First, Debug the Test Oracle

Xinrui Guo, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun

Abstract—Opposing to the oracle assumption, a trustworthy test oracle is not always available in real practice. Since manually written oracles and human judgements are still widely used, testers and programmers are in fact facing a high risk of erroneous test oracles. However, test oracle errors can bring much confusion thus causing extra time consumption in the debugging process. As substantiated by our experiment on the Siemens Test Suite, automatic fault localization algorithms suffer severely from erroneous test oracles, which impede them from reducing debugging time to the full extent. This paper proposes a simple but effective approach to debug the test oracle. Based on the observation that test cases covering similar lines of code usually generate similar results, we are able to identify suspicious test cases that are differently judged by the test oracle from their neighbors. To validate the effectiveness of our approach, experiments are conducted on both the Siemens Test Suite and `grep`. The results show that averagely over 75 percent of the highlighted test cases are actually test oracle errors. Moreover, performance of fault localization algorithms recovered remarkably with the debugged oracles.

Index Terms—Test oracle, debugging, spectrum-based fault localization

1 Introduction

A fundamental assumption in software testing, known as "the Oracle Assumption", is that there exists some mechanism, or an oracle, that is trustworthy to adjudge whether a test case is executed correctly as expected [1]. This assumption is so commonly accepted that the real world scenario, i.e., such oracles are hard or even impossible to obtain, is generally ignored. Confronting the large variety and increasing complexity of software systems, automatic test oracle generation techniques are still far from adequate [2]. In fact, manual result checking is still employed as a major approach in industry [3]. In other words, human judgement constitutes the vast majority of test oracles in practical use.

One natural but severe problem about manual test oracles is that they are error prone [3], thus breaking the oracle assumption immediately-testers are not able to decide with full correctness whether the software under test passes or fails the input test case. Sources for this unreliability exist in various ways. For complex programs, testers may be incapable of understanding the programs' functionality accurately thus making wrong judgements of the test results. Another case is when the software system needs to be tested under too large a test suite that it is nearly impossible to be correct at every judgement. In [4], the author mentioned the case where test inputs are automatically generated from test generation tools. These inputs can be unrealistic, which requires extra effort to understand, making manual judgement of test cases harder. These sources naturally exist in software developing process, therefore testers and programmers are in fact facing a high risk of erroneous test oracles.

Errors in test oracles directly hinder the debugging process, conducted either manually or aided by automatic fault localization techniques. Generally there are two kinds of errors: (1) The oracle judges the test case to be a "pass" while the software actually fails the test (called a "mis-pass"); (2) The oracle judges "fail" while the software actually passes (called a "mis-fail"). For human debuggers, "mis-passed" test cases waste precious opportunities to discover a bug since passed tests are usually discarded or only kept for regression testing [5]. Pity as it is, "mis-passed" test cases are sometimes acceptable since with large enough test suites there will always be another test case exposing the same bug. "Mis-failed" test cases, on the other hand, are misleading and even costly. Debuggers may waste much of their time on solving a bug which does not even exist. Despite the expensive time and labour consumption of manual debugging, it is still possible that errors in test oracles are corrected along with the bugs in target program. Automatic fault localization techniques, however, trust the test oracle completely. Moreover, as presented in Section 2, techniques with better accuracy tend to be more sensitive to test oracle errors. With misleading oracles, these techniques are prevented from helping saving debugging time to their full power.

A direct way to eliminate or ameliorate the impact of oracle errors is to discard the original test oracle and build a new one. However, the new oracle could be as erroneous as the original one, let alone the expensive cost of time. Besides, regardless of the fraction of errors, existing test oracles are valuable in providing most of the correct judgements of test results. Therefore a better solution is to utilize the existing test oracle and automatically correct the errors, or to "debug" it. In this paper, we propose an automatic approach to debug an "erroneous" test oracle, i.e., a test oracle that makes erroneous judgements ("mis-passes" and "mis-fails") on test results, before it is used to debug the program. Our approach is based on a simple observation

- *X. Guo, M. Zhou, M. Gu, and J. Sun are with the School of Software, Tsinghua University, Key Laboratory for Information System Security, Ministry of Education and Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China. E-mail: {gxr12, zhoumin03}@mail.tsinghua.edu.cn, {guming, sunjg}@tsinghua.edu.cn.*
- *X. Song is with the Department of Electrical and Computer Engineering, Portland State University, Portland, OR. E-mail:song@ece.pdx.edu.*

Manuscript received 29 May 2014; revised 16 Dec. 2014; accepted 20 Jan. 2015. Date of publication 21 Apr. 2015; date of current version 16 Oct. 2015. Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2425392

TABLE 1
Debugging Process of a Sorting Program

Input: (a, b, c)	(1, 2, 3)	(1, 3, 2)	(2, 1, 3)	(2, 3, 1)	(3, 1, 2)	(3, 2, 1)	$Suspicion_c$	$Suspicion_e$
1 `if (a > b){`	•	•	•	•	•	•	0.5	0.5
2 `swap(a, b);`			•		•	•	0.5	1.0
3 `if(b > c){`	•	•	•	•	•	•	0.5	0.5
4 `swap(b, c);`	•			•	•	•	1.0	0.6
5 **`if(a < b)`**	•			•	•	•	1.0	0.6
6 `swap(a, b);`	•				•		1.0	0.3
7 `}`							–	–
Oracle(correct)	×	✓	✓	×	×	×		
Oracle(error)	✓	✓	×	×	×	×		

inspired by works in field of coverage-based test suite reduction [6], [7]—tests with similar execution traces are likely to have identical test results. We try to measure the similarity between test cases and identify the test cases with unusual test results comparing to their nearest neighbours. When this "unusualness", quantified as "suspicion", reaches a threshold, we mark the corresponding test result judgement as wrong and debug it by changing the judgement to the opposite. The procedure is simple but effective. For a comprehensive evaluation, we adopted two sets of programs from SIR [8] as our subject, namely, the Siemens Test Suites and `grep`. The former is a widely used benchmark in field of fault localization while the latter is a real world program scaling up to over 13,000 lines of code. Using the Siemens Test Suites, we managed to generate erroneous test oracles for all seven programs and their 132 variants (three of which were excluded as no test cases fail on them) with error rate (meaning the ratio of faulty judgements in all judgements made by the test oracle) ranging from 0.01 to 0.1. For the five versions of `grep` provided, erroneous test oracles were generated for four of them (v5 was excluded because no test case could reveal the bug) with the same error rate range. To evaluate the oracle debugging process solely, we compared the debugged oracles with the correct ones. Results show that on average over 75 percent of the debugged test results are actually test oracle faults. Then we applied four automatic fault localization techniques on the correct, erroneous and debugged oracles respectively and compared their accuracy. Experiments show that accuracy of different fault localization techniques suffered to various degrees on the erroneous test oracles, but recovered largely on the debugged oracles.

The contributions of this paper are as follows. First we conducted an experiment on the Siemens Test Suite to demonstrate how errors in test oracles can deteriorate the accuracy of automatic fault localization techniques. To the best of our knowledge this is the first empirical study on fault localization accuracy under erroneous test oracles. Second we proposed a simple but effective approach to automatically correct the errors in test oracles is proposed. Experiments show that 75 percent errors are successfully fixed and accuracy of fault localization techniques recovered noticeably on the debugged oracles. During this experiment we also analysed quantitatively how different kinds of test oracle errors, namely, "mis-fails" and "mis-passes" can impact fault localization accuracy.

In the rest of this paper, we first demonstrate how errors in test oracles affect the debugging process in Section 2. In Section 3, we explain in detail our approach to debug the test oracle. Section 4 presents the experimental results, regarding the effect of our approach on both debugging the test oracle and the corresponding enhancement on accuracy of fault localization techniques. Section 5 discusses how "recall" and "false passes/fails" influence the fault localization accuracy. Section 6 analyses the threats to validity. Related work is introduced in Section 7 and we conclude in Section 8.

2 Oracle Errors' Impact on Debugging Process

An error in a test oracle is a faulty judgement of a test result. Programmers directed by these faulty judgements may find it very confusing if an already "passed" test case later crashed the system, or an actually successful execution is reported to be "fail". More upsetting than confusion is the associated extra and unnecessary time-consumption. Manual debuggers may need to trace the test case fully to confirm or negate the bug. Programmers aided by fault localization techniques may sense that the technique is not saving him time as its accuracy is compromised by misleading test oracles. These frustrations are better illustrated through the following example program.

2.1 An Illustrative Example

Table 1 presents a simple sorting program and a possible applied test suite. The program is intended to take three real numbers as its input, and sort them in ascending order. Unfortunately the programmer mistakenly reversed the direction of the third comparison (highlighted in red). The test suite consists of six test cases, covering six major different categories of the initial input. The execution trace of each input is marked by dots and blanks, representing a hit or a miss of the corresponding line. Two test oracles (correct and erroneous respectively) are displayed below the traces. A tick means a pass, while a cross means a fail. The erroneous oracle mis-judged the first and third test cases, which are marked in red.

2.1.1 *Impact on Manual Debugging Process*

For manual debugging process, traces and test results are almost all the information available. A typical programmer

will scrutinize the failed test cases one by one. With the correct oracle, he would examine the first failed test case, i.e., $(1, 2, 3)$. Following its trace, the programmer will notice the wrong comparison as soon as he reaches the faulty line. After the correction, all test cases should be passed and the debugging process stops. With the erroneous oracle, however, the programmer would neglect $(1, 2, 3)$, and waste his time on $(2, 1, 3)$, only to meet a correct output that conflicts the test oracle. For simple programs as in this case, the programmer can be sure that this is an oracle error. With a complex program, he may need to further consult and discuss with the testers to reach an agreement. Having solved this conflict, he finally comes to $(2, 3, 1)$ where the bug is ultimately fixed. In summary, the erroneous oracle costs the programmer an opportunity to repair the bug at early stage and a precious period of time to confirm the oracle error.

2.1.2 Impact on Automatic Fault Localization Algorithms

With the development of test automation, programmers can utilize automatic fault localization algorithms to save debugging time. An automatic fault localization algorithm aims to provide an ordered checking list of code lines that ranks the real cause of bug as front as possible. *Spectrum-based Fault Localization (SFL)* is a family of light-weight automatic fault localization algorithms. They take as input the execution traces, called "spectra", and test results, called "diagnosis". According to spectra and diagnosis, they first count four values for each line of code c, a_{ep}, a_{ef}, a_{np} and a_{nf}, demonstrating respectively the number of test cases that covered c and passed, covered c but failed, did not cover c and passed, and did not cover c but failed. Based on these statistics, a metric is applied to calculate the "suspicion value" of c. Finally, lines with higher suspicion value are ranked higher in the checking list.

