Amortising the Cost of Mutation Based Fault Localisation using Statistical Inference

Jinhan Kim
KAIST
Republic of Korea
jinhankim@kaist.ac.kr

Gabin An
KAIST
Republic of Korea
agb94@kaist.ac.kr

Robert Feldt
Chalmers University
Sweden
robert.feldt@chalmers.se

Shin Yoo
KAIST
Republic of Korea
shin.yoo@kaist.ac.kr

ABSTRACT

Mutation analysis can effectively capture the dependency between source code and test results. This has been exploited by Mutation Based Fault Localisation (MBFL) techniques. However, MBFL techniques suffer from the need to expend the high cost of mutation analysis after the observation of failures, which may present a challenge for its practical adoption. We introduce SIMFL (Statistical Inference for Mutation-based Fault Localisation), an MBFL technique that allows users to perform the mutation analysis in advance against an earlier version of the system. SIMFL uses mutants as artificial faults and aims to learn the failure patterns among test cases against different locations of mutations. Once a failure is observed, SIMFL requires either almost no or very small additional cost for analysis, depending on the used inference model.

An empirical evaluation of SIMFL using a total of 357 faults in Defects4J shows that SIMFL can successfully localise up to 103 (51%) faults at the top, and 152 (75%) faults within the top five, on par with state-of-the-art alternatives. The cost of mutation analysis can be further reduced by mutation sampling: SIMFL retains over 80% of its localisation accuracy at the top rank when using only 10% of generated mutants, compared to results obtained without sampling.

1 INTRODUCTION

As software systems grow in size and complexity, automated fault localisation techniques [35] have received a lot of attention [2, 25, 32, 36, 40, 41]. There are two driving motivations for automated fault localisation. First, various studies have shown that developers can benefit from automated fault localisation technique if the location of a real fault can be narrowed down to a sufficiently small candidate set [19, 37]. Second, Automated Program Repair (APR), another technique increasingly in demand, depends on the accuracy of automated fault localisation for its success [31, 33, 34].

Mutation analysis has been successfully applied to fault localisation, resulting in a group of techniques called Mutation Based Fault Localisation (MBFL) [12, 24, 28–30]. Mutation analysis applies random syntactic modifications (each corresponding to a mutation operator) to existing code, and observes whether the changes in the program behaviour are detected via testing [13]. Existing MBFL techniques exploit the captured dependency between the artificial faults (i.e., mutants) and the changes in program behaviours (i.e., test results). For example, if mutating a program causes test cases to fail in a pattern similar to an observed failure, the mutant may be near the root cause of the observed failure [28, 29]. Alternatively, if mutating a program causes test cases to fail in a pattern very different from an observed failure, the mutant may be far from the location of the root cause [24].

Despite their success, MBFL techniques share a major weakness with mutation testing, which is the cost of test execution [13]. The more closely mutants approximate real faults, the more accurate MBFL techniques can be. As such, MBFL benefits from a large number of mutants, generated by a diverse set of mutation operators, to be analysed. However, this directly increases the cost of inspecting whether each mutant can be *killed* (i.e., whether the behavioural differences introduced by them are detectable), as this process requires the execution of the test suite per each mutant.

With large systems, this cost can grow significantly large, to the point that MBFL techniques cannot be used just-in-time after failures are observed. This is especially the case when MBFL techniques are used in the context of Continuous Integration (CI) [6, 23]. If developers encounter a failure during the pre-commit testing, they are likely to want a just-in-time debugging technique that ensures fast and accurate feedback, so that they can remove the fault and continue to submit the changes. If, on the other hand, a failure is observed during the post-commit testing initiated by the CI, it is still crucial for a fault localisation technique to be sufficiently fast so that developers do not wait hours for feedback [22]. The cost of having to re-run MBFL for each of the possibly many different failure patterns that can arise during pre- and post-commit testing efforts over, possibly, several commits could be truly staggering.

To overcome the high cost of mutation analysis in MBFL, we introduce SIMFL (Statistical Inference for Mutation-based Fault Localisation), an MBFL technique that allows developers to perform the mutation analysis in advance against an earlier version of the code. SIMFL constructs a kill matrix using a version of the System Under Test (SUT) before any test failures are observed.

The matrix essentially captures which test cases fail when specific locations of SUT are mutated. Once an actual failure is observed, SIMFL builds predictive models and consults them using the information of which test cases pass and/or fail under the observed failure. Depending on the statistical inference technique, the actual post-hoc analysis, required after the observation of the

failure, takes either virtually no time at all (Bayesian inference), or a small fraction of mutation analysis time (Logistic Regression or Multi-Layer Perceptron). SIMFL allows developers to amortise the cost of mutation analysis and use MBFL techniques in a just-in-time manner. By doing even the model building ahead-of-time the cost can be amortized further since we only need to use the previously built model and apply it to the specific failure patterns that are observed.

We have implemented and evaluated SIMFL using multiple modelling schemes and statistical inference techniques.

Given that SIMFL uses less up-to-date (i.e., collected before the fault) information for locating faults than other MBFL and FL techniques, we expect it to be less accurate. Our focus is thus on understanding the trade-off between modelling effort and the achievable accuracy, rather than head-to-head comparison to existing approaches.

