2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Software Behavior and Failure Clustering: An Empirical Study of Fault Causality

Nicholas DiGiuseppe
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: nicholas.digiuseppe@uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: jajones@ics.uci.edu

***Abstract*—To cluster executions that exhibit faulty behavior by the faults that cause them, researchers have proposed using internal execution events, such as statement profiles, to (1) measure execution similarities, (2) categorize executions based on those similarity results, and (3) suggest the resulting categories as sets of executions exhibiting uniform fault behavior. However, due to a paucity of evidence correlating profiles and output behavior, researchers employ multiple simplifying assumptions in order to justify such approaches. In this paper we present an empirical study of profile correlation with output behavior, and we reexamine the suitability of such simplifying assumptions. We examine over 4 billion test-case outputs and execution profiles from multiple programs with over 9000 versions. Our data provides evidence that with current techniques many executions should be omitted from the clustering analysis to provide clusters that each represent a single fault. In addition, our data reveals the previously undocumented effects of multiple faults on failures, which has implications for techniques' ability (and inability) to properly cluster. Our results suggest directions for the improvement of future failure-clustering techniques that better account for software-fault behavior.**

***Keywords*-Execution Clustering; Failure Proximity; Fault Understanding; Debugging;**

I. Introduction

Fixing software faults is an expensive but necessary step in software development. One study found that attempts to reduce the number of delivered faults in software are estimated to consume 50% – 80% of the development and maintenance effort [3]. To alleviate this burden, researchers created a variety of automated techniques which facilitate the debugging process. *Failure clustering* is one such technique enabling the grouping of test cases exhibiting similar program behaviors. Failure clustering attempts to group failing executions that fail due to the same fault. Failure-clustering methods have been found to be successful [5], increase speed of the product to delivery [11], and beneficial at removing noise for other debugging methods like fault localization [23]. The clustering of failures is typically performed by inferring the semantic behavior of those failures from the executed program features. Ideally, the clustering process produces clusters that fail due to a single fault — we call such single-fault clusters *pure.*

Previous researchers have focused primarily on program spectra (e.g., [5, 14, 19, 20]) to categorize executions. Harrold *et al.* [10] defined various program spectra, such as, statement coverages, statement profiles, branch coverages, data dependences, and execution traces, and discovered that, "when failures exist on particular inputs, spectra differences are likely also to exist on those inputs." Yet Liblit *et al.* found even though spectra differ, "in reality, we do not know which failure is caused by which bug," [15].

Researchers to date, typically employ at least one of three simplifying assumptions: (1) program spectra (in our case, statement profiles) can approximate the behavior semantics of failing test cases, (2) cluster purity (whether failures within a cluster are failing due to the same fault or faults) can be ignored, and (3) each failure to be clustered is failing due to a single fault. This work investigates the validity of these assumptions, because they can greatly impact current failure-clustering research and practice by providing guidance on the creation of more pure and accurate clusters. In this paper, we examine these three assumptions on three real-world programs with a combination of over 9000 faulty versions which took over 100 days of computational time.

One factor affecting failure clustering is the inherent complexity of failure behavior in the presence of multiple faults. To facilitate early work, despite many unknowns in this problem space, researchers have continued to work despite the lack of thorough understanding of how multiple faults interact, and how those interactions can affect failure clustering. However, in order to best inform future failure-clustering efforts, we believe that it is vital to understand how program-spectra relate to behavior semantics, specifically in the presence of multiple faults. Do similarities in execution spectra approximate similarities in output fault behavior? Does the use of program spectra result in clusters of failures caused by the same faults? Do failures typically exhibit the effects of a single fault or multiple faults? Such questions, which are the subject of this work, are motivated by the fact that program spectra are often the most detailed and accessible information about program failures. Without an understanding of how faults alter spectra, clustering techniques will suffer unknown sources of inefficacy.

To gain such understanding, we present experiments that examine the relationship between program-failure output and statement profiles of real-world software, in detail. First, we investigate program output to determine its suitability as an oracle to evaluate spectra-based failure-clustering techniques. Next, we investigate one of the most common program spectra for performing failure clustering — execution profiles — to analyze how accurately they represent failure behavior. Then, we investigate features of execution-failure behavior, which demonstrate that failure behavior, and thus the ability to cluster, are more complex than assumed. Next, we present results and assess the viability of commonly used simplifying assumptions for failure-clustering research and experimentation. Finally, we investigate the potential for a pre-clustering phase, which may greatly reduce some of the complexity that impacts failure clustering.

The main contributions of the paper are:

1) An in-depth analysis and discussion of the intricacy and prevalence of complexity issues which are unresolved with current failure-clustering research. Current research accepts simplifying assumptions to facilitate early study. Our studies investigate these assumptions and provide evidence that in part corroborates and in part refutes the basis on which these assumptions can and should be made. These results can inform developers of new fruitful research topics in failure clustering, and clarify what issues need to be overcome to enable clustering methods to generate *pure* clusters in real software.
2) A novel study into software behavior with respect to clustering. Through an examination of outputs and profiles we identify and classify four previously unexplored categories of failure behavior. Additionally, we represent a possible oracle for failure clustering which demonstrates the need for a pre-clustering step which can remove failures that contribute to cluster impurity. This possible oracle could illuminate ways that current failure clustering research can better approximate perfect clustering.
3) A pilot study, which presents evidence of heuristics that can allow clustering to more closely approximate actual software behavior through an examination of software profiles and our new software behavior data. This pilot study provides guidance for new research areas focusing on cluster purity along with enabling current failure clustering methods to have a greater degree of cluster purity.

