Program Structure Aware Fault Localization *

Heng Li
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
liheng@ios.ac.cn

Yuzhen Liu
North China Electric Power University
Beijing 100190, China
codestorm04@gmail.com

Zhenyu Zhang†
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zhangzy@ios.ac.cn

Jian Liu
Institute of Software
Chinese Academy of Sciences
Beijing 100190, China
liujian@iscas.ac.cn

ABSTRACT

Software testing is always an effective method to show the presence of bugs in programs, while debugging is never an easy task to remove a bug from a program in software development. To facilitate the debugging task, statistical fault localization estimates the location of faults in programs automatically by analyzing the program executions to narrow down the suspicious code region. We observe that program structure has strong impacts on the assessed suspiciousness of the program elements. However, existing techniques inadequately pay attention to this problem in locating faults. In this paper, we emphasize the biases caused by program structure in fault localization, and propose a method to address them. Our method is dedicated to boost a fault localization technique by adapting it to various program structures, in a software development process. It collects the suspiciousness of program elements when locating historical faults, statistically captures the biases caused by program structure, and removes such an impact factor from a fault localization result. An empirical study using the Siemens test suite shows that our method can greatly improve the effectiveness of the most representative fault localization *Tarantula*.

*This research is supported in part by the National Key Basic Research Program of China (project no 2014CB340702), the National Natural Science Foundation of China (project no 61379045), and the National Science and Technology Major Project of China (grant no. 2012ZX01039-004).

†All correspondence should be addressed to Dr. Z. Zhang at Institute of Software, Chinese Academy of Sciences. Tel: (+8610)6266 1658. Fax: (+8610)6266 1627. Email: zhangzy@ios.ac.cn.

InnoSWDev'14, November 16, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3226-2/14/11...$15.00
http://dx.doi.org/10.1145/2666581.2666593

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Experimentation, Verification

Keywords

Software testing, fault localization, program structure

1. INTRODUCTION

With the fast development of software and software system, both the scales and complexities of programs are greatly increased. At the same time, software failures are still problems hard to solve. Most software failures are caused by faults (bugs) in programs. Failures revealed in testing confirm the existence of faults in programs. However, even we know there are faults in programs, a software failure will not always appear in different program runs. Debugging is the activity to locate a fault in a program and remove it from the program. It is never an easy task in software development [15, 19].

To facilitate the debugging task, automatic fault localization mechanisms are invented. In order to alleviate the problem, researchers have proposed many automated fault-localization methods. Among these techniques, there is a category of techniques known as statistical fault-localization techniques, such as *Tarantula* [11, 12], *Jaccard* [1], *CBI* [13], *SOBER* [14] and so on. The basic intuition behind these techniques is that they attempt to identify program features whose execution is correlated with the program failures, and the strengths of the correlations can be used as indicators of the degree to which those program features may explain the failures. Techniques like *Tarantula* can produce real number values (aka. *suspiciousness scores*) for program entities to represent the correlations with program failures and sort the entities by value in descending order. Then a ranked list of program entities is produced and programmers can localize the fault under the guidance of such a ranked list.

In recent years, promising experimental results with the statistical fault localization techniques have been reported.

However, we have realized huge space for improvements for such techniques. For example, the effectiveness to locate faults is unavoidably influenced by a number of factors. Coincidental correctness has been perceived as a factor that can adversely affect the effectiveness of testing [16]. This factor occurs when "no failure is detected, even though a fault has been executed [18]", which has attracted many researchers' interests (eg., [8, 20]). Noise has been reported as another factor that affects the effectiveness of the fault localization techniques. It can be estimated as the possibility of not executing a feature causing a failure [22] and some techniques have been proposed to generalize the fault localization techniques of noise-reduction. Zhang et al. [25] reported that the short-circuiting evaluation manner also has influence on statistical fault localization, and empirically showed that the use of short-circuiting evaluation information can significantly improve some predicate-based statistical fault-localization techniques.

In this paper, we perceive that the program structure may have impacts on the effectiveness of the fault localization techniques. As a result, a common but previously ignored phenomenon is that some statements always have higher suspiciousness values than others, while some statements always have lower suspiciousness values at the same time. Since a statement is not likely to correlate with all program failures across program versions, we deem the fundamental reason for that to be the impact of program structure. For example, the statements in a *catch* block in exception handler are apt to be executed more in failed program runs and less frequently in passed runs, so that the statements in such a program structure are more probable to have abnormally high suspiciousness scores.

In this paper, we proposed a program structure aware fault localization technique that can be used to capture the impacts caused by program structures. In a software development scenario, for each statements, we record its suspiciousness calculated in locating each faults in history. We use such historical information to analyze the impact of the program structure on it, i.e., whether its suspiciousness was ever overestimated due to specific program structure pattern. We then erase the impact from future fault-localization to restore the unbiased suspiciousness. To verify the effectiveness of our method, we apply it on the most representative fault localization technique *Tarantula*, and carry out a controlled experiment to evaluate. The results show that our method can greatly improve the effectiveness of *Tarantula* on the subject programs.

