Mutations: How close are they to real faults?

Rahul Gopinath
Oregon State University
Email: gopinath@eecs.orst.edu

Carlos Jensen
Oregon State University
Email: cjensen@eecs.orst.edu

Alex Groce
Oregon State University
Email: agroce@gmail.com

***Abstract*—Mutation analysis is often used to compare the effectiveness of different test suites or testing techniques. One of the main assumptions underlying this technique is the Competent Programmer Hypothesis, which proposes that programs are very close to a correct version, or that the difference between current and correct code for each fault is very small.**

Researchers have assumed on the basis of the Competent Programmer Hypothesis that the faults produced by mutation analysis are similar to real faults. While there exists some evidence that supports this assumption, these studies are based on analysis of a limited and potentially non-representative set of programs and are hence not conclusive. In this paper, we separately investigate the characteristics of bug-fixes and other changes in a very large set of randomly selected projects using four different programming languages.

Our analysis suggests that a typical fault involves about three to four tokens, and is seldom equivalent to any traditional mutation operator. We also find the most frequently occurring syntactical patterns, and identify the factors that affect the real bug-fix change distribution. Our analysis suggests that different languages have different distributions, which in turn suggests that operators optimal in one language may not be optimal for others. Moreover, our results suggest that mutation analysis stands in need of better empirical support of the connection between mutant detection and detection of actual program faults in a larger body of real programs.

I. Introduction

Mutation analysis is a fault injection technique originally proposed by Lipton [1] and is often used in software testing. It is used as a means of comparison between different testing techniques [2], as a means of estimating whether a test suite has reached adequacy [3], and as a means of emulating software faults for the purposes of estimating software reliability [4]. In fact, the validity of mutation analysis is an assumption underlying considerable work in other suite evaluation techniques, such as code coverage criteria [5].

Mutation analysis involves systematic transformation of a program through introduction of first order syntactical changes, and determines whether tests can distinguish the mutated code from the original (presumed correct) source code. A mutation score, which measures how many mutants were distinguished from the original code by at least one test, is used as a measure of the effectiveness of the test suite [6] because it is believed to correlate well with the effectiveness of the test suite in detecting real faults [7].

Mutation analysis relies on two assumptions: (1) the Competent Programmer Hypothesis and (2) the Coupling Effect [8]. The Competent Programmer Hypothesis suggests that the version of program produced by a competent programmer is close to the final correct version of a program, while the Coupling Effect claims that a test suite capable of catching first order mutations will also detect higher order mutations that contain these first order mutations. In practice, a strong Competent Programmer Hypothesis (that for programs of any size the initial version is syntactically close to correct) is fairly obviously incorrect for large programs. However, mutation analysis only rests on a weaker version: that the Competent Programmer Hypothesis holds with respect to each individual fault.

For mutation analysis to be successful, the mutants it produces should ideally be similar in character to the faults found in real software. This property has been used in practice by many researchers [2], [9]–[11], for generating plausible faults[1]. While this has been investigated by a few researchers [7], [12]–[14], the evidence is largely based on the real faults from a single program. Further, except for the study by DeMillo et al. [12], the similarities investigated were constrained to the error trace produced [13], and the ease of detection [7], [14].

The existing body of work, especially by DeMillo et al. underscores the necessity of further studies, with a much larger sample of programs, especially in light of the proliferation of programming languages and availability of open source software. We expand the work by DeMillo et al. — which investigated 296 bug-fixes from a single program (TeX) — to faults from 5,000 programs in four different programming languages (C, Java, Python, and Haskell) — a total of 240,000 bug-fixes.

We also extended our investigation to localized patches that contain just a single modification in them, which should contain a simple fault, and hence should be similar to those produced by mutation operators. The incidence of bug fixes and localized changes in the overall population is summarized in Table I. The details of our data collection are summarized in Section III.

Our analysis, summarized in Section IV, suggests there is a huge variation in the incidence of different classes of mutations, which are dependent on the kind of programming language chosen. Further, there are a significant number of change patterns which are different from the single token change captured by standard mutation operators. Hence, using all mutations equally would not be representative of the real

[1]These articles do not explicitly call upon *the Competent Programmer Hypothesis*, but we believe that their use of mutation analysis-generated faults instead of a fault seeding approach based on fault distributions is essentially based on the Competent Programmer assumption.

faults in software, and most real faults do not match any mutation operator. Further, the choice of mutation operators also needs to be guided by the programming language used. We provide a basis for future investigations in this regard.

The data for this study is available at Dataverse [15].

II. Related Work

Our work is an extension of the work done by DeMillo et al. [12], Daran et al. [13] Andrews et al. [7] and Namin et al. [14] which attempts to relate the characteristics of mutation operators to that of real faults. In the remainder of this paper, we use the term mutation operator to indicate, in context, either actual mutation operators applied during mutation analysis, or the actual small changes made to code in bug fixes.

DeMillo et al. [12] were the first researchers to investigate the representativeness of mutations to real faults empirically. The investigated the 296 errors in TeX, and found that 21% were simple faults (single token changes), while the rest were complex errors.

Daran et al. [13] investigated the representativeness of mutation operators to real faults empirically. They studied the 12 real faults found in the program developed by a student, and 24 first order mutants. They found that 85% of the mutants were similar to the real faults. While this paper highlights the importance of relating the actual mutations to real faults, they were constrained by their small sample size, a single program. More importantly, the conclusions were based on only 12 real faults.

Another important study by Andrews et al. [7] investigated the ease of detecting a fault for both real faults and hand seeded faults, and compared it to the ease of detecting faults induced by mutation operators. The ease is calculated as the percentage of test cases that killed each mutant. Their conclusion was that the ease of detection of mutants was similar to that of real faults. However, they based this conclusion on the result from a single program, which makes it unconvincing. Further, their entire test set was eight C programs, which makes the statistical inference drawn liable to type I errors. We also observe that the programs and seeded faults were originally from Hutchins et al. [16] where the programs were chosen such that they were subject to certain specifications of understandability, and the seeded faults were selected such that they were neither too easy nor too difficult to detect. In fact they eliminated 168 faults for being either too easy or too hard to detect, ending up with just 130 faults. This is clearly not an unbiased selection. More seriously, this selection can not tell us anything about the ease of detection of hand seeded faults (because the criteria of selection itself is confounding).

These acute problems were highlighted in the work of Namin et al. [14] who used the same set of C programs, but combined them with analysis of four more Java classes from JDK. They used a different set of mutation operators on the Java programs, and used fault seeding using student programmers on them. Their analysis concluded that we have to be careful when using mutation analysis as a stand-in for real faults. They found that programming language, the kind of mutation operators used, and even test suite size has an impact on the relation between mutations introduced by mutation analysis and real faults. In fact, using a different mutation operator set, they found that there is only a weak correlation between real faults and mutations. However, their study was constrained by the paucity of real faults available for just a single C program (same as Andrews et al. [7]). Thus they were unable to judge the ease of detection of real faults in these Java programs. Moreover, the students who seeded the faults had knowledge of mutation analysis which may have biased the seeded faults (thus resulting in high correlation between seeded faults and mutants). Finally, the manually seeded faults in C programs, originally from Hutchins et al. [16], were confounded by their selection criteria which eliminated the majority of faults as being either too easy or too hard to detect.