The empirical evaluation studies 357 real world faults in DeFECTS4J benchmark [16], using the Major mutation tool [17]. SIMFL can successfully localise up to 103 faults at the top, and 152 faults within the top five places. To reduce the cost of SIMFL even further, we also evaluate the impact of mutation sampling on the mutation analysis step of SIMFL. When using only 10% of the generated mutants for analysis, SIMFL can still achieve over 80% of its localisation accuracy, compared to when not using sampling (average 65.5 faults at the top with 10% sampling, compared to 79 faults at the top without sampling for the investigated inference technique).

The technical contributions of this paper are as follows:

- We introduce SIMFL, a Mutation Based Fault Localisation (MBFL) technique that allows ahead-of-time mutation analysis. Using the outcome of the mutation analysis, SIMFL builds a predictive model that allows developers to predict the location of actual future faults, using the test failure information as input. This process significantly amortises the cost of mutation analysis.
- We present the results of an empirical evaluation of SIMFL using the real world Java faults in DeFECTS4J benchmark. The empirical study concerns not only the localisation accuracy, but also various related aspects of SIMFL such as the impact of different modelling schemes, the viability of models built earlier than faults, and the impact of sampling rates.
- We discuss implications and characteristics of SIMFL and its results, using a state-of-the-art SBFL technique as a counterpart. Our observations suggest that a potentially successful hybridisation may be possible between SIMFL and other fault localisation techniques.

The rest of the paper is organised as follows. Section 2 lays out the foundations of SIMFL by describing how the results of mutation analysis are formulated into predictive models for fault localisation. Section 3 presents the details of experimental design, including the protocols of the empirical study and research questions. Section 4 presents and analyses results, while Section 5 discusses the results in the wider context of fault localisation. Section 6 considers potential threats to validity, and Section 7 presents related work. Finally, Section 8 concludes and presents future work.

Table 1: An Example Kill Matrix

Class	Method	Mutant	Test Result				0-1 Vector
			t_1	t_2	t_3	t_4	of t_1, t_2, t_3, t_4
com. acme. Foo	getType	n.isName() $\mapsto$ true (m_1)	✓	✓	✓	✗	(0, 0, 0, 1)
		n.isName() $\mapsto$ false (m_2)	✗	✓	✓	✗	(1, 0, 0, 1)
		bFlag || isInferred $\mapsto$ isInferred (m_3)	✓	✗	✓	✗	(0, 1, 0, 1)
		varType $\mapsto$ <NO-OP> (m_4)	✓	✗	✓	✗	(0, 1, 0, 1)
	resolveType	param.isTemplateType() $\mapsto$ true (m_5)	✓	✗	✓	✗	(0, 1, 0, 1)
		resolvedType() $\mapsto$ <NO-OP> (m_6)	✗	✓	✓	✗	(1, 0, 0, 1)
		argObjectType != null $\mapsto$ true (m_7)	✗	✓	✓	✓	(1, 0, 0, 0)

2 METHODOLOGY

Intuitively, the underlying assumption of SIMFL is that, for a test that has killed the mutants located on a specific program element, the same program element should be identified as the suspicious location when the same test later fails again. This is based on the coupling effect hypothesis in mutation testing: essentially we *simulate* the occurrence of real faults with artificial faults with known locations, i.e., mutants, and build predictive models for actual future faults. This section describes the models and the statistical inference techniques used by SIMFL.
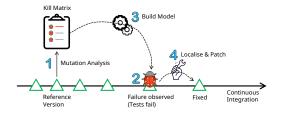

Figure 1: An expected use case scenario of SIMFL

Figure 1 depicts the expected use case scenario of SIMFL, which includes four stages:

(1) Perform mutation analysis for a version of SUT, and produce the kill matrix. The version is called the *reference* version.
(2) While testing a subsequent version, a failure is observed.
(3) Using the information of which test case(s) failed, as well as the kill matrix, build a predictive model for fault localisation.
(4) Guided by the localisation result, patch the fault.

2.1 Mutation Analysis

We perform mutation analysis on the reference version of a program **P** with a test suite **T**, and compute a kill matrix, **K**, which contains a complete report of all tests executed on all mutants. An example kill matrix is shown in Table 1: the mutant m_1 located in the method getType is only killed by the test case t_4, whereas m_2 is killed by both t_1 and t_4.

Let $\mathbf{K}_m$ denote a set of tests that kill mutant m, let $\mathbf{X}_e$ be a set of mutants located on a program element $e \in \mathbf{P}$, let M_e be an event that e is mutated, and let F_t be an event that a test case t fails on a given program. Based on the kill matrix **K**, we can calculate the probability of test case t killing the mutants located on the program element e as follows:

$$P(F_t \mid M_e) = \frac{|\{m \in \mathbf{X}_e \mid t \in \mathbf{K}_m\}|}{|\mathbf{X}_e|} \quad (1)$$

Using Bayes' rule, we can compute the revised probability of the event that the program element e has been mutated, given that the test case t fails:

$$P\left(M_e \mid F_t\right) = \frac{P\left(F_t \mid M_e\right) P\left(M_e\right)}{P\left(F_t\right)} \tag{2}$$
$$\simeq P\left(\text{fault exists in } e \mid F_t\right)$$

We argue that, if real faults are coupled to mutants, the probability above can approximate the likelihood that the fault is located on the program element e, when t is a failing test case in the future. This allows us to make ranking models that sort the program elements in descending order of the probability.