II. Background

Execution clustering attempts to group together executions based upon their semantic behavior. In practice, this means that executions are clustered by utilizing their execution profiles (gathered from instrumented versions of the code) as a proxy for measuring the behavior of executions. An early instance of such work was performed by Podgurski *et al.* [21], in which they attempted to determine an execution's pass/fail status through profile clustering. They later found that identifying pass/fail status was, "not likely, because failures are often caused by small defects in a large program" [20]. Podgurski insinuates that because a fault has little impact on the overall *profile*, it is difficult to distinguish a failing from a passing profile. However Dickinson, along with Podgurski *et al.* found that "well over half of the nearest neighbors of failures were failures," [5] and, "executions which do fail have unusual properties that may be reflected in execution profiles" [6]. These later results contradict and expand upon Podgurski's earlier findings, suggesting that profiles might be an effective identifier of failures.

An extension of execution clustering research is failure clustering. *Failure clustering* undertakes to categorize different failing executions according to those that failed due to the same fault(s). Failure clustering uses many of the same methods as execution clustering, primarily the comparison of program spectra to locate unique properties that each fault exhibits. One complication found by Liu *et al.* [16] is that failing profiles that fail due to the same fault can be quite different and the, "due-to" relationship between failing cases and underlying faults are, "hard" to identify [16].

While many different spectra have been used to isolate this due-to relationship, failure clustering has mainly followed one of two hierarchical methods in its clustering algorithm, agglomerative or divisive clustering. Divisive, or top-down clustering starts with all failures in the same cluster, and at each iteration splits the most dissimilar cluster into two. Agglomerative, or bottom-up clustering is very similar but executes this process in reverse. Agglomerative clustering begins with every failure in its own cluster, and at each iteration the two most common clusters are merged. Both of these algorithms recursively repeat until clustering is deemed to be complete. In many clustering techniques, the goal for the number of resulting clusters, or K, is pre-specified, often arbitrarily, prior to program execution.

Many researchers use this arbitrary K stopping point because it remains unclear exactly how many clusters should be generated. Researchers agree on the ideal of cluster *purity* — each resulting cluster should represent only one fault. Unfortunately, previous work performed by the authors, [7] found that faults commonly obfuscate other faults, making a determination of the quantity of faults in a program almost impossible with current techniques. Further complicating this process are situations where a failure is the result of multiple faults, resulting in confusion regarding which cluster the execution aligns with. To simplify these additional layers of complexity and facilitate preliminary study researchers have typically employed at least one of three assumptions.

The **first assumption** is that program spectra approximates fault causality (e.g., [2, 6, 11, 16, 21]). This assumption is generally made to facilitate automation. The **second assumption** is that cluster purity can be ignored. In the evaluation of these techniques there is no consideration given to the purity of the clusters (e.g., [6, 11]). These evaluations determine success if clusters are composed of executions with similar program spectra or can be effectively used for a client analysis, irrespective of how many faults are contained in each cluster. The **third assumption** is that every failure is caused by only a single fault. In these evaluations, the researchers filter the failures so that each failure executes no more than a single fault (e.g., [2, 16, 21]). Although their evaluation metrics account for cluster purity, their experimental procedures remove any failures that were caused by multiple faults, thus eliminating the concerns for non-singularity of fault-to-failure effects.

Multiple problems arise when using these assumptions. One issue is almost all real-world programs contain multiple faults, and faults are not independent. Debroy and Wong [4] and the authors [7] found when examining the Siemens suite and real-world programs, respectively, faults almost always interact to alter program behavior. Further, the authors showed real-world programs with multiple faults almost always exhibit fault-detection interference — faults interfere with other faults' ability to be localized through program spectra [8]. Thus, the single-fault-to-failure assumption reduces the problem space by removing multiple-fault failures, which makes these clustering techniques unusable at worst, and untested at best with respect to real-world (i.e., multiple fault) software. One issue stemming from ignoring cluster purity is not accurately measuring the accuracy of the resulting clusters from failure clustering. There is no conclusive data that spectra similarity represents fault synonymy. Without a quantitative analysis of cluster purity, it becomes difficult to compare metric effectiveness or determine if a metric is accurate.

III. Output as Oracle for Failure Causality

In order to demonstrate our analysis, as well as the strengths and the limitations of each approach, we present an example scenario. Figure 1 presents three versions of the same program, each version in a major column. The program, `Inc_or_Double` takes a boolean and an integer as input. If the boolean is true, the program should increment and return the integer, otherwise the program should double and return the integer. Each version executes the same test suite, presented in the minor columns to the right of each version of the program. The inputs are listed below the name of each test case. For each test case we show its profile, output, and faults that caused failure. In Version 1 and 2 there exists a single fault, fault f1 and fault f2 respectively. In Version 3, both faults are present.