These previous efforts prompted us to look at evaluating mutation analysis from a different direction. We wondered if the ease of detection was the only relevant criteria when comparing mutation operators and real faults. Why not compare them directly, by comparing the syntactical patterns of both? Even if it is argued that there may be interdependent changes that make it difficult to compare, we can still get a reasonable result by restricting our analysis to small localized changes that are limited to a single change in a single file.

There has been other research in related fields that takes a similar approach. Christmansson et al. [17], [18] analyzed field data to come up with an error model that mimics real faults, and used these to inject errors to simulate faults. Their study classified the defects based on their *semantics* using Orthogonal Defect Classification [19]. While this research is useful in its domain, it is inapplicable to mutation analysis, which is primarily a syntactical technique. We want to easily generate bugs that look like and feel like real bugs with relatively little context. We certainly don't want to understand the semantic content, e.g. whether a mutation introduces a functional error, an algorithmic error, or a serialization error (classifications of ODC).

Duraes et al. [20], [21] analyzed the change patterns in 9 open source C projects, and collected a total of 668 faults. They adapted Orthogonal Defect Classification to provide a finer classification of errors into missing, wrong, and extraneous language constructs. They find 64% of the faults were due to missing constructs, 33% due to changes, and only 2.7% were due to extraneous constructs. While this study is the closest to our approach, they are also limited by concentration on a single language (C), and a comparatively small number of faults (532 faults) to ours. Further, while the classification they provide is finer grained than ODC, it is still at a higher level than the typical mutation operator implementations. In comparison, our analysis of a larger set of data indicates that addition and deletion were relatively similar in prevalence, while changes dominated in all the languages we analyzed.

A larger study of similar nature by Pan et al. [22] extracted 27 bug fix patterns from the revision history of 7 projects, which cover up to 63.3% of the total changes, and computed the most frequent patterns. Their study, like the previous one

```
class MyClass {
  int loop(int counter) {
    int i = 0;
    while(i < counter) {
*      <count = count +1 | i++ >;
    }
*   return <count | i>;
  }
}
```

Fig 1: An example patch

by Duraes analyzed the patterns from a higher level than typical mutation operators, and hence is not directly applicable to mutation analysis. Further, their analysis is restricted to Java programs which, along with the limited number of projects reduces their applicability.

Our research uses machine learning techniques to automatically classify patches as bug-fix or non bug-fix based on an initial set of changes that we manually classified. Mokus et al. [23] first used a classifying approach that relied on the presence of keywords. They classified changes into categories of fixes, refactoring and features.

Another related study is by Purushotam et al. [24] who analyzed the change history of a large software project, specifically focusing on small (one line) changes. They were interested in finding the patterns of changes that can induce an error with high probability in software. The study is interesting for the distribution they found for small changes, which we also consider. They found that 10% of the total changes involved a single line of code, and 50% were below 10 lines, dropping to 5% for those above 50 lines. They also suggest that most changes involved inserting new lines of code. Our study found that small (localized) changes can range from 26.2% in C to 62.7% in Haskell (see Table I).[2]

	C	Java	Python	Haskell
Localized changes	26.241	27.278	43.770	62.685
Bug-fixes	44.314	29.612	34.395	31.009
Localized bug-fixes	10.464	9.053	16.541	16.486

TABLE I: Localized changes and bug-fixes prevalance in %

III. Methodology

We were primarily interested in finding answers to the following questions.

Q Can we find empirical evidence for or against the Competent Programmer Hypothesis? Can we find any support for the assumption that real faults look like those produced by typical mutation operators? Can we do this by analysis of patches (whether it be the complete set of changes or a subset that is identified as bug-fixes or localized small bug-fixes that should be fixes for simple faults)?

Q How much of an effect does programming language have on the distribution of change patterns? Can we extend the results from the distribution of syntactical changes or fault patterns in one language to another? We especially want to make sure that we compare apples to apples here and look at a common set of mutation operators across different languages.

Q What are the most common mutation operators? Are they different from the traditional mutation operators that are commonly used? Can we provide any guidance to future implementors of mutation tools so that mutation operators produced look similar to real faults?

We wanted our results to be applicable to a wide variety of languages and ensure that our analysis did not suffer from bias for a particular language group. We chose four languages, each representative of an important kind of development. We chose C as the dominant systems programming language, widely used in the most critical systems for testing. Java was chosen as a popular programming language used in enterprise applications. The choice of Python was driven by its status as one of most popular languages in the dynamically typed community, and its use in many domains including statistics, mathematics, and web development. Finally Haskell, while less popular than the other three, is a popular strongly typed functional language preferred in academic research.

To ensure that we had a relatively unbiased population from each language, we searched for projects in Github [25] with criteria `stars :>= 0` and filtered by the language side bar. We used this criteria since this is a nil-filter—the stars start from '0'—and hence no project was excluded. This search resulted in 1850 projects for C, 1128 for Java, 1000 for Python, and 1393 for Haskell.

A. Classifying patches

Each project from Github came with its entire revision history, which is accessible as a set of patches. To answer our research questions we had to differentiate between bug-fixes—where some pre-existing fault was fixed—and patches that were not bug-fixes. Since we lacked resources to manually classify our entire dataset, we made use of machine learning techniques. We manually classified 1200 patches as bugs or non-bugs for each of the languages. Out of these 4800 classified patches, we used 4000 to train our classifiers, and used 800 (200 from each) to cross validate our trained classifiers. We achieved an accuracy of 78.87% using CRM114 classifier [26] which gave us the highest accuracy out of Bayesian, Bishop, LSI, and SVM classifiers. The acceptance accuracy for bugs was 73.19%, while the rejection accuracy was 81.24%. We got overall better results by combining training examples from all the languages than by training on each in isolation. For example, using individual training, accuracy obtained for Java was 76.5% (acceptance: 64.4%, rejection 81.5%), 77% (acceptance: 76.8%, rejection: 77.1%) for Python, 71% (acceptance: 69.7%, rejection: 71.8%) for C,

[2]The percentages given in Table I are overlapping. The set of changes is divided into bug-fixes and feature updates, and orthogonal to that, as localized (single line), and non-localized (mult-line) changes

