [4], [5]. The coincidentally correct test cases were introduced in terms of PIE(Propagation-Infection-Execution) model [3], where a faulty statement is executed without being transitioned into an infectious state or propagated to the output. In other words, the coincidentally correct test cases are the test case whose executions cover faulty statements but without exhibiting any faulty behavior.

The presence of coincidental correctness introduces noise into the statistics that are captured and used for coverage-based fault localization techniques. Consequently, the noise distorts the suspiciousness scores measured for the statements and thus hurring localizing faults effectively. Masri et.al.[4] report the results of an empirical study in which the prevalence of coincidentally correct test cases has been assessed. Their results show that coincidental correctness occurs frequently and in most cases execution of test cases with coincidentally correct property negatively affects performance of coverage-based faults localizations. Due to their impact on fault localizations, coincidentally correct test cases need to be detected. Once the coincidentally correct test cases are identified, the tester can either avoid them or flip their test status to failed when conducting statistical fault localizations [5].

The problem of identification of test cases with coincidentally correct propert is in fact an instance of identification of mislabeled data in machine learning. In the scenario, we may regard coincidentally correct test cases as mislabeled data, i.e. test cases whose test status should be flipped into “failed". Therefore, existing machine learning techniques in detecting mislabeled data can be easily adapted to address this problem. For instance, Miao et.al. [6] propose a strategy to detect coincidental correctness by clustering test cases using a k-means clustering algorithm. In addition to k-means clustering, there exist some other machine learning techniques that can applied to identify mislabeled data. Therefore, it is of interest to investigate the application of other algorithms to identify coincidental correctness.

In this paper, we introduce the application of ensemble-based support vector machines (SVM) [7], a well-known technique for detecting mislabeled data to address the identification of coincidentally correct test cases. We perform two types
of experiments and compare their results. We first employ a trimming approach where coincidentally correct test cases are avoided from a test suite, once identified [5]. We then report the results of flipping the test status of coincidentally correct test cases and keep them in fault localizations [6]. We evaluate the performance of the proposed technique with respect to these two strategies. Our results show that the proposed ensemble-based SVM technique could effectively identify coincidental correctness with high accuracy, precision and recall. The key contributions of this paper are as follows:

- Introduce an ensemble-based SVM to identify coincidentally correct test cases;
- Evaluate the performance of the proposed technique with respect to the classification metrics employed;
- Compare the performance of fault localizations between trimming and flipping strategies;

The rest of this paper is organized as follows. Section II describes the background knowledge, and reviews the techniques and algorithms which are referred to in this paper. Section III formulates the research problem we aim to address, and describes the algorithms. Section IV presents an illustrative example. In Section V, we then explain the experimental procedure including the subjects chosen, data collection, research questions proposed and evaluation metrics. Section VI evaluates the proposed technique by discussing research questions. Section IX concludes our study and presents possible future research directions.

II. BACKGROUND

This section first reviews coverage-based faults localization techniques and the influence of coincidental correctness. It then describes the algorithms that are referred to in this paper including support vector machine and ensemble learning.

A. Coverage-Based Faults Localizations

Debugging based on executing test cases and gathering relevant statistics about code coverage is known as coverage-based fault localization. This statistical debugging approach measures suspiciousness of statements and ranks them according to a ranking metric. A variety of ranking metrics have been introduced [8] including the metric used in the Tarantula debugging system:

\[
Tarantula(s) = \frac{n_{ef}}{n_{ef} + n_{ef}} + \frac{n_{ef}}{n_{ef} + n_{ef}}
\]

The Ochiai metric:

\[
Ochiai = \frac{n_{ef}}{\sqrt{(n_{ef} + n_{ef})(n_{ef} + n_{ef})}}
\]

The \(\chi^2\) metric based on contingency tables [9]:

\[
\chi^2(s) = \frac{N \times (n_{ef} \times n_{ef} - n_{ef} \times n_{ef})^2}{n_{ef} \times n_{ef} \times n_{ef} \times n_{ef}}
\]

And the metric based on odds ratios [10]:

\[
\text{Odds Ratio}(s) = \frac{n_{ef} + 0.1}{n_{ef} + 0.1} \frac{n_{ef} + 0.1}{n_{ef} + 0.1}
\]

where the parameters are defined as:

- \(n_{ef}\): Number of test cases exercising the underlying statement and failed
- \(n_{ef}\): Number of test cases exercising the underlying statement and not failed
- \(n_{ef}\): Number of test cases not exercising the underlying statement but failed
- \(n_{ef}\): Number of test cases not exercising the underlying statement and not failed
- \(n_{e}\): Total number of test cases exercising the underlying statement
- \(n_{f}\): Total number of test cases not exercising the underlying statement
- \(n_{f}\): Total number of failing test cases
- \(n_{f}\): Total number of passing test cases
- \(N\): Total number of test cases

B. Coincidental Correctness

It is a common belief that the effectiveness of the coverage-based fault localization activity relies on the goodness of the ranking scheme employed. Most of the ranking metrics introduced for statistical fault localizations share similar statistics. However, there are some other factors that influence the performance of fault localizations based on coverage data. One of the influential factors is coincidental correctness, which refers to passing test cases whose executions include exercising faulty portions of the program under test without exhibiting any faulty behavior. The presence of such test cases confuses fault localization computations in accurately estimating the suspiciousness of statements.