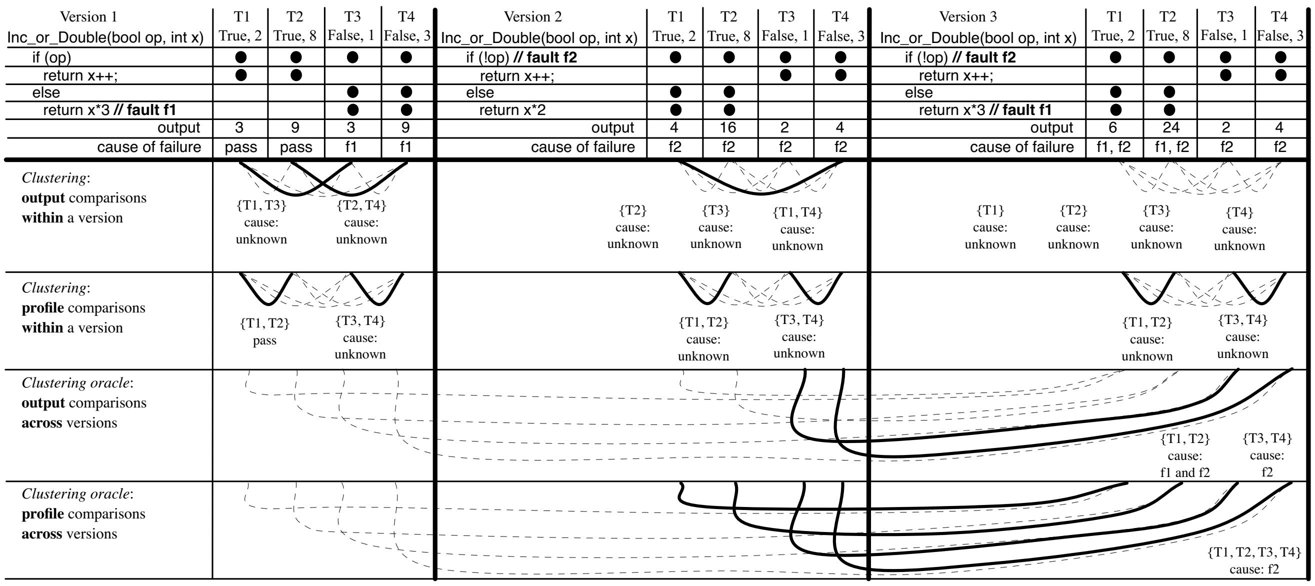
Below the bold horizontal line, we present four methods, each in its own row, for comparing test cases (or, more generally, executions). The former pair of methods (i.e., first two rows) each compares test cases *within* a version. The latter pair of methods (i.e., last two rows) each compares test cases *across* versions. The *within* methods are generally possible in practice when attempting to cluster test cases. In contrast, the *across* methods are generally infeasible in practice because developers would be unable to generate versions of their software containing different subsets of their faults until they found and identified each individual fault, at which point failure clustering would be unnecessary. Additionally, across-version profile-based clustering would require mapping program elements between the differing versions with potentially altered control flow. The *across* methods are the methods that we are evaluating in this work for their potential for providing oracles for failure-clustering research, as well as providing a means to assess the viability of the failure-clustering assumptions.

In these rows of methods, lines are drawn for each comparison performed: solid lines connect executions that are deemed similar and dashed lines connect executions that are deemed dissimilar. Below the comparisons, the composition of the resulting clusters are presented in braces, as well as the deduced cause of the failures in that cluster. For traditional failure-clustering activities (i.e., the first two rows), the faults that cause failures cannot be known simply from the cluster.

The first method of each method-type (i.e., within and across) compares the output of each test case. The second method of each method-type (i.e., within and across) compares the profile of each test case.

We form clusters based on the similarities found (represented as bold lines). Although execution clustering is typically performed using hierarchical clustering (as described in Section II), for the purpose of clarity, we express within-version clustering by the sets of executions that are deemed similar. For example, for the within-version output-clustering method on Version 2, T1 and T4 are deemed similar, and T2 and T3 are not similar with any other test case. As such, the resulting clusters are {T1, T4}, {T2}, and {T3}.

In contrast, we perform across-version clustering by comparing test-case behavior — whether output or profile — for a particular version containing a set of known faults with all versions containing every subset of its faults. For example, Version 3 (which contains both faults, f1 and f2) has the behavior of each test case compared with that same test case's behavior executed on Version 1 (which contains only f1) and Version 2 (which contains only f2). For the across-version output-clustering method, the output of test case T3 executed on Version 3 is compared with: (1) the output of T3 on Version 2, which is found to be similar; and (2) the output of T3 on Version 1, which is found to be dissimilar. Because similarity for T3 is found only with



Version 1 Inc_or_Double(bool op, int x)	T1 True, 2	T2 True, 8	T3 False, 1	T4 False, 3
if (op)	●	●	●	●
return x++;	●	●		
else			●	●
return x*3 **// fault f1**			●	●
output	3	9	3	9
cause of failure	pass	pass	f1	f1

Version 2 Inc_or_Double(bool op, int x)	T1 True, 2	T2 True, 8	T3 False, 1	T4 False, 3
if (!op) **// fault f2**	●	●	●	●
return x++;			●	●
else	●	●		
return x*2	●	●		
output	4	16	2	4
cause of failure	f2	f2	f2	f2

Version 3 Inc_or_Double(bool op, int x)	T1 True, 2	T2 True, 8	T3 False, 1	T4 False, 3
if (!op) **// fault f2**	●	●	●	●
return x++;			●	●
else	●	●		
return x*3 **// fault f1**	●	●		
output	6	24	2	4
cause of failure	f1, f2	f1, f2	f2	f2

Figure 1: Three versions of the same program, with each combination of two faults. Comparisons (dashed lines for dissimilar and solid lines for similar) are made both within a version (traditional failure clustering) and across versions (failure-causality oracles). Resulting clusters shown in braces.