2.2 Ranking Models

We regard the probability in Equation 2 as the quantitative score representing how suspicious the program element e is for the failure observed via the failure of t. This section presents the formulations of ranking models based on the scores as well as more refined inference models based on kill matrix data.

2.2.1 Exact Matching (EM). This model is an extension of Equation 2 to a set of test cases. Let $\mathbf{T} = \{t_i \mid 1 \le i \le n \le n'\}$ be the test set, which consists of two disjoint sets: $\mathbf{T}_f = \{t_1, \ldots, t_n\}$ is the set of failing test cases, and $\mathbf{T}_p = \mathbf{T} \setminus \mathbf{T}_f$ is the set of passing tests, on the faulty program. While there can be many different formulations of ranking models based on a set of test cases, we start by treating the set of all observed failures, $F_{\mathbf{T}_f}$, as a conjunctive event of individual test case failures, i.e., $F_{\mathbf{T}_f} = F_{t_1} \cap \cdots \cap F_{t_n}$. Our goal is to find the faulty program element $e_i \in \mathbf{P}$ with the highest probability of being the cause of the observed failure symptoms, that is, $P\left(M_{e_i} \mid F_{\mathbf{T}_f}\right)$. It follows that:

$$\underset{i}{\arg\max}\, P\left(M_e \mid F_{\mathbf{T}_f}\right) = \underset{i}{\arg\max}\, \frac{P\left(F_{\mathbf{T}_f} \mid M_e\right) P\left(M_e\right)}{P\left(F_{\mathbf{T}_f}\right)} \tag{3}$$

The denominator in Equation 3, $P\left(F_{\mathbf{T}_f}\right)$, can be ignored without affecting the order of ranking based on this score, because it is not related to a specific program element. Expanding the numerator yields the following:

$$\underset{i}{\arg\max}\, P\left(F_{\mathbf{T}_f} \mid M_{e_i}\right) P\left(M_{e_i}\right)$$
$$= \underset{i}{\arg\max}\, P\left(F_{t_1} \cap \cdots \cap F_{t_n} \mid M_{e_i}\right) P\left(M_{e_i}\right)$$
$$= \underset{i}{\arg\max}\, \frac{|\{m \in \mathbf{X}_{e_i} \mid \{t_1, \ldots, t_n\} = \mathbf{K}_m\}|}{|\mathbf{X}_{e_i}|} P\left(M_{e_i}\right)$$
$$= \underset{i}{\arg\max}\, \frac{|\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}|}{|\mathbf{X}_{e_i}|} \frac{|\mathbf{X}_{e_i}|}{|\mathbf{X}_\mathbf{P}|} \tag{4}$$
$$= \underset{i}{\arg\max}\, \frac{|\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}|}{|\mathbf{X}_\mathbf{P}|}$$
$$= \underset{i}{\arg\max}\, |\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}|$$

Intuitively, Equation 4 counts the mutants on e that cause the same set of test cases to fail as the symptom of the actual fault, $F_{\mathbf{T}_f}$. We call this model the Exact Matching (EM) model with failing test cases, denoted by EM(F).

Alternatively, we can include passing tests in the pattern matching as well. Let P_t be an event that a test case t passes on a given program, then Equation 4 changes as follows:

$$\underset{i}{\arg\max}\, P\left(F_{\mathbf{T}_f} \cap P_{\mathbf{T}_p} \mid M_{e_i}\right) P\left(M_{e_i}\right)$$
$$= \underset{i}{\arg\max}\, P\left(F_{t_1} \cap \cdots \cap F_{t_n} \cap P_{t_{n+1}} \cap \cdots \cap P_{t_{n'}} \mid M_{e_i}\right) P\left(M_{e_i}\right) \tag{5}$$
$$= \underset{i}{\arg\max}\, |\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m \wedge \mathbf{T}_p = \mathbf{T} \setminus \mathbf{K}_m\}|$$

Similarly to EM(F), this model is called EM(F+P): it counts the mutants on e that cause the same set of test cases to fail and pass exactly as the symptom of the actual fault. If, for example, a test case t passed under the actual fault, EM(F+P) model will not count any mutants that are killed by t.

2.2.2 Partial Matching (PM). The Exact Matching (EM) models lose any partial matches between the symptom and the mutation results. Suppose two test cases, t_1 and t_2, failed under the actual fault, but only t_1 killed a mutant on the faulty program element, i.e., $\exists t_1, t_2 \in \mathbf{T}_f, t_1 \in \mathbf{K}_m \wedge t_2 \notin \mathbf{K}_m$. The information that t_1 kills a mutant on the location of the fault is lost, simply because t_2 failed to do the same. To retrieve this partial information, we propose two additional models based on partial matches: a multiplicative partial match model and an additive partial match model.