To illustrate the impact of the existence of coincidental correctness on suspiciousness scores, let us consider an example. We assess the Ochiai metric introduced in Formula 2 where the numerator is \(n_{ef}\), i.e. the number of failing test cases executing the underlying statements, and the denominator is the square root of some combination of \(n_{ef}\), \(n_{ef}\), \(n_{ef}\), and \(n_{ef}\) where \(n_{ef}\) represents the total number of failing test cases that exercise the underlying statements. When computing the suspiciousness score for a faulty statement, the presence of coincidental correctness has no impacts on the numerator. However, it increases the denominator where \(n_{ef}\) is the total number of coincidentally correct test cases.

As a second example, let us assess the influence of coincidentally correct test cases on the odds ratio metric [10]. Empirical study shows that the odds ratios are more effective for localizing single fault [2] compared to other ranking metrics. The simplified form of the odds ratio metric is represented:
When computing the suspiciousness score for a faulty statement, the presence of coincidentally correct test cases increases the value of \( n_{ef} \), thus inflating the value of the denominator. This results in reducing the suspicious score computed for the underlying faulty statement.

Similarly, the suspiciousness scores computed by the metric used in Tarantula and also the contingency-based table \( \chi^2 \) are affected when there are instances of coincidental correctness. As a result, the scores computed underestimate the suspiciousness of the underlying faulty statements.

C. Support Vector Machine

Support vector machine (SVM) are machine learning algorithms widely used for classification and prediction problems [11], [12]. SVM utilizes a kernel function to perform both linear and non-linear classifications. For a simple binary high-dimensional linear classification problem, the SVM algorithm builds a hyper-plane with the intention of maximizing the distance between hyperplane and to the nearest data points on each side, i.e. the margins. Figure 1 illustrates the mechanics of a linear classification for binary prediction [13].

Suppose that a training set \( S \) is given as:

\[
S = (x_1, y_1), \ldots, (x_n, y_n)
\]

where \( x_i \in \mathbb{R}^p \), i.e., \( p \)-dimensional space, and \( y_i \in \{-1, +1\} \). The goal of SVM is to find a separating hyperplane:

\[
\mathbf{w}^T \mathbf{x} - b = 0
\]

and to solve the optimization problem:

\[
\min \left( \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum \xi_j \right)
\]

Subject to the following constraints:

\[
y_j \left( \mathbf{w}^T \mathbf{x}_j - b \right) \geq 1 - \xi_j
\]

\[
\xi_j \geq 0
\]

where \( x_j \) are the samples not on the correct side of the separating plane, \( C \) is the penalty parameter, and \( \xi_j \) is a slack variable. A slack variable is a variable added to make an inequality to a quality form.

SVM is widely used in solving real problems posed in various application domains. It has been applied to text categorization as their application can significantly reduce the need for labeled training instances [14]. Its applications to classification of images are also demonstrated [15]. Experimental results show that SVM performs significantly higher search accuracy. SVM is also widely useful in bioinformatics research and medical sciences to classify gene proteins with high feasibility and accuracy predictions [16].

D. Ensemble Learning

Ensemble learning is a paradigm where multiple learners, called base learners, are trained to solve the same problem [17], [7]. Different from the single learning approaches which learn one hypothesis from training data, ensemble methods construct a set of hypotheses and combine them for better predictions. Figure 2 depicts a general picture on how to make use of ensemble learning.

Several separate data sets can be derived from the original training data by different approaches. For each data set a model is trained and built. Each model makes its own decision when applied to the test data. The final result however is an aggregation of all predictions and it is usually based on the majority votes of all the models built.

It can be proved that the ensemble learning can improve the prediction accuracy significantly. Suppose the accuracy of a basic learner is \( p \), a probability value between \([0, 1]\). The
likelihood that a basic learner may produce a false prediction is therefore $1 - p$. When there are $n$ learners, the possibility of making correct predictions $P$ by majority voting is given in Formula 10:

$$P = \sum_{m=\left\lceil \frac{n}{2} \right\rceil}^{n} C_n^m (1-p)^{n-m} p^m \geq p \quad (10)$$

Where:

- $n$ is the total number of learners;
- $m$ is the number of times the $n$ learners have predicted the labeling correctly;
- $C_n^m$ is the combination of $m$ out of $n$ data points;
- $C_n^m (1-p)^{n-m} p^m$ are the binomial distribution for each learner, i.e. selection $m$ out of $n$ values forms a binomial distribution.

