Diagnosing New Faults Using Mutants and Prior Faults* (NIER Track)

[1]Syed Shariyar Murtaza, [2]Nazim Madhavji, [3]Mechelle Gittens, [4]Zude Li
[1,2,3,4] Department of Computer Science, University of Western Ontario, London, Ontario
[3] Department of Computer Science, University of West Indies, Cave Hill, Barbados
{[1]smurtaza,[4]zli263}@uwo.ca, [2]madhavji@gmail.com, [3]mechelle.gittens@cavehill.uwi.edu

ABSTRACT
Literature indicates that 20% of a program's code is responsible for 80% of the faults, and 50-90% of the field failures are rediscoveries of previous faults. Despite this, identification of faulty code can consume 30-40% time of error correction. Previous fault-discovery techniques focusing on field failures either require many pass-fail traces, discover only crashing failures, or identify faulty "files" (which are of large granularity) as origin of the source code. In our earlier work (the F007 approach), we identify faulty "functions" (which are of small granularity) in a field trace by using earlier resolved traces of the same release, which limits it to the known faulty functions. This paper overcomes this limitation by proposing a new "strategy" to identify new and old faulty functions using F007. This strategy uses failed traces of mutants (artificial faults) and failed traces of prior releases to identify faulty functions in the traces of succeeding release. Our results on two UNIX utilities (i.e., Flex and Gzip) show that faulty functions in the traces of the majority (60-85%) of failures of a new software release can be identified by reviewing only 20% of the code. If compared against prior techniques then this is a notable improvement in terms of contextual knowledge required and accuracy in the discovery of finer-grain fault origin.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Testing and Debugging – debugging aids, tracing, diagnostics, error handling and recovery.

General Terms
Reliability.

Keywords
Mutants, decision tree, faulty function, execution traces.

1. INTRODUCTION
Studies show that in large software systems about 70% [10] and 50%-90% [3] of the field failures are "rediscoveries" of previous faults. Literature also indicates that as much as 80-100% of the field faults originate in 10-20% of the code [8] --80-20 Pareto rule. Unfortunately, a field-reported rediscovered fault again requires identifying the faulty code so as to fix it. This drains invaluable resources [3],[10] as discovery of fault origin can consume 30%-40% of the time required for error correction [16].

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

We investigated the Siemens suite [14] and found that different faults in the same function occur with similar function-calls occurrences. For example, Figure 1 shows that when an out-of-bound index error occurs in an array in a function ("fooX"), then the last sequence of function-calls are the same: in Figure 1 (traces 'a', 'b' and 'c'), "fooX" always follows "foo2" though not necessarily immediately, and the program exits by throwing an exception. Thus, using the prior empirical analysis of data on rediscoveries, Pareto rule and similar function-call patterns as a basis, it can be envisaged that majority (80-90%) of the field failures can be diagnosed by focusing on the "faulty" traces of approximately 20% of the code.

This paper, therefore, addresses the problem of identifying faulty functions from function-call level traces of (crashing or non-crashing) failures in deployed software. Our previous work (called the F007 approach) [13] requires a historical collection of failed traces of actual faults of the current release to discover faulty functions from the traces of field failures. This limits it to only previously known faulty functions of the current release.

In this paper, we describe a new strategy which uses failed traces of mutants (i.e., automatically generated faults [13]) and failed traces of prior releases to identify faulty functions in succeeding releases. In this strategy, first, we collect failed traces of prior releases (if any). Second, we identify approximately 20% (or more) of the suspected functions (to be faulty) in a new software release using a prediction model based on the code metrics of functions. We generate mutants for the suspected functions, and then collect failed traces for those mutants. Third, we execute F007 on traces of mutants and prior releases to discover new and rediscovered faulty functions. The use of mutants actually facilitates F007 to discover new faulty functions.