For instance, a widely known SFL algorithm is Tarantula [9]. Tarantula decides the suspicion of each line as

$$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}.$$

The intuition of Tarantula is simple: for a certain line of code, larger proportion of failed test cases that hit this line indicates a higher suspicion. In our example, we list the suspicion value suggested by Tarantula using both the correct and erroneous oracle in the last two columns in Table 1. With the correct oracle, Tarantula reports three most suspicious lines, 4, 5 and 6. Suppose the programmer checks lines with equal suspicion in random order. Then according to this judgement, the programmer checks randomly among line 4 to 6 and is expected to reach the bug at the second line he checked. With the erroneous oracle, however, line 2 is reported to be most suspicious. Therefore before he moves on to line 4 and 5, the programmer will need to rule out line 2. As a result, he is expected to find the bug at the 2.5th line he checked.

Tarantula is impeded from saving the programmer's debugging time by two erroneous judgements in the test oracle. Though an extra 0.5 line of code to examine seems harmless, we must notice that this is only a seven-line small program. As shown in the next section, the proportion of extra code that needs to be examined is rather stable with respect to scale of the program. With a larger program, many more lines will need to be examined before finding the real cause of fault.

2.2 Experiment on the Siemens Test Suites

In support of intuitive analysis, we conducted experiments on the Siemens Test Suites, which contains seven different programs and their 132 variants. Detailed information of the Siemens Test Suite is presented in Section 2.2.3. Unfortunately, with such volume of test suites, it is nearly impossible to impartially estimate the extra time consumption due to oracle errors during human debugging process. If two test oracles for a same program are given to a same person, he would be much more familiar with the program as he debugs the second time, resulting in inaccurate time difference. If given to two different people, however, differences in experience and capability will also cause inaccuracy. To fairly reflect the impact of test oracle errors, we choose four representative SFL algorithms, and calculate how their accuracy is compromised alongside the increase of oracle errors.

2.2.1 Accuracy of SFL Algorithms

If the program under test contains only one fault, a natural criterion to score the accuracy of SFL algorithms is the percentage of code that a programmer needs to examine before he finds the bug. However real world programs usually contain multiple faults. In this case, we assume that the programmer would fix only one bug at a time and run SFL algorithms again for further bug fixing, which is referred to as "one-at-a-time" mode in [10]. Therefore we define accuracy of SFL algorithms as the percentage of code that a programmer needs to examine before he or she finds the first bug.

In cases where several lines are judged to be equally suspicious, we assume that there exists a certain tie-breaking scheme that the programmer adopts for all programs. Then we can reasonably estimate from an overall perspective the real faulty line of code would appear in a random position among his (or her) checking list. Based on this assumption, the quantitative measurement of accuracy is defined as follows:

For a program containing N lines, let $\{l_{f1}, \ldots, l_{fk}\} \subset \{1, 2, \ldots, N\}$ denote the faulty lines with the highest suspicious value, and sus_j denote the suspicion value of line j, then the score s of SFL applied to this program is defined as

$$s = \frac{|\{j | sus_j < sus_{l_{f1}}\}|}{N} + \frac{|\{j | sus_j = sus_{l_{f1}}\}|}{k+1}. \quad (1)$$

Obviously, lower score means less effort at debugging and accordingly better fault localization performance.

2.2.2 Selection of SFL Algorithms

Different SFL algorithms adopt different suspiciousness metrics. There have been more than 30 SFL algorithms proposed and their performance vary. Due to limited space we are only able to select and experiment on a representative fraction of them.

TABLE 2
Metrics of Included SFL Algorithms

Name	Formula
Tarantula	$\frac{a_{ef}}{a_{ef}+a_{nf}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{nf}+a_{ef})(a_{ef}+a_{ep})}}$
Naish2	$a_{ef}-\frac{a_{ep}}{a_{ep}+a_{np}+1}$
Russel&Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$

TABLE 3
Brief of Siemens Test Suites

Program	Versions	Tests	Description
print_token	7	4,130	lexical analyzer
print_token2	10	4,155	lexical analyzer
replace	32	5,542	pattern recognition
scheduler	9	2,650	priority scheduler
scheduler2	10	2,710	priority scheduler
totinfo	23	1,608	altitude separation
tcas	41	1,052	information measure

For reasonable selection of evaluated metrics, we referred to existing empirical and theoretical studies and their conclusions are as follows. Empirical studies [11], [12] both identified that Ochiai is consistently more accurate than Jaccard and Tarantula. Later through theoretical analysis [13], Xie et al. confirmed that Ochiai is better than Jaccard, which is better than Tarantula. They also identified a group of metrics, ER1 (including Naish1 and Naish2) to be better than Ochiai and all other metrics except for ER5 (including Wong1, Russel&Rao and Binary). Therefore ER1 and ER5 were stated to be metrics with "maximal" accuracy. However results of empirical study in [14] states that Ochiai is statistically better than ER5 and ER1, and Tarantula's score is better than Naish1 and ER5. They owe this contradiction to the unrealistic 100 percent coverage assumption in [13].

Striving to present a comprehensive evaluation within limited space, we believe that the selected metrics should:

1) cover widely used metrics or metrics of best performance (when evaluated using correct oracles),
2) be applicable to general scenarios, i.e., both single- and multi-fault scenarios and
3) have better been evaluated empirically in early works for result validation.

Comparing these requirements with the conclusions from existing works, our selection are made as follows. We first selected Naish2 from ER1 and Russel&Rao from ER5 to represent the metrics of theoretical "maximal" accuracy. Wong1 yields identical ranking with Russel&Rao under all conditions and are therefore represented. The reason not to include Naish1 and Binary is that their prominence are only proved under "single-fault" condition. Then we selected Ochiai and Tarantula as Ochiai has been reported to performed best and Tarantula covered in more empirical studies than Jaccard.

Finally, we made the selection of four SFL metrics, namely Tarantula, Ochiai, Naish2 and Russel&Rao. The corresponding formula of metrics are listed in Table 2.

2.2.3 Experiment Design

To quantify the influence of oracle errors, both erroneous and correct versions of the tested program is mandatory. However, correct oracles are difficult to acquire. One way to circumvent this obstacle is to generate standard outputs from a known correct program. This leads us to the Siemens Test Suites provided by SIR.

The Siemens Test Suites is a widely used test bench for evaluating the accuracy of fault localization algorithms. As presented in Table 3, it contains seven different real world C programs. For each program, SIR provides the correct version, a collection of faulty variants and a corresponding test suite. All variants are single-faulted, but some types of the faults, for example wrong value in "#define" statements or missing statements, can influence a great proportion of the program and may mimic the effect of multi-faults. With these basic facilities and assistant tool `gcov`, we are able to generate the following sequence of data for a particular faulty program version:

1) A correct oracle of the program by executing the test suite against the correct version.
2) A simulated faulty oracle at error rate r by randomly mutating the correct oracle at a mutate ratio $mr = r$. To be specific, we used a random number generator to generate a random double value in $[0, 1)$. If the value is less than r, then we mutate the test result judged by the test oracle to the opposite. Based on the randomness of mutation, the error rate of the generated faulty oracle should also be r. Therefore we do not distinguish between mutation rate and error rate of a test oracle mutant in the rest of the paper.
3) The spectra of the program by executing the test suite against the program and invoking `gcov` through the process.
4) The fault localization score s_c with the correct oracle of the four SFL algorithms.
5) The fault localization score s_f with the faulty oracle of the four SFL algorithms.
6) The accuracy compromise $s_f - s_c$ of the four SFL algorithms.

For all 132 variants, we recorded the accuracy compromise as mr ranges from 0.01 to 0.1 with interval 0.01 for four SFL algorithms. Statistics are presented in the next section.

2.2.4 Experimental Results

As a ground basis, we first applied four SFL algorithms to faulty variants of the Siemens Test Suites with the correct test oracles and recorded their score. The results are presented in the left graph of Fig. 1. Let V be the set of all faulty variants and $v_i \in V$ be an arbitrary variant. Let $\Omega = \{Naish2, Ochiai, RusselRao, Tarantula\}$ be the set of concerned SFL algorithms and $\omega_i \in \Omega$ be one of them. Let s_x on "score" axis be a certain score value. Then the corresponding value on "proportion" axis is defined as

$$p_{\omega_j}(s_x) = \frac{|\{k|s(v_k, \omega_j) \leq s_x\}|}{|V|}.$$

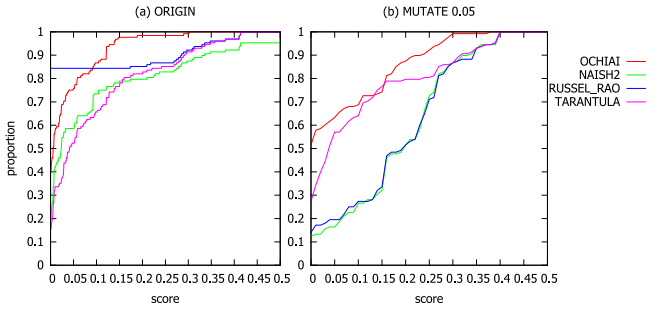


Fig. 1. Score distribution of four SFL algorithms with correct and $mr = 0.05$ oracles.

In other words, $p_{\omega_j}(s_x)$ represents the proportion of variants with better score than s_x with the corresponding SFL algorithms.