Version 2, and that version contains only f2, we deduce that T3 failed due to f2. Test case T4 exhibits this same quality of failing specifically due to f2. As a consequence of their equivalently deduced fault causality, T3 and T4 are clustered. Because the output of T1 and T2 on Version 3 are found to be dissimilar with their output of either Versions 1 or 2, their behavior cannot be attributed to either f1 or f2. As such, we deduce that they failed due to the presence of both faults and thus are clustered together. That is, their failure behavior cannot be witnessed with any subset of the faults in the program.

When assessing these methods on the example, we observe that on Version 3, the within-version output clustering failed to identify meaningful clusters, but the profile clustering produced correct clusters. In contrast, the across-version output clustering produced the correct clusters, and the profile clustering failed to recognize the difference in behavior caused by f1. Thus, while output clustering performed poorly *within* a version, it performed well *across* versions, which motivates its use as an oracle for failure causality. Conversely, profile clustering performs well *within* a version, but less effectively *across* versions, but may still be a candidate as an oracle. In our experiment, we utilize output as the oracle and evaluate profiles as an approximation, which can also be used in intra-version clustering.

IV. Experiment

To understand software behavior in the presence of multiple faults and the implications for failure clustering, we conducted an experiment. Our experiment assesses the impact of multiple faults and evaluates the viability of the past assumptions for failure clustering. We present our research questions in Table I.

In this section, we first discuss the subjects used in our experiment. Next, we discuss our independent variable along with our four dependent variables. Finally, we discuss the setup of the experiment.

A. Objects for Analysis

To enable understanding of failure behavior in the presence of multiple faults, we evaluated three real-world subjects, Gzip version 1.0.7, Sed version 3.02, and Space, that are commonly used in software-testing research. Gzip and Sed are Unix utilities of medium size. Gzip contains 7928 LOC and 214 test cases; and Sed contains 10154 LOC and 362 test cases. Space originates from the European Space Agency and contains 6445 LOC and over 13000 test cases. To enable the experiment to scale — it required months of computational time (explained in Section IV-C) — we randomly sampled and used 500 of Space's test cases. All were downloaded from the Subject Infrastructure Repository [9] along with their faulty versions and test cases.

As with previous work (e.g.,[8, 12, 13, 17, 23]), we excluded faulty versions that caused no failures or were not covered by any test cases. Excluded faults were replaced with mutants so that the subjects had a similar quantity of faults. Thus, our programs contained some real faults, and some mutants. Recent work by Ali *et al.* [1] found that mutants were identified with similar rates as real faults.

Table I: Research questions addressed in this experiment.

RQ1:	Does program spectra (in our case, coverage profiles) approximate fault causality?
RQ2:	Can cluster purity (whether failures within a cluster are failing due to the same fault or faults) be ignored in evaluations?
RQ3:	Are almost all failures caused by a single fault?

Table II: Example showing all four types of behavior.

Faults Contained	Output for: Test Case 1	Output for: Test Case 2	Output for: Test Case 3	Output for: Test Case 4
F1, F2, F3	A	B	C	A
F1	A	X	C	D
F2	F	F	F	D
F3	G	G	H	G
F1, F2	J	J	J	A
F1, F3	R	R	R	R
F2, F3	Q	Q	C	Q
Category	Singleton	New Failure	Tie	Combo

However, to ensure these mutants are representational of real faults, we follow the methods described by Offutt *et al.* [18]. Offutt's methods dictate random line selection and random mutation based upon the set of possible mutations enumerated in his study.

B. Variables and Measures

Our primary goal is to understand the impact of the behavior of faulty software on failure clustering. To accomplish this, we utilize one independent and four dependent variables. Our independent variable is the *quantity of faults*. Our four dependent variables are the quantity of failures that exhibit each of four new categories of failure behavior, which are identified in this work. We name these four failure-behavior categories: *Singleton, Combo, Tie,* and *New* failures. These failure categories correspond to how test-case outputs and profiles are affected by multiple faults and are useful in assessing our research questions. Output comparisons check for equality, whereas profile comparisons check for similarity. Our profile similarity metric is described later in this section.

Singleton failures occur when the output (profile) of a multi-fault version is equal (similar) to only one single fault that composed it. *Combo* failures occur when the output (profile) of a multi-fault version is equal (similar) to only one multi-fault subset that composed it. *Tie* failures occur when the output (profile) of a multi-fault version is equal (similar) to multiple different fault subsets that composed it. *New* failures occur when the output (profile) is not equal (is not similar) to any fault subset that composed it — that is, that the current combination of faults that compose the multi-fault version produces behavior that no subset of those faults exhibited.

To demonstrate these classifications more simply, we present an example in Table II. In the "Faults Contained" column, we present the faults which are present in a program. The remaining columns have a test case with its corresponding behavior, represented as a letter. The *primary* version in this example contains faults F1, F2, and F3. Each possible version of the same program containing a sub-combination of the primary version's faults is represented in the remaining rows, along with its behavior. The behavior of the sub-combination version is compared with the primary version, for each test case. In Test Case 1, the output for the primary version matches exactly with only the version containing fault F1. Because the primary version's output is equal to only one sub-version, and the sub-version contains one fault, we call the behavior of Test Case 1 a Singleton failure. In Test Case 2, no output in any sub-version equaled the output from the primary version. In other words, every single fault in the primary version acted upon the program to alter its behavior and thus is called a New failure. Examining Test Case 3, we see that the primary version's output equals two sub-versions' outputs, the first containing fault F1 and the second containing faults F2 and F3, and one set of faults is not subsumed by the other. In other words, different versions managed to create the exact same output, and one is not a subset of the other. We cannot determine which group of faults caused this output — F1 alone, or F2 and F3. Thus, it is a Tie failure. Finally, in Test Case 4 we see that the primary version's output equals the sub-version containing F1 and F2. Because this sub-version contains multiple faults, and it is the only sub-version which equals the primary version, we call this a Combo failure. Combo is used here to denote that a combination of faults created this output through their interaction.