The main contributions of our work are at least threefold. First, we noticed and confirmed that the program structure can have significant impact on effectiveness of a fault localization technique. Second, we proposed a general method to capture and remove the impacts of the program structure. Third, we conducted an experiment on a common data set and a medium-scaled realistic program with a representative peer technique to validate the effectiveness of our method.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 uses three examples to motivate this work. Section 4 presents our technique, which is evaluated in Section 5. Section 6 concludes this paper.

2. RELATED WORK

Many statement-level fault localization techniques have been proposed. They contrast program execution spectra of statements in passed and failed execution, and then use some heuristics to estimate the extent of a statement being related to faults. Jones et al. [12] proposed *Tarantula*, which uses the proportions of passed and failed executions to compute the suspiciousness of each statement in a program. The suspiciousness of a statement denotes the likelihood of a statement to be faulty. Specifically, the suspiciousness of statement s is computed using the percentage of failed program runs that execute s and the percentage of passed program runs that execute s. Other statement-level fault localization techniques include *Jaccard* [1], *Ochiai* [2], etc. These techniques are similar to *Tarantula* except that they use different formulas to compute the suspiciousness of a statement.

Predicates have also been adopted as fault indicators. Liblit et al. [13] developed *CBI*, which uses the number of times a predicate being evaluated true in passed and failed program runs with reconciling the specificity and sensitivity of the predicates to estimate the suspiciousness of predicates. However, for those predicates that are always evaluated true, the *CBI* is ineffective. *SOBER* [14] compares the distributions of evaluation biases between passed runs and failed runs to obtain the suspiciousness of predicates. The larger the value of evaluation biases, the more likely the predicate associates with the root cause of failure. We do not limit the use of our method, which can be applied on a general fault localization technique no matter what kind of granularity it is of.

Many factors can have impacts on the effectiveness of fault localization techniques. The noise introduced by some features on the same set of executions with the statement, has been noticed as such a factor in [22]. It can be estimated as the possibility of not executing a feature causing a failure.

Program structure can be another impact factor to localizing precision. A fault may be triggered, propagates along any execution path, and finally causes an observable failure. Zhang et al. [24] proposed the approach, which captures the propagation of infected program states through the program structure, namely the edges in a control flow graph. It associates scores of control flow edges to suspiciousness scores of basic blocks to locate faults. In this paper, instead of adopting a CFG-related approach, we argue that a better choice is to statistically capture the impact of the program structure on the effectiveness of fault localization.

3. MOTIVATION

Statistical fault localization techniques make use of two kinds of information collected in the testing process, namely testing results and program spectrum, to estimate the suspiciousness of a program element to be related to faults. The testing result of a test case represents whether the output is expected (a *passed* program run) or not (a *failed* program run), while a program spectrum is a collection of data that provides a specific view on the dynamic behavior of the program during execution [7, 17]. Generally speaking, it records the runtime profiles of various program entities for a specific test suite. The program entities could be statements, branches, basic blocks, and so on. Typically, the runtime profile is in the form of execution counts of program entity [5, 21]. We notice that the *program structure* mostly determines the execution counts, and accordingly has impacts on the suspiciousness of program entities.

3.1 Ineffective Case

Table 1: Statements in the main entry.

		passed runs		failed runs	
		r1	r2	r3	r4
	int main() {				
n	int[] arr = new int[2];	•	•	•	•
n+1	arr[0] = 1;	•	•	•	•
	...				
	}				

The code segment in Table 1 includes the *main* function of a program. Since the main function is where the program starts its execution, as an entry point, the statements in such a structure will be executed almost by all runs. As a result, a neutral suspiciousness score (e.g., 0.5 by Tarantula) will be given to such statements. If there are some faults among the statements, most such fault localization techniques are ineffective to assess the extent of suspiciousness of such statements, when contrasting their execution counts in passed executions to those in failed executions. As a result, that is an ineffective case for fault-localization.

3.2 Misleading Case

Table 2: Statements in an exception-handler.

		passed runs		failed runs	
		r1	r2	r3	r4
	try{···} catch(Exception e) {				
n	println(e.getMessage());			•	•
	...				
	}				

The example in Table 2 shows an exception-handler. When we come across the exceptions by accident during program execution, the program flow will enter the exception block and generate relevant exception information. The statements in *catch* block are apt to be executed by most failed runs and few passed runs, so that they have higher suspicious scores than the statements in other blocks. Let us take *Tarantula* to illustrate. These statements are given higher suspiciousness scores (i.e., close to 1). However, we notice that the statements in the catch block may not be faulty. As a result, such program structure will mislead the fault-localization process.