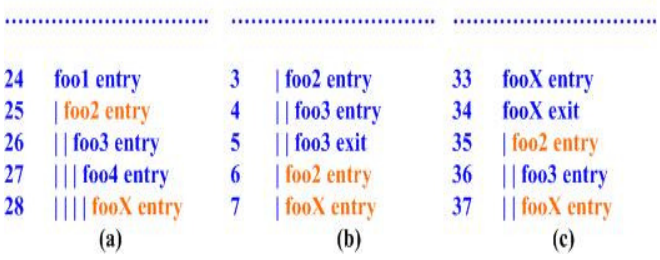

Figure 1: An example of common patterns in failed function-call level execution traces.

Previous techniques using the field failures of a program focus on: (i) using many pass-fail traces to identify the origin of faults [4],[11]; (ii) identifying the causes of only crashing failures by matching the symptoms of a problem with known faulty symptoms [3],[10]; (iii) identifying faulty coarse grain code from

* This work is in part supported by NSERC.

execution traces such as files as the source of the field failure [15]; and (iv) classifying a field trace as passing trace or failing trace [9]. Furthermore, mutants have mostly been used to measure, enhance and compare the effectiveness of testing strategies (e.g., identifying quality of test cases [14], and measuring test coverage criteria [1]).

This paper contrasts from previous research, and contributes as follows: (a) it proposes a new strategy to discover faulty functions from the traces of crashing and non-crashing (field) failures; (b) it shows that mutants can be used to discover actual faults; and (c) it shows that failed traces of approximately 20% of the functions can be used to discover faulty functions in the majority of failed traces. Note that a non-crashing failure is more difficult to diagnose than a crashing failure (e.g., segmentation fault) because a non-crashing failure (e.g., logical error) could manifest itself well after the execution of the faulty code [15].

We evaluated F007 (with new strategy) on two well known UNIX utilities: Flex and Gzip [6]. Our results on these programs show that faulty functions in approximately 60-85% of the actual failed traces of succeeding releases can be identified by using traces of previous releases and traces of mutants of the same release. This identification requires reviewing only 20% or less of the program. This is significant for deployed system when only a few traces are available from the field to discover a fault's origin because of the overhead in trace collection; e.g., it is useful for alpha testing, beta testing and corrective maintenance. In essence, the proposed strategy helps in reducing the time spent in fault discovery by automatically discovering the origin of fault. This paper continues as follows: Section 2 explains subject programs; Section 3 elaborates on the proposed strategy; Section 5 articulates results; and Section 6 concludes this paper with directions to future work.

2. SUBJECT PROGRAMS

We evaluated our strategy on two UNIX utilities (i.e., Flex and Gzip) made available by Do et al. [6]. Each program had several releases, and each release of the program had many faults. Failed traces for every fault of a release were collected by running test cases on them. We used Etrace [7] to collect function-call level traces as shown in Figure 1. There were five releases (i.e., 2.4.7 to 2.5.4) of the Flex program, having 8250-9831 lines of code and 150-168 functions. We excluded the fifth release 2.5.4 of the Flex program because only four traces had failed. In the case of the Gzip program, we excluded release 1.2.3 because no test cases failed on it, and used the four releases (i.e., 1.1.2 to 1.3) with 4032-5103 LOC and 88-110 functions. Also, following the documentation provided by Do et al. [6], we collected traces of only those faults on which more than 20% of the test cases failed. Reason for this is that faults revealed by more than 20% of the cases can be easily identified by testers [6].Thus, we collected 49 to 362 failed traces on the four releases of the Flex program, and 14 to 50 failed traces on the four releases of the Gzip program, which formed our dataset of traces.

3. PROPOSED STRATEGY

There are three main steps in our proposed strategy. First, it collects failed traces of prior releases (explained in Section 3.1). Second, it generates mutants of expected faulty functions of the current (or new) release of a program and collects (mutant) traces on them (discussed in Section 3.2). Third, it executes F007 [13] on the traces of prior releases and traces of mutants to identify faulty functions in the traces of current release (see Section 3.3). The use of mutants and prior releases actually facilitates F007 to discover new and old faulty functions in a new software release.