When comparing profiles, our goal, like previous researchers, is to find profiles that have similar semantic behavior. To this end, we first normalized loop and method profiles by dividing all their elements by the profile count of their header. We then compare the Euclidean distance, D between two profiles. To determine whether two profiles are deemed as similar given D, we compute a threshold for a given program and test case. Each test case's threshold was gained by randomly sampling 5% of all versions and using the maximum distance between any two sampled versions. To be conservative, we consider profiles similar if D is less than 1% of the maximum distance.

C. Experimental Setup

To examine software behavior for multiple faults and assess the three assumptions of previous research discussed in Section II, we designed an experiment that enables detailed comparisons of output and profiles as the quantity of faults changes. Our experiment tracks patterns in software behavior

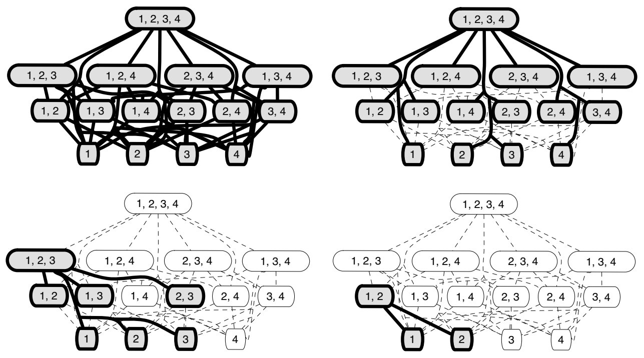


Figure 2: An example of our comparison process. The upper-right graph contains the comparisons for the four fault version, the bottom-left graph contains the comparisons for *one* three-fault version, and the bottom-right graph contains the comparisons for *one* two-fault version. The upper-left graph contains all the comparisons for all four-fault, three-fault, and two-fault versions.

as software becomes more faulty. First, we select ten faults at random and generate a version containing those faults. Then, we create a version for each possible sub-combination of faults. We made all the single faults versions, (10 choose 1), then all the two faults versions (10 choose 2), and on up to the nine fault versions (10 choose 9) that could be generated from the ten fault version. Next, for each version, we executed the entire test suite storing the output and profile for each test case. Profiles were gathered with the GNU C compiler, Gcc, and the Gcov utility.

Then, for every faulty version, f we compare each output (profile) with every output (profile) from the versions that contained faults that were subsets of f. Figure 2 depicts this process. In this example the primary program contains four faults: 1, 2, 3 and 4. We first need to compare each two-fault version with its sub-versions. The figure shows the version that contains faults 1 and 2 being compared with the the versions that contain only fault 1 and only fault 2. We repeat such comparisons for all two-fault versions (although the figure doesn't show this for space sake), then compare the three-fault versions. The figure compares the version containing faults 1, 2, and 3 with the versions containing faults 1,2; 1,3; 2,3; 1; 2; and 3. We would repeat this type of comparison for every three-fault version (although the figure doesn't show this for space sake). Finally, we need to compare the four fault version, with the versions containing 1,2,3; 1,2,4; 2,3,4; 1,3,4; 1,2; 1,3; 1,4; 2,3; 2,4; 3,4; 1; 2; 3; and 4. Thus, we compare each version with all the versions containing any subset of contained faults. The number of comparisons required is precisely defined as

$$C = 2T \sum_{i=2}^{n} \sum_{j=1}^{i-1} \binom{n}{i} \binom{n}{j} \tag{1}$$

where C is the total number of comparisons, T is the size of the test suite, and n is the maximum quantity of faults existing for any version. The multiplier of 2 is included to reflect the fact that both output and profile comparisons are performed. Thus we compare each version's output and profile with every possible sub-version's corresponding output and profile. This quantity of comparisons enables (1) the identification of output or profile changes as faults are inserted in the program and (2) the analysis of how faults affect the output and profile. A list of possible changes to output or profiles is found in Table II and explained previously in Section IV-B.

Each iteration entails creating a random ten-fault version for Gzip, Sed, or Space, creating all possible sub-versions, and performing all comparisons. Each iteration took roughly 11 days for Gzip, 14 days for Sed, and 10 days for Space. For each each iteration, 1023 unique versions are created, Gzip performs 112,103,900 comparisons of profiles and outputs while Sed performs 190,157,550, and Space performs 26,192,500,000 (due to differences in their test suite size). To gather our data we executed nine iterations — three for each subject — generated 9207 versions, and performed 4,533,921,750 comparisons, which required roughly 2,520 hours, or 105 days of computational time. The experiments were run on an Intel Core2 Duo CPU at 2.66GHz.

V. Results

We present our results relating profiles and output in Figures 3, 4, and 5. Figures 3a, 4a, and 5a display the software behavior using the output, and Figures 3b, 4b, and 5b display the software behavior approximated by profiles. For each figure the horizontal axis represents our independent variable: the quantity of faults in the base version. Along the vertical axis we represent the average frequency of each of our dependent variables as a percentage for all iterations. At every coordinate along the horizontal axis, there can be four bars, each representing a single type of software behavior: singleton, tie, combo, and new failure, as shown in the legend. Each bar aggregates all faulty versions that contain the same quantity of faults. For example, the bars at quantity 3 would include results for the program containing faults {f1, f2, f3}, {f2, f3, f4}, {f1, f2, f4}, and so on.