3.3 Unfair Case

Table 3: Statements in an if-return structure.

		passed runs		failed runs	
		r1	r2	r3	r4
n	if(x > y)	•	•	•	•
n+1	return x;		•	•	
n+2	return y;	•			•

The code excerpt in Table 3 shows a common *if-return* structure. Due to the existence of statement $n+1$, the execution counts of statement n is always no less than that of statement $n+2$. Since the suspicious score is a function of the execution counts of statements in failed runs and passed runs, we know that the suspiciousness of statement n and statement $n+2$ will be unfairly estimated. Apparently, such a program structure has impacts on the fault-localization result. However, it is still not easy to statically know whether the bias involved is positive or negative.

The above examples successfully show the impacts of program structure on the accuracy of fault localization techniques. If we can find out and remove the biases involved, the effectiveness of fault localization can be improved.

However, we still foresee some challenges related to such an approach. First, we have no confidence that we can iterate all legitimate pattern of program structures. Second, we cannot statically know whether the impact of a program structure (like the case in Section 3.3) is positive or negative for a pattern. Third, the existence of function calls and loops in program greatly increases the difficulty of frequency counting [24]. As a result, we adopt a statistical approaching not relying on any CFG-related methodology, which will be elaborated on in the next section.

4. OUR MODEL

In this section, we revisit current statistical fault localization techniques and a realistic software development scenario, and after that propose our method to capture and remove the impacts on the existing fault localization technique.

4.1 Statistical Fault Localization

Existing statistical fault localization techniques, such as *Tarantula* [11, 12], *Ochiai* [2], *Jaccard* [1], *CBI* [13], *DES* [25] and *SOBER* [14], use a suspiciousness function to assess the suspiciousness of a program entity being related to faults. They then rank all suspiciousness value in descending order, and generate a ranked list of program entities. Programmers may check the entities along the ranked list, from the most suspicious (top of the ranked list) to the least suspicious (tail of the ranked list) to locate the fault. The position of the faulty entity found in the ranked list reflects the effectiveness of a statistical fault localization technique.

Given a program $P = \langle s_1, s_2, \ldots, s_n \rangle$ with n statements and executed by a test suite of m test cases $T = \{t_1, t_2, \ldots, t_m\}$. During software testing, we can get the runtime profiles for a specific test case, namely a tuple $A_i = \langle a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np} \rangle$, where a^i_{ef} represents the number of failed test cases that execute statement s_i, a^i_{nf} represents the number of failed test cases that do not execute statement s_i, a^i_{ep} represents the number of passed test cases that execute statement s_i, and a^i_{np} represents the number of passed test cases that do not execute statement s_i.

A suspiciousness score formula is constructed using the four variables following some heuristics, and applied on each statement s_i to compute a real value that indicates the extent of statement s_i correlating to the program failures. For example, the suspiciousness score formula, *Tarantula* [12], is modeled as follows.

$$Tarantula(s_i) = \frac{\frac{a^i_{ef}}{a^i_{ef} + a^i_{nf}}}{\frac{a^i_{ef}}{a^i_{ef} + a^i_{nf}} + \frac{a^i_{ep}}{a^i_{ep} + a^i_{np}}}$$

The ranking order of two statements s_i and s_j is determined by the suspiciousness scores computed. Let us use the

technique *Tarantula* to illustrate. The order of statements s_i and s_j in the resultant ranked list are accordingly determined. The statement (e.g., s_i) having the prior order in the ranked list means that the statement s_i is deemed to be more suspicious than the statement s_j.

4.2 Realistic Developing Scenario

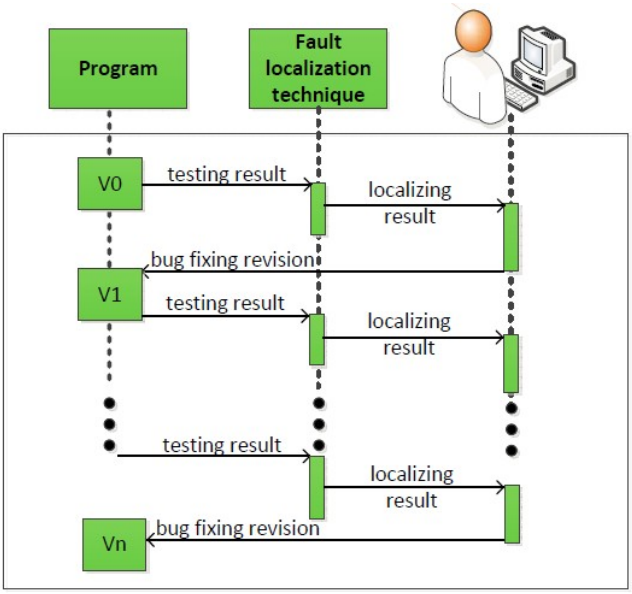


Figure 1: A scenario of software development.

Most modern software development process follows a continuous integration [10, 23] manner. Fig. 1 illustrates such a scenario. Firstly, a programmer gets an initial program (V0 in Fig. 1). When program failures are revealed during testing, debugging is carried out. A fault localization technique can be applied to estimate the fault position and the resultant ranked list is adopted by a programmer to narrow down the search region. After locating and correcting the first fault, a new version (V1 in Fig. 1) of the program is generated. Programmers need to carry out a testing task with it. If there still exist failed runs, a new round of debugging and testing starts (Vn in Fig. 1).

