An Empirical Study on the Use of Mutant Traces for Diagnosis of Faults in Deployed Systems

[1]Syed Shariyar Murtaza, [2]Abdelwahab Hamou-Lhadj, [3]Nazim H. Madhavji, [4]Mechelle Gittens

[3,4] Dept. of Computer Science, University of Western Ontario, London, Canada

[4]Dept. of Computer Science, University of West Indies, Cave Hill, Barbados

[1,2]Dept. of Electrical and Computer Engineering, Concordia University, Canada

[1]smurtaza@encs.concordia.ca, [2]abdelw@ece.concordia.ca, [3]madhavji @csd.uwo.ca, [4]mechelle.gittens@cavhill.uwi.edu

Abstract— Debugging deployed systems is an arduous and time consuming task. It is often difficult to generate traces from deployed systems due to the disturbance and overhead that trace collection may cause on a system in operation. Many organizations also do not keep historical traces of failures. On the other hand earlier techniques focusing on fault diagnosis in deployed systems require a collection of passing-failing traces, in-house reproduction of faults or a historical collection of failed traces. In this paper, we investigate an alternative solution. We investigate how artificial faults, generated using software mutation in test environment, can be used to diagnose actual faults in deployed software systems. The use of traces of artificial faults can provide relief when it is not feasible to collect different kinds of traces from deployed systems. Using artificial and actual faults we also investigate the similarity of function call traces of different faults in functions. To achieve our goal, we use decision trees to build a model of traces generated from mutants and test it on faulty traces generated from actual programs. The application of our approach to various real world programs shows that mutants can indeed be used to diagnose faulty functions in the original code with approximately 60-100% accuracy on reviewing 10% or less of the code; whereas, contemporary techniques using pass-fail traces show poor results in the context of software maintenance. Our results also show that accuracy and different faults in closely related functions occur with similar function call traces. Our results also show the challenges related to using mutants.

Index Terms— Software Debugging, Software Maintenance, Fault Diagnosis, Fault location, Software Reliability, Mutation.

1. Introduction

Typically, maintainers collect data (such as execution traces, error logs, etc.) related to software failures in order to debug the causes of failures. For example, Windows Error Reporting (WER, 2012), Mozilla crash reporting (Mozilla, 2013), and Ubuntu's Apport crash reporting (Ubuntu, 2013) collect function calls on stacks and other related information to debug the causes of crashes. Similarly, maintainers at IBM collect function call traces for DB2 (Melnyk, 2004) and WebSphere (Hare & Julin, 2007) from the field to diagnose the causes of crashing failures and non-crashing failures[1] (e.g., performance failures, unexpected outputs, etc.). However, diagnosing the origin of faults causing failures in deployed systems is time consuming and can take up to 30%-40% of the corrective maintenance time (UWO & IBM, 2008).

Prior techniques focusing on automatic fault diagnosis in deployed systems (e.g., using statistical debugging (Chilimbi et al., 2009) (Liu & Han, 2006)) propose to diagnose fault locations by collecting passing and failing traces from deployed systems at a time when the fault occurs. Other researchers (Brodie et al., 2005) (Lee & Iyer, 2000) (Murtaza et al., 2010) (Podgurski et al., 2003) focusing on deployed systems propose to correlate (function call) failure traces from deployed systems with historical traces of failures to identify recurrent faults. Another typical practice, mostly used in in-house software testing, is to reproduce faults on test machines and collect the corresponding program traces of pass-fail test cases. In this practice, passing-failing traces are collected at a finer grained level, such as statements, and fault localization techniques are executed on them: many fault localization techniques have been proposed that focus on software testing (e.g., (Agrawal et al., 1995) (Wong & Qi, 2006) (Jones & Harrold, 2005) (Zhang et al., 2009) (Wong W. E. et al., 2007), etc.).

In practice, it is usually not feasible to collect many traces from deployed systems, due to overhead incurred during trace collection that can impact business operations. Further, historical traces based techniques usually detect only known faults and it is common that historical traces are not available in many organizations. It is also time consuming to reproduce thousands of faults reported from the field in a lab environment and many faults are not easily reproducible (e.g., crashes occurring due to specific system configurations).

In the field, function call traces are commonly collected traces for crashing failures (Mozilla, 2013) (Ubuntu, 2013) and non-crashing failures (Melnyk, 2004) (Hare & Julin, 2007), and mostly a failed trace is collected for a corresponding failure. In this paper, we focus on the problem of identifying faulty functions from a function call[2] trace of a deployed software system. We approach the solution of this problem by employing the concept of software mutation for the identification of faulty functions. A software mutant is an artificially generated

[1] Note that non-crashing failures can manifest themselves long after the execution of fault and are difficult to resolve than crashes.
[2] Our focus is on fault diagnosis from function call traces because they are mostly collected from the field, not other kinds of traces.

fault in a program and Andrews et al. (Andrews et al., 2005) showed that mutants are close representative of actual faults. More precisely, we investigate whether we can generate mutants for functions of a program and use their (mutants) traces to locate faulty functions in the traces of actual faults of the program.

The use of mutants for fault localization is a novel approach as mutants have mostly been used to measure, enhance, and compare the effectiveness of testing strategies (Offutt & Untch, 2001) and test coverage criteria (Andrews J. et al., 2006)). If traces of mutants can be used to diagnose actual faults, then it can relieve the collection of historical failed traces or pass-fail traces from deployed systems and facilitate the diagnosis of faults without spending time in fault reproduction. The use of mutants will also reduce the overhead of multiple trace collection from deployed systems but at the expense of time to generate traces of mutants before deployment (i.e., offline). However, savings in time and overhead of trace collection for systems in operations (deployed systems) is more critical than offline trace generation. This paper therefore addresses the following novel research question:

(Q1) Can we diagnose actual faults in traces of deployed software systems by using only the traces of mutants (i.e., automatically seeded artificial faults) of software systems?

In on our earlier work (Murtaza et al., 2010), we showed that different actual faults in the same function occur with similar function call traces. In this paper, we extend this investigation further by determining how different artificial (mutants) and actual faults in functions are related to each other in terms of function call traces. This can be beneficial in understanding the relationship among different faulty functions and improving the fault diagnosis process. Therefore, a secondary research question that follows from (Q1) is:

(Q2) Do different artificial faults (mutants) and actual faults in functions occur with similar function call traces?

We determine the answers to these novel research questions by training decision trees on the traces of mutants of functions in a program and predicting faulty functions in actual faulty traces of that program. Our results on public programs show that mutants can be used to diagnose actual faults and different faults in a group of functions occur with similar function call traces. These results are novel and contribute to the knowledge of corrective maintenance and the literature on fault diagnosis.

The rest of the paper is as follows. We present our approach in Section 2, and case studies to evaluate our

approach in Section 3. In Section 4, we describe the related work by comparing our technique with the other techniques. Section 5 explains the threats to validity and Section 6 concludes this paper with the directions to future work.

2. Approach

The steps of our approach are shown in Figure 1. Initially, we generate mutants (artificial faults) for the functions of the program. The next step is to collect function call traces from executing the mutants. We call these traces *mutant traces*. This step requires generating the mutants and running test cases on them. A mutant trace is collected when the output of a test case is different from the original program (deemed correct). Function call traces are then stored in a database. The records of each mutant in the database are labeled with the corresponding faulty function.

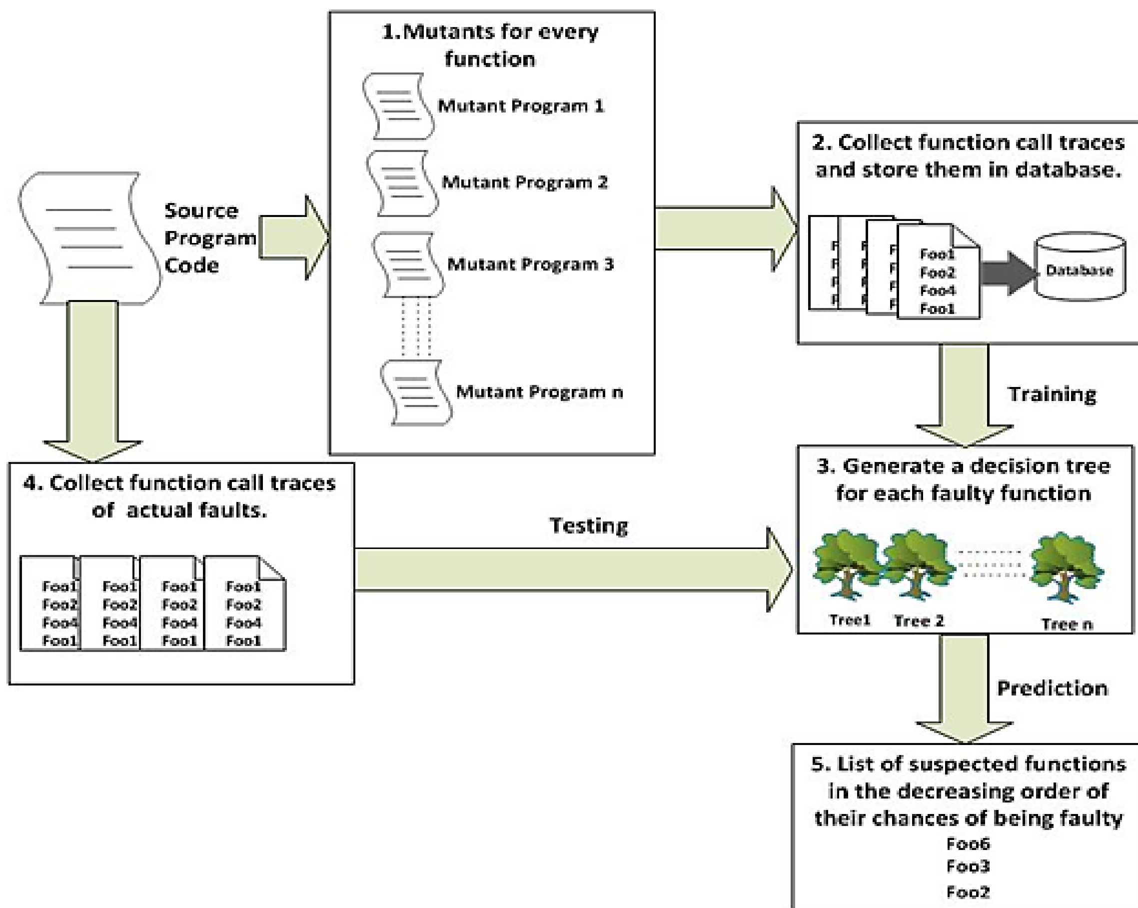

Figure 1: Steps of our approach.

Using the database of mutant traces, we build decision tree models. These models are used later to identify the actual program functions that may be faulty. We use decision trees as our learning technique but several other classification algorithms (e.g., neural network, support vector machines, etc.) can also be used. In one of our earlier papers, we empirically determined that there is no significant difference between classification algorithms when use on call traces (Murtaza et al., 2012). We chose decision trees because of their efficacy in training and human readable rules.

The next step is to collect traces of actual faults from the original program. Ideally, these traces should be collected from the field but due to the limitation of publicly available datasets as required by our study, we run the actual program using test cases and collect traces of test cases that cause the system to fail (we discuss this further in Section 3.1). We call these traces *actual failed traces* to distinguish them from *mutant traces*. The actual failing traces are provided to the trained decision trees. Each decision tree predicts its faulty function with a probability for a given trace. Functions are then arranged in the decreasing order of the probability. The functions with the highest probabilities are the ones that are most likely to be faulty. The list is then evaluated by determining if an actual faulty function of a trace is present in the predicted list.

In Section 2.1, we describe the fundamentals of mutation and show examples of mutants. In Sections 2.2 and Section 2.3, we explain the training and testing of the decision trees, respectively. Finally, in Section 2.4 we explain the implementation details.

2.1 Generating Mutants

The term mutation refers to the generation of mutants (faulty variant) of a program by applying mutant operators (e.g., replacing an arithmetic operator with another operator in a statement). Mutants are automatically generated (virtual) faulty versions of the system. A mutant is considered dead (or killed) if the output of a test case on the mutant differs from the output of that test case on the original program (Offutt & Untch, 2001). Mutants which are not killed by test cases are called live mutants. Live mutants actually show inadequacy and weaknesses of the test suite in exposing faults (Offutt & Untch, 2001). If a test suite misses some control flow paths of a program then it would be weak in detecting mutants (i.e., faults) on those paths of a program. Sometimes mutants become equivalent to the original program and they cannot be killed (Offutt & Untch, 2001)—i.e., they produce the same output as the original program. Identifying equivalent mutants is a tedious task and it is an undecidable problem (Andrews et al., 2005) (Offutt & Untch, 2001)—not even automatic solutions can identify all equivalent mutants.