- PM*(F): Multiplicative Partial Match Model w/ Failing Tests

$$\underset{i}{\arg\max}\, \prod_{t \in \mathbf{T}_f} \left(P\left(M_{e_i} \mid F_t\right) + \epsilon\right)$$
$$= \underset{i}{\arg\max}\, \prod_{t \in \mathbf{T}_f} \left(|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{K}_m\}| + \epsilon\right) \tag{6}$$

- PM+(F): Additive Partial Match Model w/ Failing Tests

$$\underset{i}{\arg\max}\, \sum_{t \in \mathbf{T}_f} P\left(M_{e_i} \mid F_t\right)$$
$$= \underset{i}{\arg\max}\, \sum_{t \in \mathbf{T}_f} |\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{K}_m\}| \tag{7}$$

Intuitively, instead of counting exact matches, we want to aggregate scores from the relationship between individual failing test cases and all mutants on a specific program element. PM*(F) and PM+(F) respectively aggregate individual scores by multiplication and addition. Note that the PM*(F) model requires a small positive quantity ϵ to prevent the value of the entire formula from being zero when there exist one or more terms that evaluate to zero.

Similarly to the case of EM models, we can also include the information of test cases that pass under the actual fault. These two models are called PM*(F+P) and PM+(F+P), and defined as follows:

- PM*(F+P): Multiplicative Partial Match Model w/ All Tests

$$\underset{i}{\arg\max}\, \left(\prod_{t \in \mathbf{T}_f} \left(P\left(M_{e_i} \mid F_t\right) + \epsilon\right) \prod_{t \in \mathbf{T}_p} \left(P\left(M_{e_i} \mid P_t\right) + \epsilon\right)\right)$$
$$= \underset{i}{\arg\max}\, \prod_{t \in \mathbf{T}} \left(|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{T}_f \iff t \in \mathbf{K}_m\}| + \epsilon\right) \tag{8}$$

- PM$^+$(F+P): Additive Partial Match Model w/ All Tests

$$\underset{i}{\text{argmax}} \left(\sum_{t \in T_f} \left(P\left(M_{e_i} \mid F_t\right)\right) + \sum_{t \in T_p} \left(P\left(M_{e_i} \mid P_t\right)\right)\right)$$

$$= \underset{i}{\text{argmax}} \sum_{t \in T} |\{m \in X_{e_i} \mid t \in T_f \iff t \in K_m\}| \tag{9}$$

2.2.3 Linear and Non-linear Classifiers. Scores from the Bayesian inference models described in Section 2.2.1 and 2.2.2 can be directly computed from the kill matrix, and requires virtually no additional analysis cost when scores are needed to be computed. However, all these models simply rely on counting matches between test results under the actual fault and kill matrix from the ahead-of-time mutation analysis.

To investigate if more sophisticated statistical inference techniques can improve the accuracy of SIMFL, we apply both linear and non-linear classifiers to build predictive models for SIMFL. These classifiers take the test results as input, and yield the most suspicious method, as well as the suspiciousness score of each method as output. Let α_{T_i} denote a 0-1 vector of the test results of T_i, where 0 indicates that test case fails, and 1 indicates that test case passes. We first build a training set using the kill matrix K: test results per mutant T_i are transformed into α_{T_i}, and the class is labelled based on the method where the mutant located.
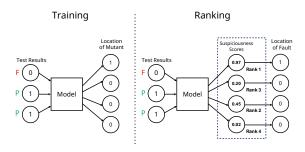

Figure 2: Model Training and Ranking

We train linear and non-linear classifiers using Logistic Regression (LR) and Multi-Layer Perceptron (MLP) [11, 42]. In the serving phase, we use the suspiciousness score of each program element, which is obtained before the model computes the most suspicious method (see Figure 2). Only using the observed failures, we can compose 0-1 vectors (i.e., LR(F) and MLP(F)), or compose 0-1 vectors by including the information of passing tests (i.e., LR(F+P) and MLP(F+P)). Note that, unlike the Bayesian inference models described in Section 2.2.1 and 2.2.2, training these classifiers requires additional analysis cost to SIMFL, although training cost of these models is much lower than the cost of mutation analysis.

3 EXPERIMENTAL DESIGN

This section describes the design of our empirical evaluation, including the way we use DEFECTS4J benchmark, the research questions, as well as other environmental factors.

3.1 Protocol

One foundational assumption of SIMFL is that existing test cases can be fault revealing also for future changes. That is, for future faults to which SIMFL will be applied, test cases that would reveal them are available at the time of the ahead-of-time mutation analysis. We believe this is a likely scenario mainly in two contexts: regression faults, which are defined as failures of existing test cases, and pre-commit testing, for which developers depend on existing test cases for a sanity check. SIMFL is designed to reduce the cost of MBFL for these scenarios[1].

However, this makes realistic experiments on real-world data challenging since a majority of failure triggering changes are not likely to have been committed to the main branch of the Version Control System (VCS): one of the purposes of Continuous Integration is to prevent such commits. Consequently, fault benchmarks, such as DEFECTS4J, contain faults that have been reported externally (e.g., from issue tracking systems), and provide fault revealing test cases that have been added to the VCS with the patch itself [16]. This presents a challenge for the realistic evaluation of SIMFL in the context it was designed for. To address this issue, we introduce two experimental protocols.

3.1.1 Faulty Commit Emulation (FCE). This scenario emulates a faulty *commit* that would trigger failures of existing test cases simply by reversing a fix patch in DEFECTS4J. We take the fixed version (V_{fix}) in DEFECTS4J as the reference version and performs the mutation analysis, including the test cases from the same version. Subsequently, we reverse the fix patch, execute the same test cases, and try to localise the fault using the results with SIMFL. While this may seem unnatural, we argue that this is more realistic than injecting faults artificially. Since artificially injected faults are exactly what SIMFL uses to build its models, we argue that it would get an unfair advantage if it was also evaluated on them. Instead, we emulate faulty commits using faults that some developers actually had introduced in real-world software. We use FCE to evaluate the accuracy of the SIMFL approach itself.