Fig. 1 shows the typical process of a software development scenario. We notice that generally there are more than one fault found in the history of development. Also, there are often more than one fault localization rounds conducted in the history of the software development. That provides us the possibility to capture the impact of program structure on fault-localization. We will elaborate on our idea in the next section.

4.3 Our Proposal

Our method consists of two steps. First, we use historical fault localization information to capture the impacts of program structure. Second, we remove the captured impacts from a fault localization result when locating fault in a program version in hand.

4.3.1 Capturing the Impacts of Program Structure

To capture the impacts of program structures, we investigate the historical suspiciousness ranked lists. Our basic heuristics is that "**a statement always having high suspiciousness score in all previous program versions has a chance to be overestimated to be fault-relevant by a fault localization technique, due to the bias caused by specific program structure related to it**". In our model, we formalize the impacts by statistically investigating the historical suspiciousness scores associated to a statement. We denote the impacts on statement s_i by program structures as $Imp^v(s_i)$, where v represents the v-th version of the program. $Imp^v(s_i)$ is calculated as follows.

$$Imp^v(s_i) = \begin{cases} 0 & \text{if } v = 1 \\ \frac{1}{v-1}\sum_{k=1}^{v-1} Tarantula^k(s_i) & \text{O.W.} \end{cases}$$

Note that for the first version of the program, $Imp^1(s_i) = 0$, which means that no history can be referenced and the impact is accordingly set to zero.

4.3.2 Removing the Impacts of Program Structure

After the impacts are captured, we can remove them to restore the unbiased effectiveness of fault localization techniques. In our model, we adopt a simply strategy to reduce the impacts from the suspiciousness score computed by a fault localization technique. Suppose we are using *Tarantula* as the fault localization tool, we use $Tarantula^v(s_i)$ to denote the suspiciousness score of s_i calculated by *Tarantula*. To differentiate from that, we use $Ours^v(s_i)$ to denote the suspiciousness score of our method, which is as follows.

$$Ours^v(s_i) = Tarantula^v(s_i) - Imp^v(s_i)$$

Note that for the first version of the program, $Ours^1(s_i) = Tarantula^1(s_i)$, which means that no history can be referenced and our method gives results identical to those of *Tarantula*.

4.4 Complexity

The basic complexity of applying our method is $O(V \times n)$, where V is the number of program versions in history and n is the number of statements in the program version in hand. The complexity can be reduced to $O(n)$ if we calculate $Imp^v(s_i)$ in an incremental manner.

5. EMPIRICAL EVALUATION

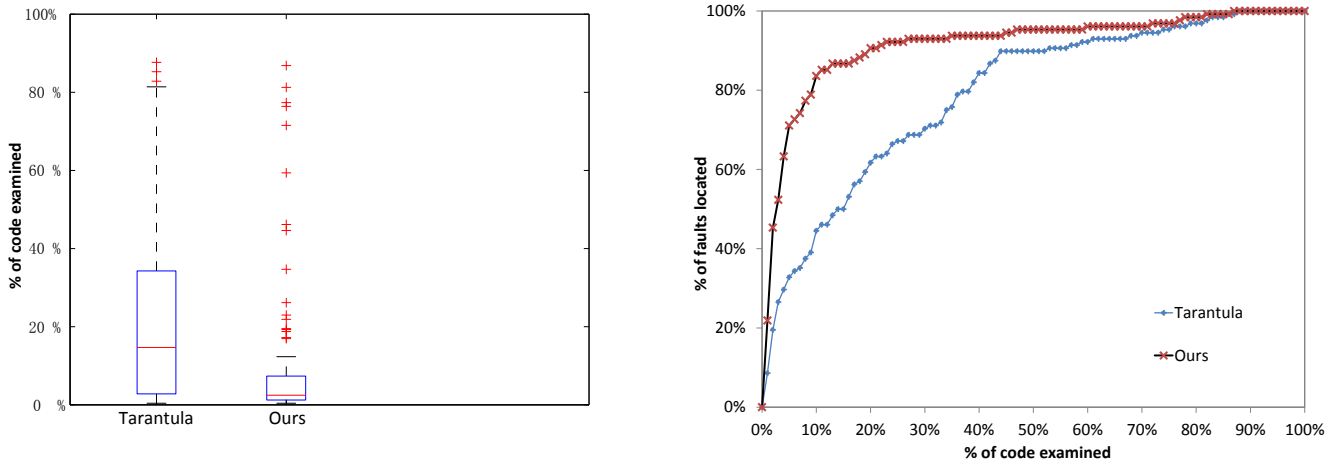
In this section, we conduct experiments to evaluate the effectiveness of our method.

5.1 Experiment Setup

5.1.1 Subject Programs

Table 4: Descriptive statistics of subject programs.