The Gzip figures demonstrate a monotonically increasing number of Combo failures as the quantity of faults increases in the base version, while the Singleton failures decrease for both output and profiles. Also, the approximation of Gzip's profiles to output is fairly accurate.

The Sed figures demonstrate a monotonically increasing number of Singleton failures, at fault quantities greater than 1, which by definition always produce 100% Singletons.

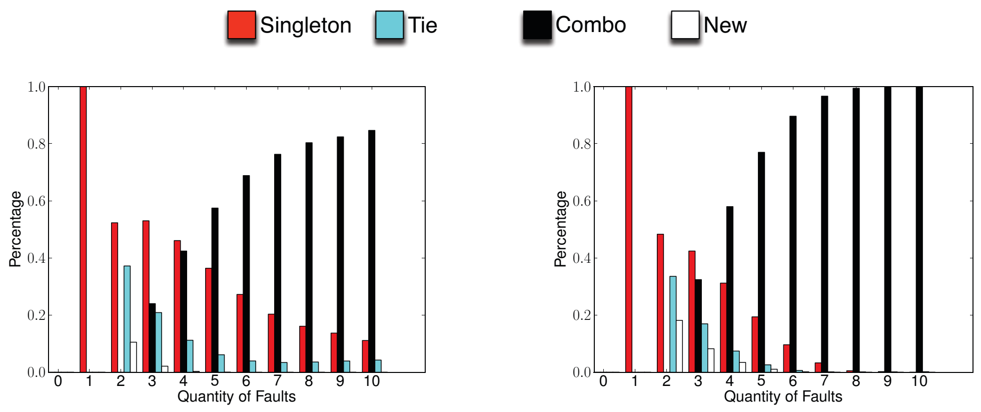


Figure 3: Average software behavior for outputs (a) and profiles (b) of Gzip.

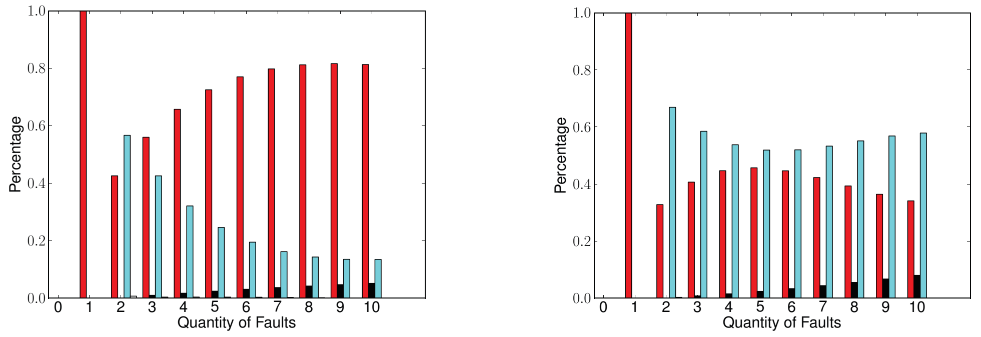


Figure 4: Average software behavior for outputs (a) and profiles (b) of Sed.

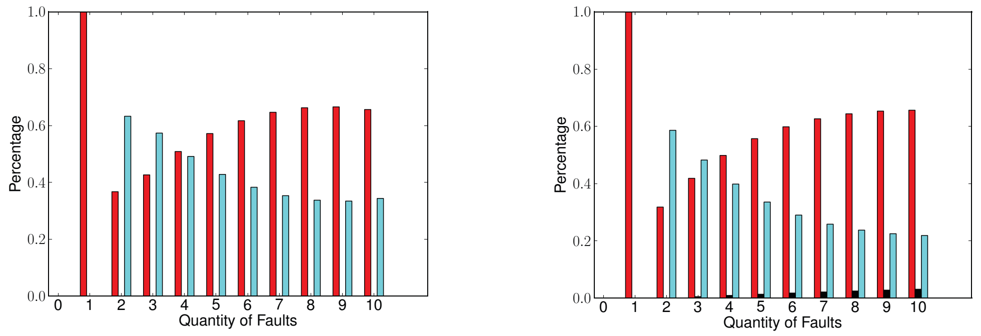


Figure 5: Average software behavior for outputs (a) and profiles (b) of Space.

Sed also exhibits a monotonically decreasing number of Tie failures. The Sed profile figure exhibits an initial decrease in ties and then increases, while demonstrating an initial rise in singletons and then a decrease. The profile approximation is generally accurate, but less so than the other subjects.

Much like Sed, the Space figures demonstrate decreasing numbers of Tie failures, and increasing number of Singleton failures, as the fault quantity in the base version increases. The approximation of the Space profiles closely matches the results for output.

These results demonstrate a few trends. The first is that Sed and Space exhibit similar multi-fault interaction, whereas Gzip behaved differently. Gzip produces a majority of combos, a smaller set of singletons, and only a small percentage of ties or new failures, whereas our other two subjects primarily produce ties and singletons and only a small subset of combos. Such differences suggest that multi-fault interaction and behavior is program-specific.

Gzip demonstrates highly similar output and profile results. Both output and profile results exhibit that failures caused by the interaction of subsets of all faults (i.e., Combo failure) is proportional to the quantity of faults present. Additionally, only in rare cases does Gzip fail due to all faults (i.e., New failure). We speculate that this paucity of New failures is due to the difficulty of individual test cases executing all faults in the program.