Figure 3: Visualisation of the Test Existence Emulation Scenario

3.1.2 Test Existence Emulation (TEE). This scenario uses original faulty commits that led to the faulty versions (V_{bug}) in DEFECTS4J, but simply *pretends* that the *fault revealing test cases existed earlier*. We have checked whether the fault revealing test cases in DEFECTS4J can be executed against versions that precede the actual

---

[1]Although we do note that the more mature a software system is and the stronger and more complete its test suite is, the more likely it is that these conditions hold.

faulty version. Since system specifications evolve over time, executing a future test case against past versions is not always successful: we have identified 28 previous versions for which the future fault revealing test cases can be executed and *do not fail*. We use these 28 versions as references, and use their mutation analysis results to localise the corresponding faults that happened later. Figure 3 visualises the TEE scenario. Compared to FCE, TEE follows the ground truth code changes, and only assumes the earlier existence of fault revealing test cases. We use TEE to evaluate whether training SIMFL models with kill matrices of earlier versions degrades its localisation accuracy.

3.1.3 Experimental Premise. Building a full kill matrix requires huge computational cost: mutation analysis of all faulty versions of Closure in Defects4J exceeded our 24 hours timeout, and other subject programs also required significant amounts of analysis time. To address this practical concern, for the purpose of experimentation, we have constructed the kill matrix using only the *relevant test cases* as defined by Defects4J[2], which include the failing test cases as well as any passing test cases that makes the JVM to load at least one of the classes modified by the fault introducing commit.

Note that this procedure has been adopted strictly to reduce experimental cost. Since we only have the kill matrix for the relevant test cases, models that use F+P test cases actually use the full set of relevant test cases. However, if construction of the full kill matrix is feasible, the same input used by SIMFL in this paper is naturally available. The F+P models can be trained either using the full set of test cases (increased training cost but also richer input information), or using the relevant test cases (relevancy information is still cheaper than full coverage instrumentation).

We argue that, in general, the limitation to only the relevant test cases is a conservative one and should reduce rather than improve the fault localisation accuracy of SIMFL since other test cases could also be informative for its statistical models.

Table 2: Subjects

Subject	# Faults	kLoC	# Methods	# Mutants	# Test cases
Commons-lang (Lang)	65	22	1,527	21,178	2,245
JFreeChart (Chart)	26	96	4,903	75,985	2,205
Joda-Time (Time)	27	28	1,946	21,689	4,130
Closure compiler (Closure)	133	90	5,038	58,515	7,927
Commons-math (Math)	106	85	2,713	79,428	3,602
Total	357	321	16,126	256,792	20,109

3.2 Subject Programs

In our study, we use 357 versions of five different programs from the Defects4J benchmark set. They provide reproducible and isolated faults of real-world programs. Table 2 summarises the subject programs we used with the average number of generated mutants, methods, lines of code, and test cases across all faults belonging to each subject respectively. We could not include Mockito as we failed to compile the majority of its versions and their mutants, using the build script provided by Defects4J on Docker[3] containers.

---
[2]See https://github.com/rjust/defects4j#export-version-specific-properties
[3]https://www.docker.com

To filter out infeasible subjects, we consider a fault that is fixed by a patch that only introduces new methods, as an *omission fault*, and preclude such faults from our study as the introduced method does not exist at the point when we run the mutation analysis so that cannot be localised. With this condition, we exclude total seven faults: three in Lang, one in Time, one in Closure, and two in Math.

Additionally, to contrast models with F and F+P configurations, we exclude faults whose relevant tests (see Section 3.1.3) set (F+P) is equal to the failing tests set (F): one fault each in Lang and Chart have been excluded due to this criterion. Finally, due to resource constraints, we exclude any fault from the study of F and F+P models, if the mutation analysis for its relevant tests set requires more than 24 hours. For F models, we exclude two faults in Math. For F+P models, all faults in Closure, and nine faults in Math, have been excluded due to the 24 hour timeout.

In contrast, model training is much cheaper. EM and PM models are computed virtually instantly, and the longest LR models take is typically five to six minutes against kill matrices with about 80K rows. MLP models can take up to 30 minutes on a CPU, but we expect GPU-based parallelisation to significantly speed them up.

3.3 Research Questions

This paper asks the following research questions to evaluate SIMFL.

RQ1. Localisation Effectiveness: Does the models of SIMFL produce accurate fault localisation? RQ1 is answered by computing the standard evaluation metrics, $acc@n$, wef, and MAP (see Section 3.4 for definitions) on the eight models of SIMFL under the FCE scenario outlined in Section 3.1.1.

RQ2. Model Viability: How well does SIMFL hold up when applied using prior models built earlier? RQ2 is answered by computing the standard evaluation metrics using prior models built under the TEE scenario outlined in Section 3.1.2.