In this paper, we used a program developed by Andrews et al. (Andrews et al., 2005) to generate mutants for C programs. In order to generate mutants for a source file, the authors applied mutation operators (when

possible) sequentially to each line of code. This results into one mutant for every valid application of a mutation operator on each statement. Andrews et al. used the following four classes of mutation operators (Andrews et al., 2005):

- First class replaces an integer constant C by 0, 1, -1, ((C)+1), or ((C)-1).
- Second class replaces an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class.
- Third class negates the decision in an `if` or `while` statement.
- Fourth class deletes a statement.

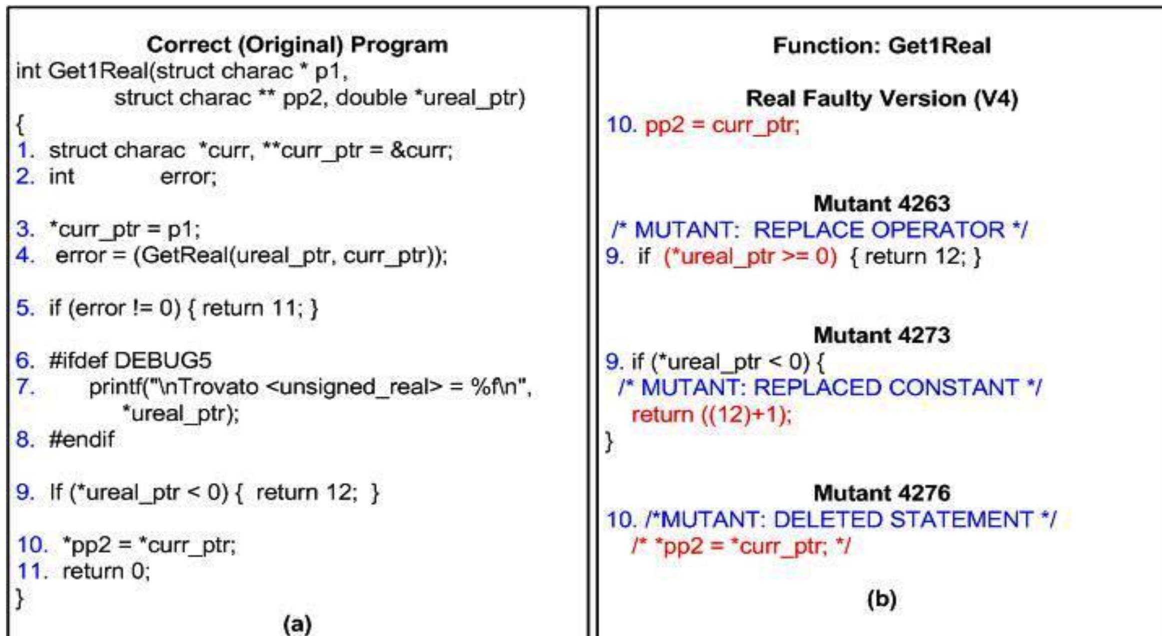

Figure 2: Correct source code of the function "Get1Real" of the Space program, its real faulty version and its faults generated using mutants.

In Figure 2, we show examples of mutants using the above mutation operators. Part 'a' shows the source code of a correct program for the function "Get1Real" of one of the subject programs, called Space[3], used in the case study (Do et al., 2005). Part 'b' of Figure 2 shows the faulty statements of the same function "Get1Real" found in real faulty version of the program, and randomly selected mutants for the same function "Get1Real". These three mutants are obtained after applying three different mutant operators from the above list.

Andrews et al. have also examined the relationship between mutant and actual faults (Andrews et al., 2005). They have found that mutant faults are close representative of actual faults, but they are different from hand

[3] Space is a C program, an interpreter for an antenna array definition language written for the European Space Agency

seeded faults. They determine this by executing several testing strategies on mutants, actual faults and hand seeded faults. They have also observed that the hand seeded faults are harder to detect than the actual faults. Moreover, hand seeded faults require human effort; whereas, mutants can be generated automatically which can save valuable time.

The process of mutation can result in a large number of mutants with several mutants for every single statement. For example, for the Space program (6218 LOC), the process of mutation can result in 12,262 mutants. It can be quite expensive to run test suites on all the mutants of a program and collect the failed (mutant) traces for each mutant. Therefore, we decided to randomly generate three mutants for every function of the subject programs. Ideally we should have one mutant per function but we use three mutants per function in order to avoid a situation of no mutant traces when a mutant does not compile or when no test cases fail on the mutant (Andrews J. et al., 2006). In our study, three mutants per function seem to work well; however, adequate number of mutants can vary from one program to another. For example, a function with large number of statements (LOC) might require more than three mutants per function and function with fewer LOC might settle with only one mutant. Determining an adequate number of mutants for fault diagnosis such that effort of running test suites also remains feasible requires another empirical study and is beyond the scope of this work.

Furthermore, it can still be quite resource draining to run all the test cases on selected mutants and collect traces. We decided to put an upper limit on the number of traces we generate for each mutant. We generate traces for a maximum of 10 failed test cases per mutant; i.e., 10 mutant traces. This results in 30 mutant traces per function and allows us to determine the minimum number of mutant traces necessary for adequate fault diagnosis (e.g., 10, 20 or 30 per function). The intuition behind limiting number of mutants and number of mutant traces is to keep the use of mutation scalable from the perspective of resource utilization. After collecting mutant traces, we trained decision trees on them to predict faulty functions in the failed traces of actual faults. In the machine learning terminology, the mutant traces form a training set and the actual failed traces form a test set.

2.2 Training the Decision Trees on Mutant Traces

The decision tree algorithm requires mutant traces to be converted into a form on which the decision tree can be applied (Witten & Frank, 2005). This transformed representation of the mutant traces is shown in Figure 3, part a. Figure 3 shows selected examples of function calls and mutant traces for the Space program (used in the case study) (Do et al., 2005). A row represents a mutant trace and a cell represents the occurrence of function calls in that trace. The last column shows faulty functions corresponding to the mutants. In data mining terminology, function calls are independent variables and the faulty function is referred to as the dependent

variable.

The reason for selecting single function calls as independent variables in Figure 3 lies in the empirical investigation of our earlier paper (Murtaza et al., 2010), where we have empirically investigated that the patterns (sequences) of function calls do not yield better results than the single function calls when used with the decision tree. For example, if a trace has four functions {F1, F2, F3, F4} and a decision tree is trained on patterns like {"F1,F2","F2,F3", "F3,F4", etc.} and on only individual functions, then the decision tree yields the same accuracy—implying, training on individual functions is as efficient as when using patterns. Thus, we can avoid using patterns as their extraction causes an additional overhead.

	Functions							
	adddef	addscan	doubmax	elemdef	……….	versdef	waitcont	Faulty Functions
M373_T5605	4	0	0	1	.	0	1	adddef
M376_T2755	2	0	0	1	.	0	1	adddef
M025_T9506	1	0	0	1	.	0	1	Get1Real
M336_T13053	3	0	0	1	.	1	1	grgeodef
……………								…………
M228_T5123	4	1	1	1	.	0	1	simamp

(a) Original dataset for all categories

M373_T5605	4	0	0	1	.	0	1	others
M035_T9506	1	0	0	1	.	0	1	Get1Real
M336_T13053	2	1	0	1	,	0	1	others
M4276_T9507	1	0	0	1	,	0	1	Get1Real

(b) Dataset for "Get1Real" against all others

Figure 3: Faulty functions and traces from mutants of the Space program.

Following the transformation of data shown in Figure 3, we trained the decision tree on it using the one-against-all approach (Witten & Frank, 2005). In the one-against-all approach, a dataset (as in part 'a' of Figure

3) with M categories of dependent variables (faulty functions) is decomposed into M new datasets with binary categories. Each new binary dataset ‘Di’ has category ‘Ci’ (where i = 1 to M) labeled as positive and all other categories labeled as negative. An example of a dataset of a faulty function “Get1Real”, against all “others” faulty functions, is shown in part b of Figure 3. The columns for part ‘b’ of Figure 3 are the same as for part ‘a’ of Figure 3.

According to the one-against-all approach, on each new datasets ‘Di’ the decision tree algorithm is trained, resulting in ‘M’ trees in total. Whenever a new faulty trace (we mean here an actual failed trace) comes, each decision tree predicts its category ‘Ci’ of the dependent variable (i.e., the faulty function) along with a probability of being faulty. A category ‘Ci’ (i.e., faulty function) with the highest probability is considered as the correct prediction. Empirical evidence shows that training multiple decision trees (one-against-all) on several binary datasets yields better results than training a single decision tree on a dataset with many categories of dependent variables (Polat & Güneş, 2009).

```
| | _portspec <= 0
| | | _adddef <= 0
| | | | _recgrdef <= 0
| | | | | _Get1Real <= 1: Get1Real
| | | | | _Get1Real > 1
| | | | | | _GetKeyword <= 9: Get1Real
| | | | | | _GetKeyword > 9: others
| | | | _recgrdef > 0
| | | | | _GetKeyword <= 9
| | | | | | _TapeGet <= 96: Get1Real
| | | | | | _TapeGet > 96: others
| | | | | _GetKeyword > 9
| | | | | | _GetKeyword <= 15
| | | | | | | _TapeGet <= 140
| | | | | | | | _lenunit <= 0
| | | | | | | | | _Get1Real <= 1
| | | | | | | | | | _hexgrdef <= 0: Get1Real
| | | | | | | | | | _hexgrdef > 0
| | | | | | | | | | | _TapeGet <= 108: others
| | | | | | | | | | | _TapeGet > 108
| | | | | | | | | | | | _TapeGet <= 110: Get1Real
| | | | | | | | | | | | _TapeGet > 110: others
| | | | | | | | | _Get1Real > 1
| | | | | | | | | | _TapeGet <= 110: Get1Real
| | | | | | | | | | _TapeGet > 110: others
| | | | | | | | _lenunit > 0: others
```

Figure 4: The C4.5 decision tree model for the function “Get1Real” of the Space program from failed traces of mutants by using one-against-all approach.

An excerpt of the trained decision tree generated for part ‘b’ of Figure 3 is shown in Figure 4. Each row contains a function, its frequency of occurrence, and the name of a faulty function after a colon sign if any. The function name represents a tree node and the faulty function name after the colon sign represents the leaf of the

tree. For example, the decision tree of Figure 4 shows that if in a failed trace the occurrence of the function "portspec", "adddef" and "recgrdef" is less than or equal to "0" and the function "Get1Real" is <=1, then the faulty function is "Get1Real".

The tree of Figure 4 was obtained by applying the J48 algorithm in the data mining tool Weka (Witten & Frank, 2005), which is an implementation of the C4.5 decision tree algorithm. The C4.5 decision tree algorithm is the most widely and practically used algorithm. It is suitable for a dataset with numerical values (e.g., see Figure 4) of independent variables, unlike ID3 decision tree algorithm (Witten & Frank, 2005) which works only with nominal values of independent variables. The details of the C4.5 algorithm can be found in standard text by Quinlan (Quinlan, 1993) and Witten and Frank (Witten & Frank, 2005).

2.3 Testing the Decision Trees on Actual Failed Traces

Following the training of the decision trees on mutant traces, actual failed traces are provided as input to the decision trees for prediction. Each decision tree predicts a faulty function with a probability. The probability of prediction in the C4.5 algorithm is determined by measuring the number of training instances correctly classified at a leaf and dividing it by the total number of instances (correct and incorrect) reached that leaf (Quinlan, 1993).