As shown in Fig. 1, Russel&Rao performed best since the faulty line of over 80 percent of V is ranked first. Ochiai is the second as it can localize the fault within 15 percent of the code on over 95 percent of V. Naish2 is moderately better than Tarantula, but both performed inferior than Ochiai. This is in accordance with empirical study [15] as Ochiai outperforms Tarantula consistently. It also partially agrees with [14] as Ochiai performs superior than Naish2, but inferior than Russel&Rao. This is because [14] compares the average accuracy of all program variants while we plotted the proportion of program variants with accuracy score less than s_x when s_x walks through $[0, 1]$. Results from both us and former empirical studies contradict the conclusion in [13] as Naish2 should be better than Ochiai theoretically. In addition to the unrealistic assumption of 100 percent coverage pointed out by [14], we believe there are another two reasons causing this disagreement. First proofs in [13] are based on "single fault assumption". Though each faulty version of the Siemens programs contains only one fault, certain types of fault have the effect of multi-fault. For example in eight variants the error is a faulty constant value defined in a "#define" declaration. In one version a variable is declared in type "FLOAT" while it should be a "DOUBLE". Another reason is that missing statement errors are not considered in the theoretical analysis, but in 13 variants a complete statement is missing while missing conditions in "IF" statements are much more common. Nevertheless, all four algorithms can successfully locate the bug of over 80 percent of V within less than 20 percent of the code.

To quantify the impact of errors in oracles, let $s_0(v_i, \omega_j)$ denote the score of ω_j applied to v_i with the correct test oracles, and $s_r(v_i, \omega_j)$ with test oracles of an error rate at r, then the absolute performance compromise of ω_j for v_i at r is defined as

$$\Delta^{-}_{|\cdot|}(\omega_j, v_i, r) = s_r(v_i, \omega_j) - s_0(v_i, \omega_j).$$

Based on the correct oracles, we managed to generate erroneous oracles by randomly negate the correct oracle at mutation rate r. As r ranges from 0.01 to 0.1, performance of SFL algorithms is compromised at different degree. For a first impression, the score-proportion slice when $r = 0.05$ is presented in the right of Fig. 1. As shown in the figure, there is a moderate decline of $p(s_x)$ when $s_x \in [0.15, 0.25]$ for Tarantula. Curves of Ochiai, Naish2 and Russel&Rao cave in remarkably. Originally Ochiai is able to find the fault within 15 percent of the code for over 95 percent of V, while now this ratio drops to less than 75 percent. For Naish2 and Russel&Rao, only 30 percent of V can be debugged within 15 percent of code, contrasting to the original 80 percent.

For a complete view of performance compromise, we recorded $\Delta^{-}_{|\cdot|}(\omega_j, v_i, r)$ for all $\omega_j \in \Omega$, $v_i \in V$, $r \in \{0.01, 0.02, \ldots, 0.1\}$. This is presented in four separate graphs of Fig. 2, depicting absolute performance compromise of each algorithm respectively.

In Fig. 2, with a certain error rate r, the "proportion" value p relative to a certain value Δ_x with respect to a certain SFL algorithm ω_j is defined as

$$p_{\omega_j}(\Delta_x, r) = \frac{|\{k|\Delta^{-}_{|\cdot|}(\omega_j, v_k, r) \leq \Delta_x\}|}{|V|}$$

that is, the ratio of variants whose score compromised less than Δ_x. Value of p is visualized by a colour spectra. Contour stating value in $[0.5, 1)$ are plotted on both the surface and projected on x-y panel at interval 0.1.

As is shown in the figure, different SFL algorithms suffer differently. In accordance with the slice at $r = 0.05$, Tarantula is the most stubborn algorithm. For averagely 80 percent programs only 2 percent extra code needs to examined. Ochiai is the next, suffering over 15 percent loss of fault localization accuracy on around 20 percent programs. Score compromise of Naish2 and Russel&Rao are similar. Approximately accuracy dropped over 15 percent on over 50 percent

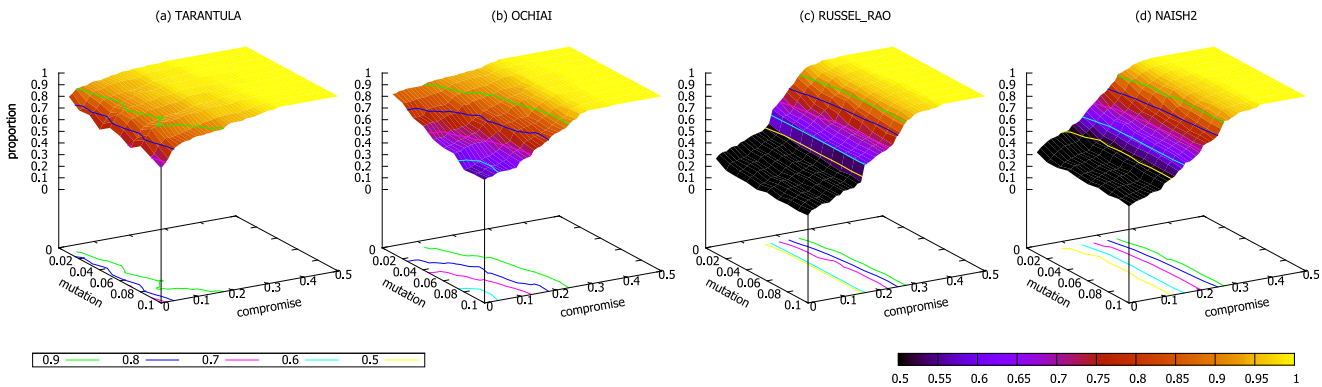


Fig. 2. Accuracy compromise of four SFL algorithms as mr ranges from 0.01 to 0.1.

programs. One must notice that we recorded absolute compromise instead of relative compromise. Therefore 15 percent drop maps to 15 percent of actual lines of code, which will lead to an observably increase of debugging effort.

This result is not coincidence. For Russel&Rao, a_{ef} is the only factor that decides the final ranking result. As we randomly mutate the oracle judgements, the "mis-fails" will cause an increase in a_{ef} for many lines of code, therefore the real faulty line of code will not be ranked in a position as high as with the correct oracle. For Naish2, since $\frac{a_{ep}}{a_{ep}+a_{np}+1}$ is negligible compared to a_{ep}, the decisive factor is still a_{ep}. Therefore the oracle errors should impact Naish2 similarly as Russel&Rao. For Ochiai, since the execution trace of each test case is unaffected by oracle errors, the only part affected is $\frac{a_{ef}}{\sqrt{a_{ef}+a_{nf}}} = \sqrt{a_{ef}}\sqrt{\frac{a_{ef}}{a_{ef}+a_{nf}}}$. Since we mutated the oracle judgements randomly, $\frac{a_{ef}}{a_{ef}+a_{nf}}$ should remain almost the same, then the affected factor is left to be $\sqrt{a_{ef}}$. This explains why Ochiai is less affected than Russel&Rao and Naish2, but shows similar trend in the accuracy compromise. Finally for Tarantula, both $\frac{a_{ef}}{a_{ef}+a_{nf}}$ and $\frac{a_{ep}}{a_{ep}+a_{np}}$ be affected limitedly, which agrees with the result that Tarantula appears to be fault-tolerant.

A reasonable doubt may rise here. Since Tarantula is fault-tolerant, why bother correcting oracle errors or trying other algorithms? First, Tarantula performs inferior to Ochiai, Naish2 and Russel&Rao with correct oracles. Therefore with a corrected or partially corrected oracle, Ochiai, Naish2 and Russel&Rao may still outperform Tarantula, and it would be unwise to discard them. Second, Tarantula is not completely fault-tolerant. Debug the test oracle will certainly improve its accuracy as well. What's more, results for different programs may vary and Tarantula may not be fault-tolerant all the time. To sum up, existence of Tarantula does not diminish the importance of a correct oracle.

Reading from the contour projected on x-y panel, performance compromise is seen to aggravate as error rate r increases. This is more obviously reflected by Ochiai, as projections of contour lines deviate axis "mutation" with growing error rate. Though we did not conduct experiments with $r \geq 0.1$, we are confident that this trend will still hold.

An interesting phenomenon seen here is that the two theoretically optimal algorithms suffered much more accuracy compromise than Ochiai, which suffered more than Tarantula. This is in accordance with the intuition that optimal algorithms should be more sensitive to the "quality" of the test results. However our empirical study results show that Naish2 performed inferior than Ochiai. Therefore the performance hierarchy of these algorithms under both correct and erroneous oracles remains to be an open issue. Nevertheless, our experimental results show that essentially effective fault localization algorithms suffer from errors in test oracles. Higher error rate leads to larger performance compromise. Therefore it is necessary to debug the oracle before it is used to debug the program.

3 Debug the Test Oracle

Despite the necessity to be eliminated, errors in a test oracle are hard to identify. First of all, test oracles are designed to examine the program rather than being examined by other facilities. In other words, there is no other criteria to decide the correctness of test oracles' judgements. Since test oracles are fundamentally established on human understanding of the expected behaviour of the program, a manual double-check seems to be a solution. However, a thorough check of the test oracle can be as time-consuming as its creation. Even if the time consumption is considered as acceptable, testers may make the same mistakes again since no other source can point out their error, which makes the double-checking process meaningless.

Fortunately, for most situations, oracles may contain only a small fraction of errors. With help of the correct majority, we may be able to predict the programs' behaviour to some degree and decide whether the program "should" pass a certain test case or not.

Inspired by work in field of coverage-based test suite reduction [6], [7], we observed that test cases covering similar execution traces usually "pass" or "fail" simultaneously. The authors stated that if block coverage is maintained, test suites can be reduced greatly without excessive decrease of fault detection effectiveness. In other words, test cases sharing the same or similar execution traces, thus achieving the same or similar block coverage goals, often share the same test results (a "fail" or a "pass"). Based on this observation, we propose to utilize the coverage information of each test case and correct the unusual oracle judgements.

Algorithm 1 presents the main framework of our approach. It takes as input the program P, test oracle $O(T)$ for test suite T, the number of voting test cases n and a suspicion threshold $thres$, and return a debugged test oracle $O'(T)$. There are mainly two phases of the procedure. The first phase tests all $t_i \in T$ and record the execution traces $Tr(P(t_i))$. In the second phase, we first find n test cases $T_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\}$ whose execution traces are most similar, or nearest to t_i. Then we adopt a voting strategy and let T_i decide if t_i is supposed to pass or fail. The voting strategy returns a $Suspicion$ value which indicates the possibility that $O(t_i)$ is wrong. Suppose the voters in T_i decide that t_i should be "passed" while the input oracle judged it to be "fail", and that $Suspicion(t_i)$ is above the debug threshold $thres$, then the judgement is changed to "pass" in the output oracle O'.