3.1 Collecting Actual Traces

In this step, we collected resolved failed traces from prior releases—i.e., traces with discovered faulty functions. If prior releases are not available for a software system, then in-house failed traces can be used initially; from one of our prior studies on a very large commercial application there is an overlap in the location of in-house faults and field faults [8]. Note that these traces are the traces of actual faults and the traces of mutant are the traces of artificial faults, discussed in the next section.

3.2 Mutant Generation

In this step, we generated mutants of each release of a program. A mutant is an automatically generated faulty version of a program [13], obtained by applying mutation operators. Examples of mutation operators include: deleting a statement, negating a decision in "If" or "While" statement, and others. We used the tool created by Andrews et al.[1] to generate mutants for the Flex and Gzip program. Mutants are generated for every statement of a program [1]; implying generation of at least one mutant (faulty version) per statement of a program. Collecting failed traces for mutants of every statement is definitely not feasible because running test cases and collecting traces consume time and effort. For example, an instrumented program can take two orders of magnitude of a normal run [5].

To make mutant generation process efficient, we generated mutants only for functions that could be suspected as faulty in a release of a program. This is because not all the functions are faulty; literature indicates that 20% of the code is responsible for 80-100% of the faults [8]. The suspected functions were identified using the techniques proposed in the literature [2],[12] These third-party techniques [2],[12] identify probable faulty components in a current release from using machine learning models [2],[12] on the code metrics of components of past releases. We followed a similar approach to these third party techniques to identify suspected functions in a release of a program, and is described below:

Step 1. First we measured four code metrics for every function of current and prior releases of a program, and labelled them as "faulty" and "not-faulty". The four code metrics were: executable lines of code, cyclomatic complexity, max nesting of control constructs, and the ratio of comments. We used these four code metrics because they yielded the best results in our experiments.

Step 2. Secondly, we trained the decision tree on the data collected in Step 1 for all the prior releases before the current release. The trained decision tree then used the code metrics of each function of the current release (measured in Step 1) and predicted them as "faulty" or "not-faulty". For example, we trained the decision tree on the code metrics of release 1-3 of the Flex program to identify expected faulty functions in release 4. We generated mutants[†] for those expected faulty functions, and collected failed traces on such mutants by running test cases. Furthermore, collecting many failed traces on mutants can be time consuming too; therefore, we collected only ten failed traces per mutant—i.e., a maximum of 30 failed traces per function.

A minor problem with Step 2 above is that prior releases contains a large proportion of "not-faulty" functions in the training-set: the decision tree gets biased towards the "not-faulty" functions and

[†] Sometimes mutants do not compile, and test cases do not fail on mutants. To mitigate this threat and keep the overhead minimum we generated three random mutants per function.

predicts most of the functions as "not-faulty". If a function is not identified as "faulty" for the release of a program then we cannot generate mutants for that function, but a few extra mutants for few more faulty functions (incorrectly predicted as faulty) will not result in much extra effort. Therefore, we used the cost sensitive learning strategy [17] to train the decision trees on the code metrics to predict the faulty functions in future releases in Step 2. Costs in cost sensitive learning are the values which force the decision tree to make fewer errors on one type of predictions (e.g., faulty) than the other (i.e., not-faulty). Selection of cost values is based on the subjective judgment of users [17]. We developed our own criteria for selecting the cost values: (a) we selected those cost values on which approximately 70% of functions were correctly predicted as faulty in the *training-set (prior releases)*; and (b) we used these identified cost values to predict expected faulty functions in the test-set (current release). Thus, we identified 44-56 functions as expected faulty functions for Flex and 29-45 as expected faulty functions for Gzip—that is, approximately 25-40% functions were predicted as expected "faulty" and mutant traces were collected for them.