In recent years, besides the traditional classification techniques, the application of ensemble learning in detecting mislabeled data has been also introduced [19], [20]. The data set can be divided into two training and test sets interactively. Each data point is tested several times. The final decision for each data point is performed by majority voting. If the decision is different from its actual label, the data point is considered to be mislabeled.

III. TRIMMING TEST SUITS THROUGH SVM ENSEMBLES

In this section, we first give a formal definition of the problem we address, and then describe the algorithm developed along with the steps to be performed.

A. The Problem Formulation

Consider a faulty program with a set of passing and failing test cases devised for testing purposes. Besides localizing faults, we aim at determining whether there are any coincidentally correct test cases in the set of passing test cases. Once identified, we may avoid the coincidental correctness from fault localization procedure or flip their test status with the intention of improving the performance of the fault localization. We present a number of notations to ease the representation:

- $TP = \{t_{p_1}, t_{p_2}, ..., t_{p_i}\}$ be a suite of $i$ passing test cases.
- $TF = \{t_{f_1}, t_{f_2}, ..., t_{f_j}\}$ be a suite of $j$ failing test cases.
- $CC \in TP$, which contains coincidentally correct test cases.

In terms of the notations we have developed, we define Test Suit Trimming as: Given a faulty program, with a suite of passing test cases $TP$ and a suite of failing test cases $TF$, the trimming procedure aims at 1) identifying a subset $CC$ of $TP$; 2) moving $CC$ from $TP$ to $TF$ and generating new $TP$ and $TF$ sets.

B. Vector Constructions

We present the procedure to apply linear SVM classifiers to the test suite trimming problem. Machine learning algorithms often involve building vectors, in which intermediate data are held for further processes. The vectors represent a high-dimensional space and hold values for possible features that have been taken into account while performing the classifications. SVM expects numerical values for each feature. On the other hand, the coverage information achieved by each test case is represented by a binary variable with 0 and 1 values where the former and latter indicate the test case has/has not exercised the underlying statement, respectively. To overcome this hurdle, instead of using basic coverage information, i.e. 0 for passing and 1 for failing, in building the required vector, we utilize the number of times each statement is executed by the underlying test case.

For instance, consider a given program with 10 lines of executable statements. A desired vector for this program will be a vector with 10 elements each representing an executable statement. Then, for each test case we build a vector whose elements represent how many times the underlying test case has exercised the statement. For example, the vector constructed for a test case whose execution trace is ‘2,3,4,2,3,4,7,8,9’, where each number represent the line number of the program statement, can be represented as $(0,2,2,0,0,1,1,0)$. The elements of the vector are also called features in machine learning literature. In the particular example, the features are the line number of the program under test and their values are the number of times the underlying test case has exercised the corresponding statement. Any typical code coverage tool, e.g. gcov and ATAC, is capable of determining the number of times each statement is executed by each test case.

C. Partitioning Data Into Training/Test Sets

Figure 2 illustrates the general mechanic of ensemble learning where $m$ learners are employed to classify mislabeled data. The number of times the partitioning process is performed depends on the desired number of SVM learners employed for training ($m$ times). A subset of all possible data, i.e. Training Data, should be built for each learner, i.e. $Data-1$, $Data-2$, ..., $Data-m$. In followings, we describe the partitioning procedure we employed for building each subset $Data-i$ where $1 \leq i \leq m$.

Each SVM learner needs to learn classifying coincidentally and non-coincidentally correctness test cases. Hence, the training data set $Data-i$ must include both types of failing and passing test cases. Each $Data-i$ consists of some subset data from both failing ($TF$) and passing $TP$ test cases. In practice, the number of passing test cases is always much greater than the number of failing test cases and in most cases only one failing test case is available. Therefore, we include the entire set of failing test cases $TF$ into each $Data-i$.

In addition to failing test cases, we also need to randomly select a subset of $TP$ to be included in building each $Data-i$. For each $Data-i$, we randomly select a small subset of passing test cases $TP_i \in TP$ and add them to the $Data-i$. More precisely:

$$Data_i = \{\{TF\} \cup \{TP_i \in TP\}\}$$
Each data is the training sets for each SVM learner. The test set for each learner is then $TP \setminus TP_i$, i.e. the remaining test cases in $TP$ when the test cases in $TP_i$ are taken away. Using each data data set, each SVM classifier is trained. It is apparent that the proposed technique works only when there is at least one failing test case, i.e. $TF \neq \emptyset$.

D. Identifying Coincidentally Correct Test Cases

Algorithm 1 Identifying coincidentally correct test cases through SVM ensembles.