Sed demonstrates a slightly more pronounced difference between its output and profile results. The output results suggest that failures caused by individual faults (i.e., Singleton failures) are proportional to the quantity of faults present. Upon further investigation, we found this phenomenon to be due to an increasing likelihood to include a fault that produces behavior in the program that precludes other faults manifesting (otherwise known as fault obfuscation [7, 8]). This phenomenon is discussed more in Section VI-A.

Space demonstrates highly similar output and profile results. Both output and profile results exhibit that as more faults are added, the program is more likely to fail due to a single fault (i.e., Singleton failures). Much like Sed, we found the presence of dominating faults, which were more likely to be included as the fault quantity increased.

VI. Discussion and Analysis

In order to better understand the results from Section V, we present an interpretation and analysis here. We discuss the results, the complexity that is found within them, and how those complexities affect the three assumptions for failure clustering made by past researchers. In this section, we also present a pilot study in which we analyze a lightweight heuristic that may enable more pure clusters.

A. Questioning Assumptions

To assess the viability of the first assumption — that execution profiles accurately approximate fault causality — and to answer the research question RQ1, we compare the output results with the profile results. For two of the three subjects, Space and Gzip, the profile results closely approximate the output results, and for Sed, the profile results are a reasonably accurate approximation. These results suggest that profiles are useful approximations of output behavior. However, we investigated several output-to-profile execution pairs and found that while the profile-based execution behaviors were correctly categorized, the set of faults deduced as the failure cause differed. For example, at the 10-fault level for Gzip, where most failures are categorized as Combos, the output oracle specified that Fault 12 and Fault 18 were together causing most failures, whereas the profile results blamed Fault 3, Fault 11, Fault 12, Fault 13, and Fault 18. Although the profile-based assessment of the faults causing failures subsumed the output-oracle, many spurious faults were also blamed.

To RQ1, we assess: Profiles can accurately approximate the general behavior of failures, however do suffer from a degree of inaccuracy in terms of fault-causality approximation.

To assess the viability of the second assumption — cluster purity can be ignored — and to answer the research question RQ2, we observe the ratio of Singleton failures in the presence of multiple faults. Although we did not perform traditional within-version clustering in our experiment, our results for our subject programs exhibited many Combo and Tie failures, which would likely cause clusters to be impure. However, our results also demonstrate the high degree of Singleton failures, especially when the number of faults present in the program is high, for two of our three programs.

To RQ2, we assess: The high degree of failures caused by multiple faults suggest that clusters are unlikely to be pure. Future research is advised to evaluate cluster purity rather than assume it.

To assess the viability of the third assumption — failures that are caused by multiple faults can be ignored and discarded, a priori, from an evaluation — and to answer research question RQ3, we compare the number of Singleton failures with the combined numbers of Combo, New, and Tie failures. Our results suggest that the prevalence of failures caused by multiple failures may be program-specific. Whereas Sed and Space exhibit relatively few failures caused by multiple faults at higher fault quantities, Gzip exhibited mostly failures due to multiple faults. For Sed and Space, evaluations that ignored multiple-fault failures would remove a small, but sizable, portion of test cases from evaluation. For Gzip, several versions would be left with no failures at all — or, at least, an unrepresentative and small sample of the population.

In further examinations of the Singleton failures, we found an interesting phenomenon: *fault domination*. For example, for Sed, several of the Singleton failures contained its Fault 20. We found that sub-versions that contain Fault 20, almost always demonstrate Singleton behavior: Fault 20 solely causes those failures. That is, it prevented the behaviors of the other faults from manifesting. This phenomenon has been observed and reported by past researchers: Zheng *et al.* [23] and DiGiuseppe and Jones [7, 8]. We also observed the prevalence of fault domination in Space. Although fault domination gives some confidence in the ability to discard failures caused by multiple faults, in order to simplify experimentation, a non-trivial number of failures will be discarded and thus may bias experimental results. Additionally, subject-program choice may play a significant factor in such decisions, and may even prohibit evaluation in cases where all or almost all failures are caused by multiple faults.

To RQ3, we assess: Experimental procedures that prescribe the elimination of failures caused by multiple faults are likely to introduce bias in the results and may even prohibit evaluation.

B. Seeking the Pure Cluster

Ideally, clustering techniques would enable the production of clusters that are pure (i.e., all failures are caused by exactly the same faults) and the composition of those clusters contains only Singleton failures (i.e., failures that fail due to a single fault). Researchers have found that failures that contain multiple faults are more difficult and time-consuming to debug (e.g., [7, 8, 13, 22, 23]). Unfortunately, our subjects demonstrated large quantities of non-Singleton failures.

In order to facilitate research to produce pure, single-fault clusters, we investigated the possibility to heuristically identify non-Singleton (i.e., Combo, Tie, and New) failures. If such a heuristic could be found to be adequately effective, a "pre-clustering" might be performed in order to remove non-Singleton failures from clustering, and thus eliminate the concern for the third assumption (and its impact on experiments addressed by RQ3) and enable more pure clusters, and thus easier debugging.

C. Looking for Purity in All the Wrong Places

To assess the potential for future lightweight heuristics that can distinguish Singleton failures from non-Singleton failures, we performed a pilot study to examine a measurable execution characteristic, which we speculated may be a useful indicator. The statement-instance count for each test case was captured. We define a statement-instance as a single execution of a single statement: in other words, the total statement-instance count can be computed by summing the statement-profile count for every statement. Our intuition was that test-case failures that have higher total statement-instances (i.e., longer-running executions, which executed more code) were more likely to have executed more faults. If our intuitions proved true, we speculated, we may be able to automatically and heuristically produce pre-clustering techniques that discard test-case executions that were more likely to be caused by more than one fault.