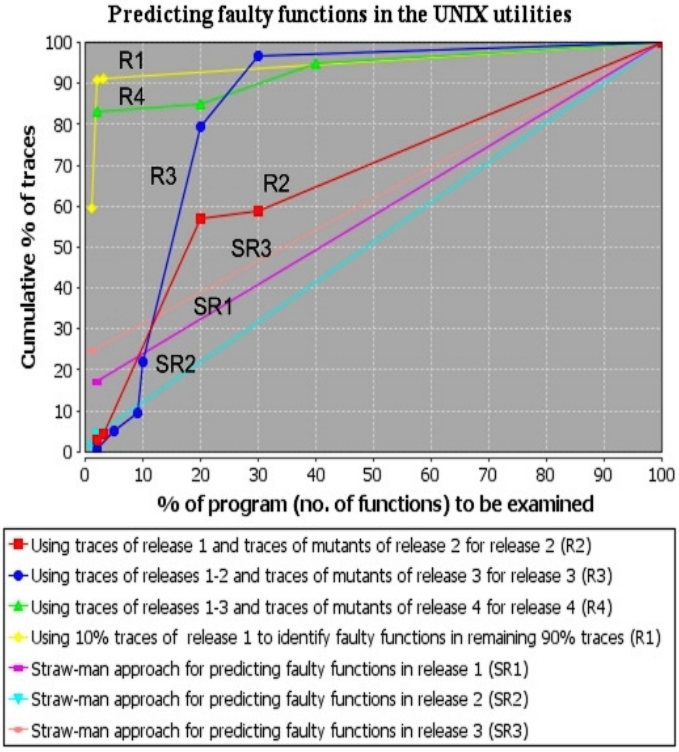

Figure 2: Functions-effort in identifying faulty functions.

3.3 Executing the F007 Technique

After collecting actual traces of different releases and traces of mutants we applied F007 on them, as described below:

Step 1. F007 first extracted function-calls and occurrences from the traces of prior failures, and labelled them with faulty functions corresponding to traces. Prior failed traces were the traces of: (a) faults in previous releases; and (b) mutants of a current release.

Step 2. Secondly, F007 trained the decision trees using one-against-all [17] approach on the extracted function-calls and their corresponding faulty functions identified in Step1. In the one-against-all approach, one decision tree is trained for each faulty function and its corresponding function-calls. For example, in the case of release 3 of the Flex program there were 63 faulty functions in the traces of release 1 and 2 and the traces of mutants of release 3. Therefore, 63 decision trees were generated for each function. The process of decision tree generation and the prior trace collection was only done once and was not time consuming.

Step 3. Thirdly, when a new failed trace arrived, F007 extracted function-calls and occurrences from that trace and provided it to the trained decision trees. Each decision tree predicted a faulty function with a probability. Functions were then arranged in the decreasing order of probability with the intuition that the top most functions were more likely to be faulty than the lower ones; e.g., in the case of a trace of the function "gen_NUL_trans" of release 3 of the Flex program, "gen_NUL_trans" was ranked at position 3 by F007--i.e., three functions were required to be reviewed.

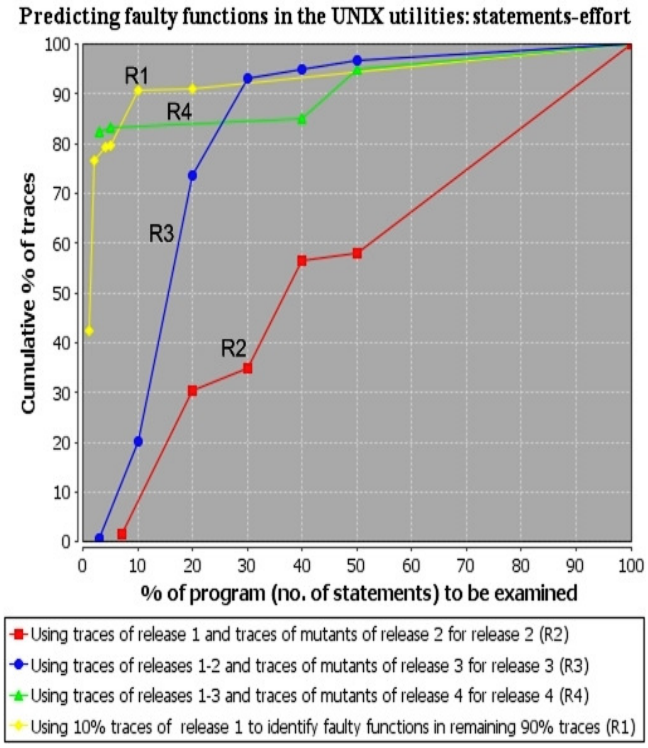

Figure 3: Statements-effort in identifying faulty functions.