Require: Inputs:
1) passing test cases $TP = \{tp_1, ..., tp_i\}$
2) failing test cases $TF = \{tf_1, ..., tf_j\}$
3) SVM learning algorithm $L$
4) number of partitions $N$

Ensure: $CC$
1: $CC = \{\}$ \{initialization of coincidentally correct set\}
2: for $n = 1$ to $i$ do
3: $Z_n = \{\}$ \{initialization of each label pool\}
4: end for
5: Randomly divide $TP$ into $N$ partitions $TP_1, ..., TP_N$
6: for $n = 1$ to $N$ do
7: $tr_n = TP_n \cup TF$
8: $ts_n = TP \setminus TP_n$
9: $SVM_n = L(tr_n)$ \{Train an individual learner\}
10: for each test data point $tp_k$ in $ts_n$ do
11: $label_k = SVM_n(tp_k)$ \{test each data point\}
12: $Z_k = Z_k \cup \{label_k\}$
13: end for
14: end for
15: for $n = 1$ to $i$ do
16: $coi=0$ \{number of labels of coincidental correctness\}
17: $non=0$ \{number of labels of non-coincidental correctness\}
18: for each $z$ in $Z_n$ do
19: if $z$ labels coincidental correctness then
20: $coi++$
21: else
22: $non++$
23: end if
24: end for
25: if $coi \geq non$ then
26: $CC = CC \cup \{tp_k\}$ \{take the major voting\}
27: end if
28: end for

Algorithm 1 describes the algorithm developed for identifying coincidentally correct test cases. In addition to two sets of passing ($TP$) and failing ($TF$) test cases, the number of learners ($N$) employed during the course of procedure, $N$ also indicates the number of partitioning required for generating training sets for learners. The output is a set of coincidentally correct test cases $CC$.

The entire procedure consists of five major steps:

1) **Stage I - Initialization.** Initialize the final output $CC$, a set of the coincidental correctness. Initialize each label pool for the passing test cases (Lines 1–4);
2) **Stage II - Partitioning.** Divide passing test cases into $N$ partitions (Line 5);
3) **Stage III - Building Training Sets.** For each partition build training set $tr_n$, i.e. the union of $TP_i$ and $TF$. The corresponding test data set $ts_n$ consists of the remaining passing test cases (Lines 7–8);
4) **Stage IV - Build Labeling Vectors.** Build support vector machines ($SVM_n$) for each training data set and using each $SVM_n$ to predict the corresponding decision on labeling for each test case $tp_k$. Finally, for each test case, store the corrected labels into a vector, called labeling vector (Lines 9–13); The labeling vectors will record all the resulting labeling and possible corrections performed by $N$ learners for each test case.
5) **Stage V - Majority Voting.** Make final decisions on coincidental correctness by majority voting for each passing test case involved (Lines 13–28); in this part for each passing test cases, two temporary values, the number of labels of coincidental correctness $coi$ and the number of labels of non-coincidental correctness are used, so that a comparison of the two values would reflect the decision, i.e. vote, performed by majority.

In making the final decision based on majority voting, the procedure utilizes the labeling vectors built during the course of the algorithm. Each labeling vector, built for a test case, holds the results of each learner regarding whether a test case should be considered as a coincidentally or non-coincidentally correct test case. A simple counting on the number of decisions, i.e. votes, provided by each learner, can determine whether a test case should be coincidentally or non-coincidentally correct.

E. Trimming Test Suites

Once the coincidentally correct test cases are identified, we may either avoid these test cases or flip their test status to “failing.” Since either of these approaches will affect the values of $n_c f$ and $n_e f$, the results could improve the suspiciousness scores measured for each statement.

By avoiding coincidentally correct test cases from the test suites before performing faults localization, the set of passing test cases are updated, i.e reduced, but no operations are performed on failing test suites. Using the notations defined above, the avoiding approach will produce a reduced $TP$ by $TP = TP \setminus CC$. On the other hand, the flipping strategy would cause re-labeling the test status of coincidentally correct test cases from passing to failing, and update the passing and failing test suites simultaneously as following: $TP = TP \setminus CC$ and $TF = TF \cup CC$.

IV. AN ILLUSTRATIVE EXAMPLE

In order to have a better insight of how the proposed technique works for detecting coincidentally correct test cases and also how the trimming strategy improves the performance of faults localizations, we present an illustrative example.

The Java code snippet given in Table I implements a class to obtain the abstract value of a given variable. Since the input numbers could be either double or integer, an override method is implemented that can take various input types. The code is composed of 8 lines with one class AbsVal along with two
overriding methods. A fault is injected at line 6 by replacing the predicate \( a \leq 0 \) to \( a \leq 2 \). Five test cases, of which four of passing and one is failing are devised for the purpose of testing the class and its functionality.