Program	# of executable statements	# of Versions	# of Cases
print_tokens	194–195	7	4130
print_tokens2	196–200	10	4115
replace	241–246	32	5542
schedule	151–154	9	2650
schedule2	128–130	10	2710
tcas	63–67	41	1608
tot_info	122–123	23	1052



(a) Box-Whisker plot

(b) Cumulative plot

Figure 2: Overall effectiveness

To evaluate our approach, we choose the common data set — Siemens suite of programs as our subject programs, which are downloaded from the Software-artifact Infrastructure Repository (SIR) [6]. Each of them have many sequential versions. Table 4 shows the descriptive statistics of the subject programs, including the number of applicable faulty versions, the number of executable statements (LOC), and the size of the test pools. Take *print_tokens* as an example, it is a program with 194 to 195 lines of executable statements, depending on which subversion. Five single-fault versions are used in this experiment, and they share a test suite consisting of 4130 test cases.

Following the documentation of SIR [6] and previous work [11, 13, 24], we exclude the versions whose fault cannot be revealed by any test case. That is because that our technique and the peer techniques rely on the existence of failed runs [1, 11, 13, 24]. At the same time, we mark the directly affected statement or an adjacent executable statement a faulty when the faulty statement is non-executable (such as [9]). The remaining 128 single-fault versions are used in the experiment. Meanwhile, we use some tactics associated with *gcov* to keep track of the traces of program executions to make the experiment more accurate, even though there are some crashing errors during execution.

5.1.2 Peer Technique

In our experiment, we choose the most representative statistical fault localization technique, *Tarantula*, to test. *Tarantula* [12] is chosen because it pioneers the work in this field and has many variants. The formula of *Tarantula* has been introduced in previous section.

5.1.3 Effectiveness Metrics

Tarantula generates a ranked list of all the executable statements in descending order of their suspiciousness scores. Then we check all statements along the ranked list, until a faulty statement is found. When two statements share identical suspiciousness score, we use the *Confidence* metric to further distinguish their degree of suspiciousness. A statement having a high Confidence value will have a high rank.

$$Confidence(s) = max(\%failed(s), \%passed(s))$$

It can be understood that a higher confidence is given to statements that are executed by more test cases.

Since a programmer is suggested to search for fault along the ranked list, the *Expense* metric is used to measure the fault-localizing quality of a ranked list.

$$Expense = \frac{rank\ of\ faulty\ statement}{number\ of\ executable\ statements} * 100\%$$

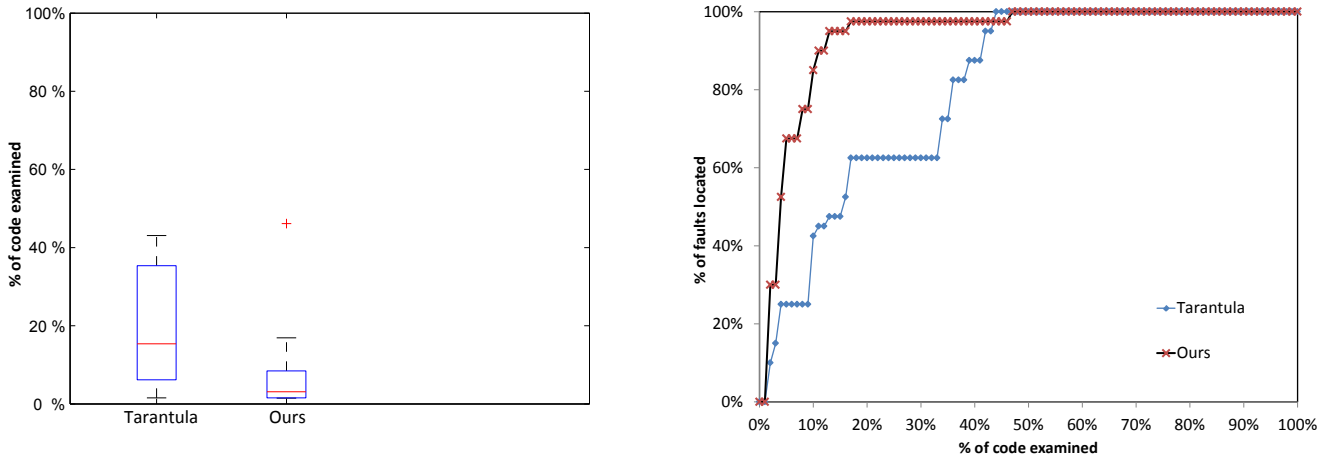
It is calculated as the percentage of the rank of the faulty statement to the total number of executable statements. The result can be understood as the code examination effort to locate a fault by examining a ranked list generated. The lower the value, the better the effectiveness of the fault localization is.

5.2 Results

In this section, we apply our method on *Tarantula* and evaluate the improvement. We use "*Ours*" to refer to the result of our method applied on *Tarantula* in the rest of the paper.