The intuition behind F007 is that if traces of different faults in various functions are similar (see Figure 1) then the decision tree can associate function-calls in traces to common faulty functions.

4. RESULTS AND DISCUSSION

In Figure 2, we show the results of our proposed strategy by using the pictorial convention used by other researchers [4],[5]. The X axis measures the percentage of program to be examined -- as functions reviewed -- divided by the total number of functions. The Y axis is measured as the cumulative percentage of failed traces correctly resolved by reviewing some percentage of code. For example, the blue series (marked by ● and annotated as R3) shows that by training F007 on the actual failed traces of release 1 to 2 and the traces of mutants of selected functions of release 3, faulty functions in 80% of the actual traces of release 3 can be identified by reviewing approximately 20% of the program (functions). This is the accuracy of identifying faulty functions for both Flex and Gzip using their respective traces. Similarly, in the cases of releases 2 and 4, approximately 60-85% accuracy was obtained by reviewing 20% of the code. Thus, this show that most

of the faulty functions in the failed traces of succeeding releases can be identified by reviewing less than 20% of the code.

In the case of release 1, no prior data of mutants and previous releases existed. We trained F007 on only 10% of the failed traces of release 1 (of Flex and Gzip) to predict faulty functions in the remaining 90% traces. This is to demonstrate that we can use the traces of in-house testing or the traces of initial field faults to identify faulty functions in the traces of new field faults. If 50-90% [3],[10] of the field failures are rediscoveries then this is highly significant for deployed systems.

Note that our results were obtained with the knowledge of faults in a small percentage (25-40%) of functions (see Section 3.2). In Section 1, we already characterized previous techniques in the literature. A direct comparison of our technique with other techniques does not exist because other techniques: (a) identify a fault from a collection of passing traces and failing traces of a fault [4],[11]; whereas, our technique focuses on identifying a fault from a single trace; and (b) identify faulty files [15] for a failed trace; whereas, our technique identifies (finer-grain) faulty functions. Moreover, the technique to classify a field trace as passing or failing [9] is complementary to our work when it is not known that a field trace is captured at the time of a fault. Our technique is significant for deployed systems when only a few traces of failures are available from the field to discover its fault origin—this is usually the case due to overhead in trace collection.

Nonetheless, in Figure 2 we compare our approach against the straw-man approach. That is, we considered that for every new failed trace in a current release, a developer predicts the function with the largest LOC and the function found faulty in the largest number of failed traces of previous releases. The results are shown in Figure 2 by annotations SR1, SR2 and SR3 for release 1, 2 and 3 respectively; note that no prediction resulted for release 4. For example, for release 1(SR1), only 5% of the failed traces were correctly resolved by reviewing 2% of the code. For the remaining 95% of the traces, 100% of the code was required to be reviewed.

Finally, in order to measure the accuracy of prediction of a faulty function in terms of statements we summed all the statements of a function reviewed to identify the faulty function, and divided by the total number of statements. This is shown in Figure 3 which is similar to Figure 2. Figure 3 is actually a pessimistic approach because a developer would never review all the statements of a function due to experience and contextual information.

5. CONCLUSIONS

This paper shows that by using the traces of faulty functions of prior releases and mutants of approximately 25-40% of the functions, we can identify faulty functions in about 60-85% failed traces by reviewing 20% of the code. This paper also shows that mutants can be used to discover actual faults. Our technique is a notable improvement over other techniques (e.g., [3],[4],[15]) because it can accurately identify finer-grained origin of faults (i.e. faulty functions) without the need for much contextual information. Finally, we used code metrics to identify expected faulty functions for the current release. This area still requires improvement to exactly identify the expected faulty functions, which will result in reduced effort to generate mutants and traces.