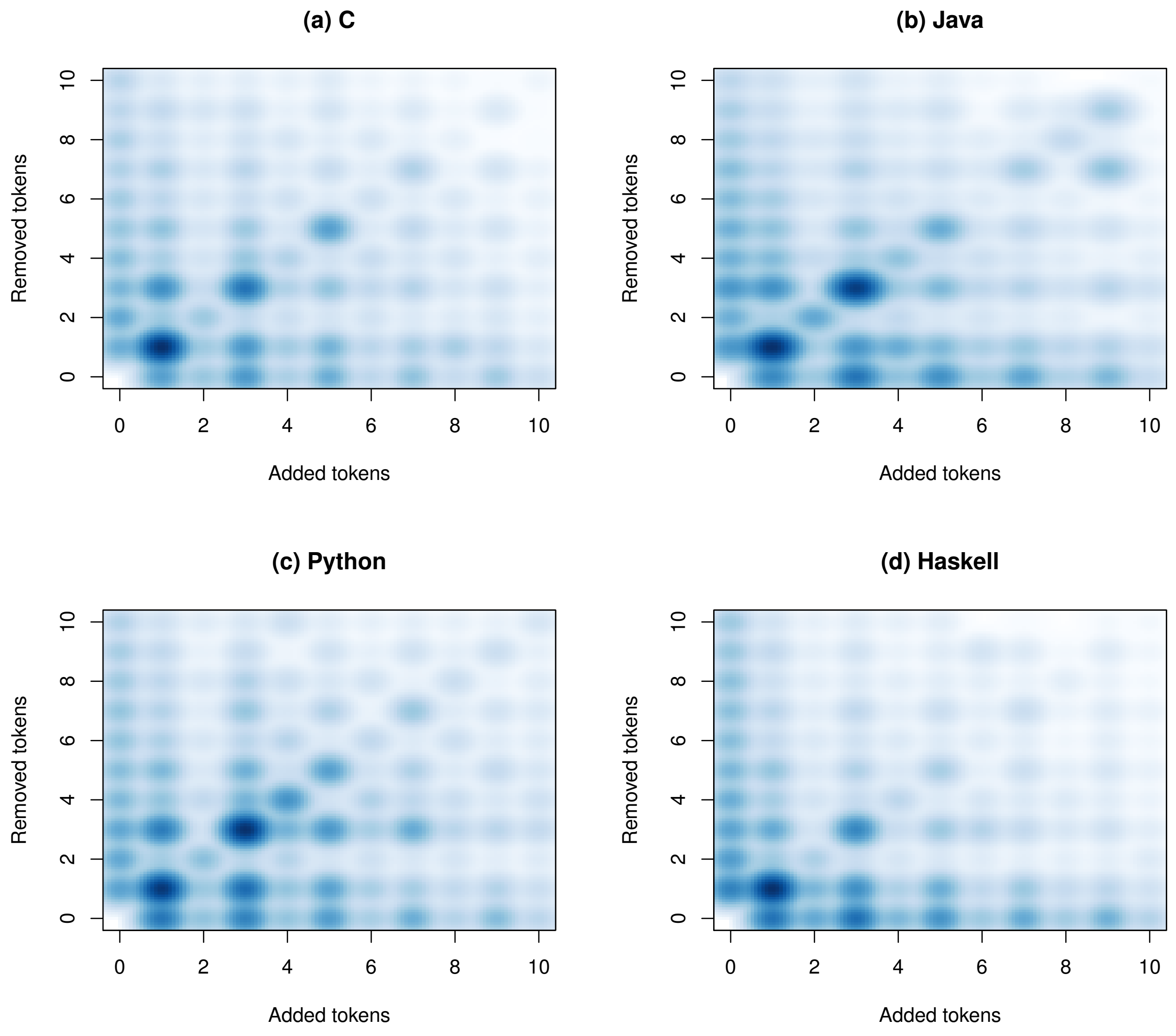

Fig. 2: Density plot of added vs removed number of tokens in replacement changes for full distribution

and 76% (acceptance: 70.9%, rejection: 76.9%) for Haskell. This rate is close to the rate obtained by leading research [27] in classification of bugs and non-bugs, which obtained an accuracy between 77% to 82% using change tracking logs.

After classification, we found that 44.31% of commits in C were bug fixes, 29.6% for Java, 34.39% for Python and 31.01% for Haskell. The distribution is given in Table I.

B. Generating normalized patches

Next, we wanted to collect the patches in each project, after discounting for the differences due to whitespace and formating changes. To accomplish this, for each project, the following procedure was applied to collect normalized patches for each projects.

First, the individual revisions of files were extracted, and they were cleaned up by stripping comments, joining multi-line statements, and hashing string literals. These were then re-formated by passing through a pretty-printer. This removed the differences due to addition or removal of comments or due to formating changes. Next, successive revisions were diffed against each other using a token-based diffing algorithm, and the patches thus produced were collected.

C. Sampling

We were interested in finding the distribution of token changes, unbiased by effects of project size, developer or project maturity, or other unforeseen factors. For statistical inference to be valid to a given population, the observations from which the inference is drawn should be randomly sampled from the targeted population. For this purpose, we decided to generate random samples of patches from the projects we had.

We generated 10 random samples, with each containing 1,000 patches for combinations of the following sets—whether they are bugs or not (bug, nonbug, all), whether the bug fix was localized or otherwise (small, all), and each of the languages