Suppose we would like to perform coverage-based faults localization where Ochiai metric for measuring suspiciousness score and for ranking statements. The outcome of the simple statistical fault localization based on Ochiai is listed in Table II. According to the literature, the fault localization cost is usually defined as the percentage of statements in the program that must be examined before reaching the first faulty statement [21]. Therefore, the cost for localizing the fault injected into Line 6 will be \( \frac{4}{8} = 0.5 \).

<table>
<thead>
<tr>
<th>Statement</th>
<th>( n_{ef} )</th>
<th>( n_{ef}' )</th>
<th>( n_{T} )</th>
<th>( n_{T}' )</th>
<th>ochiai</th>
<th>ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0.7</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0.7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0.7</td>
<td>4</td>
</tr>
</tbody>
</table>

We notice that one passing test case \( TC4 \) whose execution covers the faulty statement, did not expose any failure. It is apparent that this test case is coincidentally correct.

We aim at identifying this test case by the proposed ensemble-based SVM using Algorithm 1. First we divide passing test cases into \( N \) groups. In this example, since the size of the data set is small, we set \( N = 4 \). In other words, each group contains exactly one passing test case along with the failing pass. We build four training date sets:

- \( trn_1 = \{ TC1, TC5 \} \);
- \( trn_2 = \{ TC2, TC5 \} \);
- \( trn_3 = \{ TC3, TC5 \} \);
- \( trn_4 = \{ TC4, TC5 \} \);

Using each training data set, we construct a single SVM classifier, and predict the labeling of the remaining passing test cases, which are regarded as test data. In this example, the predict result for the four classifier are listed in Figures 3 and 4:

For each passing test case, we take the majority decisions or votes. For instance, 3 of the 3 decisions predict \( - \) for labeling \( TC1 \). Therefore, the final label for \( TC1 \) will be \( - \), which indicates that \( TC1 \) is not labeled as a coincidentally correct test case. Whereas, 3 of the 3 decisions for \( TC4 \) is \( + \), which demonstrates that \( TC4 \) is predicted as a coincidentally correct test case.

The results showed that only \( TC4 \) is a coincidental correctness. Following the flipping strategy, we flip the label of \( TC4 \) from passing to failing, and re-perform the fault localization procedure using the updated statistics shown in Table III. The cost will improve, i.e. \( \frac{3}{8} = 0.37 \), which is an indication of improvement. Similarly, we remove \( TC4 \) following removing strategy. Although the statistics in Table IV varies from the original in both scores and rankings, the cost remains unchanged since the rank of the faulty statements
TABLE III. OCHIAI STATISTICS AFTER FLIPPING COINCIDENTAL CORRECTNESS.

<table>
<thead>
<tr>
<th>Statement</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>ochiai</th>
<th>ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0.7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
</tbody>
</table>

TABLE IV. OCHIAI STATISTICS AFTER REMOVING COINCIDENTAL CORRECTNESS.

<table>
<thead>
<tr>
<th>Statement</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>( n_{ef} )</th>
<th>ochiai</th>
<th>ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0.7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1.0</td>
<td>3</td>
</tr>
</tbody>
</table>

does not change, i.e. 4/8 = 0.5.

V. EXPERIMENTAL PROCEDURE

We have conducted a number of experiments to assess the performance of the proposed technique. In this section, first we describe the subjects programs and the experimental setup, and then introduce three metrics and the result of the analysis. Throughout the experiments, we used Ochiai metric to assess the proposed technique in enhancing fault localization.

A. Subject Program

Table V lists the subject programs used for experiments. We obtained these extensively used Java programs from the Software Infrastructure Repository (SIR) [22]. The faults are hand-seeded by other researchers. We studied one release of nanoXML with 7 different faults, two releases of XMLsec with three different faults, two releases of jmeter with six different faults, and three releases of jtopas with four different faults.

The nanoXML and XMLsec are TSL (test specification language) test suites, and the other two are based on Junit test framework. Nanoxm is an xml parser for Java. Jtopas is a Java small library for tokenizing and parsing text. Jmeter is a Java desktop designed to load test functional behavior and measure performance. Xml-security library includes a mature digital signature and encryption implementation. There are four reasons that we choose these subject programs. Firstly, they are wildly studied by other researcher; secondly, the programs already include some faults injected; third, test cases generated are ready to use; and fourth, the size of these program ranges from 5 to 43 thousand lines of the code, which make the study results more meaningful.

B. Experimental Setup

To perform test case classification, we require to collect the frequency of exercising each statement by each test case and results. Although it was possible to use a Java-based code coverage tool to capture these information, we decided to obtain these statistics by running test cases on the instrumented subject program. We instrumented each program by inserting print statement into each block where block is defined as a section of codes grouped together without any branches inside. For the result of the each test case, we simply compared the output of each faulty version against the output of the original version of the program with no faults. When the output of the faulty program was different than the output of the original program, the test case was labeled as failing. Similarly, when the outputs were exactly the same, the test case was tagged as passing.