5.2.1 Overall Effectiveness

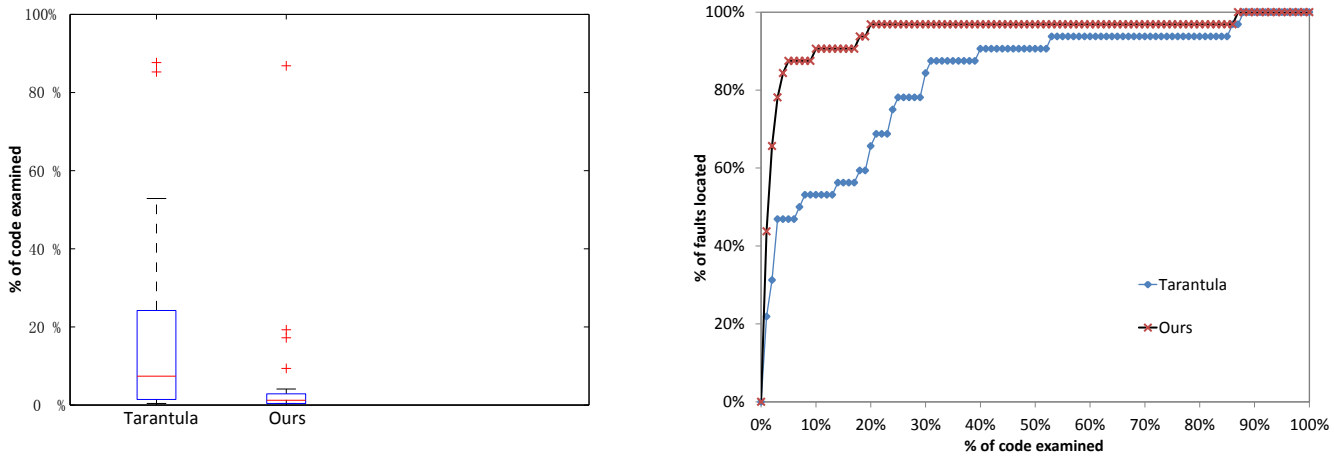
Fig. 2 shows the overall effectiveness of *Tarantula* and our technique. To give a better presentation, both the box-whisker plot and the cumulative plot are given to show the effectiveness of each technique. In the box-whisker plot (see Fig. 2(a)), we use two columns to show the effectiveness of *Tarantula* and our method, respectively. For each column, the cross in the box indicates the median value of the code examination effort to locate faults in each faulty version of the specific program. The bottom of the box corresponds to the 25% percentile, while the top of the box corresponds to the 75% percentile. The upper whiskey shows the maximum code examination effort within 1.5 IQR [4] of the upper quartile, while the lower whiskey shows the minimum code examination effort with 1.5 IQR of the lower quartile. The



(a) Box-Whisker plot

(b) Cumulative plot

Figure 3: Effectiveness on programs of scale $0 - 100L$ (tcas)



(a) Box-Whisker plot

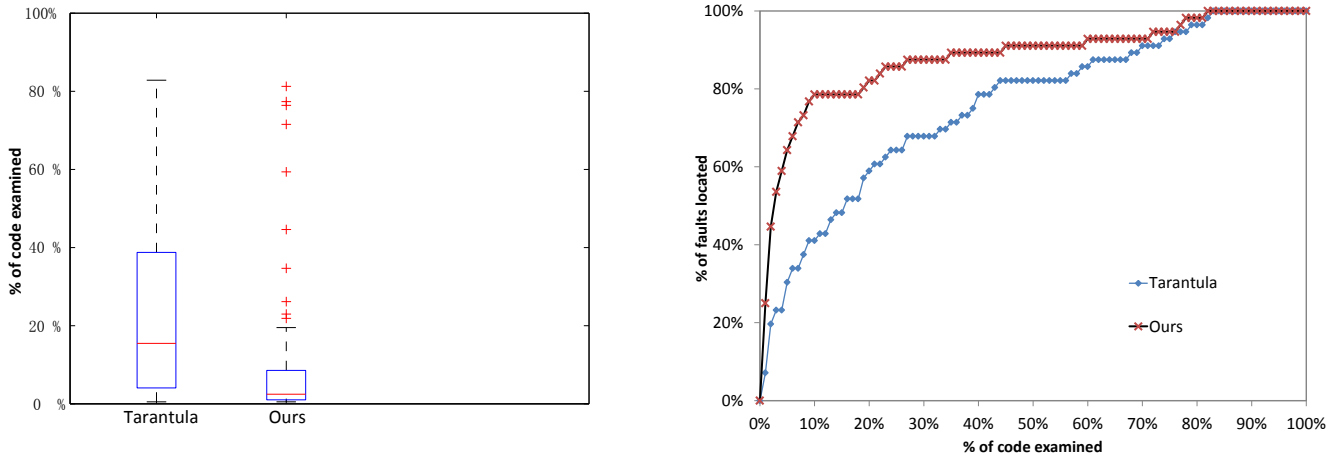
(b) Cumulative plot

Figure 4: Effectiveness on programs of scale $200L+$ (replace)