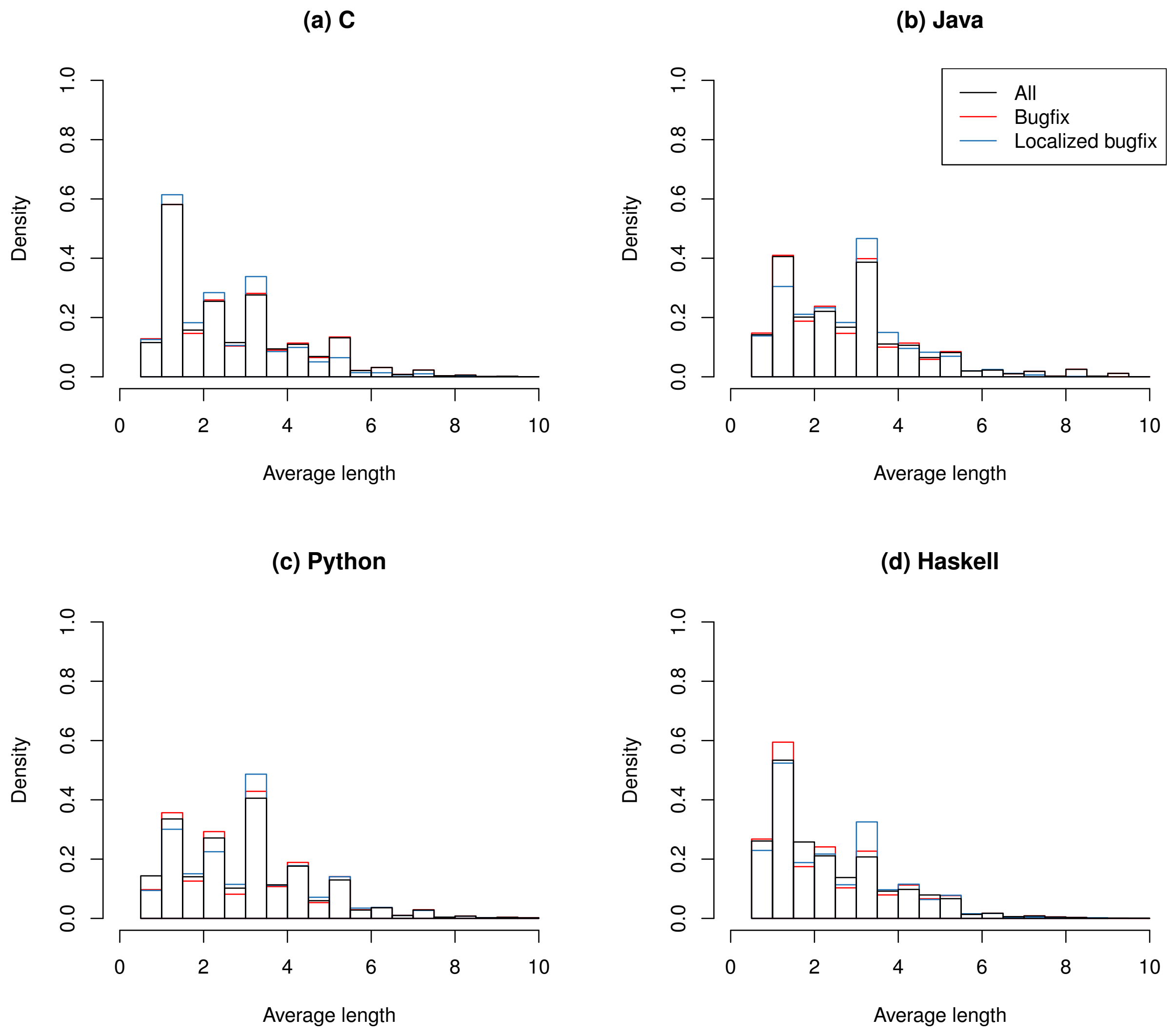

Fig. 3: Average length of added vs removed tokens

(C, Java, Python, Haskell). This generated $3 \times 2 \times 4 \times 10 = 240$ samples (240,000 patches, but some may be repeated in multiple samples).

D. Collecting chunks

Each patch is composed of multiple segments in the file where some text was removed, or added, or some text was replaced (remove + add). An example patch is given in Figure 1. This patch contains two chunks. The first chunk is in line 6 and involves removal of 5 tokens, and addition of two tokens. The second one is in line 8, and involves removal of a single token and addition of another. These chunks were extracted and processed further by eliminating syntactic sugar elements such as parenthesis[3], commas, etc. and collapsing strings to their checksums for easier processing. The tokens thus identified were then passed through a lexical identifier which replaced each lexical element by its class. We use chunk and change interchangeably in this paper.

[3]We may therefore miss some changes that involved semantically meaningful parenthesis additions, but this is also not a standard mutation operator.

E. Identifying mutation operators

For ease of comparison between different languages, we chose to use a single set of mutation operators applicable across different programming languages. We started with the original 77 operators proposed for the C programming language by Agrawal et al. [28]. We then removed operators that could not be matched from the context of changes. The mutation operator variants that were mirror images were collected under a single name. Further, a few mutation operators were discarded because they were inapplicable in other languages. The mutation operators were further grouped into classes for analysis. Further, the added, removed and changed patterns that could not be classified under any existing mutation operators were grouped in their own categories, resulting in ten mutation operator categories. A complete listing is provided in Table VI. The distribution of average token count is provided in Table XIII where $\max \epsilon$ is the largest percentage detected in the remaining token bins.

	C	Java	Py	Hs
Add:oth	16.483	17.757	17.082	30.114
Change:oth	32.219	25.115	29.443	32.363
Rem:oth	13.372	14.526	12.215	23.263
Twiddle	0.219	0.057	0.047	0.070
Const	5.425	2.515	6.205	2.270
Var.Const	4.981	2.045	3.372	1.045
Var	26.641	37.744	31.487	10.721
BinaryOp	0.119	0.031	0.033	0.026
Negation	0.428	0.186	0.098	0.102

TABLE II: Summary of mutation operators for all changes

	C	Java	Py	Hs
Add:oth	28.781	29.805	22.699	31.607
Change:oth	23.352	21.078	26.439	29.294
Rem:oth	12.877	13.139	11.614	19.742
Twiddle	0.614	0.286	0.094	0.160
Const	12.126	13.086	21.442	8.545
Var.Const	5.533	4.095	4.384	2.240
Var	14.979	17.279	12.881	7.979
BinaryOp	0.453	0.307	0.158	0.054
Negation	0.932	0.728	0.189	0.332

TABLE III: Summary of mutation operators for localized changes

	C	Java	Py	Hs
Add:oth	15.705	19.519	18.714	29.788
Change:oth	32.823	27.470	29.971	33.395
Rem:oth	13.284	15.418	13.529	23.352
Twiddle	0.126	0.049	0.010	0.020
Const	4.242	2.741	7.614	2.101
Var.Const	3.648	2.240	4.808	1.511
Var	29.776	32.314	25.094	9.726
BinaryOp	0.058	0.013	0.019	0.011
Negation	0.296	0.222	0.229	0.086

TABLE IV: Summary of mutation operators for bug-fixes

	C	Java	Py	Hs
Add:oth	29.861	33.103	26.629	33.162
Change:oth	21.686	19.032	28.097	28.149
Rem:oth	12.168	13.082	11.308	18.696
Twiddle	0.852	0.392	0.161	0.259
Const	11.899	10.779	14.226	8.060
Var.Const	5.359	3.634	4.461	2.156
Var	15.856	18.366	14.678	8.773
BinaryOp	0.646	0.326	0.163	0.130
Negation	1.198	1.093	0.165	0.490

TABLE V: Summary of mutation operators for localized bug-fixes

IV. Analysis

According to a naive interpretation of the Competent Programmer Hypothesis, a majority of changes we see should be simple and localized and look like traditional mutants. The traditional mutation operators all operate on changing a single token. In order to investigate whether this is the case, we plotted the number of tokens added versus the number of tokens deleted in each change. The result of this analysis is shown in Figure 2 for each language. This figure shows that while there are a significant number of changes that are one token (ϵ changes), there is a large number of changes that include more than one token in both added and deleted counts. We note that these are not captured by the traditional mutants.

A second concern we had was about the difference between the distributions of bug-fixes and other changes, and the impact of different languages. We plotted the histograms of average change lengths (computed as the average of added and removed tokens per change) for each of the languages. This is shown in Figure 3. The plot indicates that bug-fixes do not significantly differ from the main change patterns. However, the figure indicates a difference in distribution between different languages.

To confirm our finding, we use statistical methods. Students two-sample t-test is a statistical test that checks whether two sets of data differ significantly. We use it to determine whether essential characteristics of changes differ between bug-fixes and other commits, and between different languages. We also provide a comparison with difference in mean between the two distributions obtained by running Students t-test. These were significant for $p < 0.05$ except where indicated. The difference between the bug-fix changes and others are tabulated in the Table VII. We note that the difference between bug-fix and others changes is universally very low for all four programming languages, confirming our initial finding from Figure 3.