Algorithm 1. Correct the Test Oracle

Input: P, $O(T)$, n, $thres$
Output: $O'(T)$

```
1: for each t_i ∈ T do
2:    Record Tr(P(t_i));
3: end for
4: for each t_i ∈ T do
5:    Find T_i = {t_i1, t_i2, ..., t_in} ⊂ T nearest to t_i;
6:    Suspicion(t_i) = vote (T_i);
7:    if Suspition(t_i) > thres then
8:       O'(t_i) = ¬O(t_i);
9:    else
10:      O'(t_i) = O(t_i);
11:   end if
12: end for
13: return O'(T);
```

To implement the framework in detail, several issues need to be solved. First, we need to record the execution trace of test cases. Solution to this problem is not discussed in this paper since there are many tools available. Second, we need to define a metric to measure the "similarity" between test cases according to their execution traces. This is critical as it directly influences the composition of T_i, which will further decide $O'(t_i)$. The voting strategy and the value of all parameters required, including the number of voting cases n, the debug threshold $thres$ should also be carefully selected or estimated. These implementation decisions affect not only the accuracy of oracle debugging, but also time and space complexity of the debugging process. We discuss these problems in the following sections.

3.1 Similarity Measurement

A proper similarity metric should cluster similar test cases and meanwhile disperse the dissimilar ones. Let $tr_i = \{i_0, i_1, \ldots, i_{n_i}\}$ denote the lines covered by test case t_i, or the *trace set* of t_i. A widely used metric to measure the similarity between two sets A and B is probably $\frac{|A \cap B|}{|A \cup B|}$. Applying it to our case, similarity between two test cases t_i, t_j and their corresponding trace set tr_i, tr_j could be defined as $\frac{|tr_i \cap tr_j|}{|tr_i \cup tr_j|}$, i.e. the more common lines they share, the more similar they should be considered. However, practically many lines of code are executed commonly by many test cases. As these lines take up much of the traces, two rather different test cases may be calculated to be pretty similar.

Generally, line covered by less test cases are more distinctive and should be weighted more when calculating similarity. To evaluate the importance of code lines, we refer to a commonly used weighting metric called $tf - idf$ [16]. $Tf - idf$, short for *term frequency - inverse document frequency*, is originally used as a numerical statistic that reflects how representative a word is to a collection of corpus. Let D be the document collection, $d \in D$ be a document and t be a term that appears in D. *Term frequency* $tf(t, d)$ is the number of times t appears in d. *Inverse document frequency* is a measure of how frequently t appears in D:

$$idf(t, D) = \log \frac{|D|}{|\{d | d \in D \wedge t \in d\}|}.$$

Then for document d, the importance of $t \in d$ is calculated as

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D).$$

In our case, each line of code can be seen as a term, and each execution trace as a document. Since we only recorded the covered set of lines, for $t_i \in T,\ i_p \in tr_i,\ tf(i_p, t_i) = 1$. Therefore,

$$tfidf(i_p, t_i, T) = idf(i_p, T) = \log \frac{|T|}{|\{t_l | t_l \in T \wedge i_p \in t_l\}|}.$$

Putting weight on the covered lines, we finally define the similarity between t_i and t_j as

$$Sim(t_i, t_j) = \frac{\sum_{i_p \in tr_i \cap tr_j} tfidf(i_p, t_i, T)}{\sum_{i_p \in tr_i \cup tr_j} tfidf(i_p, t_i, T)}.$$

Notice here that $tfidf(i_p, t_i, T)$ is in fact irrelevant to t_i, therefore $Sim(t_i, t_j) = Sim(t_j, t_i)$. In addition, $tr_i \cap tr_j \subset tr_i \cup tr_j$, so $0 \leq Sim(t_i, t_j) \leq 1$ always holds.

3.2 The Voting Strategy

Test cases with the highest similarity form the "jury" for t_i. They decide the "supposed" test result of t_i following a simple voting strategy. Let $T_{i,p} = \{t_{iq} | O(t_{iq}) = \mathcal{P}, t_{iq} \in T_i\}$ and $T_{i,f} = \{t_{iq} | O(t_{iq}) = \mathcal{F}, t_{iq} \in T_i\}$, representing the test cases judged by the original oracle O to be "pass" and "fail" respectively. We define $Vote_{ip} = \sum_{t_{iq} \in T_{i,p}} Sim(t_i, t_{iq})$ and $Vote_{if} = \sum_{t_{iq} \in T_{i,f}} Sim(t_i, t_{iq})$ as the count of votes for "pass" of "fail" respectively. Then suspicion value of t_i is calculated as

$$Suspicion(t_i) = \begin{cases} \frac{Vote_{ip}}{Vote_{ip} + Vote_{if}} & if\ O(t_i) = \mathcal{F} \\ \frac{Vote_{if}}{Vote_{ip} + Vote_{if}} & if\ O(t_i) = \mathcal{P}. \end{cases}$$

3.3 Parameter Settings

As parameters to Algorithm 1, it is important that the number of voting cases n and the debug threshold $thres$ are chosen properly. Intuitively, if n is too small, for example $n = 2$, it is very likely that one of the voting two test cases is mis-judged by the test oracle itself. In this case, the debugging result would be very error prone. If n is too large, for example $n = 20$, it is possible that many of the execution traces of these test cases are not similar enough to represent the targeted test case, in which case the debugging result would also be unreliable. Meantime, if $thres$ is set too high, it is possible that many of the test oracle errors are not corrected, i.e., there is a high false negative rate. If $thres$ is set too low, it is possible that many of the test oracle judgements are wrongly modified, i.e., there is a high false positive rate.

However, choosing proper values to achieve best performance is not easy. An ideal set of parameter value should let as many oracle errors be fixed and as much correct part of oracle judgements be remained, or in other words, minimize false negatives and false positives. Unfortunately, without an existing correct oracle, it is impossible to count exactly how many false negatives and false positives are generated with a certain n and $thres$. Therefore even if we traverse all possible combination of n and $thres$ (which is obviously impractical), we are still not able to identify the optimal parameter settings before we check the lines of code suggested by SFL algorithms.

To help with the choice of n and $thres$, we performed probability analysis and managed to quantitatively estimate the number of false negatives and false positives with respect to n and $thres$ without a correct test oracle available. The analysis is based on two assumptions:

Assumption 1. $\forall t_{iq} \in T_i \cup \{t_i\}, O_c(t_{iq}) \in \{\mathcal{P}, \mathcal{F}\}$ *i.i.d.* .

Reason. The test result of one test case does not affect the test result of another, therefore for each $t_{iq} \in T_i \cup \{t_i\}$, $O_c(t_{iq})$ is independent of each other. Since we assume that all $t_{iq} \in T_i$ are similar enough to t_i, it is reasonable to assume that the probability distribution of their test results are identical.

Assumption 2. $\forall t_{iq_1}, t_{iq_2} \in T_i \cup \{t_i\}, iq_1 \neq iq_2$,
$\mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = sim_i\ (constant)$.

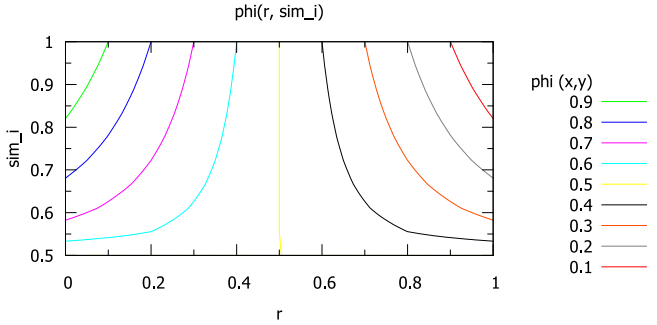


Fig. 3. Value range of ϕ_i relative to r and sim_i.

Reason. Since $O_c(t_{iq}) \in \{\mathcal{P}, \mathcal{F}\}$ $i.i.d.$, it is obvious that $\exists \alpha_i \in [0,1]$ is a constant that $\forall t_{iq_1}, t_{iq_2} \in T_i \cup \{t_i\}$, $iq_1 \neq iq_2$, $\mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = \alpha_i$. According to our observation, test cases with similar execution traces are more likely to pass or fail simultaneously. Therefore it is reasonable that we use sim_i as an estimation of α_i.

Under these two assumptions, we conclude that for any $t_i \in T$, the probability that the debugging result of t_i is a false negative, is

$$\mathbf{P}(fn(t_i)) \approx r \sum_{w=0}^{\hat{n}} \mathrm{C}_n^w \phi_i^w (1-\phi_i)^{(n-w)} \tag{2}$$

and the probability that the debugging result of t_i is a false positive, is

$$\mathbf{P}(fp(t_i)) \approx (1-r) \sum_{w=\hat{n}+1}^{n} \mathrm{C}_n^w (1-\phi_i)^w \phi_i^{(n-w)}, \tag{3}$$

where $\phi_i = \beta_{i2} + r - 2\beta_{i2} r$, $\beta_{i2} = \frac{1+\sqrt{2sim_i - 1}}{2}$, sim_i is the average similarity of all test cases in the same T_i, $\hat{n} = \lfloor n \times thres \rfloor$, r is the error rate of the test oracle.

Detailed deduction of equation (2) and (3) is included in the appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2015.2425392.

With these concise equations, we are now able to analyse quantitatively how choice of n and $thres$ affect the debugging result and make informed decisions. First of all, we look at the value range of ϕ_i relative to r and sim_i (See Fig. 3).

As shown in the figure, $0 < \phi_i < 1$. Since we assume that majority of the oracle judgements are correct, then $r > 0.5$, and in this range of r, $\phi_i > 0.5$. Therefore it is obvious that $\mathbf{P}(fn(t_i))$ increases sharply as $\hat{n}$ increases, and that $\mathbf{P}(fp(t_i))$ decreases sharply as $\hat{n}$ increases. A proper set of $thres$ and n would try to lower down false negatives and positives simultaneously, and thus intuitively it is best to let $thres = 0.5$ so that neither of them is too high. We have listed the value of $P(fn(t_i))/r$ and $P(fp(t_i))/(1-r)$ with $n = 3, 4, 5$, $\hat{n} = 1, \ldots, 4$ with $\phi_i = 0.7, 0.8, 0.9$ in Table 4 and the results support this intuition. The best decisions (highlighted in red) are chosen based on two considerations. First, r is relatively small, then it is reasonable to tolerate a bigger $P(fn(t_i))/r$ than $P(fp(t_i))/(1-r)$ (when $n = 4$, we chose $\hat{n} = 2$). Second, as our experimental results show later, false negatives have a greater impact on fault localization accuracy, therefore it is better to move $\hat{n}$ slightly away from n for better fault localization performance (when $n = 5$, we chose $\hat{n} = 2$).