	Function	Probability
	Space_Fault11_t1915	
Rank 1	intmin	0.044
Rank 2	mksnode	0.042
	Space_Fault4_t4455	
Rank 1	circspec	0.119
Rank 2	prnfile	0.0166
Rank 3	Get1Real	0.0165

Figure 5: Ranking of suspected faulty functions in real failed traces obtained from the decision tree model of failed traces of mutants

When predicting faulty functions using the one-against-all approach, we made a minor modification. Instead of selecting a predicted faulty function with the highest probability, we ranked the predicted faulty functions in the decreasing order of their predicted probabilities. This allows the developer to have multiple options in case

the function with the highest probability is not the actual faulty function. The function list is then presented to the developer. The developer's effort to go through the list of probable faulty functions can be quantified using common metrics for effort estimation when diagnosing a fault (Jones & Harrold, 2005) (Di Fatta et al., 2006) (e.g., percentage of functions reviewed to diagnose faulty functions).

An example of a ranked list of faulty functions for two different traces of actual faulty versions of the Space program is shown in Figure 5. This figure first shows ranking for the trace "t1915", corresponding to "fault 11" of the Space program, according to the probabilities predicted by the decision trees. In "fault 11" "mksnode" is the faulty function ranked at position 2. Similarly, for the trace "t4455" corresponding to "fault 4", the function "Get1Real" is the faulty function.

2.4 Implementation

We implemented this application in Java, and used MySQL database to store processed traces (e.g., functions and occurrences). We optimized the application for bulk reads of large traces from hard disk, bulk inserts of large records into the database, and used different table-indexes in MySQL database. We used SWI Prolog based mutant generation tool developed by Andrews et al. (Andrews et al., 2005). The tool generates mutants for almost every statement of the program, which results in a large number of mutants. We modified this tool to automatically extract functions and their locations in source code and automatically generate three random mutants for every function by using Java and Shell scripting.

3. Case Studies

This section answers the two research questions of Section 1: (Q1) Can we diagnose actual faults in traces of deployed software systems by using only traces of mutants (i.e., automatically seeded artificial faults) of software systems? (Q2) Do different artificial (mutants) and actual faults in functions occur with similar function call traces?

To answer the questions, we applied our approach to different systems, namely, the Space program and the UNIX utilities (i.e., Grep, Gzip and Sed) (Do et al., 2005) as the dataset. The description of these programs is provided in Section 3.1. Second, we generate mutants of the subject programs, explained in Section 3.2. Third, we apply our approach on the traces of mutants and traces of actual faults (see Section 3.3 to Section 3.4). Fourth, we compare our findings with the related techniques in Section 3.5, fifth we repeat our experiments due to their random nature in Section 3.6, and sixth we discuss them in Section 3.7.

3.1 Target Systems

Our approach requires access to both crashing and non-crashing traces of failures along with the source code. Ideally, our target systems should be traces of a commercial software system collected from the field. However, the publicly available traces are only limited to stack traces of crashes. In the case of proprietary software systems, it is not trivial to obtain access to source code and traces. It has been shown in the literature that faults in field overlap with the faults in testing (Gittens et al., 2005). Therefore, we decided to use publicly available dataset of programs such that we can collect both crashing and non-crashing failure traces by running test cases. Our failure traces are composed of both non-crashing failures (e.g., due to logical faults) and crashing failures (e.g., due to segmentation faults). Please make note that the use of failure traces using test cases should not be confused with the in-house software debugging techniques.

Table 1 Characteristics of the subject programs

Flex, Sed, Grep and Gzip are well known UNIX utilities. Space is an interpreter for an antenna array definition language written for the European Space Agency.						
Program	Test Cases	LOC (excludes comments & blank lines)	Functions	Distinct Faulty Functions	Faulty Versions (# Faults)	Actual Failed Traces
Grep (release 2.4)	809	9041	149	4	5	247
Gzip (release 1.1.2)	214	4032	89	7	7	246
Sed(release 4.0.7)	370	4735	143	1	3	60
Space	13585	5767	136	26	34	71958

We used the Space program (Do et al., 2005) and three open source UNIX utilities (Do et al., 2005), namely, Grep, Gzip and Sed for our experiments. Space is a C program, an interpreter for an antenna array definition language written for the European Space Agency, and the faults were found during actual development. The UNIX utilities are well-known open source public applications. Although faults in the UNIX utilities were hand seeded, a specific procedure was followed to keep them realistic (Do et al., 2005). The important steps of fault insertion procedure in the UNIX utilities were:

(a) Identification of the changes in source code of different releases.

(b) Insertion of faults at the changes in the code by multiple programmers working independently.

(c) Insertion of faults associated with definition, redefinition, deletion, and change of values of variables.

(d) Insertion of faults associated with control flow, such as deletion of path, addition of new block of code,

redefinition of execution condition, modification to external function-calls, etc.

(e) Insertion of faults associated with memory; e.g., erroneous use of pointers, memory not allocated, etc.

(f) Merging of all the faults and removal of overlapping faults such that programs should compile.

The Space program and the UNIX utilities are made available by Do et al. (Do et al., 2005) at the Subject Infrastructure Repository (SIR). The Space program has been used in a number of fault localization studies (e.g., (Bowring et al., 2004) (Wong & Qi, 2006) (Jones & Harrold, 2005)). The UNIX utilities are also used in different studies including the identification of faulty statements using edge profiles (Zhang et al., 2009). Table 1 provides more details on these programs. In Table 1, each program consists of one original version deemed correct because it passed all the test cases, and several faulty versions. A faulty version is a variant of the original version by one fault—that is, one fault is equivalent to one faulty version. A fault is equivalent to incorrect statements in the code. In Table 1, the first column shows the name of the program (with actual release number) and the second column shows the number of test cases. In the UNIX utilities and Space, the test cases are shared across faults; i.e., all the test cases are run on each fault. Third and fourth columns show the number of lines of code and the number of functions in the program. The last three columns of Table 1 show the number of distinct faulty functions, the number of faulty versions or faults in a program, and the number of failed test cases for the program.

We used Etrace (Devillard & Chudnovsky, 2004) to collect the function call traces for both mutants and original programs. A failed trace was collected when a test case failed on a faulty version. A test case was considered failed when the output of the same test case on the faulty version differed from the original version of the program.

3.2 Generating Mutants

Recall from Section 2.1 that we randomly selected three mutants per function and each mutant belongs to one of the four types. We were able to generate 246-517 mutants for the four subject programs and it took 23-64 sec for their generation. Table 2 shows the percentages of four classes of mutants in the four subject programs. The last row in Table 2 shows the total percentage of three randomly selected mutants out of the total number of mutants for each program. We collected a mutant trace if the output of a test case on the mutant version of a program and the original (deemed correct) version of a program differed. As for the number of mutant traces, recall from Section 2.1 that we decided to limit this to ten failed traces per mutant. In total, we had a maximum of 30 failed traces for every function of a program.

During our investigations, we observed that for some functions the number of failed traces per function were

less than the maximum limit of 30. This is because, sometimes, the randomly selected mutants did not compile, a few test cases failed, or no test cases failed on the mutant. In short, there were 30 or less failed mutant traces per function. The average size of a mutant trace was 48.2-595.16 KB; whereas, the average size of the actual trace was 33.62-440.19 KB. The average time to parse a trace and store it in database was 0.24-1.81 seconds. In order to investigate how many failed traces of mutants per function are enough to identify faulty functions in the actual failed traces, we gradually experimented with 5, 10, 15, 20, 25 and then 30 mutant traces per function.

Table 2 Percentages of mutants in the four programs

Mutant Type	Grep	Gzip	Sed	Space
Replace Constant	31.8%	43.9%	26.0%	47.0%
Negate Decision	7.50%	6.50%	8.50%	7.20%
Replace Operator	35.80%	28.0%	30.0%	17.50%
Delete Statement	24.0%	21.50%	35.27%	28.27%
Percentage of randomly selected mutants out of total mutants				
	3.17%	2.44%	4.86%	3.55%

3.3 Experiments on the Space Program and the UNIX utilities

We trained the decision tree on these failed traces to identify faulty functions in the actual traces. Figure 6 shows the results obtained for 5, 10, 15, 20, 25 and 30 mutant traces per function for the Space program. In Figure 6, the X-axis represents the percentage of the program to be examined in diagnosing faulty functions. It is measured by the percentage of functions reviewed from the predicted list of faulty functions up to the discovery of the actual faulty functions in the program, as shown in Equation 1. It should be noted that this metric does not measure complete developer's effort to review functions since it does not take into account the complexity of each function.

$$\begin{bmatrix} \%\ of\ \ program \\ to\ review \end{bmatrix} = \frac{Functions\ reviewed\ upto\ the\ faulty\ function}{Total\ functions} * 100$$

Equation 1: Estimating program review effort in functions.

The X-axis represents the percentage of a program that needs to be examined and is divided into segments. Each segment is 10 percentage points except for the first 10 segments which are divided into 1 percentage points, i.e., 1-10% segments are divided into 1 percentage points and 90-100% segments are divided into 10

percentage points each.

The Y-axis measures the cumulative percentage of mutant traces that achieve a score within a segment. We have taken this approach from the similar graphical convention used for evaluation of the developer's effort by other researchers (Jones & Harrold, 2005) (Wong W. E. et al., 2007) (Di Fatta et al., 2006). For example, in part 'a' of Figure 6, the point (10, 60) on a series "using 25 traces per function" shows that faulty functions in approximately 60% of the actual failed traces were diagnosed by reviewing 10% or less of the code (functions) for the Space program. The straight line at the end of the series, when there are no more points visible on the series, means that there are no predictions by the decision trees for the remaining 15% traces.

It is possible that the approach lists two or more functions at the same rank, then the best case effort entails that the first function to be examined is faulty, and the worst case effort entails that the last function to be examined is faulty. For example, suppose there is one function listed at rank 1, and five functions listed at rank 2. The best case effort is that the faulty function is the second function to be examined (i.e., one at rank 1 and one at rank 2); whereas, the worst case is that the faulty function is the sixth to be examined.

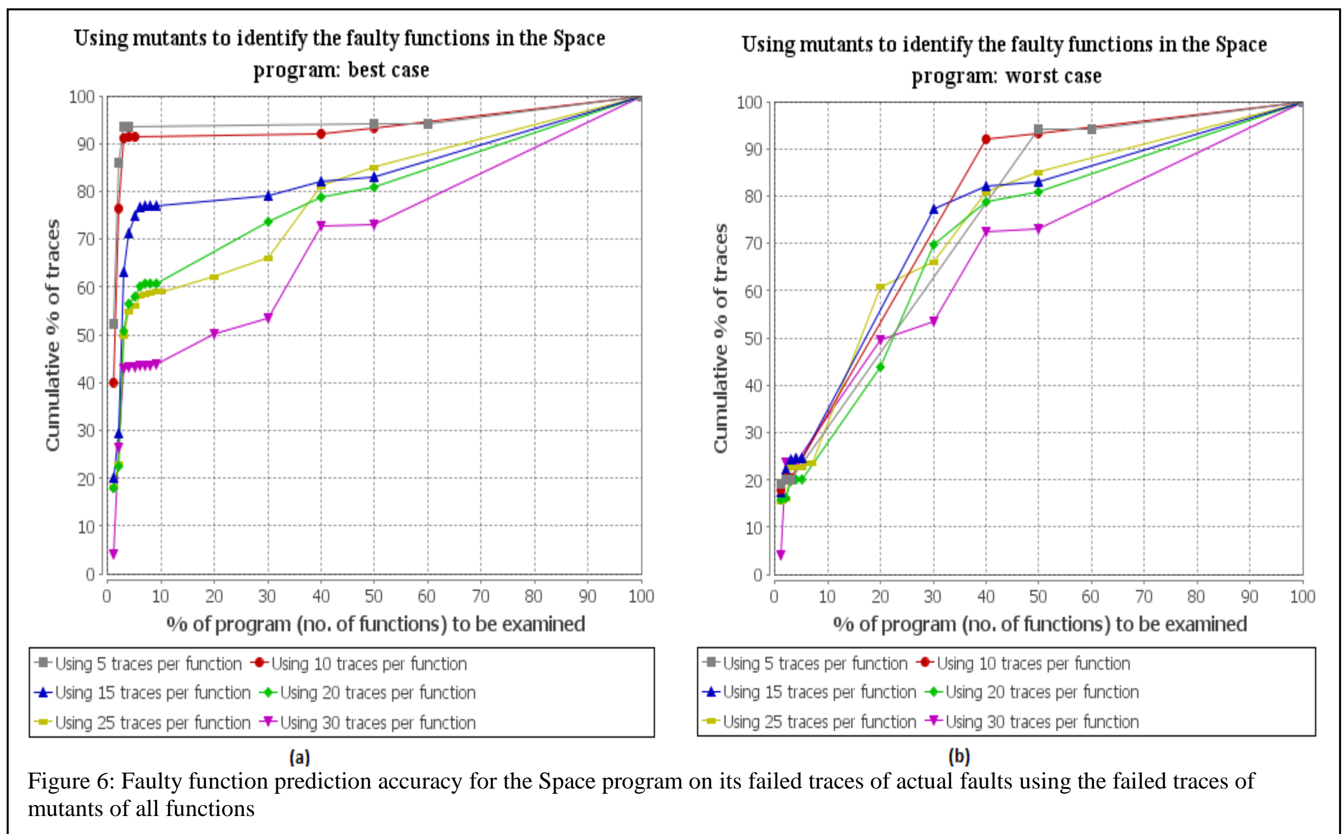

Figure 6: Faulty function prediction accuracy for the Space program on its failed traces of actual faults using the failed traces of mutants of all functions

In Figure 6, part 'a' shows the best case effort of the programmer with 5-30 mutant traces per function of the Space

program, and part ‘b’ of Figure 6 shows the worst case effort of the programmer using the same number of mutant traces.