	Bug-fix	Nonbug	*SBug	LowCI	HighCI	MeanD	Pval
C	4.19	4.17	3.08	-0.06	0.09	0.02	0.65
Java	4.22	4.18	3.18	-0.07	0.14	0.03	0.53
Python	4.39	4.22	3.91	0.07	0.27	0.17	0.00
Haskell	4.48	4.46	3.93	-0.08	0.13	0.02	0.69

TABLE VII: Average tokens changed between bug-fixes and other changes

Next we compare the distributions of tokens between different languages. The mean difference from Students t-test is given in Table VIII. These were not significant for the pair C and Java, but was significant with $p < 0.05$ for all other language pairs.

	C			Java			Python			Haskell		
	MD	LCI	HCI	MD	LCI	HCI	MD	LCI	HCI	MD	LCI	HCI
C	0	0	0	-0.02	-0.09	0.04	-0.1	-0.2	-0.06	-0.3	-0.4	-0.2
J	-0.02	-0.09	0.04	0	0	0	-0.1	-0.2	-0.03	-0.3	-0.3	-0.2
P	-0.1	-0.2	-0.06	-0.1	-0.2	-0.03	0	0	0	-0.2	-0.2	-0.09
H	-0.3	-0.4	-0.2	-0.3	-0.3	-0.2	-0.2	-0.2	-0.09	0	0	0

TABLE VIII: Mean difference for average tokens changed between different languages (p < 0.05 except C x Java)

We observe here that while the difference between languages seems small, there is a large similarity between C and Java patterns, and Haskell is closer to Python than others. This seems somewhat intuitive if we consider that C and Java are descendants of the Algol family, while Python to a large part supports functional programming paradigms, of which Haskell is an exemplar.

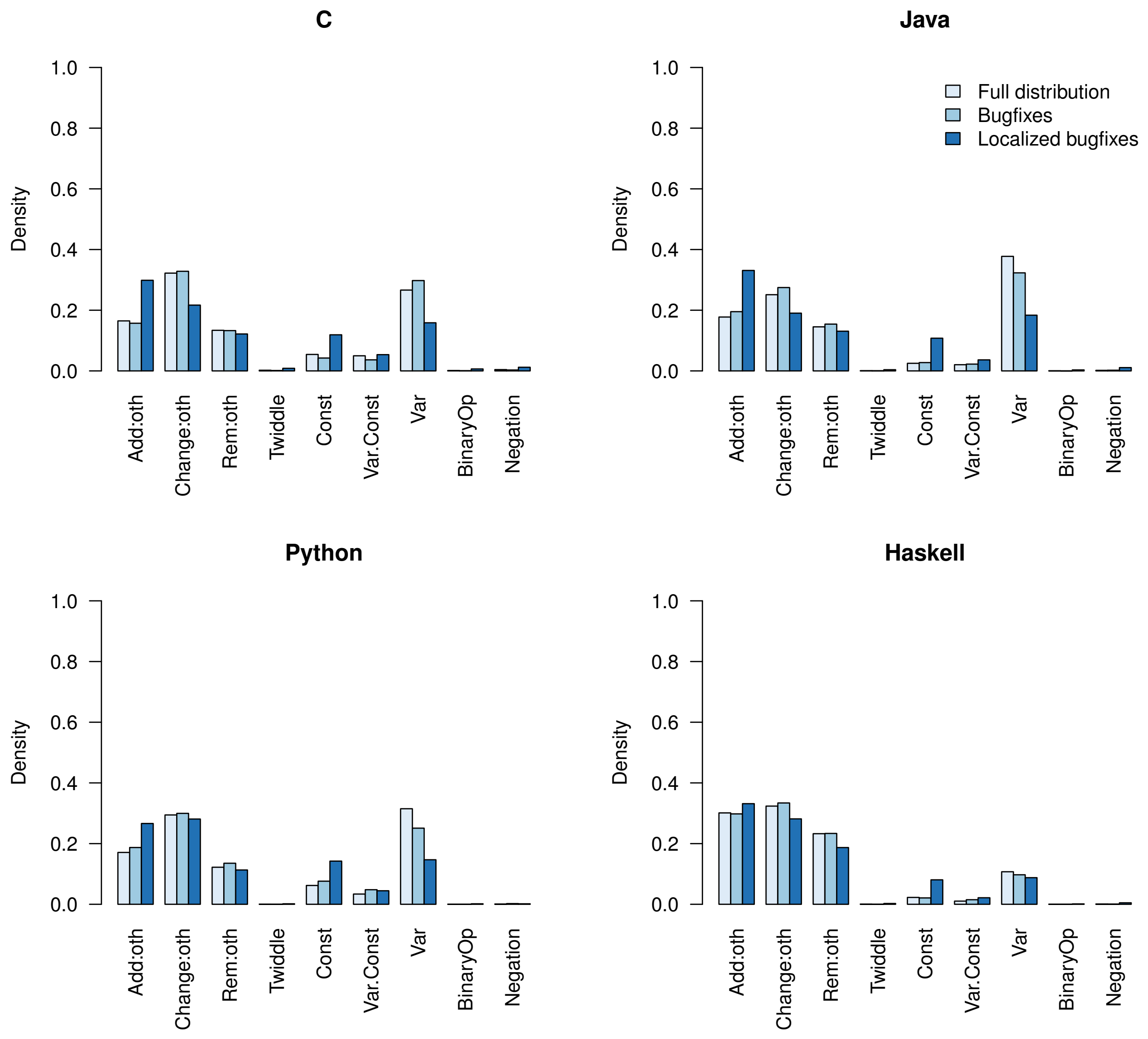

Fig. 4: Relative occurrence of mutation operators

A. Mutation operator distribution

A major part of our analysis is the comparison of mutation operator distributions across different languages and kinds of patches. We analyze the difference between the complete distribution, that of just bug-fixes alone, and localized bug-fixes. This is visualized in Figure 4. The summary of mutation operators are also provided as in Table II, and a summary of mutation operators for localized changes are given in Table III. Finally, Table IV tabulates the distribution of mutation operators for bug-fixes, and Table V the distribution of localized bug fixes. The mutation operator class explanations are given in Table VI.

B. Regression Analysis

Regression analysis is a statistical process that helps us to understand the relative contributions of different variables. Here, we make use of regression analysis to assess the contribution of class of mutation operator, programming language, and the kind of change (bug-fix or otherwise) to the prevalence of the mutation operator.

First we run our analysis for the complete distribution, analyzing which model fits best. Next, we run our analysis on only the patches classified as bug-fixes, and finally on those localized bug-fixes. We use the keys given in Table IX to refer to the variables in the model.