data not included between the whiskers are plotted as stars. Let us take *Tarantula* to illustrate, the lower whisker shows that the minimum code examination effort to locate a fault in one of the 128 faulty programs is 0.41%. The upper whiskey shows that the maximum code examination effort to locate a fault in one of the 128 faulty programs is 81.4%. The bottom and top of the box, which shows the 25% and 75% percentiles for code examination efforts with respect to each of the 128 faulty versions, are 2.80% and 34.27%, respectively. The cross in the box indicates that the median value of the code examination effort for the 128 faulty versions is 14.67%. While using our method, the median value of the code examination effort for the 128 faulty versions is 2.44%, the 25% and 75% percentiles for code examination efforts are 1.23% and 7.36%, respectively.

Fig. 2(b) shows the cumulative plot of the overall effectiveness of *Tarantula* and *Ours*. The x-axis indicates the code examination effort. The y-axis indicates the percentage of faults located within the code examination effort indicated by the x-coordinate. Both the curves for *Tarantula* and *Ours* starts from the point (0%, 0%) and finally reaches the point (100%, 100%). Apparently, no faults can be located when examining 0% of the code, while all faults can be located when examining 100% of the code. The plot shows that *Tarantula* can locate about 44.53% of all the faults by examining no more than 10% of the code in a faulty version, while our method can locate about 83.59% of all the faults. We observe that our method can always locate more faults than *Tarantula* in the code examination range between 0% and 88%.

In Fig. 2, we have the basic observation that applying our method on *Tarantula* can improve *Tarantula* greatly. Furthermore, we also want to know the detailed information on the effectiveness on each individual programs.



(a) Box-Whisker plot

(b) Cumulative plot

Figure 5: Effectiveness on programs in scale $100L - 200L$ (others)

5.2.2 *Effectiveness on Individual Programs*

We further compare the effectiveness of *Tarantula* and our method on each subject program. To give a clear view, we categorize the programs according to their scales (measured by the number of executable statements in a program). The program with executable statements less than 100, *tcas*, is extracted. And the program with executable statements more than 200, *replace*, is extracted. The programs with executable statements between 100 and 200 are grouped together, which includes 56 faulty program versions.

Fig. 3 shows the effectiveness of the two techniques over the program *tcas*. Fig. 3(a) shows the effectiveness using box-whisker plot, while Fig. 3(b) shows the cumulative plot. In Fig. 3(a), the median using *Tarantula* is 15.38%, the 25% and 75% percentiles for code examination efforts with respect to each of the 40 faulty versions are 6.16% and 35.38%, respectively. While the median using our method is 3.13%, the 25% and 75% percentiles for code examination efforts are 1.54% and 8.46%, respectively. In Fig. 3(b), the x-axis indicates the code examination effort in this paper, and the y-axis indicates the percentage of faults located. From it, we can see that the curve *Tarantula* passes through the point (10%, 42.5%), while the curve Ours passes through the point (10%, 85%). It means by examining up to 10% of code in each faulty program, our method can locate 85% of faults, while *Tarantula* can only locate 42.5% of faults. Both plots in Fig. 3 show that our method outperforms *Tarantula* on programs having a scale of 100 lines of executable code or less.

Fig. 4 shows the effectiveness of the two techniques over the program *replace*. Like the program *tcas*, *replace* shows the effectiveness using the box-plot and the cumulative plot. In Fig. 4(a), the median using *Tarantula* is 7.38%, the 25% and 75% percentiles for code examination efforts with respect to each of the 32 faulty versions are 1.44% and 24.18%, respectively. While the median using our method is 1.23%, the 25% and 75% percentiles are respectively 0.41% and 2.88%. In Fig. 4(b), the meanings of the x-axis and y-axis are similar with Fig. 2. From here, we can see the curve

Table 5: Improvements on subject programs

Program	Tarantula	*Ours*	Increment Ratio
print_tokens	24.72%	15.70%	36.49%
print_tokens2	18.44%	11.81%	35.95%
replace	17.18%	5.31%	69.09%
schedule	3.87%	2.13%	44.96%
schedule2	52.17%	23.52%	54.92%
tcas	19.39%	6.01%	69.00%
tot_info	24.05%	10.68%	55.59%
Average	22.83%	10.74%	52.29%

Tarantula passes through the point (10%, 53.13%), while the curve *Ours* passes through the point (10%, 90.63%). Fig. 5 shows the effectiveness of the two techniques over the program set *other*, which consists of six programs with 56 faulty versions. Like the above two programs *tcas* and *replace*, the box-plot and cumulative plot are shown. In Fig. 5(a), the median using *Tarantula* is 15.45%, the 25% and 75% percentiles for code examination efforts with respect to each of the 56 faulty versions are 4.07% and 38.76%, respectively. While the median using our method is 2.44%, the 25% and 75% percentiles are respectively 1.02% and 8.54%. From Fig. 5(b), we can see the curve Tarantula passes through the point (10%, 41.07%), while the curve Ours passes through the point (10%, 78.57%). Results in Fig. 4 and Fig. 5 also show that our method outperforms *Tarantula* on programs having a scale of 100+ lines of executable code.