With these intuitions, we examined our subjects' data at the 10-fault level and classified them according to our output oracle for fault causality. We present the data for our pilot study in Table III, which shows the average and standard deviation statement-instance counts for each Singleton, Tie, and Combo failures.

The results show that for Gzip, Combo failures do indeed show longer executions, but strangely, Singleton failures were also lengthy, and Ties were remarkably short. For Sed, the results were quite different: Combos were the shortest, and Singletons and Ties were, on average, almost five times as long. And, for Space, the results were yet again different:

Table III: Average and standard deviation of execution length for each failure category at the 10-fault level.

	Gzip	Sed	Space
Singleton$_{avg}$	205,669	1549	6175
Singleton$_{std}$	63,994	899	3168
Tie$_{avg}$	304	1543	5408
Tie$_{std}$	218	568	2556
Combo$_{avg}$	323,827	361	.
Combo$_{std}$	137,516	172	.

Singletons were long, and so too were Ties (and Combo failures did not exist at the 10-fault level).

D. Divining Purity Clues

Despite our inability to find a generalizable early indicator of the number of faults causing failure, the results are nevertheless interesting. These results demonstrate a clear difference in the length of execution for each of the categories for at least two of the three subject programs. Although the results do not follow our intuition, this within-subject differentiation across categories is a potentially promising aspect to be further explored in future research, and may be a factor that should be considered in the effort to produce pure failure clusters. The results here hint at the possibility to unravel these behaviors toward that goal.

VII. Threats to Validity

One difficulty in constructing external validity for this work is the generalizabilty our results. We only use three subjects and due to the diversity of software sizes and complexity, it is difficult to assure that our results create a representational set. However, while we only use three subjects, all are real-world programs. Further, we executed more than 4 billion comparisons with roughly 105 days of computational time. This large quantity of data gives us confidence that our results are similar to programs of relative size and complexity. Another complication with our external validity is that some results gained seem fairly program specific. However while each programs gave different results, all indicate similar answers for our three research questions.

A difficulty in creating construct validity for this work lies in our use of mutants. Although the majority of our faults are real, a small subset are mutants to create enough diversity to enable our experiment. This work investigates the impact of fault behavior and fault interaction on failure clustering, and by definition, mutants are not true faults. However, a recent study by Ali *et al.* [1] found that mutants were often found with the same frequency as real faults when using fault localization tools, which utilize the same program spectra as this work. Further, Offutt *et al.* [18] created a framework to create mutants which are representational of real faults. To ensure our mutants behaved similarly to real faults, we follow the methods outlined by Offutt.

VIII. Conclusions and Future Work

In this paper, we present an empirical study, which demonstrates the effects of multiple faults on failure behavior. We examine the assumptions made by past failure-clustering research, describe a novel experiment, classify failure behaviors, present a potential oracle for failure clustering, and provide a pilot study that examines a potential factor for promoting failure-cluster purity.

In our analysis of the simplifying assumptions of previous research, we found that *program spectra can perform quite well at classifying semantic behavior.* We found that profiles were able to fairly accurately approximate the type of behavior (singleton, tie, combo, or new failure), though often produce inaccurate fault-causality determinations, which may indicate some limitations for its use in failure clustering. We *identified dangers in the assumption of cluster purity*, and we found evidence that *ignoring multiple-fault failures in research experiments may introduce bias and produce results that do not generalize.*

Further, our demonstration of output as a potential failure-causality oracle allows for an automated analysis of semantic program behavior and provides new insights into how to improve failure clustering.

Lastly, our pilot study offers promise that lightweight heuristics — such as statement-instance execution count — may provide indications that enable future techniques to filter test-case failures that are likely to be caused by multiple faults. If such a pre-clustering stage is made possible, cluster purity (and the resulting benefits for debugging) may be more achievable. These results open new avenues for research along with providing specific direction for how failure clustering can improve.

While we examined software behavioral patterns with respect to failure clustering, more research is needed. We plan on performing similar experiments on more subjects to gain a better understanding and attempt to locate software-behavior generalizations. Further work also needs to be done on lightweight heuristics that can be used in a pre-clustering step to remove failures caused by multiple faults. We intend to perform a comparative analysis of multiple heuristics and measure their contributions.

IX. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Research Award.

References

[1] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang. Evaluating the accuracy of fault localization techniques. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[3] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.*, 1989.

[4] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2009.

[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering*, 2001.

[6] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2001.

[7] N. DiGiuseppe and J. A. Jones. Fault interaction and its repercussions. In *Proceedings of the International Conference on Software Maintenance*, 2011.

[8] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 9th ACM/IEEE International Symposium on Software Testing and Analysis*, 2011.

[9] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, 2004.

[10] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 1998.

[11] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, 2007.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*, 2005.

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

[14] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd international conference on Software engineering*, 2000.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[16] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2006.

[17] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of 10th European Software Engineering Conference and 13th Foundations on Software Engineering*, 2005.

[18] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5, 1996.

[19] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the International Conference on Software Engineering*, 2003.

[20] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8, 1999.

[21] A. Podgurski and C. Yang. Partition testing, stratified sampling, and cluster analysis. In *Proceedings of the symposium on Foundations of software engineering*, 1993.

[22] M. Srivastav, Y. Singh, C. Gupta, and D. Chauhan. Complexity estimation approach for debugging in parallel. In *Computer Research and Development, 2010 Second International Conference on Computer Research and Development*, 2010.

[23] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, 2006.