RQ3. Sampling Impact: What is the impact of mutation sampling to the effectiveness of SIMFL? Since the cost of mutation analysis is the major component of the cost of SIMFL, we investigate how much impact different mutation sampling rates have. We evaluate two different sampling techniques: uniform random sampling, which samples from the pool of all mutants uniformly, and stratified sampling, which samples as equal number of mutants from each method as possible.

3.4 Metrics and Tie Breaking

We use three standard metrics to evaluate the effectiveness of SIMFL:

- $acc@n$: counts the number of faults located within top n ranks. We report $acc@1$, $acc@3$, $acc@5$, and $acc@10$. If a fault is patched across multiple methods, we take the highest ranked method to compute $acc@n$.
- wef: approximates the amount of efforts wasted by developer while investigating non-faulty methods that are ranked higher than the faulty method. It is one less than the highest rank of the faulty methods.
- Mean Average Precision (MAP): measures the mean of the average precision values for a group of all faults. For each fault, when each faulty program elements are ranked at

$R = \{r_1, \ldots, r_n\}$, where r_i is the higher rank than r_{i+1}, the average precision is calculated as $\frac{1}{|R|} \sum_{i=1}^{n} \frac{i}{r_i}$. The faulty methods not retrieved get a precision score of zero.

If multiple program elements have the same score, resulting in the same rank, we break the tie using max tie breaker that places all program elements with the same score at the lowest rank.

Table 3: Mutation Operators

Op.	Description	Op.	Description
AOR	Arithmetic Operator Replacement	ROR	Relational Operator Replacement
LOR	Logical Operator Replacement	LVR	Literal Value Replacement
SOR	Shift Operator Replacement	ORU	Operator Replacement Unary
COR	Conditional Operator Replacement	STD	Statement Deletion

3.5 Mutation Tool and Operators

In the study, we use Major version 1.3.4 [18] as our mutation analysis tool, and choose mutation operators listed in Table 3. Note that some operators have been turned off only for specific classes to avoid errors occurred due to the exceptionally large numbers of generated mutants[4].

3.6 Configuration & Environment

In PM* models, ϵ has been set to 0.001. We trained LR and MLP models using algorithms implemented in `scikit-learn` version 0.20.2 on Python version 3.7.2. For all MLP models, we use only one hidden layer whose size is 50, set maximum number of iterations to 50, and adopt Adam optimizer for weight optimisation with a constant learning rate 0.01. We performed mutation analysis on machines equipped with Intel i7-8700 CPU and 32GB RAM, running Ubuntu 16.04.4 LTS.

4 RESULT

This section contains the results of our empirical evaluation.

4.1 Effectiveness (RQ1)

Table 4 shows the results of each evaluation metric for all studied faults, following the FCE scenario. The numbers $X(Y)$ in the column "Total Studied" represent the number of faults that we can localise (X), and the number of faults provided by DEFECTS4J (Y). Evaluation metric values representing the best outcome (i.e., the largest $acc@n$ and MAP, and the smallest wef) are typeset in bold. See Section 3.2 for the details of exclusion criteria we used: note that more faults are excluded from the study of F+P models shown on the right.

Overall, MLP(F+P) shows the best performance in terms of $acc@n$ metrics, placing 42 out of 61 faults at the first place for Lang, and 38 out of 91 faults at the first place for Math. Considering that MLP(F+P) is evaluated on fewer faults (203) than MLP(F) (348), the result suggests that MLP(F+P) shows better performance on average. We argue that including results of passing tests gives richer information when compared to only using results of failing tests. However, we also note that only MLP significantly benefits from

---

[4]Due to the internal design of Major, some classes that yield too many mutants may lead to the violation of bytecode length limit imposed by Java compiler. See https://github.com/rjust/defects4j/issues/62 for technical details.

the additional information: MLP(F+P) places 32 more faults at the top than MLP(F). Two linear models, LR(F) and LR(F+P), on the other hand, do not show any significant difference in performance. This suggests that exploiting this information requires more sophisticated, non-linear inference methods.

The reason that $PM^+(F)$ shows comparable results to MLP(F) may be that it is relatively easy to simply count the matching patterns of failing tests, which are much rarer than passing tests. We also note that $PM^*(F)$ and $PM^+(F)$ both produce better results than EM(F), suggesting that partial matches are better than exact matches. This is because even the fault revealing test case may not be able to kill all mutations applied to the location of the fault (e.g., the mutant may be an equivalent one). In such a case, the EM(F) model will lose the information, while PM(F) models will benefit from other killed mutants from the same location.

Finally, the addition of passing test information to PM models actually degrades the performance significantly, as the metrics for $PM^*(F+P)$ and $PM^+(F+P)$ show. Partially matching test cases that did not fail against the faulty version with test cases that did not kill mutants at the location of the fault will directly dilute the signal, as failing tests and killed mutants are likely to provide more information about the location of the fault in general.

Based on this analysis, we answer RQ1 that SIMFL can localise faults accurately: SIMFL places at the top 23.28% of studied faults using F models, and 50.74% of studied faults using F+P models.

4.2 Model Viability (RQ2)

Following the TEE scenario described in Section 3.1.2, we seek reference versions preceding the faulty version, i.e., the versions before the faulty version that pass all test cases of the fixed program, including the fault revealing test cases. Assuming that more recent versions are more likely to serve as references, given a faulty version n, we check $n-1, \ldots, n-10, n-20$, and $n-30$ previous program versions, as it is impractical to inspect all of them. Starting from 357 faulty versions of subject programs, we found 28 preceding reference versions that correspond to seven different faulty versions. We have built $PM^+(F)$ models on these 28 reference versions to localise the fault in the faulty version.