Variable	Name
P	Prevalence of mutation operator
O	Operator (Mutagen operator)
L	Language
B	Bug-fix or otherwise

TABLE IX: Explanations of model variables

1) Complete Distribution: We started with the full model containing the full interactions between all given variables.

$$\mu\{P|O,L,B\} = O+L+B+O\times L+O\times B+L\times B+O\times L\times B$$

Class	Explanations
Add:oth	Added tokens not including twiddle, negation, unary and statement mutation operators
Change:oth	Replaced tokens not classified under any of other changes
Rem:oth	Removed tokens not including twiddle, negation, unary and statement mutation operators
Twiddle	Addition or removal of +/- 1 or the use of unary increment or decrement operators
Const	Change in constant value
Var.Const	Changing a constant to a variable or reverse
Var	Changing a variable to another variable
BinaryOp	Changing a binary operator to another
Negation	Negation of a value (includes arithmetic, bitwise, and logical)

TABLE VI: Explanations of mutation operator classes

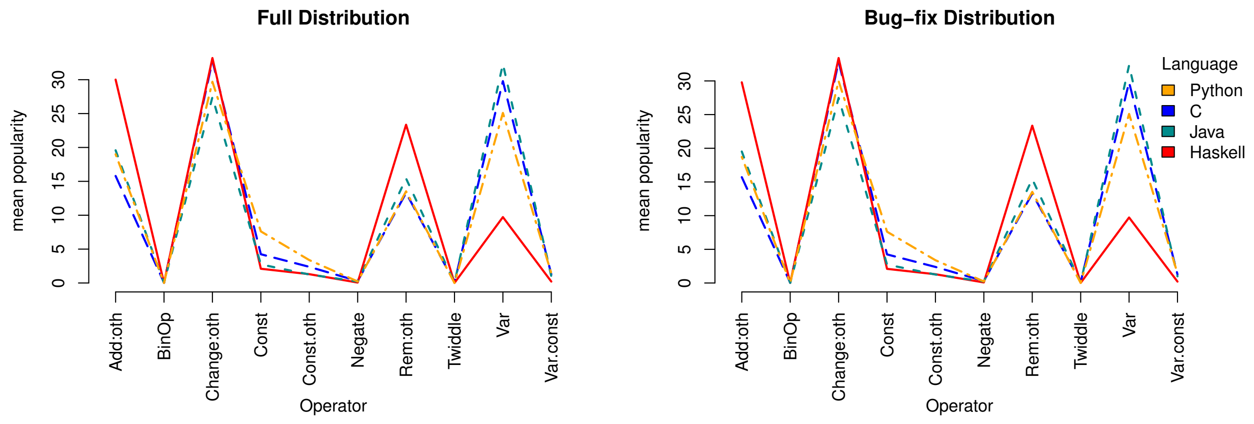

Fig. 5: $Op \times Language$ interaction

However, not all the variables were significant contributors towards the prevelance of the mutation operator. We sequentially eliminated non-significant variables resulting in

$$\mu\{P|O, L, B\} = O + L + O \times L$$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	101703.67	11300.41	1724.49	0.0000
Language	3	0.00	0.00	0.00	1.0000
Op:Language	27	10669.20	395.16	60.30	0.0000
Residuals	760	4980.20	6.55		

TABLE X: Results of the model fit for complete distribution

This provides us the best fit given in Table X, and has correlation coefficient R^2 =0.955. This suggests that a patch has similar change patterns irrespective of whether it is a bug-fix or otherwise. This is also suggested by the interaction plot between mutation operator bug-fixes given in Figure 6. Further, we also see the evidence of non additive interaction between mutation operators and language in Figure 5 and in the ANOVA results in Table X.

2) Localized Change Distribution: Next, we analyze the localized changes. These are changes that modify only a single file in a single part such that the change is restricted to a single chunk. We investigate localized changes because they are closest to the changes produced by mutation operators.

$$\mu\{P|O, L, B\} = O + L + B + O \times L + O \times B + L \times B + O \times L \times B$$

Interestingly, for localized distribution, the interaction between mutation operators, language, and bug-fix is significant, which makes the full model the one with the best fit. The model has R^2 =0.955, and the model ANOVA is given in Table XI.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	79759.07	8862.12	4088.18	0.0000
Language	3	0.00	0.00	0.00	1.0000
Bug	1	0.00	0.00	0.00	1.0000
Op:Language	27	5319.62	197.02	90.89	0.0000
Op:Bug	9	1378.17	153.13	70.64	0.0000
Language:Bug	3	0.00	0.00	0.00	1.0000
Op:Language:Bug	27	860.12	31.86	14.70	0.0000
Residuals	720	1560.77	2.17		

TABLE XI: Results of the model fit for localized distribution

3) Localized Bug-fix Distribution: The previous result induced us to also look at the distribution of localized bug-fixes. These are localized changes that were also identified as bug-fixes. This results in a very close fit model with a coefficient of correlation R^2 =0.991. The result of ANOVA is given in Table XII.

$$\mu\{P|O, L\} = O + L + O \times L$$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	42910.04	4767.78	4880.83	0.0000
Language	3	0.00	0.00	0.00	1.0000
Op:Language	27	2002.98	74.18	75.94	0.0000
Residuals	360	351.66	0.98		