It can be observed from Figure 6 (part ‘a’ and ‘b’) that when using fewer mutant traces per functions, the best case effort is higher than using a larger number of mutant traces per function, whereas, the worst case effort is lower. If the difference between the worst case and the best case is too high for a series then it means most of the functions are listed at the same rank, and the use of particular numbers of mutants per functions represented by that series is ineffective. This also means that using fewer mutant traces, the decision tree did not have sufficient information to

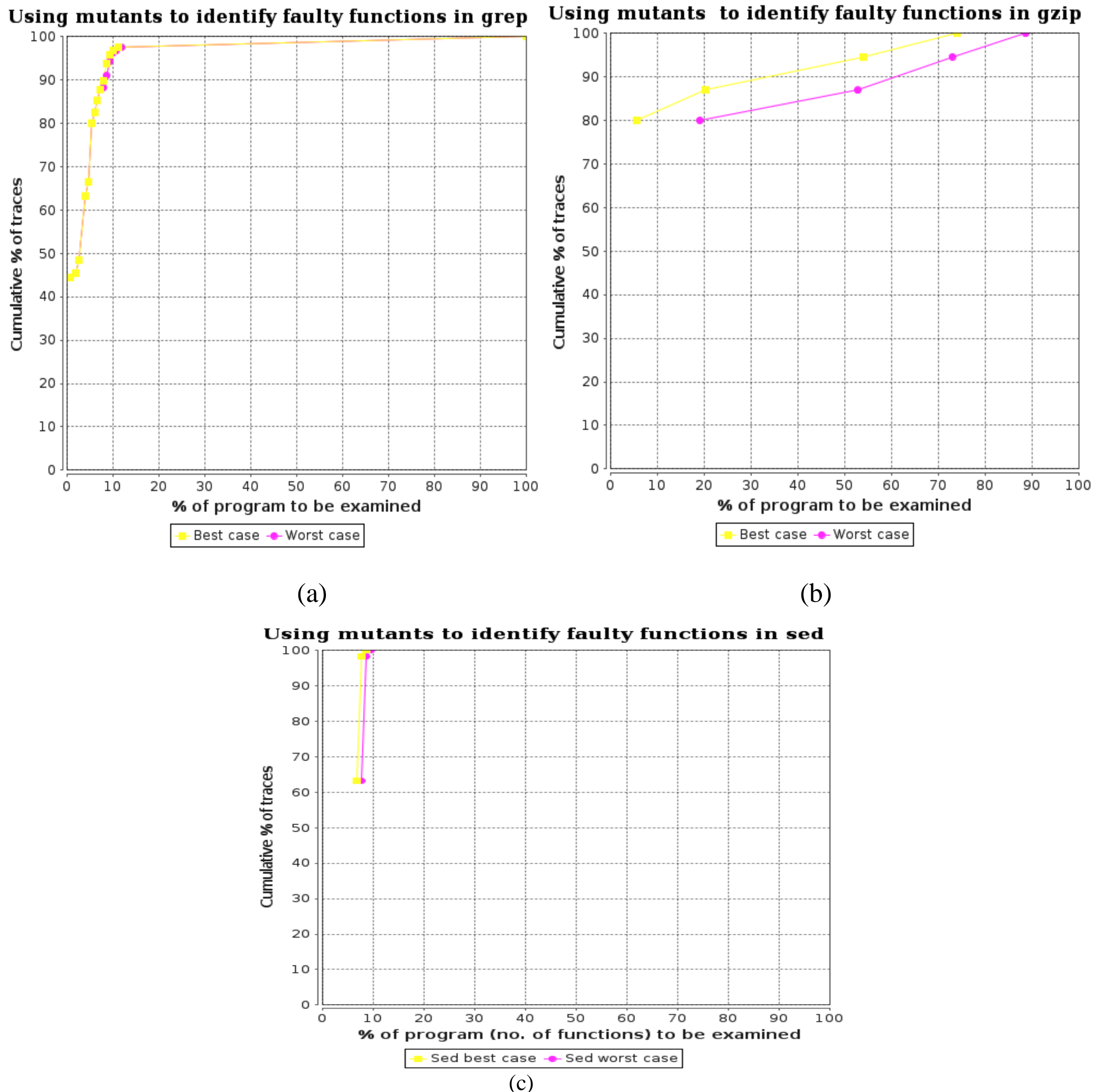

Figure 7: Accuracy of identification of faulty functions in the actual traces using mutant traces on the UNIX utilities

predict faulty functions in actual traces, and most of the suspected faulty functions were predicted with the same

probability.In the case of Figure 6, the use of 25 and 30 traces per functions series have a small gap between their worst and best cases, respectively, implying that there are fewer functions listed at the same rank for 25 and 30 mutant traces per function. In the case of 25 mutant traces per function, both the worst case effort and the best case effort are better than the 30 mutant traces per function. Thus, 25 mutant traces yield the best results for the Space program as we can identify faulty functions in 60% of the actual failed traces on the review of 20% of the code in both the best and worst cases.

The test suites in the Space program were more extensive than the ones would usually be produced in practice. That is, approximately 13,000 test cases for approximately 6000 lines of code. This means almost all of the functions and control flow paths were exercised by the test cases. However, in the UNIX utilities (i.e., Grep, Gzip and Sed), although the sizes of the programs were almost the same as the Space program, the test cases were not as extensive as the Space program. The test suites of the UNIX utilities mimic the real world scenario closely. On the other hand, recall from Section 2.1, if the test suite does not exercise all the paths then this shows the weakness of the test suite in detecting faults, and, eventually, will leave many mutants live or equivalent. The test cases of the UNIX utilities provided average (approx.) 70% functions coverage; whereas, the test cases for the Space program provided approximately 85% functions coverage.

We performed experiments on Grep, Gzip and Sed programs again by using 5, 10, 15, 20, 25 and 30 mutant traces per function. Figure 7, part 'a', 'b' and 'c', shows the results on Grep, Gzip and Sed programs, respectively. In these three programs, the best diagnosis accuracy was obtained by using 15 mutant traces per functions. In the case of 5-10 mutant traces per function and 20-30 mutant traces per function the accuracy of diagnosing faulty functions was lower than 15 mutant traces per function. In other words, the accuracy of diagnosis of faulty functions in real traces increases up till 15 mutant traces per function and then started to decrease. Figure 7 only shows results with 15 mutant traces per function to avoid cluttering in this paper.

Figure 7 shows the accuracy of the identification of faulty functions on all of the three UNIX utilities in the same manner as in Figure 6 (Space program). We also show the best case and the worst case accuracy for the UNIX utilities in Figure 7 for 15 mutant traces per function. It can be again observed from Figure 7 that the use of 15 mutant traces per function resulted in almost identical best and worst case effort for the UNIX utilities in Figure 7. This means there were fewer or almost zero functions listed at the same rank for 15 mutant traces per function. Overall, the accuracy of identification of faulty functions on the UNIX utilities in Figure 7 is quite high if compared to the Space program in Figure 6. For example, part 'a' of Figure 7 shows that faulty functions in 80% of the failed traces were diagnosed correctly in the Grep program by reviewing only 5% or lesser code (functions) when we train the decision trees on 15 mutant traces per function. Similar results can also be observed for the Sed and Gzip program.

The results on the UNIX utilities are better than the Space program because the test suites were not as exhaustive as in the case of the Space program. This resulted into fewer failed test cases due to mutants, hence, lesser overlapping execution paths for a faulty function and lesser noise for the decision trees. For example, in the case of

the Sed program, we were able to collect failed traces for only 37 functions out of 143 total functions. Similarly, we collected mutant traces for 79 functions out of total 89 of the Gzip, and 68 out of 149 for the Grep program. In the case of the Space program, which has a very large test suite, mutant traces for 116 functions were collected out of total 136 functions. The test cases of the UNIX utilities mimic the real world scenario, where test suites do not exhaustively cover the source code and gives priority to critical functionalities. It has been found in the literature that 20% of the code causes 80-100% of the faults (Gittens et al., 2005) (Ostrand et al., 2005) in a software system because a small percentage of the functionality is mostly used. Thus, using a test suite that tests the critical functionality of a software system we can generate mutant traces of critical functions and achieve higher prediction accuracy.

This implies that less extensive test suites would result in a better accuracy (as in Figure 7) than the extensive test suites (as in Figure 6). However, note that the accuracy in the case of extensive test suite (Figure 6) is 60% on the review of 10% or less of the code (i.e., best case for 25 traces per function). This accuracy is also high and the difference between Figure 6 and Figure 7 is not too wide.

3.4 Case of Multiple Faults

Thus far, our experiments only showed results for single faulty functions with only single faults in them. In the case of UNIX utilities, it was possible for us to enable multiple faults and make multiple functions faulty simultaneously. We randomly enabled multiple faults simultaneously for the Grep, Gzip and Sed programs. For the Grep program, we enabled a different combination of faults to make different combinations of functions faulty simultaneously. The list of functions that we randomly made faulty for the Grep program is shown in Table 3. In the case of Sed and Gzip, we enabled all the faults, which resulted into 16 functions being faulty for Gzip and only one function being faulty for Sed. In the case of Sed program, there were six faults but all of them were faulty in the same function "do_subst". Failed traces for multiple faults were collected by running test cases in the same way as mentioned in Section 3.1.

We identified multiple faulty functions by training decision trees on mutant traces of single faulty functions (as earlier) and predicting the list of single faulty functions for the traces of multiple faulty functions. If one of the faulty functions out of few faulty functions of the actual trace is predicted correctly, then we consider that a faulty function has been diagnosed for that trace. We show in Figure 8 the accuracy of predictions of multiple faulty functions on subject programs. After fixing one faulty function, if a failure appears again then the process can be repeated to diagnose other faulty functions. It is also likely that if a programmer diagnoses (or fixes) one faulty function then other faulty functions would get diagnosed (or fixed) too.

Figure 8 shows the accuracy of diagnosis of multiple faulty functions in the Grep and Sed programs. In the case of Gzip, the diagnosis accuracy resulted into 100% accuracy on the review of 5% of the code in the best case and 100% accuracy on the review of 19% of the code in the worst case. We have not shown the graph for Gzip because a single point would be difficult to visualize. These results for multiple faulty functions are obtained by training decision trees on 15 mutant traces per function, similar to Section **Error! Reference source not found.**. These results show that

accuracy of prediction of multiple faulty functions is high and approximately 95-100% of the faulty functions are identified by reviewing approximately 10% or less of the code in the best case.

Table 3: List of multiple faulty functions the Grep program.