Table 5 shows the rank of the faulty method based on $PM^+(F)$ models built on each preceding reference versions. For 23 out of 28 preceding references, $PM^+(F)$ produced the same rank as the FCE result, regardless of whether the reference result was good or bad. One notable exception is Math 46 (f0b12de) that places the fault in the second place, which is a significant improvement over the FCE scenario rank, 2,958. We examined the kill matrix of this reference version, and found that some mutants in the future faulty method have been additionally killed due to timeout (enforced by Major itself), contributing to the high rank (these mutants were not killed in other preceding reference versions of Math 46). We suspect that this is due to the non-determinism in the process of building the kill matrix: the mutation may have brought in flakiness that have been removed for the original program. We consider this as one of the threats to internal validity.

We answer RQ2 that performance of SIMFL using models built with preceding reference versions tends to be stable when compared to the FCE results: only 4 out of 28 preceding reference versions

Table 4: Effectiveness of SIMFL

Model	Project	Total Studied	acc @1	@3	@5	@10	wef med	mean	std	MAP
EM (F)	Lang	62 (65)	33	42	48	49	0.0	269.61	578.63	0.6000
	Chart	26 (26)	5	11	**14**	15	4.0	1134.19	2060.39	0.3088
	Time	26 (27)	4	**8**	9	13	11.0	164.85	510.85	0.2357
	Closure	132 (133)	9	32	36	59	15.5	423.14	1319.94	0.1602
	Math	102 (106)	20	41	52	67	4.0	433.55	1060.59	0.3198
	Total	348 (357)	71	134	159	203				
PM* (F)	Lang	62 (65)	36	45	**52**	54	0.0	125.19	417.59	0.6576
	Chart	26 (26)	5	11	**14**	16	4.0	204.46	937.00	0.3401
	Time	26 (27)	4	**8**	10	13	10.0	92.42	365.56	0.2429
	Closure	132 (133)	10	35	44	68	9.0	173.75	811.74	0.1806
	Math	102 (106)	21	43	**57**	**74**	4.0	112.08	522.54	0.3548
	Total	348 (357)	76	142	**177**	**225**				
PM+ (F)	Lang	62 (65)	**38**	46	51	54	0.0	125.11	417.62	**0.6807**
	Chart	26 (26)	5	11	**14**	**17**	3.0	**195.08**	938.03	0.3566
	Time	26 (27)	4	**8**	10	13	10.0	91.92	365.81	0.2441
	Closure	132 (133)	**12**	**37**	**46**	66	9.5	173.77	811.82	**0.1831**
	Math	102 (106)	20	44	56	**74**	4.0	111.88	522.57	**0.3583**
	Total	348 (357)	79	**146**	**177**	224				
LR (F)	Lang	62 (65)	**38**	**47**	**52**	53	0.0	**45.74**	201.31	0.6669
	Chart	26 (26)	5	9	12	14	6.0	312.00	947.63	0.2976
	Time	26 (27)	4	**8**	10	13	11.5	**18.38**	23.55	0.2463
	Closure	132 (133)	9	33	44	70	8.0	140.72	516.51	0.1828
	Math	102 (106)	25	43	54	67	4.0	141.71	460.55	0.3539
	Total	348 (357)	**81**	140	172	217				
MLP (F)	Lang	62 (65)	34	**47**	**52**	**56**	0.0	57.68	268.03	0.6459
	Chart	26 (26)	6	**12**	**14**	16	3.0	203.81	936.91	0.3643
	Time	26 (27)	**5**	**8**	9	14	8.0	19.54	26.34	0.2494
	Closure	132 (133)	9	24	34	59	11.5	**117.80**	480.67	0.1620
	Math	102 (106)	17	**45**	**57**	72	3.5	**109.88**	470.82	0.3243
	Total	348 (357)	71	136	166	217				