6. REFERENCES

[1] Andrews, J., H.; Briand, L.,C.; Labiche, Y.; Namin, A., S.; "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria." *IEEE Trans. on Softw. Eng.,* Vol.32, No.8, Aug. 2006, pp.608-624.

[2] Basili, V., R.; Condon, S., E.; El Emam, K.; Hendrick, R., B.; and Melo, W.; "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components." *Proc. 19th Intl. Conf. on Softw. Eng. (ICSE)*, ACM Press, Boston, USA, May 1997, pp. 282-291.

[3] Brodie et al.; "Quickly Finding Known Software Problems via Automated Symptom Matching." *Proc. 2nd Int'l Conf. on Autonomic Computing,* IEEE CS, Seattle, USA, June 2005, pp. 101-110.

[4] Chilimbi et al.; "HOLMES: Effective Statistical Debugging via Efficient Path Profiling." *Proc. 31st Intl. Conf. on Softw. Eng*., IEEE CS, Canada, May 2009, pp. 34-44.

[5] Dallmeier, V.; Lindig, C.; Zeller, A.; "Lightweight Defect Localization for Java", *ECOOP 05- Object Oriented programming*, Lecture Notes in Computer Science, Springer, Glasgow, UK, Aug. 2005,pp 528-550.

[6] Do, H.; Elbaum, S., G.; and Rothermel, G.; "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." *Empirical Softw. Eng.*, Vol. 10, Springer, Oct. 2005, pp. 405-435.

[7] Etrace (Tracing Tool): http://ndevilla.free.fr/etrace/; Mar. 2008.

[8] Gittens, M.; Kim, Y.; and Godwin, D.; "The Vital Few Versus the Trivial Many: Examining the Pareto Principle for Software." *Proc. 29th Int'l Computer Softw. and Appl. Conf.*, IEEE CS, Edinburgh, Scotland, July 2005, pp. 179-185.

[9] Haran et al.; "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks." *IEEE Trans. on Softw. Eng.* ,Vol. 33, No.5, May 2007, pp.287 -304.

[10] Lee, I.; and Iyer, R.; "Diagnosing Rediscovered Problems Using Symptoms." *IEEE Trans. on Softw. Eng.*, Vol. 26, No. 2, Feb. 2000, pp.113-127.

[11] Liu, C.; and Han, J.; "Failure Proximity: A Fault Localization-based Approach." *Proc. of the 14th SIGSOFT Symp. on Foundations of Softw. Eng.*, ACM, Portland, USA, Nov. 2006, pp.45-56

[12] Lounis, H.; Ait-Mehedine, L.;"Machine Learning Techniques for Software Product Quality Assessment." *Proc. Conf. on Quality Software (QSIC), IEEE CS*, Germany, Sep. 2004, pp. 102-109.

[13] Murtaza, S.,S.; Gittens, M.; Li, Z., Madhavji, N., H.; "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field." *Proc. conf. of centre of advanced studies on collaborative research: meetings of minds (CASCON)*, ACM Press, Ontario, Canada, Oct. 2010, pp. 57-71.

[14] Offutt, A., J.; and Untch, R., H.; "Mutation 2000: Uniting the Orthogonal," in *Mutation Testing for the New Century*, Wong W,E., Ed., USA: Kluwer Academic Publishers, 2001, pp. 34-44.

[15] Podgurski, A.; Leon, D.; Francis, P.; Masri, W.; Minch, M.; Sun, J.; & Wang., B.; "Automated Support for Classifying Software Failure Reports". *Proc. Intl. Conf. on Softw. Eng.*, IEEE CS, Portland, USA, May 2003, pp. 465-475.

[16] Proprietary workshop on large commercial software, Sep. 2008.

[17] Witten, I., H.; and Frank, E.; *Data Mining: Practical Machine Learning Tools and Techniques,* USA, Morgan Kaufmann, 2005.