Group #	Function names
Group 1	gcompile; nlscan; grepfile; page_alloc
Group 2	fillbuf; grepdir
Group 3	fillbuf; init_syntax_once; page_alloc'
Group 4	reset; lex
Group 5	grepfile; lex; init_syntax_once
Group 6	prtext; closure
Group 7	fillbuf; prline; lex

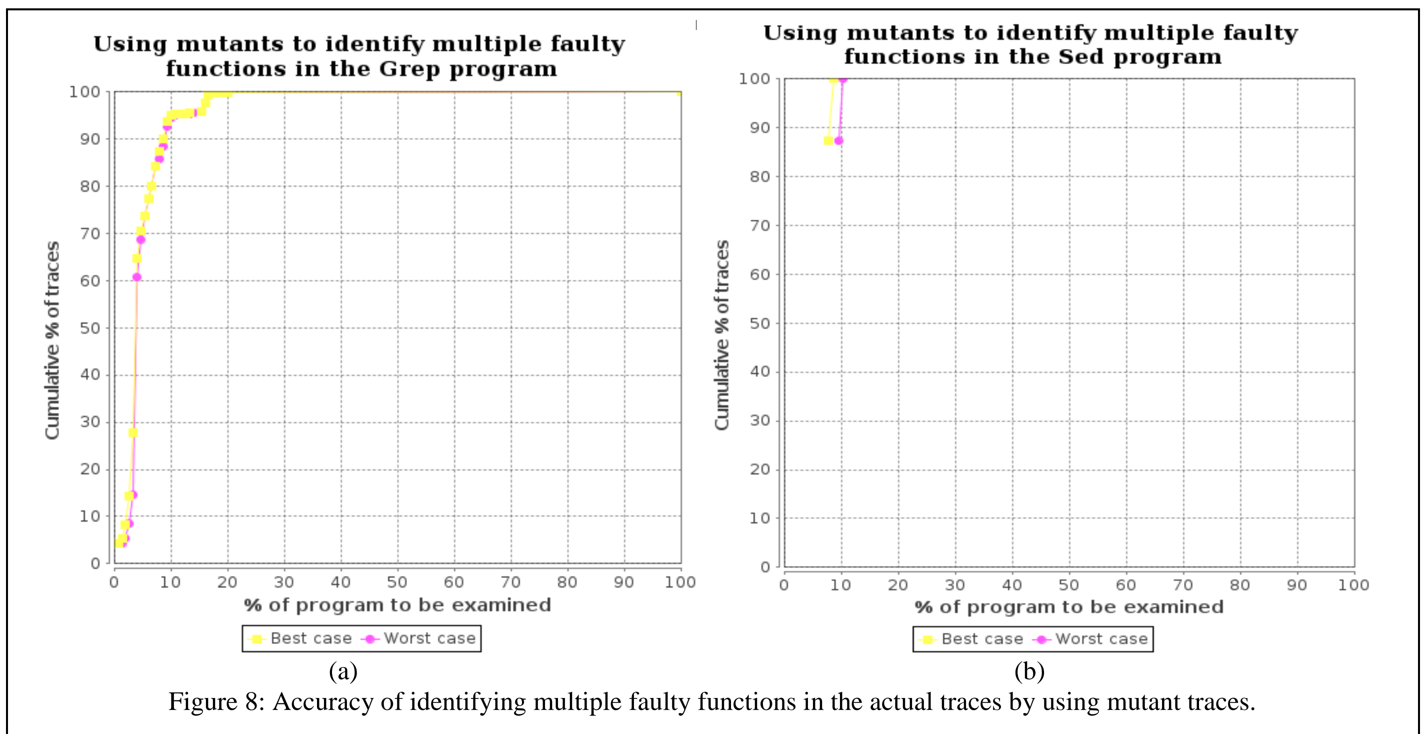

Figure 8: Accuracy of identifying multiple faulty functions in the actual traces by using mutant traces.

3.5 Comparison

A direct comparison of our technique does not exist with other techniques. The focus of our technique is to diagnose faults in every single trace of failure collected from the field during software maintenance. The focus of closely related techniques is to diagnose faults from a set of failure traces of the same fault usually collected during software testing. For example, (Jones & Harrold, 2005) (Chilimbi et al., 2009) (Di Fatta et al., 2006) (Dallmeier et

al., 2005) (Wong W. E. et al., 2007) (Wong et al., 2012) use a set of passing traces and a set of failing traces pertaining to a fault to diagnose faulty statements, classes, predicates, paths, and functions. Their focus is usually software testing where a set of passing-failing traces at different levels of granularities (e.g., statement, paths, branches, etc.) are easily available for fault localization. For the sake of comparison, we compare our technique with a well-known benchmark Tarantula technique (Jones & Harrold, 2005), and a recent technique based on RBF (Radial Basis Function) neural network (Wong et al., 2012). Wong et al. has already shown that RBF performs better than other fault localization techniques in the literature. Therefore, our comparison with their technique will also indirectly account for the comparison with other techniques. Both these techniques are proposed for statement-level traces, we have employed them on function call traces of the subject programs. Recall from Section 1 that we use function call traces because they are usually collected for failures in operational systems. In addition to these two techniques, we also compare our technique against a technique, called HOLMES, proposed specifically for traces collected from server side software in operation in the field (Chilimbi et al., 2009). HOLMES proposes to install instrumented software components on a need basis when a fault occurs and collect passing-failing traces, containing path coverage, for fault localization.

We evaluated these techniques by first collecting passing traces for our subject programs via execution of test cases on the non-faulty versions of the programs. Second, we enabled actual faults (not mutants) one by one and collected actual failed traces for failed test cases (same as described in Section 3.1 for our case study). Third, we removed passing traces of those test cases which failed when faults were enabled (to avoid overlapping test cases). Fourth, we evaluated the related techniques by having them to identify faulty functions in every single failed trace rather than a set of failing traces for every fault.

All three techniques generate ranking of paths or statements (functions in our case). Tarantula generates ranking by using a simple heuristic shown in Equation 2; where, F(e) represents the number of failed traces that executed an entity, P(e) represents number of passing traces that executed an entity, P represent total number of passing traces, and F shows total failing traces (which is always one in our case). The entity 'e' in our case is a function and functions with higher suspiciousness values are ranked first in the list.

Similarly, HOLMES uses a combination of different measures to calculate the suspiciousness of an entity. First HOLMES measure the "Sensitivity" of an entity by using "log(F(e))/log(F)", which is always zero when there is only one fail trace. Second, HOLMES distinguishes between observed and executed entities, and measures a score (called "Context") for observed entities. In our case, we have only executed entities, therefore this also results in zero. Third, HOLMES measures "Increase" of an entity by using "F(e) / (P(e) +F(e))" and finally the suspiciousness (called "Importance") is measured using Equation 3. However, Equation 3 always results in 0 when there is only one failed trace, showing that HOLMES require multiple failed traces of the same fault for fault localization and is not applicable for only a failed trace.

$$suspiciousness(e) = \frac{F(e)/F}{\left(\frac{P(e)}{P}\right) + \left(\frac{F(e)}{F}\right)}$$

Equation 2: Suspiciousness of an entity for Tarantula

$$suspiciousnes(e) = \frac{2}{\frac{1}{Sensitivity(e)} + \frac{1}{Increase(e)}}$$

Equation 3: Suspiciousness of an entity for HOLMES

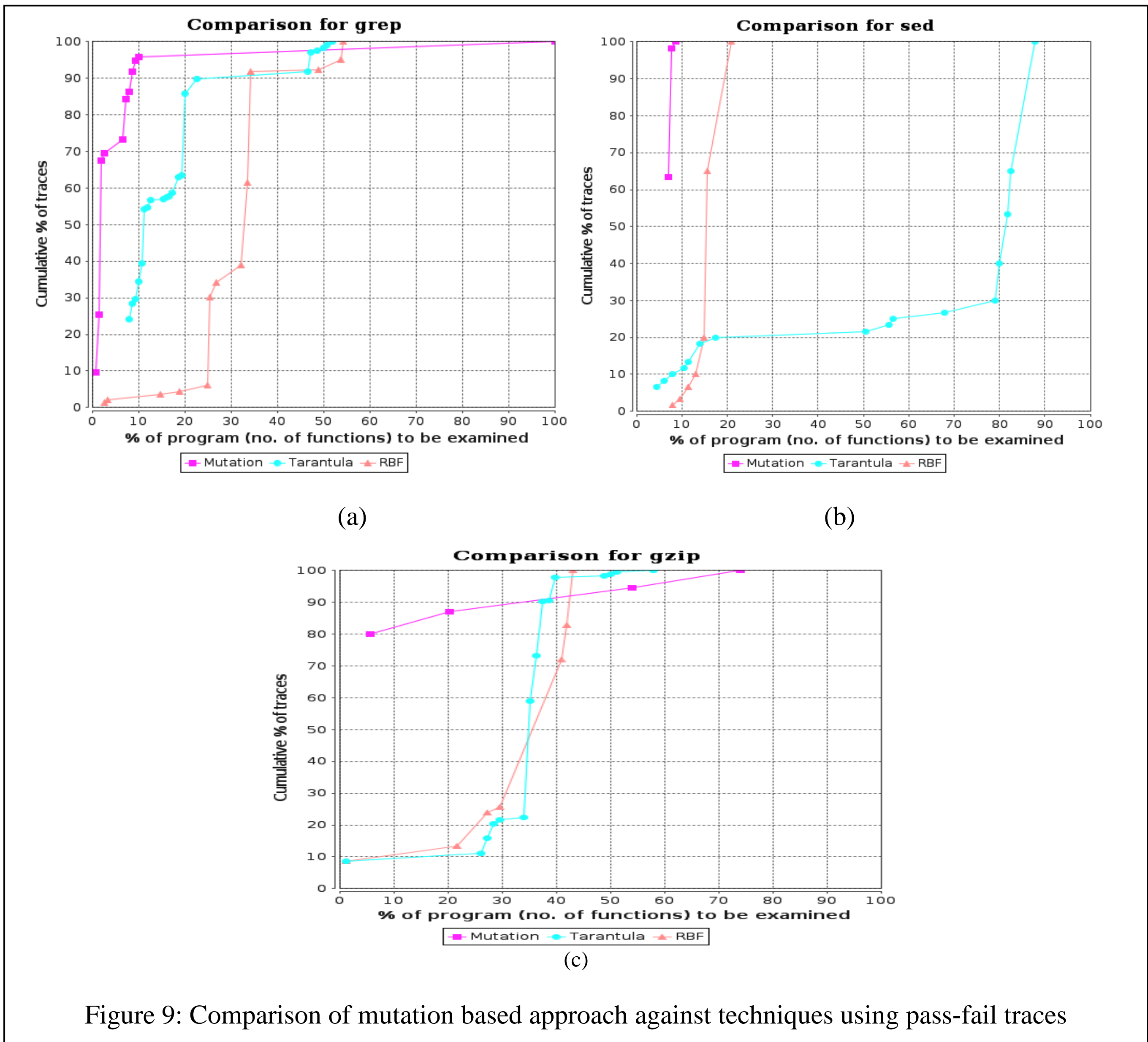

Figure 9: Comparison of mutation based approach against techniques using pass-fail traces

In the case of RBF neural network technique, the suspiciousness of an entity is measured using the output of the neural network (Wong et al., 2012). An RBF neural network is trained on a set of passing-failing traces and every

executed statement is provided one by one to the trained network to measure its suspiciousness. We implemented the same method as described by Wong et al. for a set of passing traces and a failing trace containing function calls. The results for Tarantula, RBF and our technique (mutation) are shown Figure 9. We show the best case accuracies for all the techniques in Figure 9. The results clearly show that our technique diagnoses faulty functions in majority (80-100%) of the failed traces by reviewing less than 10% of the program (functions); whereas, other techniques diagnosed 80% of the failed traces by reviewing 20%-85% of the program. In the case of Space program, Tarantula and RBF listed most of the functions on the same rank, and resulted in approximately 100% accuracy of diagnosis of faulty functions on approximate 1% of the code review in the best case and approximately 80% of the code review in the worst case. This renders the comparison useless because of the wide gap between the worst and the best case. The results of our mutation approach on the Space program are already shown in Figure 6.

Moreover RBF technique requires training neural network for every failed trace which is quite time consuming when the program is large or number of traces are many. For example evaluation of RBF based technique took a day on large number of traces of the Space program, while we implemented it as an efficient Java class.

Other closely related techniques that focus on failures of systems in operations (software maintenance) use clustering algorithms on a historical collection of failed traces containing function calls (Brodie et al., 2005) (Lee & Iyer, 2000) (Podgurski et al., 2003). Each cluster is formed on the basis of similar crashing reason and developers then explore the traces in clusters to diagnose the location of faults. The focus is mostly on recurrent field failures. These and other similar techniques are discussed in detail in Section 4. Our technique is different from these techniques as it focuses on directly pointing out faulty functions in the traces of unknown field failures, and in fact can complement clustering based techniques by pointing out the faulty function in the traces of clusters.