The three figures confirms the existence of the impact of program structure on fault-localization effectiveness and validates the effectiveness of our method in removing the impacts. Further, we want to know how many improvements we made,which details are listed out in the next section.

5.2.3 *Statistics of Improvements*

Table 5 shows the code examination effort of *Tarantula* and *Ours*, accompanied with the increment ratio on code examination effort from *Tarantula* to that of *Ours*. We define the increment ratio as the ratio between the increment in

the code examination effort from *Tarantula* to *Ours* to that of *Tarantula*. Take the program *print_tokens* as an example (the second row), we can see that the average code examination effort using *Tarantula* on *print_tokens* is 24.72%, while our method only used 15.70%, and the increment ratio is $\frac{24.72\% - 15.70\%}{24.72\%} = 35.49\%$. Furthermore, the increment ratios on other programs are all between 35.95% to 69.09%. From the last row of the Table, we can see that the average code examination using *Tarantula* in all the faulty program versions is 22.83%, while our method only used 10.74%, and the average increment ratio is 52.29%. We observe that our method always has an improvement over *Tarantula*.

5.3 Threats to Validity

We use *gcov* to keep track of the traces of program executions. Meanwhile, we use some tactics associated with *gocv* to get the traces for runtime executions for crashing cases. As exception information in runtime contains plenty error information, we take the runtime exception runs as failed runs. Different experiment setup may result in different observation.

Our method is general because it needs history suspiciousness list. Given a fault localization technique and history suspiciousness list, our method can be applied. However, the strategy we use to capture the impacts from program structure is only one possible solution.

We follow the SIR documents to manipulate the common data set to simulate a software development scenario. Experiments in realistic scenario may manifest different observations.

6. CONCLUSION

Existing statistical fault localizations utilize the executing information to estimate the positions of faults and narrow down the region of the faulty statements. However, most of them pay little attention to the biases caused by program structure.

In this paper, we studied the impacts by the program structure on the effectiveness of statistical fault localization techniques. We proposed a strategy to capture the impacts caused by the program structure on the fault localization techniques, and remove them. We conducted a controlled experiment on the representative technique *Tarantula* over Siemens test suite to evaluate the effectiveness of our method. The experimental result confirms the existence of the concerned impacts and validates the effectiveness of our method in improving the effectiveness of *Tarantula*.

Further work includes a thorough study on other fault localization techniques and the performance on other programs. In order to enhance the effectiveness of our method, a prospective is to integrate date flow profiles as well. Further, Abreau et al. [3] proposed an approach to locate faults in multi-fault programs. Our future work can incorporate this aspect because our method has the potential to downplay the statements that always have with unreal high suspiciousness.

7. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemud. On the accuracy of spectrum-based fault localization. In *Proceedings of TAICPART-MUTATION 2007*, pages 89–98, Washington, DC, USA, 2007.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing(PRDC 2006)*, pages 39–46, 2006.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 88–99, 2009.

[4] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008(ICPC 2008)*, pages 182–191, 2008.

[5] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *International Conference on Software Engineering, 2009(ICSE 2009)*, pages 34–44, 2009.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, pages 405–435, 2005.

[7] M. J. Harrold, G. Rothermel, R. W., and L. Yi. An empirical investigation of program spectra, 1998.

[8] R. M. Hierons. Avoiding coincidental correctness in boundary value analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(3):227–241, 2006.

[9] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA 2008, pages 167–178, 2008.

[10] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen. How well does test case prioritization integrate with statistical fault localization? *Information and Software Technology*, 54(7):739–758, 2012.

[11] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[13] B. Liblit, A. Aiken, M. Naik, and Alice X. Zheng. Scalable statistical bug isolation. In *In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26. ACM Press, 2005.

[14] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[15] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Transactions on Software Engineering and Methodology*, 23(1):8:1–8:28, 2014.

[16] D. Muriel. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on*

Software Testing and Analysis, pages 158–171. ACM Press, 1996.

[17] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ACM Software Engineering Notes*, pages 432–449. Springer-Verlag, 1997.

[18] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *Software Engineering, IEEE Transactions on*, 19(6):533–553, 1993.

[19] I. Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man and Cybernetics*, 16(5):621–637, 1986.

[20] X. Wang, S. C.Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 45–55, 2009.

[21] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software of Engineer and Methodology*, 22(4):31:1–31:40, 2013.

[22] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, and S. Li. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, 55(5):880 – 896, 2013.

[23] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, 2013.

[24] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE 2009, pages 43–52, New York, NY, USA, 2009.

[25] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. Debugging through evaluation sequences: A controlled experimental study. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pages 128–135, 2008.