We developed a Java utility program to build the labeling vectors, i.e. features, for each execution following the description in section III-B. We then developed some R scripts to perform the classification process. The package e1071 developed for the R system [23] offers an interface to the SVM classification implementation in C++. By simply calling svm() function in “LIBSVM” library, which is a library for SVM, we observed the predicted coincidental correctness. Furthermore, we set \( N = 10 \), the number of learners and partitioning in our experiments.

We carried out the experiments on a Dell laptop with Windows XP environments. The machine is equipped with a Intel processor (1.66Hz, 2 cores) and with 1 GB physics memory.

C. research question

We pose three major research questions and throughout the experiments we intend to address these questions.

1) How prevalent is the coincidental correctness? It is a common held belief that the presence of coincidental correctness hinders the effectiveness of coverage-based fault localization. However, to the best of our knowledge, only one empirical study has assessed its prevalence. It is important to replicate the study and assess the external threats to the validity of the experimentations. In this paper, we are interested in investigating this research question on a number of Java-based subject programs.

2) Is the propose technique Effective? This paper proposes a technique which employs support vector machine ensembles to detect coincidental correctness, grounded on the theory that coincidental correctness is treated as an identification of mislabeled data problem. Since the profiles of coincidental correctness are similar to failing test cases but labeled as passing test cases, we would like to investigate whether the proposed technique indeed improves fault localizations?

3) Which approach works the best? Avoiding or Flipping? The coincidental correctness could be either removed or flipped. An interesting research question and implication for testers is then which strategy is more effective in term of improving the performance of fault localization?
D. Evaluation Metrics

In this section, we evaluate the proposed technique with respect to both prediction and faults localization’s performance.

1) Prediction Metrics: In the field of statistics and in particular for the prediction purposes, four key terms including true positives(tp), true negatives(tn), false positives(fp), and false negatives(fn) are usually computed for assessing the performance of a classifier. The terms positive and negative refer to the classifier’s prediction, also known as the expectation, and the terms true and false refer to whether that prediction corresponds to the external judgment, also known as the observation [24]. These terms and their associations are illustrated in Table VI for classification of coincidental correct test cases(CC):

<table>
<thead>
<tr>
<th>Predicted CC</th>
<th>Truely CC</th>
<th>Truely Non-CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>tp</td>
<td></td>
<td>fP</td>
</tr>
<tr>
<td>fN</td>
<td></td>
<td>tn</td>
</tr>
</tbody>
</table>

Three major measurement metrics, i.e. accuracy, precision and recall are usually used to assess how well a binary classification is performed. The accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity’s actual (true) value. It is the percentage of the sum of all true positives and false negatives out of the sum of all the true positives, true negatives, false positives, and false negatives [24].

\[
\text{Accuracy} = \frac{tp + tn}{tp + fp + fn + tn} 
\]  

The precision of a measurement system, also called as reproducibility or repeatability, is the degree to which repeated measurements under unchanged conditions show similar results [24]. It is formulated as the fraction of the number of true positives to the sum of the true positives and false positives.

\[
\text{Precision} = \frac{tp}{tp + fp} 
\]  

The recall measurement in this context, is also referred to as the true positive rate or sensitivity, is the ratio of true positives over the sum of true positives and false negatives or the percentage of flows in an application class that are correctly identified [24].

\[
\text{Recall} = \frac{tp}{tp + fn} 
\]

2) Faults Localization Cost: Similar to the other researchers [21], we compute the fault localization cost by measuring the percentage of statements in the program that must be examined before reaching the first faulty statement in the code. More precisely, cost for localizing faults is defined as the total number of statements exercised before reaching (examining) the first statement containing faults over the total number of statements in the program under test.

\[
\text{Cost} = \frac{\text{rank of fault}}{\text{size of program}}
\]  

It may be possible that some statements rank equally, i.e. tie ranking scores. In calculating the overall fault localization cost when some non-faulty statements are ranked tie with a faulty statement, we take into computation exercising all those statements that were ranked tie with the faulty statement, i.e. the worst case where the debugger inspects all non-faulty and faulty statements with equal ranks.

VI. ANALYSIS

In this section we report the result of the experiments, and address the three research questions posed in Section V-C.