3.6 Repetition of Experiments

The technique proposed in this paper employed random strategies when selecting mutants and traces. In order to ascertain the results of the technique were not significantly affected by random selection of mutants and traces, we repeated all the experiments. We followed the same procedure as in earlier experiments. First, we executed mutation procedure to randomly select three mutants for each function of the subject program. The percentages of different types of mutants are shown in Table 4. Second, we collected a maximum of ten mutant (failing) traces for each function. Third, we trained the decision trees on the mutant traces and tested them on the actual traces. The results are shown in Figure 10. We only show the best results for different randomly selected traces for each of the subject programs to avoid cluttering, and show both the best case and worst case efforts. The best results were obtained by using 30 traces per function for the Grep program, 15 traces per function for the Sed program, 10 traces per function for the Gzip program, and 30 traces per function for the Space program. Due to random nature of experiments, the results of Figure 10 are not exactly the same as the results of Figure 6 and Figure 7. However, the results are quite similar and show good accuracy of diagnosis of faulty functions. We also measured the mean accuracy and standard deviation of diagnosis of faulty function for the results obtained in earlier experiments and in this repetition of experiments; shown in Table 5.

Table 4: Percentages of different randomly selected mutants in four programs

Mutant Type	Grep	Gzip	Sed	Space
Replace Constant	31.8%	52.24%	33.91%	50.75%
Negate Decision	8.79%	4.49%	8.91%	6.53%
Replace Operator	34.91%	22.04%	23.27%	13.81%
Delete Statement	24.46%	21.22%	33.91%	28.89%
Percentage of randomly selected mutants out of total mutants				
	3.19%	2.40%	5.77%	3.24%

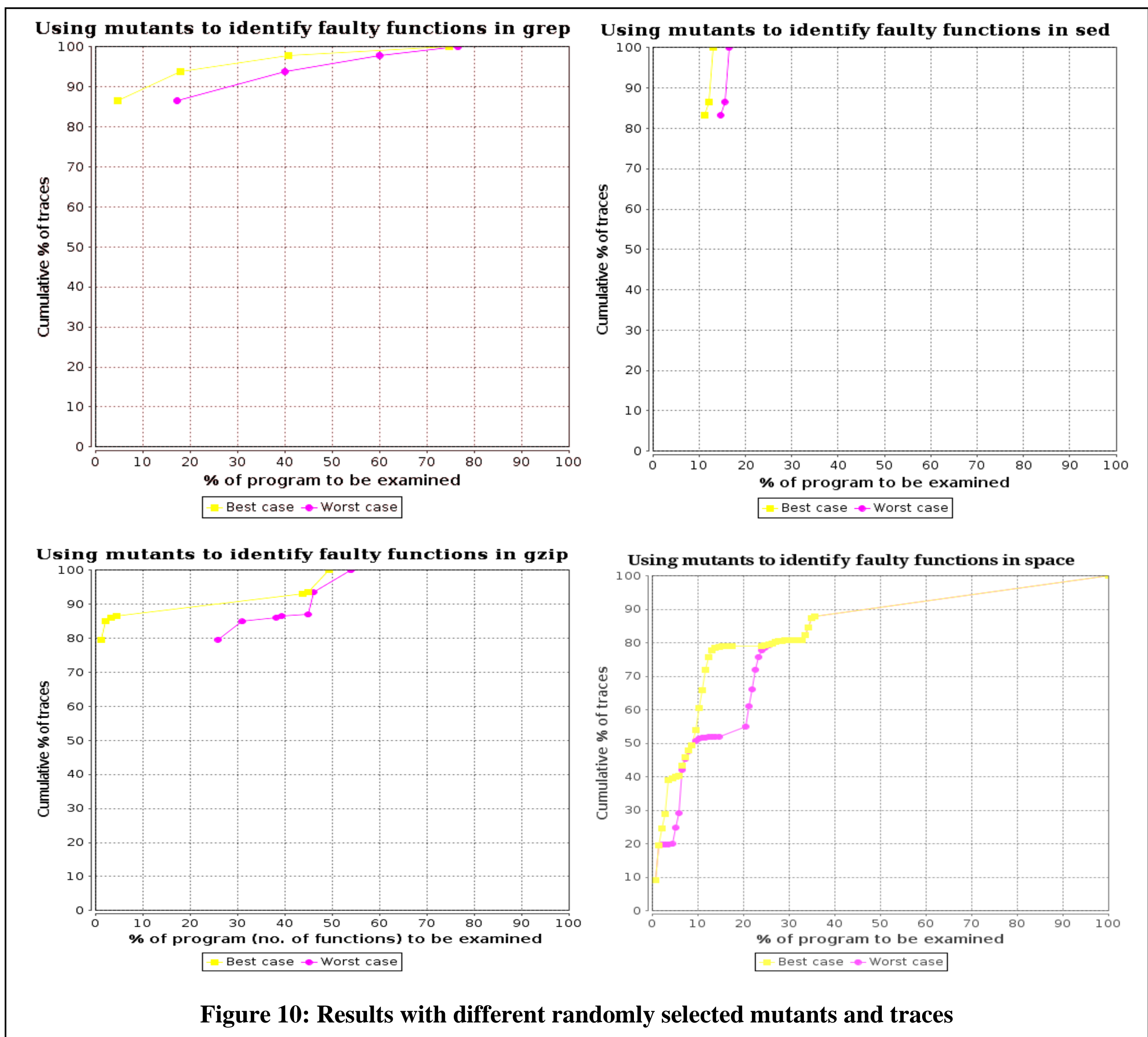

Figure 10: Results with different randomly selected mutants and traces

Table 5 shows the mean accuracy and standard deviations for the best case efforts. We measured the mean accuracy (and standard deviation) of diagnosis of faulty functions in traces for different program review percentages. We divided the program review percentages into segments of 5 up to the 20th percent and then into segments of 10 up to the 100th percent. The smaller segments for up to 20th percent are selected because a higher percentage of program review for an automated technique would deter developers from using it. In Table 5, the standard deviation is high for UNIX utilities for a program review of less than 5%; however, the standard deviation is low afterwards and the mean accuracy approaches to approximately 90%. These results show that there may be minor variations in results but the mutants have the potential to diagnose faulty functions in actual traces.

Table 5: Mean accuracy of diagnosis of faulty functions and standard deviations for different selections of mutants and traces

Program Review	**Grep**		**Sed**		**Gzip**		**Space**	
	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
1	22.50	31.82	0.00	0.00	40.50	57.28	13.50	6.36
5	78.50	12.02	0.00	0.00	84.00	5.66	48.00	11.31
10	92.50	3.54	90.50	13.44	84.00	5.66	60.00	0.00
15	93.50	2.12	100.00	0.00	84.00	5.66	69.50	13.44
20	94.50	0.71	100.00	0.00	88.00	0.00	70.50	12.02
30	95.50	0.71	100.00	0.00	88.00	0.00	73.50	10.61
40	96.50	2.12	100.00	0.00	88.00	0.00	85.00	5.66
50	97.00	2.83	100.00	0.00	94.00	8.49	86.00	4.24
60	97.00	2.83	100.00	0.00	96.50	4.95	86.00	4.24
70	97.00	2.83	100.00	0.00	96.50	4.95	86.00	4.24
80	97.50	3.54	100.00	0.00	100.00	0.00	86.00	4.24
90	97.50	3.54	100.00	0.00	100.00	0.00	86.00	4.24
100	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00

3.7 Discussion

The results in Figure 6, Figure 7 and Figure 8 show that by using mutant traces of every function, the faulty functions in the actual failed traces are not entirely distinguishable. This is because 100% or closer accuracy was not obtained on reviewing 1% (or less) of the code (i.e., faulty functions in all the actual failed traces are diagnosed correctly on the review of first function in the ranked list). This also implies that function call traces of faults in a function overlap with the function call traces of faults in some other functions.

In fact, the results show that there are M groups of closely related functions, and functions in each group make

calls to each other or call the same functions regularly. When a fault occurs in one of the functions of a group (e.g., M_i) then function calls overlap. When a fault occurs in a function in another group M_k, then there are fewer overlapping function calls with the function calls of faults in groups other than M_k. The reason is that if function calls of all the functions had overlapped, then we would have had to review about 100% (or closer to 100%) of the program to identify the faulty functions in any trace. We could still find faulty functions in 60% of the failed traces of the Space program (see Figure 6) by looking at 10% percent of the program (functions), when using 25 mutant trace per function. Similarly, we were able to find faulty functions in 80-90% of the traces of the UNIX utilities by reviewing 10% of the code (see Figure 7) when using 15 mutant traces per function. These results and the comparison section show that mutants have a better potential to detect actual faults (faulty functions) than the related techniques.

Thus, we come up with two conclusions:

- A group 'Mi' of related functions has similar function call traces when a fault occurs in any of the functions of that group 'Mi', and function-call traces of 'Mi' are different from the function-call traces of another group of function 'Mk' if a fault occurs in the functions of group 'Mk'; where i,k= 1-n and $i \neq k$ and $Mi \subset N$ and $Mk \subset N$ and N={functions | functions ∈ program}. This answers our research question (Q2) that artificial faults and actual faults in functions occur with similar function call traces.
- Due to the similarity of function call traces as mentioned in point 'a', function call traces of mutants can be used to identify faulty functions in function call traces of actual faults. This answers our research question (Q1) that mutants can be used to diagnose actual faults.

A limitation of our approach is that it requires the generation of mutants and mutant traces. This can be tedious and time consuming, particularly if test cases are not automatically executed. If the process of generation of mutant traces is automated then the effort require in mutant trace collection can be reduced. To make the approach scalable, we also tried to overcome this limitation by randomly sampling up to three mutants which reduced the number of mutants by 95% (see Section 3.2). The fact that we use one-against-all classification (i.e., only two types of faulty functions per decision tree) and generate a ranking of faulty functions after they are predicted from the decision trees for a trace, faulty functions in programs with large number of functions can be predicted in the same manner as in our case studies. Due to one-against-all approach, the large number of functions would not reduce the accuracy of prediction of faulty functions from each decision tree because there will always be only two classes (faulty functions) in each decision tree. In case, if there is an effect on the accuracy, the use of ranking will facilitate in negating the effect on accuracy by ordering the predicted faulty functions. We have seen this with different number of functions in our case studies. The time to train decision trees take only few minutes, the only time of significance is generation of mutant traces.

Another approach that can be used to reduce the mutants and mutant traces is to generate mutants of only critical

functions. Usually, in commercial applications, the test suites do not exhaustively cover 100% of the functions of a program and target critical functionality more than other functionality. Our results show that when test suites are not extensive then the use of mutants will yield higher prediction accuracy. To save time in execution of test cases, mutants can be generated for critical functions, the functions that test cases target instead of the whole source code. Anyways, mutants for parts of the source which test cases do not cover are not going to result in mutant (failed) traces. Critical functions can also be identified with developers' knowledge and software bug history. In addition, critical functions can be identified by focusing on 20% of the components that cause 80% of the faults in software applications—80-20 Pareto rule for software faults (Gittens et al., 2005) (Ostrand et al., 2005). Further research on this is currently beyond the scope of this paper. Nonetheless, this technique can be extended to large programs if the process of generation of mutant traces is automated.

4. Related Work

This section describes closely related techniques grouped into three categories: fault localization techniques for in-house faults, fault localization techniques for field failures, and the techniques using mutation.