The calculation and parameter choice in Table 4 only considers one particular test case t_i. To estimate the overall expected number of false negatives or positives it is required that we know ϕ_i for all test cases. However this is impossible without running all tests and calculate sim_i. Fortunately one very important conclusion we can get from Table 4 is that given a fixed n, the best $\hat{n}$ stays the same despite the changing ϕ_i. Therefore although sim_i varies among different test cases, once n is chosen we would not need to adjust $thres$ to fit in for different test cases. In other words, though we would not be able to calculate the exact probability value of false negative and positives for each test case, we are still able to choose a proper $thres$ that is optimal for all test cases.

The analysis above simplifies the problem as now we only need to choose a proper n. According to Table 4, generally bigger ϕ_i generates lower false negative and positive rates, which complies with intuition. With smaller ϕ_i, bigger n is better to guarantee performance. However when $\phi_i > 0.8$, $n = 4$ would generate rather satisfactory performance. If estimation from experience is available, Table 4 should serve as a reference for choice of n. If estimation from experience is unavailable, we can(1) run all the tests and gather their traces, (2) sample a small proportion of them and find the closest 10 test cases they have and (3) calculate the similarity value between them and check the results. In this way, we are able to estimate sim_i and get a glimpse of how many close enough neighbours each test case owns. If sim_i is too low or the neighbours are few, it is probably better to supply new test cases into the test suite. If sim_i is high enough and neighbours are plenty, then as long as r is not too large, it is still possible that the final performance is good enough. Nevertheless, even with

TABLE 4
Probability of $fp(t_i)$ and $fn(t_i)$ Related to ϕ_i, n and $\hat{n}$

n		3		4			5			
$\hat{n}$		1	2	1	2	3	1	2	3	4
$\phi_i = 0.7$	$\mathbf{P}(fn(t_i))/r$	**0.2160**	0.6570	0.0837	**0.3483**	0.7599	0.0307	**0.1630**	0.4717	0.8319
	$\mathbf{P}(fp(t_i))/(1-r)$	**0.2160**	0.0270	0.3483	**0.0837**	0.0081	0.4718	**0.1630**	0.0308	0.0024
$\phi_i = 0.8$	$\mathbf{P}(fn(t_i))/r$	**0.1040**	0.4880	0.0272	**0.1808**	0.5904	0.0067	**0.0579**	0.2627	0.6723
	$\mathbf{P}(fp(t_i))/(1-r)$	**0.1040**	0.0080	0.1808	**0.0272**	0.0016	0.2627	**0.0579**	0.0067	0.0003
$\phi_i = 0.9$	$\mathbf{P}(fn(t_i))/r$	**0.0280**	0.2710	0.0037	**0.0523**	0.3439	0.0005	**0.0085**	0.0814	0.4095
	$\mathbf{P}(fp(t_i))/(1-r)$	**0.0280**	0.0010	0.0523	**0.0037**	0.0001	0.0814	**0.0085**	0.0005	0.0000

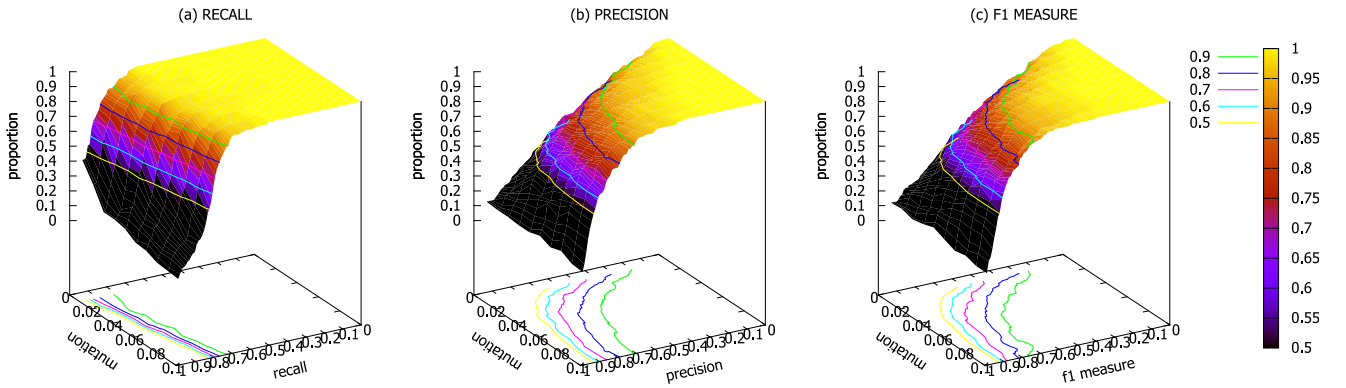


Fig. 4. Recall and precision distribution as mr ranges from 0.01 to 0.1.

situations where new test cases are needed, we do not increase the burden of testers since the test oracle is erroneous itself and needs debugging anyway.

3.4 Time and Space Complexity

Our algorithm contains two main phases, i.e., test case profiling and oracle debugging. In the first phase time and space complexity depend on the profiler, for example `gcov` in our case. But to record the traces of all tests the space complexity is $O(|T|N)$. In the second phase, first we need to calculate the $tf-idf$ value of each line of code, which requires $O(|T|N)$ time complexity, but no new space. Then we calculate the "similarity" between any two test cases, which causes $O(|T|^2O(sim))$ time complexity, where $O(sim)$ is the time complexity of similarity calculation between two test cases and $O(|T|^2)$ space complexity. In our implementation, $O(sim)=O(N)$. Then we need to select the closest n test cases for each test case, which requires at most $O(|T|^2n)$ time complexity and $O(|T|n)$ space complexity. Finally we need to calculate the "suspicion" value for each test case, which causes $O(|T|O(Sus))$ time complexity, where $O(Sus)$ is the time complexity of suspicion calculation and at most $O(|T|)$ space complexity. In our implementation $O(Sus)=O(n)$. Therefore the overall time complexity is $\max\{O(|T|^2O(sim)), O(|T|^2n), O(|T|O(Sus))\}=O(|T|^2N)$ and the overall space complexity is $O(|T|N)$. This result indicates that time complexity grows linearly with size of the program but quadratically with size of the test suite, while space complexity grows linearly with both size of the program and test suite.

4 Experiments

A comprehensive evaluation of our approach should inspect its performance on two aspects, (1) how much "correcter" is the debugged oracle compared to the original one and (2) how much the debugged oracle improves SFL algorithms' performance. To answer these two questions, we conducted two corresponding experiments. The results are presented in the following two sections.

4.1 Repair the Erroneous Test Oracle

Our first experiment evaluates the quality of the debugged oracle. In this experiment, we used the Siemens Test Suite as our subject because it contains enough similar entities of programs, which ensures a statistical significant result. Same as when we inspected erroneous oracles' impact on SFL algorithms, we first generated the correct oracles of the Siemens Test Suites and randomly mutated them with a mutation rate mr ranging from 0.01 to 0.1. Remember the value of two input parameters needs to be decided, i.e., the number of voters n and the debugging threshold $thres$. To achieve better performance, we first analysed a sample of the subject test suites and discovered that nearly all test cases own more than 10 neighbours with similarity above 0.9. This was confirmed by a later analysis as approximately over 95 percent test cases own such number of neighbours. Referring to Table 4 we decided $n=4$ is proper for gaining satisfactory performance. As with $thres$, according to our analysis, as long as $2<\hat{n}<3$ the final performance should be satisfactory. For convenience we fixed it on 0.58 for all experiments.

With $n=4$ and $thres=0.58$, we applied our debugging algorithm to the oracle mutants and compared the output oracles to the original correct ones. According to [10] we repeated our experiment five times to achieve a stable and convincing result. Three quantitative index values are selected as assessment criteria, namely, "recall", representing our algorithm's ability to acutely discover the error, "precision", indicating the ability to preserve the already correct judgements and "f1 measure", synthesizing the former two.

Fig. 4 presents the overall distribution of recall, precision and f1 measure for all oracle mutants and error rates. For recall rec and a mutation rate mr, the proportion value $p(rec, mr)$ is defined as the ratio of mutants whose corresponding output oracle achieved recall value greater or equal than rec in all mutants mutated at mr. For precision pre and f1 measure $f1$, $p(pre, mr)$ and $p(f1, mr)$ are similarly defined.

As shown in the figure, averagely our algorithm achieves 85 percent recall rate on over 90 percent mutants, meaning over 85 percent oracle errors are successfully identified. As to precision, mutants at different mutation rates respond differently. As the median, when $mr=0.05$, precision reaches 75 percent for over 80 percent mutants, i.e., over 75 percent of the highlighted test cases are truly misjudged in the input oracle. This fraction expands to 80 percent as mr increases to 0.1. Figure of F1 measure is similar to precision as value of recall is close to 1.

Contour projection on the x-y panel presents some interesting phenomena. First, when mr is lower than 0.02,

over 20 percent highlighted test cases are false positives for 50 percent mutants, and over 40 percent for 30 percent mutants. This accords with intuition when considering the relative high recall. 90 percent errors in around 80 percent oracle mutants are marked out, thus it is very likely that many other oracle judgements are wronged. Second, with the increase of mr, recall falls slightly while precision rises promptly. This matches our theoretical analysis as with a fixed $thres$ and n, both $\mathbf{P}(fn(t_i))/r$ and $\mathbf{P}(fp(t_i))/(1-r)$ stay almost the same. Therefore as r increases, $\mathbf{P}(fn(t_i))$ increases while $\mathbf{P}(fp(t_i))$ decreases.

4.2 Cure SFL Algorithms

The second experiment is designed to evaluate how much our algorithm can reduce the extra time consumption caused by test oracle errors during the debugging process. This evaluation is conducted on not only the Siemens Test Suite, but also on `grep`, a real world program which scales up to over 13,000 lines of code. The results are presented as follows.

4.2.1 Results on the Siemens Test Suite

As a reference, we first applied the four SFL algorithms on each program variants of the Siemens Suites with the erroneous oracle mutants we generated and recorded their performance. Then we performed the oracle debugging process and generated a corresponding "debugged oracle" for each oracle mutant. Finally we applied the four SFL algorithms with the "debugged oracles" and compared their performance with the record with the erroneous oracle mutants. In consistency with Section 2, time required for debugging is quantified as the score defined by equation (1). Again the experiment was repeated five times. The results are presented in Fig. 5.