A. Prevalence of Coincidental Correctness

<table>
<thead>
<tr>
<th>Faulty Program</th>
<th>Tests</th>
<th>#Failing</th>
<th>#Passing</th>
<th>#Coincidental</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoxml(v1)-F1</td>
<td>214</td>
<td>36</td>
<td>178</td>
<td>31</td>
</tr>
<tr>
<td>nanoxml(v1)-F2</td>
<td>214</td>
<td>34</td>
<td>180</td>
<td>34</td>
</tr>
<tr>
<td>nanoxml(v1)-F4</td>
<td>214</td>
<td>20</td>
<td>194</td>
<td>48</td>
</tr>
<tr>
<td>nanoxml(v1)-F5</td>
<td>214</td>
<td>1</td>
<td>213</td>
<td>0</td>
</tr>
<tr>
<td>nanoxml(v1)-F6</td>
<td>214</td>
<td>10</td>
<td>204</td>
<td>0</td>
</tr>
<tr>
<td>nanoxml(v1)-F7</td>
<td>214</td>
<td>29</td>
<td>185</td>
<td>0</td>
</tr>
<tr>
<td>xmlsec(v1)-F1</td>
<td>83</td>
<td>1</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td>xmlsec(v1)-F2</td>
<td>83</td>
<td>5</td>
<td>78</td>
<td>8</td>
</tr>
<tr>
<td>xmlsec(v1)-F4</td>
<td>83</td>
<td>3</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>jmeter(v1)-F1</td>
<td>58</td>
<td>2</td>
<td>56</td>
<td>4</td>
</tr>
<tr>
<td>jmeter(v1)-F2</td>
<td>58</td>
<td>1</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>jmeter(v1)-F3</td>
<td>58</td>
<td>1</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>jmeter(v1)-F4</td>
<td>58</td>
<td>3</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>jtopas(v1)-F1</td>
<td>54</td>
<td>4</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>jtopas(v1)-F2</td>
<td>54</td>
<td>4</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>jtopas(v2)-F1</td>
<td>54</td>
<td>4</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>jtopas(v3)-F1</td>
<td>54</td>
<td>4</td>
<td>50</td>
<td>1</td>
</tr>
</tbody>
</table>

Overall 10 out of 19 versions exhibited some instances of coincidental correctness, where the faulty statement was executed but no failure exhibited. One faulty program, nanoxml version 1 fault 3, did not expose any failure, so we excluded it from the experimentation. The prevalence percentage was around 50%. Figure 5 demonstrates the percentage of failing test cases (colored in red), coincidental correctness (colored in yellow), and non-coincidental correctness (colored in yellow) for each one of the 10 faulty versions where coincidental correctness occurs. The percentage of coincidental correctness is in the range from 1.2% to 22.2%, which is consistent with the study of other researchers [4].

B. Detecting Coincidentally Correct Test Cases

<table>
<thead>
<tr>
<th>Program</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoxml(v1)-F1</td>
<td>94.94%</td>
<td>77.50%</td>
<td>100.00%</td>
</tr>
<tr>
<td>nanoxml(v1)-F2</td>
<td>96.67%</td>
<td>85.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>nanoxml(v1)-F4</td>
<td>97.06%</td>
<td>88.89%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jmeter(v1)-F1</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jmeter(v1)-F2</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>xmlsec(v1)-F1</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>xmlsec(v1)-F2</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jmeter(v1)-F3</td>
<td>92.00%</td>
<td>90.48%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jtopas(v1)-F1</td>
<td>92.00%</td>
<td>90.48%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jtopas(v2)-F1</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>jtopas(v3)-F1</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

Table VIII reports the performance of our proposed approach for identifying coincidently correct test cases. Overall
the decrease in the debugging expense is around 5%. We observe that the recall of 98.10% and the precision of 95.14% are significant with the improvement of flipping or removing. The recall ratio is 98.10% indicating that we can effectively identify most of the actual coincidentally correct test case.

C. Flipping or Removing

Table IX reports the fault localization cost when either flipping or removing strategy is adopted. In summary, compared to the original average cost which is measured as 11.67%, the improvement of flipping or removing are significant with the updated cost of 6.12% and 8.31%, respectively. In other words, the decrease in the debugging expense is around 5% and 3%.

Table IX. The Performance of the Operation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Original</th>
<th>Flip</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoxml(v1)-F1</td>
<td>20.31%</td>
<td>10.94%</td>
<td>18.75%</td>
</tr>
<tr>
<td>nanoxml(v1)-F2</td>
<td>25.00%</td>
<td>4.63%</td>
<td>7.81%</td>
</tr>
<tr>
<td>nanoxml(v1)-F4</td>
<td>10.94%</td>
<td>10.94%</td>
<td>10.94%</td>
</tr>
<tr>
<td>jmeter(v3)-F2</td>
<td>17%</td>
<td>7%</td>
<td>11%</td>
</tr>
<tr>
<td>jmeter(v1)-F1</td>
<td>3.33%</td>
<td>3.33%</td>
<td>3.33%</td>
</tr>
<tr>
<td>xmlsec(v2)-F3</td>
<td>9.80%</td>
<td>9.80%</td>
<td>9.80%</td>
</tr>
<tr>
<td>xmlsec(v2)-F4</td>
<td>1.96%</td>
<td>1.96%</td>
<td>1.96%</td>
</tr>
<tr>
<td>jtopas(v1)-F1</td>
<td>5.56%</td>
<td>1.11%</td>
<td>1.11%</td>
</tr>
<tr>
<td>jtopas(v1)-F2</td>
<td>5.56%</td>
<td>1.11%</td>
<td>1.11%</td>
</tr>
<tr>
<td>jtopas(v3)-F1</td>
<td>17.25%</td>
<td>10.34%</td>
<td>17.24%</td>
</tr>
<tr>
<td>Ave</td>
<td>11.67%</td>
<td>6.12%</td>
<td>8.31%</td>
</tr>
</tbody>
</table>