4.1 Fault Localization Techniques for In-house Faults

Many researchers have proposed techniques for diagnosing fault locations by using the difference between passing traces and failing traces pertaining to a fault, such as diagnosing faulty statements using statement-level traces (e.g., (Agrawal et al., 1995) (Renieres & Reiss, 2003) (Cleve & Zeller, 2005) (Wong & Qi, 2006) (Jones & Harrold, 2005) (Zhang et al., 2009) (Wong W. E. et al., 2007) (Santelices et al., 2009) (Wong et al., 2008) (Wong et al., 2012)), diagnosing faulty functions (e.g., (Di Fatta et al., 2006)) and faulty classes (e.g., (Dallmeier et al., 2005)) using function call traces, and identifying assertions (e.g., null pointer checks) using statistical debugging (e.g., (Liblit et al., 2005) (Zheng et al., 2004) (Liu et al., 2005)). Mostly these techniques compare a set of passing and a set of failing traces, and produce a rank list of respective artifacts (e.g., statements, functions, etc.) for a particular fault. In the case of multiple faults, they need to group traces due to the same faults (e.g., by clustering) to reduce a multi-bug problem to a set of single bug problems (Wong et al., 2012). Dallmeier et al. (Dallmeier et al., 2005) has explicitly shown the execution of their approach on one failing trace and a set of passing traces to predict faulty classes. Similarly, execution slicing techniques (e.g., (Agrawal et al., 1995) (Renieres & Reiss, 2003) (Cleve & Zeller, 2005) (Wong & Qi, 2006)) provide a difference/intersection of a failing trace with a passing trace and continue the comparison with other passing traces until the fault is found. Another technique that uses only a failing trace to detect faulty statements is proposed by Jose and Majumdar (Jose & Majumdar, 2011). They compute a maximal set of unchangeable statements for a program to remain correct on a given input. The complement of this set of statements form a diagnosis set. However, their work does not allow evaluation of multiple inputs simultaneously, which is a significant limitation for deployed systems. In fact, the inputs may not be known at all for deployed systems and using multiple simultaneous inputs will also not work.

These techniques are mostly suitable for in-house testing for several reasons. In order to operate majority of them

need to have a set of failing traces due to the same fault but this is not known for hundreds of field traces that arrive regularly and they can have multiple faults too. Different customers' usages can also cause many normal execution paths that are not observed in the passing executions of in-house testing and they may be considered failing traces. In such cases passing traces should be collected from field as shown by (Chilimbi et al., 2009) (Liu & Han, 2006); however, it is not always feasible to collect many traces from customers due to tracing overhead. These techniques can still be applied by reproducing the same faults on test machines but reproducing different many faults can be time consuming or difficult. In our approach, we predicted faulty functions in every single actual failed trace by using a set of mutant traces and it does not employ passing traces. A disadvantage is the time to generate mutant traces but it is only required to be done once for a release. A comparison with passing-failing traces techniques is shown in Section 3.5. In fact prior techniques can be benefitted by using mutant traces along with an original failing trace as a set of failing traces for a particular region of code. They can then apply their methods on that set of traces and repeat the process for different regions of code until the fault is found. They can use mutants to diagnose multiple faults too in this way without using additional clustering techniques.

4.2 Fault Localization Techniques for Field Failures

Podgurski et al. (Podgurski et al., 2003) propose a fault diagnose approach that forms clusters of execution traces of field failures based on common faulty source files. The granularity in the Podgurski et al.'s approach is a faulty file where majority of the clusters encompass failed traces with different files. In our approach, we predict faulty functions for a trace. Their clustering technique requires a historical collection of failed traces and mutant traces can help in preparing historical traces when they are not available.

Liu and Han (Liu & Han, 2006) cluster failing runs according to the rank list of assertions obtained using the statistical debugging tool SOBER (Liu et al., 2005). They propose to collect passing and failing traces from the field to predict fault locations (assertions). They insert light weight assertions (i.e., check points) in code, collect traces and if a fault is not found they insert new assertions. Their work also suffers from the limitations mentioned in Section 4.1. Mutant traces can be helpful in this work too as mention in Section 4.1.

Another statistical debugging tool, HOLMES (Chilimbi et al., 2009), identifies suspicious fault locations from the traces containing executed paths of deployed software. This tool can only be applied to server side applications, because Chilimbi et al. (Chilimbi et al., 2009) have to redeploy software components with instrumentation of selected functions to collect passing traces and failing traces pertaining to a fault. The use of passing-failing traces is also a limitation, as discussed in Section 4.1. Also, redeployment of instrumented software components on servers may not be feasible in some cases. Our approach does not incur any additional overhead on client machines other than the collection of a trace, and as mentioned before predict a fault in a single trace not in a collection of failed traces. Again mutant traces can facilitate HOLMES in collecting a set of failed traces for a particular region of code (see Section 4.1).

Elbaum et al. (Elbaum et al., 2007) propose a technique that compares the field failure traces (function sequences) to in-house passing traces in order to anticipate the occurrence of a failure such that data collection for a fault in the

field can be started. Bowring et al. (Bowring et al., 2004) and Haran et al. (Haran et al., 2007) develop a technique based on the Markov model (Bowring et al., 2004) and the decision tree (Haran et al., 2007) to characterize statement or branch level executions as being passing or failing runs. This is because sometimes it is not known from the field whether the trace is passing or failing. These techniques complement fault localization techniques by separating passing-failing traces and they also complement our work.

Brodie et al. (Brodie et al., 2005) use string matching techniques to group one function call trace of a crash with other groups of function call traces for different known crashes. Lee and Iyer (Lee & Iyer, 2000) propose a technique to classify a rediscover crashing failure by literal matching of its function call trace with already known failure traces. They consider a variety of heuristics to match several function call paths followed by the same fault. Murtaza et al. (Murtaza et al., 2010) propose a decision tree based method to discover faulty functions in crashing and non-crashing field failures. These techniques require a historical collection of failed traces. All of these techniques can benefit from the use of mutation traces to prepare historical failed traces for fault localization.

4.3 Techniques Using Mutation

Mutants are automatically generated variants (faulty version) of a program obtained by applying mutation operators to the source code (Andrews et al., 2005) (Offutt & Untch, 2001). For example, mutation operators include changing an arithmetic operator with another in a statement, negating a decision in if or while statements, or deleting a statement. Moreover, mutation analysis is used as a measure of quality of test cases (Offutt & Untch, 2001). Mutation analysis has mostly been used to measure, enhance, and compare the effectiveness of testing strategies. For example, Mayer and Schneckenburger (Mayer & Schneckenburger., 2006) use mutation analysis to compare the effectiveness of all the adaptive random testing techniques in detecting failures. Similarly, Do and Rothermel (Do & Rothermel, 2006) employ mutation analysis to evaluate the ability of several test case prioritization techniques in improving the fault detection rate on Java programs. Test case prioritization has been used to reduce the cost of regression testing by running important test cases first—i.e., test cases which have more chances of detecting faults (Do & Rothermel, 2006). Andrews et al. (Andrews J. et al., 2006) use mutation analysis to compare the cost-effectiveness of data and control flow coverage criteria (i.e., Block, Decision, C-Use, and P-Use). Andrews et al. have also determined using empirical studies that mutation faults are similar to real faults (Andrews et al., 2005) but different from hand seeded-faults (Andrews et al., 2005). Hao et al. (Hao et al., 2005) use mutants as faulty versions of a program to evaluate fault localization techniques. A recent survey about fundamentals, advances, trends, tools, and challenges in mutation testing can provide curious readers further details on mutation testing (Jia & Harman, 2011).

Papadakis and Traon (Papadakis & Traon, 2012) use mutants to diagnose faulty statements from statement level traces. They use a set of passing traces and a set of failing traces (similar to techniques in Section 4.1) for a fault. They generate a ranking of faulty statements by first generating many mutants of one statement and running test cases on each mutated statement. Nica et al. (Nica et al., 2010) propose a technique to reduce the number of suspicious statements generated by automatic (in-house) fault localization for passing-failing test cases. They select

those suspicious statements whose mutants could lead to passing of test cases and further filter out suspicious statements by determining distinguishing test cases. They show improvement in overall results. Zhang et al. (Zhang et al., 2013) also propose the use of mutation to improve the ranking schemes for suspicious statements generated by automated (in-house) fault localization techniques for passing-failing test cases. They actually identify code coverage of failing test cases, get mutants for the same area of code, execute test cases on those mutants, measure how many test cases pass and fail on those mutants compared to actual pass and fail test cases, and then generate a new ranking of statements. They use similar measures as other fault localization techniques for ranking and show improvement in results for most of the cases. Debroy and Wong propose a technique that automatically suggests fixes using mutation for the ranked list of faulty statements generated using the Tarantula (Jones & Harrold, 2005) fault localization technique (Debroy & Wong, 2010).

In contrast, the focus of this paper is on using the faults generated from mutation to diagnose the origin of actual faults from function call traces of deployed systems. We focus on diagnosing faults in every single failed trace rather than the collection of failed traces (see Section 3.5 for comparison) because it is not known whether a failed trace in deployed system belong to the same or different fault. In our earlier work (Murtaza et al., 2011), we have proposed the combined use of mutant traces and historical traces of actual faults of prior releases for the diagnosis of faults in the filed traces. Since historical failure traces are not easily available in many organizations and there could be substantial changes in the new releases, we decided to ignore historical traces and to only use mutant traces in this paper. We have also investigated in this paper, the relation between mutant and actual faults in the context of diagnosis of faults in function call traces from the field (see the research question (Q2)). In addition, this paper addresses several empirical aspects of the process of mutation for fault diagnosis, such as: the effect of test suite coverage on mutant based fault diagnosis; the use of different number of mutant traces; and effectiveness against pass-fail traces based fault diagnosis techniques. This paper also uses additional programs not used in our earlier study. In short, the use of only mutation for diagnosis of faults in failures of deployed systems is a novel aspect and has not been discussed before.

5. Threats to Validity

In this section, we describe certain threats to the validity of the research results obtained through our employed research process. We classify threats into four groups: conclusion validity, internal validity, construct validity, and external validity (Wohlin et al., 2000).

A threat to conclusion validity belongs to random variations in mutant traces. We randomly chose 3 mutants per function, but on some mutants in functions, test cases did not fail. It is possible that different selected mutants for the same function might result in failing test cases and variations in accuracy for some programs. A variation could occur due to different function call paths that faults could take. However, this threat is mitigated by the fact that we used up to 30 mutant traces per faulty function, repeated experiments on 4 real world programs, and collected different faulty paths for a function by selecting three different mutants per function. Also, we have seen in the results that

function call traces of different faults in related functions are similar and not different (see Section 3.7).

A threat to internal validity can exist in the implementation because an incorrect implementation can influence the output. For example, we wrote shell scripts to automate mutant trace collection, developed a Java program to automatically extract functions and their locations from C programs, modified the mutant tool, wrote a Java program to process traces, and implemented related techniques in Java. In our investigation, this threat is mitigated by manually investigating the outputs. Another threat exists when mutant traces are not present for a function and that function is actually faulty in the original traces. In such cases the faulty function will not be predicted by our approach.

A threat to construct validity exists in the use of only decision trees to detect actual faulty functions. Other fault localization techniques (e.g., (Brodie et al., 2005) (Lee & Iyer, 2000) (Podgurski et al., 2003) (Chilimbi et al., 2009) (Liu & Han, 2006) (Murtaza et al., 2010)) proposed in the literature can be used to demonstrate the use of mutation in fault diagnosis and validate the results. This can be a potential future work to show the comparison of different fault localization techniques on mutation. Currently, it is out of the scope of this paper. Another threat exists in the measurement of effort of programmer in terms of only functions. One can argue that the sizes of functions vary and the effort could be different. However, we did measure the effort in statement for the subject programs, and the efforts were similar to effort in functions as shown in this paper. Our results showed that sizes of functions in these public programs remain in proportion and did not vary from huge towards small.

A threat to external validity is that we have experimented only on medium-sized commercial programs. This technique has yet to be validated on very large industrial scale software application. A major issue could be in the scalability of the approach of using mutants on large systems. We have discussed this issue in Section 3.7.

6. Conclusions and Future Work

A number of fault diagnosis techniques proposed for deployed software focus on: the classification of field profiles into failed or successful executions (Bowring et al., 2004) (Haran et al., 2007), clustering field profiles (Liu & Han, 2006) (Podgurski et al., 2003), rediscovery of crashing faults (Brodie et al., 2005) (Lee & Iyer, 2000), statistical debugging (Chilimbi et al., 2009) (Liu & Han, 2006), and rediscovery of crashing and non-crashing faults (Murtaza et al., 2010). These techniques either require many passing-failing traces at the time of a failure or historical failed traces. Collecting many traces from deployed systems is not feasible due to the overhead incurred in trace collection and could be detrimental to business operations. Sometimes even reproducing fault in test machines could be time consuming and some faults might not be reproduced at all.

This paper therefore investigates the use of traces generated using automatic artificial faults with a particular focus on using mutation for the diagnosis of actual faults. It addresses the research questions: (a) Can the faults generated using mutants (artificial faults) be used to diagnose actual faults? (b) Do different faults in functions occur with similar function call traces? The question is important because it can help researchers and maintainers in the preparation of dataset for fault diagnosis and improve the software maintenance process. We investigated the answers

to these research questions by experimenting with the Space program, Flex, Grep, Gzip and Sed programs (Do et al., 2005).