TABLE XII: Results of the model fit for complete distribution

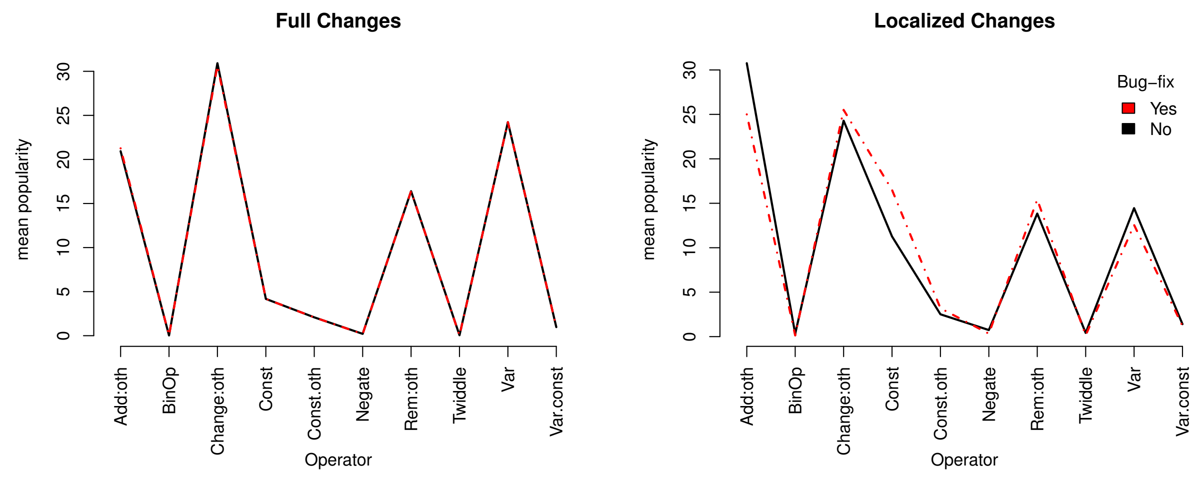

Fig. 6: $Op \times Bug$ interaction

	C all	J all	P all	H all	C bug	J bug	P bug	H bug
0.5	4.9	6.1	6.2	10.5	5.5	6.3	4.2	10.8
1	29.7	23.6	20.5	31.9	30.2	23.9	19.4	34.7
1.5	36.4	32.2	26.5	42.3	36.4	32.0	24.8	41.7
2	47.2	41.7	38.2	50.7	47.5	42.2	37.3	51.4
2.5	52.1	48.9	42.5	56.3	51.9	48.5	40.8	55.5
3	63.9	65.5	59.9	64.6	63.9	65.7	59.2	64.7
3.5	67.9	70.3	64.8	68.3	67.7	69.9	63.7	67.8
4	72.6	74.9	72.3	72.2	72.5	74.8	71.8	72.4
4.5	75.5	77.6	74.9	75.4	75.3	77.4	74.1	75.0
5	81.1	81.2	80.4	78.1	81.0	81.0	80.2	78.1
5.5	82.8	83.1	82.6	80.2	82.7	82.9	82.1	79.9
6	85.1	84.6	84.8	82.0	85.0	84.5	84.6	82.1
6.5	86.2	86.0	86.2	83.7	86.1	85.8	85.8	83.5
7	88.1	87.5	88.3	85.1	88.0	87.5	88.1	85.0
7.5	89.0	88.5	89.3	86.6	88.9	88.4	89.0	86.3
8	90.0	90.2	90.4	87.5	89.9	90.1	90.3	87.5
8.5	90.6	90.9	91.3	88.4	90.5	90.8	91.0	88.2
9	91.4	91.9	92.2	89.2	91.4	91.8	92.1	89.1
9.5	92.0	92.3	92.9	90.0	91.9	92.2	92.6	89.8
10	92.7	92.9	93.5	90.6	92.6	92.8	93.3	90.6
$\max_\epsilon$	0.5	0.6	0.6	0.7	0.5	0.6	0.7	0.6

TABLE XIII: Cumulative density(%) of average token changes

V. Results

Our first question was whether we could quantify the Competent Programmer Hypothesis, and verify whether real faults look like mutation operators. Our analysis shows that a significant number of changes are larger than the common mutation operators. A typical change modifies about three to four tokens in all the programming languages surveyed. This increases to addition or removal of about six to eight tokens if we consider addition or removal changes rather than replacement. This increases to five tokens (ten tokens for addition or removal) if we wish to include at least 80% of the real faults, and remains relatively the same even when we consider localized bug-fixes which we had expected to have a distribution similar to that produced by mutation analysis, provided the Competent Programmer Hypothesis is applicable to the mutants produced. This suggests that our understanding of the Competent Programmer Hypothesis, at least as suggested by typical mutation operators, may be incorrect.

This also suggests that in at least one dimension—that of patterns of change—mutations are different from real faults.

Our next effort was to identify whether programming language had any effect on the distribution of mutants, first without considering the different mutation operators, and later, including the differences between mutation operators. Our initial analysis in Table VII and Table VIII indicated that while there are interesting affinities between different languages with regard to the syntactical distance, the effect itself was weak when different mutation operators were not considered. However, once we consider the different classes of mutation operators, as shown in the interaction plot in Figure 5, there is a significant difference in mutation distribution between different programming languages. Finally we conclusively showed using regression analysis that language is an important contributor to the mutation operator distribution in Table X. The result holds true even for localized bug-fixes as shown in Table XII.

This quite strongly suggests that while the average change involves touching about four tokens in all languages examined, different languages encourage different mutation patterns. This suggests that we have to be careful while adapting the results from a different language.

As our final step, we investigated the most common mutation operators. Our results shown in Figure 4, and tabulated in Table II, Table III, and Table V show that different languages have different mutation patterns. Addition, deletion, and the replacement of tokens, especially those that did not come under traditional mutation operators, dominated the mutation operator distribution. This suggests a need for more effective ways to simulate real faults.

An interesting result is also that the distribution we identified between changes of addition and removal (which are somewhat similar in magnitude in each language surveyed) is somewhat at odds with previous research [20] which finds addition of statements to be the highest category (64%), while deletion was small at 2.7%.

Another interesting finding is also the difference between Haskell and other languages in the prevalence of localized changes. We found that for Haskell, more than 60% of the changes were localized changes. Further, we also found that Haskell showed a higher affinity with Python than other languages with regard to change length distribution (Table VIII).

In order to ensure that our automatic classification was not a source of error, we also analyzed manually classified patches separately, the results of which were in line with the results of this paper. Due to lack of space, the results of this analysis are not shown here, but these are available online and summarized in the appendices of this paper [29].

VI. Discussion

Mutation analysis is a very useful technique that is commonly used by researchers as a stand-in for test suite quality. Its theoretical foundations rely on two important concepts: that of the *Competent Programmer Hypothesis*, and the *Coupling Effect*. While the Coupling Effect has been investigated to some extent both theoretically [30], [31] and empirically [32], relatively little research has investigated Competent Programmer Hypothesis.

In this paper, we investigated the Competent Programmer Hypothesis. According to Budd et al. [33] and DeMillo [34], a competent programmer constructs programs that are at most one simple fault [32] away from correctness, and the program, together with the mutants generated—the finite neighborhood $\Phi(P)$—would include the correct program. The implicit claim is that real world programmers are in fact competent, at least most of the time and with regard to a particular program unit and fault. Mutation analysis looks for tests that are adequate relative to Φ.

For the ease of discourse, we define different versions of the Competent Programmer Hypothesis, differentiated by their syntactical finite neighborhood $\bar{\Phi}_\delta(P)$, that is, $\bar{\Phi}_1(P)$ are all the mutants that are at most one token away.

The current generation of mutation operators are overwhelmingly members of $\bar{\Phi}_1$ (excepting a few OO operators for Java [35] and the statement deletion operator [36]). However, our finding is that real faults appear to have a mean token distance of three to four, for all languages examined.

This also brings us to the question of effectiveness of the Coupling Effect on these larger changes. Coupling has been demonstrated to work only using the entire domain of higher order faults. We note that the actual empirical data indicates that real faults occur in such a way as to ensure that the real higher order faults are drawn not from the entire domain, but a much restricted domain of (what we suspect is) a semantic neighborhood of the correct program. It could be that detecting mutants from the $\bar{\Phi}_1$ family does detect 90% or more of mutants from the full $\bar{\Phi}_{\delta>1}$ family, but that real faults fall heavily into the 10% of mutants hard to detect, for example, since the distributions do not resemble the syntactic space of higher order operators. Hence, we suggest further research needs to be done to empirically show that the Coupling Effect holds on real faults, especially on those belonging to $\bar{\Phi}_{\delta>1}$.