We observe that for 4 out of the 10 faulty versions, the rank of faulty statements remains the same although the flipping of coincidental correctness results in the increase of the score, and for the rest 6 faulty versions, the rank of faulty statements increases. Similarly, we measure the percentage of decrease, increase, and no-change in cost when removing strategy is applied. The percentage is 5, 0 and 5 out of 10. We may conclude that flipping outperforms removing in terms of both average improvement and prevalence of improvement.

VIII. RELATED WORK

As the most widely discussed fault localization approach, coverage-based faults localization leverages the coverage and status information to rank statements in terms of their suspiciousness. Metrics such as the one used in Tarantula fault localization tool [1] and Ochiai [8] are two predominant ranking metrics using the necessary information. Similarly some crosstab-based statistical method based on Chi-square or odds ratio is proposed in [9] and [10]. Empirical studies show that these ranking metrics outperform other fault localization metrics when either single or multiple faults are present [1][2].

The study of coincidental correctness began with the analysis of faults status [3], and it was defined when a fault is executed but no failure is detected. Empirical study showed that coincidental correctness is prevalent and demonstrated that it is a safety reducing factor for coverage-based fault localization [4]. Masri [5] introduced the strategy of cleansing test suites from coincidental correctness to enhance fault localization. Wang [26] proposed an approach which refine code coverage of test runs using context patterns including control and data-flow patterns prescribed by different fault types, and thus strengthen the correlations between program failures and the coverage of faulty program entities. Furthermore, Miao[6] employed an k-means clustering based technique to identify coincidental correctness.

Support vector machine is an important supervised learning models in machine learning domain, it has already been applied to solve various real world problems including medical science [27], text classification[14], image retrieval[28], protein subcellular localization[29] and so on. The idea of the SVM ensemble has been released in [7], with further study in [30]. The advantage of SVM ensemble is that it can improve the accuracy of prediction. In recent years, the use of ensemble learning approaches for mislabeled detection has become increasingly popular with extensive research[19][20].

External Threats. The subject programs used in this paper are Java applications with hand-seeded faults. Different languages may have different features and thus the performance of the proposed technique for different cases may vary. Hand-seeded faults may not exhibit the characteristics of real faults [25] and thus the proposed technique may exhibit varying performances. Another issue related to the external threats is that we imported the ‘e1071’ package in R and called ’SVM’ function to simulate the classification process.

Internal Threats. We implemented the Ochiai metric using Java scripts instead of employing any fault localization tools. A tracking application developed by our team monitored the intermediate values computed for each statement of the program including all the statistics mentioned in section II-A. A random testing methodology was then utilized to test the accuracy of the intermediate values computed. We activate each fault manually without calling any scripts. Furthermore, we instrumented each block manually to capture the statistics.

Construct Threats. For the evaluation of prediction, we used accuracy, precision, and recall to measure the performance. However, in practice there may be other metrics and representation demonstrating how well a classifier performs. We utilized the formulas in [21] to compute the expense and suspiciousness scores for each statement. In practice, there might be other computation methodologies to assess the suspiciousness of statements in a given program.
IX. CONCLUSION

The presence of coincidental correctness hinders the effectiveness of coverage-based faults localization. In this paper, we proposed a technique which first employs support vector machine ensemble to detect coincidental correctness, then trimming the test suite by removing or flipping the detected coincidentally correct test cases. Based on our experimental study and analysis, we concluded that 1) the coincidental correctness occurs frequently in Java faulty programs; 2) the proposed technique could identify coincidental correctness with high accuracy, precision and recall; 3) flipping the coincidental correctness could reduce the cost more effectively compared to removing in terms of the localizing the bug.

Further experiments are needed to validate the results we obtained in this paper. In particular, experiments on either other language based programs, and the study on multiple faults programs are of interest. Also, the idea of the proposed technique is to use machine learning algorithm, which are grounded on features study. In practice there are other ways to construct features, we would explore further in this direction.

We also plan to continue extending our study. There exists some other clustering or classification techniques, and we would explore the possibility of applying them into coincidental correctness detection for enhancing faults localization.

REFERENCES