The results shows that faulty functions in actual failed traces can be identified by using traces of mutants of these functions—i.e., mutants have a potential to diagnose actual faults. We have also found out that function call traces are similar for closely related functions. The high accuracy of fault diagnosis using mutants improves with the test suites that focus more on critical functions.

In future, we plan to investigate the use of mutants for fault diagnosis by experimenting on very large programs to address the issue of scalability on large programs. For our results to be generalizable to other programs, we need to conduct more experiments to ascertain these findings in unproven contexts.

We also need to study the problem of determining the proper number of mutants per function. In this paper, we showed that three mutants worked well for our systems. This, however, might not always be the case for other systems unless empirical studies show so. In addition, we expect that the number of mutants will vary depending on the function complexity. For example, a function with 100 lines should have more mutants than the one with less lines so as to cover all potential faults.

7. References

Agrawal, H., Horgan, J. R., London, S., & Wong, W. (1995, June). Fault Localization using Execution Slices and Dataflow Tests. *Proc. Int'l Soft. Symp. on Reliability Eng.* (pp. 143-151). Paris: IEEE.

Andrews, J. H., Briand, L. C., & Labiche, Y. (2005, May). Is mutation an appropriate tool for testing experiments? *Proc. of the 27th Intl. Conf. on Sof. Engg.* (pp. 402-411). St. Louis: ACM.

Andrews, J., Briand, L., Labiche, Y., & Namin, A. (2006, Aug.). Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Softw. Eng., 32*(8), 608-624.

Bowring, J., Rehg, J., & Harrold., M. (2004, July). Active Learning for Automatic Classification of Software Behavior. *SIGSOFT Soft. Eng. Notes, 29*(4), pp. 195-204.

Brodie, M., Sheng, M., Lohman, G., Mignet, L., Modani, N., Wilding, M., . . . Sohn, P. (2005, June). Quickly Finding Known Software Problems via Automated Symptom Matching. *Proc. of Second Int'l Conf. on Autonomic Computing* (pp. 101-110). Seattle: IEEE CS.

Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., & Vaswani, K. (2009, May). HOLMES: Effective Statistical Debugging via Efficient Path Profiling. *Proc. of 31st Intl. Conf. on Soft. Eng.* (pp. 34-44). Vancouver: IEEE CS.

Cleve, H., & Zeller, A. (2005). Locating causes of program failures. *Proc. of the 27th Intl. Conf. on Softw. Engg.* (pp. 342-351). St. Louis, USA: ACM.

Dallmeier, V., Lindig, C., & Zeller, A. (2005, Aug.). Lightweight Defect Localization for Java. *Proc. ECOOP 05- Object Oriented programming* (pp. 528-550). Glasgow: LNCS Springer.

Debroy, V., & Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. *Intl. Conf on Software Testing, Verification and Validation* (pp. 65-74). IEEE.

Devillard, N., & Chudnovsky, V. (2004, March). Retrieved March 2008, from Etrace--Runtime Tracing Tool: http://ndevilla.free.fr/etrace/

Di Fatta, G., Leue, S., & Stegantova, E. (2006, Nov.). Discriminative Pattern Mining in Software Fault Detection. *Proc. of 3rd Int'l Workshop on Soft. Quality Assurance* (pp. 62-69). Oregon: ACM.

Do, H., & Rothermel, G. (2006, Sep.). On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Soft. Engg., 32*(9), 733-752.

Do, H., Elbaum, S. G., & Rothermel, G. (2005, Oct.). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Journal of Empirical Soft. Eng., 10*(4), 405-435.

Elbaum, S., Kanduri, S., & Andrews, A. (2007, Oct.). Trace anomalies as precursors of field failures: an empirical study. *Journal of Empirical Soft. Engg., 12*(5), 447-469.

Gittens, M., Kim, Y., Godwin, & D. (2005, July). The vital few versus the trivial many: Examining the pareto principle for software. *Proc. of 29th Int'l Computer Software and Applications Conf.* (pp. 179-185). Edinburgh: IEEE CS.

Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., & Sun, J. (2005, Nov.). A Similarity-aware Approach to Testing Based Fault Localization. *Proc. of 20th IEEE/ACM Intl. Conf. on Automated Soft. Engg.* (pp. 291-294). CA: ACM.

Haran, M., Karr, A., Last, M., Orso, A., Porter, A., Sanil, A., & Fouche, S. (2007, May). Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks. *IEEE Trans. on Soft. Engg., 33*(5), 287-304.

Hare, D., & Julin, D. (2007, April). *The Support Authority: Interpreting a WebSphere Application Server trace file.* Retrieved from IBM WebSphere Developer Technical Journal: http://www.ibm.com/developerworks/websphere/techjournal/0704_supauth/0704_supauth.html

Jia, Y., & Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering, 37*(5), 649-678.

Jones, J. A., & Harrold, M. J. (2005, Nov.). Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *Proc. of 20th Int'l Conf. on Automated Soft Engg.* (pp. 273-282). Long Beach, CA: ACM.

Jose, M., & Majumdar, R. (2011). Cause Clue Clauses: Error Localization Using Maximum Satisfiability. *Conf. on Programming Language Design and Implementation* (pp. 437-446). ACM.

Lee, I., & Iyer, R. (2000, Feb.). Diagnosing Rediscovered Problems Using Symptoms. *IEEE Transactions on Soft. Engg., 26*(2), 113-127.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable Statistical Bug Isolation. *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (pp. 15-26). Chicago: ACM.

Liu, C., & Han, J. (2006, Nov.). Failure proximity: a fault localization-based approach. *Proc. of 14th SIGSOFT Sym. on Foundations of Software Engineering* (pp. 45-56). Portland: ACM.

Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005, Sep.). SOBER: Statistical Model-Based Bug Localization. *SIGSOFT Softw. Eng. Notes, 30*(5), 286-295.

Mayer, J., & Schneckenburger., C. (2006, Sep.). An Empirical Analysis and Comparison of Random Testing Techniques. *Proc. of Intl. Symp. on Empirical Soft. Engg.* (pp. 105-114). Rio de Janerio: ACM.

Melnyk, B. R. (2004, Sep). *DB2 Basics: An Introduction to the DB2 UDB Trace Facility.* Retrieved from DB2 Information Development, BM Canada Ltd.: http://www.ibm.com/developerworks/data/library/techarticle/dm-0409melnyk/index.html

Mozilla. (2013, Jan). Retrieved from Mozilla Crash Report: http://crash-stats.mozilla.com

Murtaza, S. S., Gittens, M., Li, Z., & Madhavji, N. (2010, Oct.). F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field. *Proc. 2010 conf. of the centre for advanced studies on collaborative research: meeting of minds (CASCON 2010)* (pp. 57-71). Toronto: ACM.

Murtaza, S. S., Madhavji, N., Gittens, M., & Li, Z. (2011). Diagnosing new faults using mutants and prior faults (NIER track). *Proc. 33rd Inl Conf on Soft Engg (ICSE '11)* (pp. 960-963). Honolulu: ACM.

Murtaza, S., A, R., Hamou-Lhadj, A., & Couture, M. (2012). On the Comparison of User-space and Kernel-space Traces in Identification of Software Anomalies. *Proc. of 16th Conference on Software Maintenance and Reengineering* (pp. 127-136). Hungary: IEEE.

Nica, M., Nica, S., & Wotawa, F. (2010). Does testing help to reduce the number of potentially faulty statements in debugging? *Testing--Practice and Research Techniques*, 88-103.

Offutt, A. J., & Untch, R. H. (2001). Mutation 2000: uniting the orthogonal. In W. E. Wong (Ed.), *Mutation Testing for the New Century* (pp. 34-44.). USA: Kluwer Academic Publishers.

Ostrand, T., Weyuker, E., & Bell, R. (2005, May). Predicting the location and number of faults in large software systems. *IEEE Trans. on Softw. Engg., 31*(4), 340-355.

Papadakis, M., & Traon, Y. (2012). Using Mutants to Locate "Unknown" Faults. *Intl Conf. on Software Testing* (pp. 691-700). Montreal: IEEE.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., & Wang, B. (2003, May). Automated Support for Classifying Software Failure Reports. *Proc. Intl. Conf. on Software Eng.* (pp. 465-475). Portland: IEEE CS.

Polat, K., & Güneş, S. (2009, Mar.). A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-Against-All Approach for Multi-Class Classification Problems. *Journal of Expert Syst. Appl., 36*(2), 1587-1592.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc.

Renieres, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. *Proc. of the 18th Intl. Conf. on Automated Softw. Engg.*, (pp. 30-39). Montreal.

Santelices, R., Jones, J. A., Yu, Y., & Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. *Proc. of the 31st Intl. Conf. on Soft. Engg.* (pp. 56-66). Vancouver: IEEE CS.

Ubuntu. (2013). Retrieved from Apport crash reporting: https://wiki.ubuntu.com/Apport

UWO, & IBM. (2008, Sep.). *Proprietary workshop on a large commercial software.*

WER. (2012, Feb). *Windows Error Reporting.* Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/windows/hardware/ gg487440.aspx

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques.* USA: Morgan Kaufmann Publisher.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., & Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction.* Norwell, USA: Kluwer Academic Publishers.

Wong, W. E., & Qi, Y. (2006, July). Effective Program Debugging Based On Execution Slices and Inter-Block Data Dependency. *J. Syst. Softw., 79*(7), 891-903.

Wong, W. E., Qi, Y., Zhao, L., & Cai, K. (2007, July). Effective Fault Localization using Code Coverage. *Proc. of 31st Int'l Conf. on Comp. Soft. & App. 1*, pp. 449-456. Beijing: IEEE CS.

Wong, W. E., Wei, T., Qi, Y., & Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. *Proc. of 1st Intl. Conf. on Software Testing, Verification and Validation* (pp. 42-51). Lillehammer, Norway: IEEE CS.

Wong, W., Debroy, V., Golden, R., Xiaofeng, X., & Thuraisingham, B. (2012, March). Effective Software Fault Localization Using an RBF Neural Network. *IEEE Transactions on Reliability, 61*(1), 149-169.

Zhang, L., Zhang, l., & Khurshid, S. (2013). Injecting Mechanical Faults to Localize Developer Faults for Evolving Software. *Proc. of ACM SIGPLAN Conf. on Object Oriented Programming Systemes, Languages and Applications.* Indianapolis, IN.

Zhang, Z., Chan, W. K., Tse, T. H., Jiang, B., & Wang, X. (2009, Aug.). Capturing Propagation of Infected Program States. *Proc. Intl. Conf. on Foundations of Soft Engg.* (pp. 43-52). Amsterdam: ACM.

Zheng, A., Jordan, M. L., & Aiken, A. (2004). Statistical Debugging of Sampled Programs. In S. Thrun, L. Saul, & B. Schölkopf (Eds.), *Advances in Neural Information Processing Systems* (pp. 9-18). Cambridge, USA: MIT Press.

Syed Shariyar Murtaza received his Ph.D. from the University of Western Ontario in 2011. He received his MS in Computer Engineering from Kyung Hee University in 2006 and BS from the University of Karachi in 2004. He has been working as a research scientist with Concordia University and Defence Research and Development Canada since 2011. He specializes in software maintenance with a focus on fault localization and anomaly detection. He also has a keen interest in the applications of machine learning in software engineering and information management.

Abdelwahab Hamou-Lhadj is a tenured associate professor in the Department of Electrical and Computer Engineering at Concordia University. His research interests include software modeling, software behaviour analysis, software maintenance and evolution, anomaly detection, business process management, and organizational performance. He has worked with and consulted for many government and industrial organizations. He holds a Ph.D. degree in Computer Science from the University of Ottawa (2005). He also holds the OMG Certified Expert in Business Process Management Certification - advanced business and technical tracks. He is a Licensed Professional Engineer in Quebec, and a long-lasting member of IEEE and ACM.

Nazim H. Madhavji is a Professor at the University of Western Ontario, London, Canada. His research interests include: requirements engineering, software architectures, system compliance, software and process quality, software evolution and feedback, and empirical studies.

Mechelle Gittens is a research-adjunct professor at the University of Western Ontario, and also a professor at the University of West Indies, Cave Hill, Barbados. Her research interests include software testing, quality control and user profiling.