We also note that the effectiveness of mutation analysis need not be tied to its theoretical basis. That is, if suites that effectively kill mutants based on $\bar{\Phi}_1$ also have a very high likelyhood, in a purely empirical sense, of also detecting faults very well, that the mutants do not resemble the faults does not matter. However, this itself is in fact the real Coupling Effect that needs to be demonstrated, and as we noted in Section II the current evidence is not strong enough to place mutation analysis on a sound footing.

VII. Threats to Validity

While we have taken utmost care to avoid errors, our results are subject to various threats. First, our samples have been from a single source—open source projects in Github. This may be a source of bias, and our inferences may be limited to open source programs. However we have not seen any evidence of open source programs differing from closed source programs in terms of fault patterns.

Github selection mechanisms favoring projects based on some confounding criteria may be another source of error. However, we believe that the large number of projects sampled more than adequately addresses this concern.

Another source of error is in the bug classification of patches. However, we have followed current research recommendations, and obtained a result in classification that is close to that obtained from current best research in the field.

VIII. Conclusion

One of the main assumptions in mutation analysis is the Competent Programmer Hypothesis, which claims that real programs are very close to correct. If this assumption holds true, then mutation analysis will produce faults that are similar to real faults. However, except for an initial small scale research by DeMillo et al., there has been a lack of research quantifying the syntactic changes involved in real faults, especially with an adequate number of subjects.

Our research attempts to quantify the syntactic differences found in real faults, and finds that faults produced by typical mutation operators are not representative of real faults. Therefore the Competent Programmer Hypothesis, at least from a syntactical perspective, may not be applicable. This suggests that mutation analysis requires further research to place the use of mutants to evaluate suites on a firm empirical footing. Moreover, the differences between results for different programming languages suggest that mutation operators may need to vary even more than has been suspected in order to work in new languages.

Appendix

A. Average tokens changed in each language

The Table XIV represents the average tokens changed in each language for the manually classified set of commits. This suggests that the average changes are indeed larger than single tokens. This corresponds to our observation in the paper, that the changes made by developers do not correspond to what we expect if the competent programmer hypothesis is true.

	C	Java	Python	Haskell
Bugs	6.96	6.53	10.67	10.74
Features	7.77	7.79	8.11	6.5

TABLE XIV: Average number of tokens changed for manually classified patches

B. *Summary of mutation operators*

Table XV and Table XVI provide the average prevalence of operators in manually classified commits. We note that while the values differ somewhat from those obtained using the automatically classified data set, they still support our conclusion, which was that there is a significant difference between different languages in patterns of errors, even when considering operators that are common across languages.

	C	Java	Python	Haskell
Add	17.07	15.8	20.33	45.93
BinaryOp	0.0	0.0	0.0	0.0
Change	27.91	16.31	26.71	10.97
Const	6.34	2.77	0.76	0.62
Negation	0.12	0.15	0.0	0.0
Rem	11.19	18.57	23.67	29.02
Twiddle	0.12	0.0	0.3	0.0
Var	34.26	45.16	27.47	6.96
Var.Const	3.0	1.24	0.76	0.11

TABLE XV: Summary of mutation operators for all changes

	C	Java	Python	Haskell
Add	17.28	27.4	12.5	36.49
BinaryOp	0.0	0.0	0.0	0.0
Change	30.04	20.62	25.0	4.03
Const	6.17	6.5	6.25	0.95
Negation	0.21	0.28	0.0	0.0
Rem	12.76	13.56	31.25	54.27
Twiddle	0.21	0.0	0.0	0.0
Var	32.1	27.4	25.0	1.66
Var.Const	1.23	4.24	0.0	0.0

TABLE XVI: Summary of mutation operators for all bugfixes

References

[1] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, no. 8, pp. 608–624, 2006.

[3] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[4] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer, 2003, vol. 23.

[5] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *ACM ISSTA*. ACM, 2013.

[6] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Software Engineering, 2005. Proceedings of the 27th ICSE*. IEEE, 2005, pp. 402–411.

[8] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[9] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[10] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert, "Benchmarking testing strategies with tools from mutation analysis." IEEE, 2008, pp. 360–364.

[11] M. Patrick, R. Alexander, M. Oriol, and J. A. Clark, "Probability-based semantic interpretation of mutants," in *Workshop on Mutation Analysis*, 2014.

[12] P. U. S. E. R. C. (SERC)., R. DeMillo, and A. Mathur, *On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software*. Software Engineering Research Center, Purdue University, 1991.

[13] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '96. New York, NY, USA: ACM, 1996, pp. 158–171. [Online]. Available: http://doi.acm.org/10.1145/229000.226313

[14] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2011, pp. 342–352. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001461

[15] R. Gopinath, "Mutagen census," http://dx.doi.org/10.7910/DVN/24329, 2014.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 191–200.

[17] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, 1996, pp. 304–313.

[18] J. Christmansson and P. Santhanam, "Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on software faults," in *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, 1996, pp. 175–184.

[19] R. Chillarege, "Orthogonal defect classification," *Handbook of Software Reliability Engineering*, pp. 359–399, 1996.

[20] J. Duraes and H. Madeira, "Definition of software fault emulation operators: a field data study," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 105–114.

[21] J. A. Duraes and H. S. Madeira, "Emulation of software faults: a field data study and a practical approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 849–867, 2006.

[22] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[23] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120–130.

[24] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 511–526, 2005.

[25] GitHub Inc., "Software repository," http://www.github.com.

[26] CRM114, "Crm114 classifier," http://crm114.sourceforge.net.

[27] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: ACM, 2008, pp. 23:304–23:318. [Online]. Available: http://doi.acm.org/10.1145/1463788.1463819

[28] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Software Engineering Research Center, Purdue University, West Lafayette, IN,, Tech. Rep. SERC-TR41-P, March 1989.

[29] Rahul Gopinath, Carlose Jensen, Alex Groce, "Appendix to mutations: How close are they to real faults?" http://research.engr.oregonstate.edu/hci/sites/research.engr.oregonstate.edu.hci/files/papers/gopinath2014issre-appendix.pdf.

[30] K. Wah, "A theoretical study of fault coupling," *Software testing, verification and reliability*, vol. 10, no. 1, pp. 3–45, 2000.

[31] K. How Tai Wah, "An analysis of the coupling effect i: single test data," *Science of Computer Programming*, vol. 48, no. 2, pp. 119–161, 2003.

[32] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[33] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1980, pp. 220–233.

[34] R. A. DeMillo, "Completely validated software: Test adequacy and program mutation (panel session)," in *Proceedings of the 11th ICSE*, ser. ICSE '89. New York, NY, USA: ACM, 1989, pp. 355–356. [Online]. Available: http://doi.acm.org/10.1145/74587.74634

[35] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*. IEEE, 2002, pp. 352–363.

[36] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.