Figs. 5a and 5b compare the performance of four SFL algorithms with the erroneous oracle mutants and the oracles debugged by our algorithm. For a mutate ratio mr, a certain score s_x, the proportion value $p(mr, s_x)$ in both figures is defined as the proportion of program variants that, with the erroneous oracle at error rate mr, can be successfully debugged by checking less than s_x of its code. Visually, surfaces in Fig. 5b for all four SFL algorithms are clearly "lifted" from their place in Fig. 5a, demonstrating a remarkable improvement in fault localization accuracy. Typically, under the erroneous oracles, Russel&Rao achieved score 0 (hitting the bug at the first line it suggests to be checked) on approximately 20 percent program variants, while this number expands to 50 percent and even higher for less erroneous oracles. For Ochiai, this rate rose from less than 60 percent to over 70 percent. Even for Tarantula, the least sensitive algorithm to oracle errors, this rate rose from 25 to 40 percent.

For clearer view of the differences between Figs. 5b and 5a, we plotted two other set of figures demonstrating the absolute (Fig. 5c) and relative (Fig. 5d) improvement. To be precise, in addition to $s_0(v_i, \omega_j)$ and $s_r(v_i, \omega_j)$ defined in Section 2.2.4, we define $s_d(v_i, \omega_j)$ as the score of ω_j applied to v_i with the debugged oracle. Then the absolute performance improvement of ω_j for v_i at error rate r is defined as

$$\Delta^{+}_{|\cdot|}(\omega_j, v_i, r) = s_r(v_i, \omega_j) - s_d(v_i, \omega_j)$$

and the relative improvement is defined as

$$\Delta^{+}_{\%}(\omega_j, v_i, r) = \frac{s_d(v_i, \omega_j) - s_0(v_i, \omega_j)}{s_r(v_i, \omega_j) - s_0(v_i, \omega_j)}.$$

For both figures, proportion p relative to a certain improvement value Δ_x at mutate rate r represents the rate of program variants on which the SFL algorithms achieved greater improvement than Δ_x using oracles mutated and debugged at mutate rate r. As shown in Fig. 5c, different algorithms improve at various extent. For Russel&Rao, approximately 30 percent program variants can be successfully debugged by checking 10 percent less code. These numbers reflect dramatic reduction in debugging time.

One may notice that there seems to be not much absolute improvement on a majority of program variants. This is because for most of the cases, the faulty line can be localized within 30 percent of the total code even with the erroneous oracle mutants, which leaves not much space of absolute improvement. However, as presented in Fig. 5d, there is a relative improvement for over 50 percent program variants. The most obvious improvement appeared in Russel&Rao, where accuracy for approximately 50 percent program variants improved 50 percent.

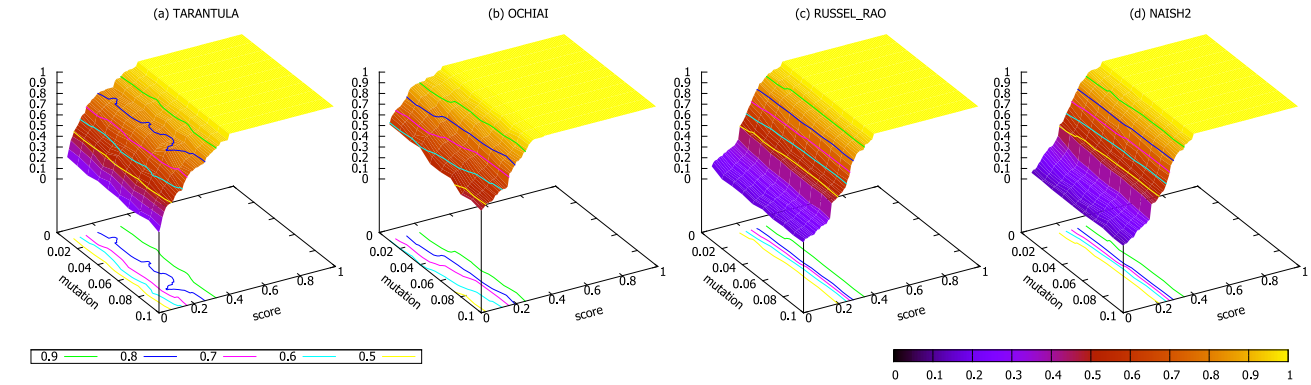
From Fig. 5, we conclude that after debugging the test oracles, effectiveness of four subject SFL algorithms recovered remarkably. Since each of these four algorithms are either identified as "optimal" metrics or widely known, we believe they are representative of the whole SFL family.

4.2.2 Results on `grep`

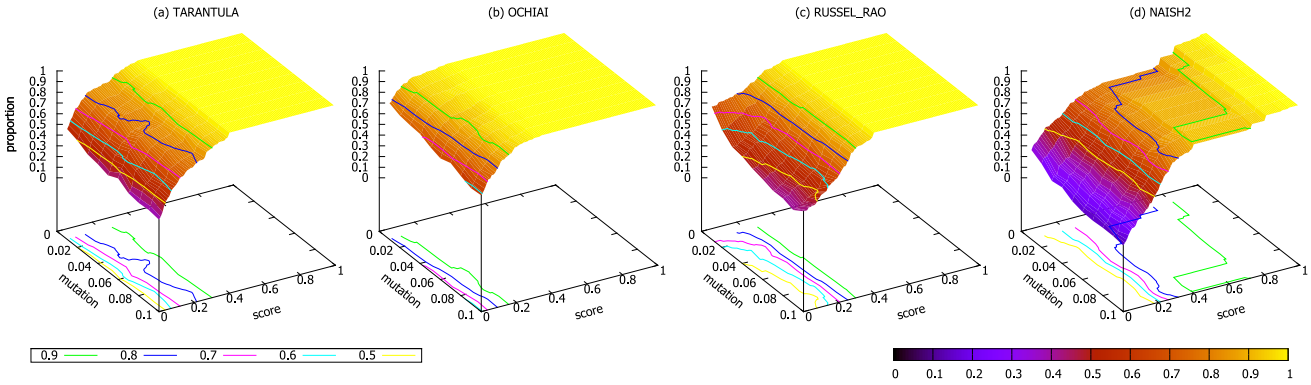
`grep` is a command-line utility for searching plain-text data sets for lines matching a regular expression widely used on Unix-like systems. SIR [8] provided five versions of `grep`, each containing from 1 to 18 fault seeds and several corresponding test sets. The original test results showed that plenty of the fault seeds could not be detected, thus were excluded in our experiment. Especially, only one fault seed was provided for version 5 but was undetectable, so version 5 was not included. A summary of the usage of `grep` is presented in Table 5.

Following the same scheme as with the Siemens Test Suite, we first generated the correct test oracles for four versions of `grep` using the corresponding unseeded programs. Then we randomly mutated the test oracles at a mutate rate mr from 0.01 to 0.1. For comparison, we recorded the accuracy of the four SFL algorithms applied to the seeded programs using both the mutated oracles and the debugged oracles. The experiment was repeated five times. The results are plotted in Fig. 6.

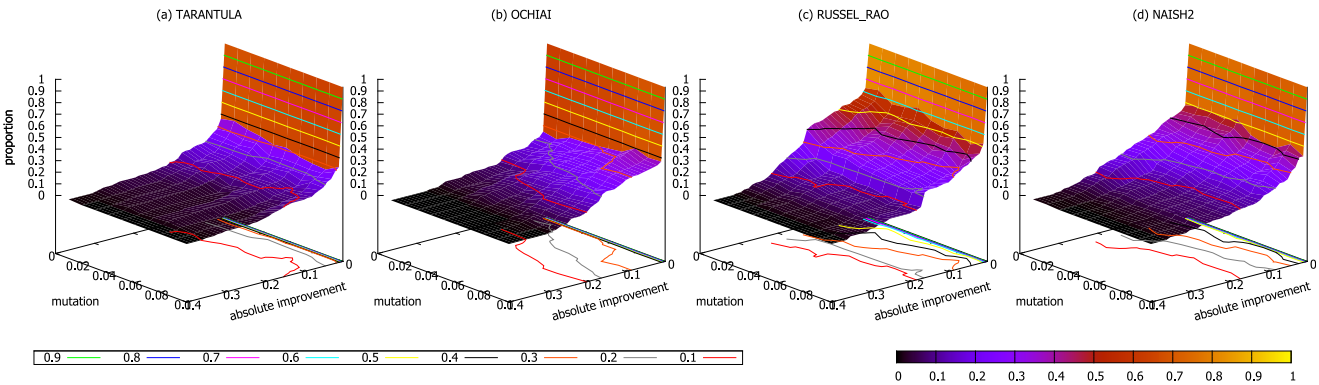
Fig. 6 presents the evaluation results on four versions of `grep` using the four SFL algorithms. Since there are only four versions of programs we were able to look closer at the effect of our approach on each program. For each version using the same SFL algorithm, the red line represents the change of accuracy as mr changes from 0.01 to 0.1 with the mutated oracles, while the green line represents the accuracy with the debugged oracles. Obviously, accuracy



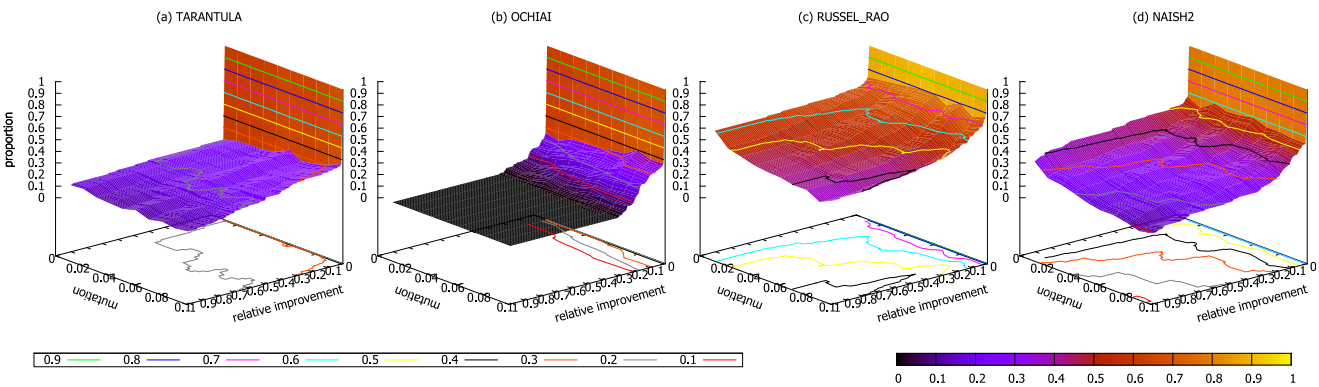
(a) Score distribution on erroneous oracle mutants



(b) Score distribution on debugged mutants



(c) Distribution of absolute improvement



(d) Distribution of relative improvement

Fig. 5. Performance improvement of four SFL algorithms on the debugged oracle.

TABLE 5
Summary of `grep`

version	fault seeds	detectable seeds	line of code
v1	18	2	12,654
v2	8	1	13,231
v3	18	2	13,374
v4	12	2	13,360
v5	1	0	13,293

improved on most of the versions for different SFL algorithms. This is especially prominent with Russel&Rao. There seems to be no obvious improvement in Ochiai, but accuracy with debugged oracle is at least not worse than with the original oracle.

Another thing worth mentioning is the test sets provided by SIR. For programs in Siemens Test Suite, SIR provided approximately 4,000 test cases each. Thus it is natural that we can debug the test oracle taking advantage of the duplicated test cases. However, for `grep` we adopted the test set which aimed to be *"representative of those that might be created in practice for these programs"*[8]. The test set contains merely 199 test cases, yet we still managed to improve the accuracy of SFL algorithms. This proves that our approach is applicable for realistic programs with realistic test suites.

5 Discussion

During the second experiment in Section 4, we observed that as error rate rises, performance improvement of SFL algorithms on the debugged oracles decreases slightly. This trend coincides with the change of recall rate relative to error rate, which leads us to wonder if there is a direct connection between recall rate of the debugged oracles and performance improvement of SFL algorithms irrelevant to the error rate of the input oracles. Since Russel&Rao showed a more obvious improvement, we calculated the absolute and relative improvement distribution related to "recall" and the results are presented in Fig. 7.

In Fig. 7, we plotted the proportion of debugged oracles of identical recall rate that achieved absolute or relative improvement values between 0 and 1. Despite the layered surface, from the contour projection we are able to recognize the rough tendency that improvement grows along with the increase of recall.

A direct conclusion from this tendency is that generally a higher recall rate will lead to better improvement with the debugged oracle regardless of the actual error rate. Therefore, our approach can be applied to scenarios where the error rate is over 0.1 (the upper bound in our experiment), with only adjustment of parameters to achieve a proper recall rate.

Another deduction answers an important question: which kind of error, "false passes" or "false fails", impacts SFL algorithms more and needs to be avoided more carefully? This is a meaningful question since early prevention is always cheaper than later correction. Intuitively we covered this issue in Section 1 and decided that "false fails" are worse. But statistical evidence is still required.

For most program variants of the Siemens Test Suite, only around 5 percent test cases can expose the error. Consequently the majority of oracle errors are "false fails" and "false passes" rarely happen. At a high recall rate, almost all errors in the oracle mutant would be corrected, which dramatically reduces the ratio of "false fails" while "false passes" are less influenced. As a result, "recall" can actually be used as an indicator for error composition of the debugged oracle, that is, a higher recall usually means less "false fails" whereas more "false passes". As the tendency shown in Fig. 7, less "false fails" relates to more obvious performance improvement. Therefore, seen from the other side "false fails" do impact more than "false passes" and should be especially prevented in the initial creation of test oracles.

6 Threats to Validity

Similar to existing work, empirical studies on SFL algorithms are all subject to validity problems. Referring to [10], we summarize the potential threats to validity of our work as follows.

First is the representativeness of the subject programs. In our empirical study in Section 2, the subject programs are the Siemens Test Suites. One issue is that the size of the Siemens programs is rather small, therefore the conclusions may not fully reflect the exact impact of oracle errors on large scale programs. But this does not impede us from confirming the negative influence of oracle errors. In Section 4, we adopted `grep`, a widely used real world program scaling up to 13,000 lines of code as a subject program and proved that the fault localization accuracy improved with debugged test oracles in Section 4. Therefore we believe that our approach is scalable and applicable to real world programs.

The second issue is that all faulty variants are single-faulted, while in real world multi-faults are common.

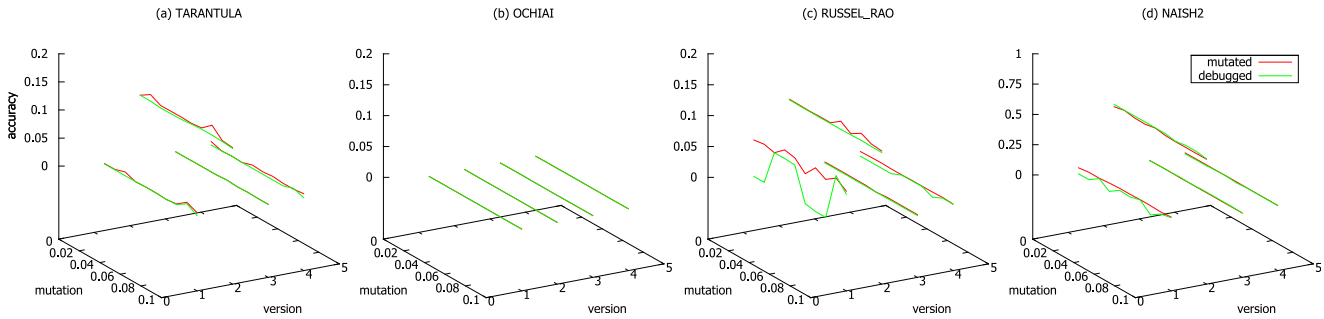


Fig. 6. Accuracy of four SFL algorithms on `grep` with mutated and debugged oracles.

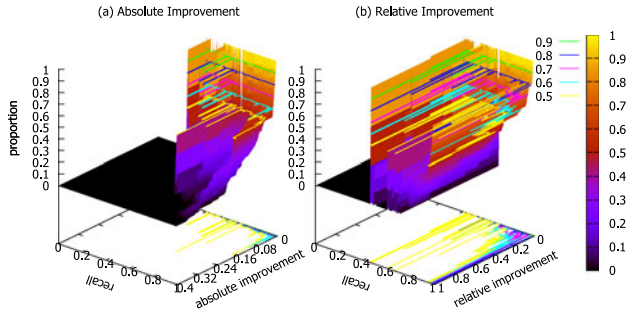


Fig. 7. Absolute and relative improvement related to recall.

Fortunately as we have pointed out in Section 2.2.4, many of the faults in fact influence a great proportion of the program and may even mimic multiple faults. For example, a wrongly defined constant value in "#define" would map to wrong values scattering all over the program. We kept these kind of errors intentionally to ameliorate the bias between our experiment and real world case.

The third issue is the sample size, or the number of subject programs. Steimann et al. [10] states that to reach stable results at least 300 subject programs with errors are required. Therefore we repeated our experiments five times to smooth away the randomness.

Another factor mentioned in [10] is the quality of the test suites, including size and coverage. Siemens Test Suite provides ample test cases for each variant and achieves satisfactory test coverage. Test suite for `grep` contains only 199 test cases but was designed to be *"representative of the those that might be created in practice for these programs"*[8]. Therefore we believe that quality of test suites in our experiment is sufficient to come to stable and convincing conclusions.

7 Related Work

To the best of our knowledge, this is the first paper investigating the problem of debugging the test oracle errors. The most closely related work falls in three categories: (1) evaluation on accuracy of SFL algorithms, (2) identifying related factors and evaluating their impacts on SFL accuracy and (3) automatic test oracle generation. The following sections present a brief introduction and discussion on these three categories respectively.

7.1 Evaluation on Accuracy of SFL Algorithms

There has been much work evaluating the accuracy of SFL algorithms, both empirically and theoretically. Jones and Harrold [17] compared Tarantula with early fault localization techniques including Set union, Set intersection, Nearest neighbour and Cause transitions. It concluded that Tarantula outperformed the other techniques. Abreu et al. [15] evaluated four similarity coefficients used in SFL algorithms and concluded Ochiai performed consistently better than Jaccard and Tarantula, while Jaccard performed no worse than Tarantula. AMPLE was occasionally better or worse than the others. Both [17] and [15] used the Siemens Test Suite as target programs. Abreu et al. [12] further compared Ochiai, Tarantula and Jaccard with other six metrics on both the Siemens Test Suite and `space` and concluded that Ochiai performed the best. Naish et al. [18] analysed 41 metrics and identified six groups of metrics that are equivalent for ranking under single-fault condition. Xie et al. [13] analysed 30 metrics and classified them by accuracy into 14 equivalent groups. They further proved theoretically the "better than" relation among these groups and identified group ER1 (Naish1, Naish2) and ER5 (Binary, Russel&Rao, Wong1) to be "maximal" metrics under single-fault condition. Le et al. [14] empirically studied the accuracy of Tarantula, Ochiai and all metrics in ER1 and ER5. Results show that Ochiai is better than ER1 and ER5, and Tarantula achieved better average accuracy than Naish1 and ER5. The authors concluded that the 100 percent coverage assumption used in theoretical analysis is unrealistic. All these evaluations are based on the assumption of an existing correct test oracle. Our results in Section 2.2.4 complement existing evaluations as we present the impact of test oracle errors on accuracy of SFL algorithms.

7.2 Factors Influencing SFL Accuracy

Accuracy of one SFL algorithm varies under different conditions. Many factors have been identified that can influence accuracy. In [12], the authors looked at how accuracy is influenced by the number of total, passing and failing test cases. They conclude that near-optimal diagnostic accuracy (exonerating over 80 percent of the blocks of code on average) is already obtained for low-quality error observations and limited numbers of test cases. In [11] the authors studied the effects of 10 test-suite reduction strategies. The study shows that the effectiveness of fault-localization techniques varies depending on the test-suite reduction strategy used. Masri et al. [19] assessed the prevalence of four scenarios related to tests or test suites, and put forward the concepts of "coincidental correctness", where the condition for failure is met but the program does not fail, and "weak coincidental correctness", where the faulty statement is executed but the program does not fail. Based on this observation, later researchers proposed to mitigate the impact of coincidental correctness through test case clustering [20], [21] and coverage refinement with context patterns [22]. Despite what the name suggests, "coincidental correctness" are results of the correct judgement of test oracles. Our work, on the other hand, analyses the scenario where test results are incorrectly judged to be "pass" or "fail". [23], [24] noticed misclassification of bug reports and their impact on bug localization and prediction. They resemble our work as they do not assume the bug reports, mostly submitted by human, always correctly classify the perceived program behaviour. However they analyse the possible location of bugs at the granularity of source files instead of statements.

7.3 Test Oracle Generation

Test oracles are the fundamental criteria to decide the correct behavior of software in the testing process. However, generation of reliable test oracles is no easy work. Though manual creation is still common practice, an increasing amount of research aim at automatic generation of test oracles that are trustworthy and time-saving.

Solutions on test oracle automation mainly fall in two categories. The first one generates an explicit test oracle. For example, in [25], the authors focused on generating test oracles for GUIs. The tested GUI is first formalized as sets of

objects, object properties and actions. Then the test oracle automatically derives the expected state for every action in the test case and compares it with the actual GUI state when the test case is executed. Though the test oracle is generated completely automatically, it relies on the formal description of GUI, thus cannot be extended on other kinds of programs. In [26], the authors studied the performance of artificial neural networks (ANN) as an automatic test oracle generator on the triangle classification problem. The authors concluded that ANN could be used as a test oracle with reasonable degree of accuracy, however the total mean relative error is found to be 19.02 percent. Comparing it to our study of oracle errors' impact on debugging process, we believe this error rate would exceed the tolerance value for a wide range of programs. Another interesting approach is presented in [27], where the authors tried to automatically generate test oracles from the design documentation. Still, the documents are required to precisely define the behavior of the system and be written in a certain data structure.

The second category circumvents the generation of an explicit oracle. In [28], Davis and Weyuker first proposed to use independently produced programs that fulfill the same specification as the original program as "pseudo oracles" for non-testable programs. They suggested using two or several versions of programs and adopt certain kind of voting algorithms to decide the correct output when there is a conflict among different versions of programs. Obviously, much work is involved to create completely independent different program versions. A developed version of this approach is presented in [29], where implementation by manual coding are compared against the automatically generated version which serves as an implicit test oracle. This is a big step forward since the only extra labor needed is to write formal specifications of the target program. However, not all programs are suitable for formal specifications. Other than depending on specifications, metamorphic testing is also used to test the software without oracles. Metamorphic testing examines if the outputs of two related inputs satisfy a certain relationship deducted from the expected behavior of the program. In [30] and [31], the authors proposed to combine metamorphic testing and fault-based testing with symbolic input, but with no concrete implementation design. Later in [32] and [33], a tool called Amsterdam is proposed to automate metamorphic testing. Unfortunately, despite the automated testing process, testers still need to manually specify the applications' metamorphic properties, which include description of how the input should be transformed and the expected change of to the output. In [34], the complete testing process is claimed to be automated, which inevitably includes test oracle automation. However, formal specifications in forms of assertions, pre- and post-conditions and class invariants written in java modeling language (JML) are required as an implicit test oracle. Strictly, these operations are still manually conducted, i.e., the testing process is still human involved.

To sum up, automatic generation of test oracles are still far from mature. This is also one of the motivations of our work–before we are able to automatically generate test oracles, we try to utilize the imperfect manually written oracles in a better way.

8 Conclusion

In this paper, we first proposed the problem of test oracle errors' impact on debugging process. To verify this impact, we conducted experiments on the Siemens Test Suites and tested the performance of four Spectrum-based Fault Localization algorithms using randomly mutated erroneous test oracles. Compared with when using the correct test oracles, fault localization accuracy is remarkably compromised. Therefore it is necessary to debug the test oracle before it is used to debug the program.

Based on a simple observation that test cases covering similar lines of code usually pass or fail simultaneously, we proposed to measure the similarity between test cases using coverage information and identify the unusually judged test cases from their neighbours. As demonstrated by later experiments, these test cases proved to be mostly wrongly judged by the erroneous test oracle. We further compared the accuracy of four SFL algorithms on both the Siemens Test Suites and `grep` using the debugged oracle with using the erroneous oracle mutants. Experimental results show that fault localization ability of all four SFL algorithms recovered in different degrees. Finally, we analysed the experiment data and concluded that our approach is applicable to oracles unlimited to an upper bound of error rate. In all, we proposed a simple but effective method to promote the quality of existing erroneous test oracles.

Acknowledgments

This research was sponsored in part by NSFC Program (Grant No. 91218302), National Key Technologies R&D Program (Grant No. 2012BAF16G01) of China, Postdoctoral Science Foundation of China (Grant No. 2014M560082), National Natural Science Foundation of China (Grant No. 61402248). M. Gu is the corresponding author.

References

[1] D. Peters and D. L. Parnas, "Generating a test oracle from program documentation: Work in progress," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1994, pp. 58–65.

[2] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," Dept. Comput. Sci., Univ. Sheffield, Sheffield, England, Tech. Rep. CS-13-01, 2013.

[3] L. Manolache and D. G. Kourie, "Software testing using model programs," *Softw.: Practice Experience*, vol. 31, no. 13, pp. 1211–1236, 2001.

[4] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. 1st Int. Workshop Softw. Test Output Validation*, 2010, pp. 1–4.

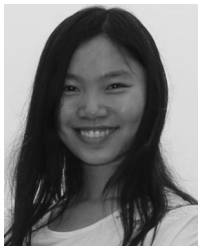
[5] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proc. 8th Int. Symp. Future Softw. Technol.*, 2004, pp. 346–351.

[6] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proc. 17th Int. Conf. Softw. Eng.*, 1995, pp. 41–41.

[7] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application," *J. Syst. Softw.*, vol. 48, no. 2, pp. 79–89, 1999.

[8] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.: An Int. J.*, vol. 10, no. 4, pp. 405–435, 2005.

[9] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 467–477.
[10] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 314–324.
[11] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 201–210.
[12] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. (2009). A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* [Online]. *82(11)*, pp. 1780–1792. Available: http://www.sciencedirect.com/science/article/pii/S0164121209001319
[13] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, p. 18, 2013.
[14] T.-D. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *Proc. 29th IEEE Int. Conf. Softw. Maint.*, Sep. 2013, pp. 380–383.
[15] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput.*, Dec. 2006, pp. 39–46.
[16] Wikipedia. (2014). Tf-idf—wikipedia, the free encyclopedia [Online]. Available: http://en.wikipedia.org/w/index.php?title=Tf5
[17] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2005, pp. 273–282.
[18] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.
[19] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proc. 2nd Int. Workshop Defects Large Softw. Syst.: Held Conjunction ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2009, pp. 1–5.
[20] W. Masri and R. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Proc. 3rd Int. Conf. Softw. Testing, Verification Validation*, Apr. 2010, pp. 165–174.
[21] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 8:1–8:28, Feb. 2014.
[22] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 45–55.
[23] P. S. Kochhar, T.-D. B. Le, and D. Lo, "It's not a bug, it's a feature: Does misclassification affect bug localization?" in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 296–299.
[24] K. Herzig, S. Just, and A. Zeller, "Its not a bug, its a feature: How misclassification impacts bug prediction," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 392–401.
[25] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 30–39, Nov. 2000.
[26] K. K. Aggarwal, Y. Singh, A. Kaur, and O. P. Sangwan, "A neural net based approach to test oracle," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 3, pp. 1–6, May 2004.
[27] D. Peters and D. Parnas, "Using test oracles generated from program documentation," *IEEE Trans. Softw. Eng.*, vol. 24, no. 3, pp. 161–173, Mar. 1998.
[28] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proc. ACM Conf.*, 1981, pp. 254–257.
[29] D. Brown, R. Roggio, I. Cross, J.H., and C. McCreary, "An automated oracle for software testing," *IEEE Trans. Rel.*, vol. 41, no. 2, pp. 272–280, Jun. 1992.
[30] T. Chen, T. H. Tse, and Z. Zhou, "Fault-based testing in the absence of an oracle," in *Proc. 25th Annu. Int. Comput. Softw. Appl. Conf.*, 2001, pp. 172–178.
[31] T. Chen, T. Tse, and Z. Q. Zhou. (2003). Fault-based testing without the need of oracles. *Inf. Softw. Technol.*[Online]. *45(1)*, pp. 1–9. Available: http://www.sciencedirect.com/science/article/pii/S0950584902001295
[32] C. Murphy and G. E. Kaiser, "Automatic detection of defects in applications without test oracles," Dept. Comput. Sci., Columbia Univ., New York, NY, USA, Tech. Rep. CUCS-027-10, 2010.
[33] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 189–200.
[34] Y. Cheon, M. Y. Kim, and A. Perumandla, "A complete automation of unit testing for Java programs," in *Proc. Int. Conf. Softw. Eng. Res. Practice*, 2005, pp. 290–295.



Xinrui Guo received the BS degree from the School of Software, Tsinghua University, in 2012. She is currently working toward the PhD degree at the School of Software, Tsinghua University. Her research interests include software testing and automated debugging.

Min Zhou received the BS degree from the Department of Mathematical Science in 2007 and the PhD degree from the Department of Computer Science and Technology in 2014. He is a post-doc at the School of Software, Tsinghua University. His research interests include model checking, program analysis, and testing.

Xiaoyu Song received the PhD degree from the University of Pisa, Italy, 1991. His current research interests include formal methods, design automation, embedded system design, and emerging technologies.

Ming Gu is a researcher, vice-director of School of Software, Tsinghua University, vice-director of Ministry of Education Key Laboratory of Information Security. His research areas include software formal methods, software trustworthy, and middleware technology.

Jiaguang Sun is a professor, member of the Chinese Academy of Engineering, dean of School of IST (Information Science & Technology) and dean of the School of Software in Tsinghua University, director of Ministry of Education Key Laboratory of Information Security, director of National Engineering Research Center of CAD Supporting Software, director of Tsinghua National Laboratory for Information Science & Technology (IST), president of Executive Council of China Engineering Graphics Society. His research areas include computer graphics, computer-aided design, formal verification of software, software engineering, and system architecture.