Model	Project	Total Studied	acc @1	@3	@5	@10	wef med	mean	std	MAP
EM (F+P)	Lang	61 (65)	36	43	44	44	0.0	424.39	683.37	0.6073
	Chart	25 (26)	6	9	11	12	28.0	2360.48	2465.15	0.2988
	Time	26 (27)	**10**	13	14	15	3.0	824.92	962.11	0.3800
	Closure	0 (133)	-	-	-	-	-	-	-	-
	Math	91 (106)	33	46	49	50	2.0	1255.23	1478.38	0.4031
	Total	203 (357)	85	111	118	121				
PM* (F+P)	Lang	61 (65)	5	21	21	25	24.0	141.51	268.08	0.2237
	Chart	25 (26)	4	4	6	7	58.0	706.24	1358.71	0.1738
	Time	26 (27)	0	0	2	3	34.0	95.50	123.78	0.0468
	Closure	0 (133)	-	-	-	-	-	-	-	-
	Math	91 (106)	6	8	10	14	152.0	464.09	747.43	0.0799
	Total	203 (357)	15	33	39	49				
PM+ (F+P)	Lang	61 (65)	0	3	8	16	49.0	209.54	292.71	0.0660
	Chart	25 (26)	0	0	0	0	649.0	1136.56	1353.73	0.0088
	Time	26 (27)	0	0	0	0	84.5	144.77	157.84	0.0166
	Closure	0 (133)	-	-	-	-	-	-	-	-
	Math	91 (106)	1	1	1	2	376.0	688.35	796.09	0.0106
	Total	203 (357)	1	4	9	18				
LR (F+P)	Lang	61 (65)	35	40	41	45	0.0	65.07	223.30	0.5991
	Chart	25 (26)	6	11	12	14	6.0	324.44	966.36	0.3308
	Time	26 (27)	9	13	16	18	2.5	38.69	125.24	0.3927
	Closure	0 (133)	-	-	-	-	-	-	-	-
	Math	91 (106)	29	31	35	39	46.0	488.23	866.36	0.3129
	Total	203 (357)	79	95	104	116				
MLP (F+P)	Lang	61 (64)	**42**	**52**	**53**	53	0.0	89.95	322.14	**0.7105**
	Chart	25 (26)	**13**	**16**	**17**	19	0.0	241.36	965.69	**0.5744**
	Time	26 (27)	**10**	**14**	**19**	**22**	1.0	24.23	100.02	**0.4671**
	Closure	0 (133)	-	-	-	-	-	-	-	-
	Math	91 (106)	**38**	**58**	**63**	70	1.0	185.49	591.56	**0.4982**
	Total	203 (357)	**103**	140	152	164				

Table 5: Model Viability Results using TEE Scenario

Fault #	Commit (# of rev. diff.)	Rank	Fault #	Commit (# of rev. diff.)	Rank
Closure 21	FCE Rank	3	Math 46	FCE Rank	2958
	32a12ba (2)	3		bbb5e1e (1)	2958
	43a5523 (3)	3		37680e2 (2)	2958
Closure 61	FCE Rank	6		1861674 (3)	2958
	f5529dd (3)	6		f0b12de (4)	**2**
	b12d1d6 (4)	6		8581b76 (5)	2958
	245362a (7)	6	Math 89	FCE Rank	1964
	8abd1d9 (8)	6		43336b0 (1)	**1963**
	37b0e1b (9)	6		cdd62a0 (2)	**1965**
Closure 62	FCE Rank	1		90439e5 (3)	1964
	245362a (2)	1		36a8485 (4)	1964
	8abd1d9 (3)	1		dbe7842 (5)	1964
	37b0e1b (4)	1		d84a587 (6)	1964
Closure 115	FCE Rank	28		d27e072 (7)	1964
	b9262dc (5)	**26**		3590bdc (8)	1964
	911b2d6 (6)	28		6b108c0 (9)	1964
Closure 120	FCE Rank	8		9c55428 (10)	1964
	2aee36e (3)	**29**			

Table 6: Uniform Random Sampling Results

Sample Ratio	Project	Total Studied	acc @1	@3	@5	@10	wef mean mean	std	MAP
0.1	Lang	62 (65)	31.85	40.55	44.10	45.10	198.70	30.79	0.5527
	Chart	26 (26)	4.95	8.45	10.30	13.65	868.84	123.12	0.2564
	Time	26 (27)	3.65	6.45	7.95	10.55	163.34	35.59	0.1906
	Closure	132 (133)	8.40	18.95	24.65	34.20	858.59	72.02	0.1129
	Math	102 (106)	16.65	34.45	43.25	55.40	471.25	41.80	0.2611
0.3	Lang	62 (65)	36.10	46.30	49.40	52.75	145.42	22.87	0.6466
	Chart	26 (26)	5.35	10.40	12.20	16.00	668.84	181.53	0.3210
	Time	26 (27)	4.00	8.90	10.10	12.50	103.62	28.80	0.2342
	Closure	132 (133)	9.10	26.35	37.20	50.50	561.29	104.82	0.1490
	Math	102 (106)	19.20	39.45	50.15	66.35	331.82	59.10	0.3172
0.5	Lang	62 (65)	37.05	**46.75**	50.65	53.35	140.10	15.70	0.6666
	Chart	26 (26)	**5.45**	10.40	12.85	16.95	467.43	218.73	0.3413
	Time	26 (27)	3.95	**9.00**	10.20	12.25	94.86	23.47	0.2367
	Closure	132 (133)	10.05	29.95	42.25	59.15	369.23	57.28	0.1676
	Math	102 (106)	20.60	42.95	53.50	69.70	232.14	47.88	0.3397
0.7	Lang	62 (65)	37.30	46.65	**51.65**	**54.20**	134.20	12.53	0.6758
	Chart	26 (26)	5.30	**11.65**	**14.00**	**17.90**	353.07	171.64	**0.3594**
	Time	26 (27)	**4.35**	8.80	**10.35**	12.65	94.09	19.05	**0.2514**
	Closure	132 (133)	10.15	32.80	45.20	63.45	270.03	59.02	0.1741
	Math	102 (106)	**20.80**	43.65	55.30	72.00	161.95	26.00	0.3512
Full	Lang	62 (65)	**38.00**	46.00	51.00	54.00	**125.11**	0.00	**0.6807**
	Chart	26 (26)	5.00	11.00	**14.00**	17.00	**195.08**	0.00	0.3566
	Time	26 (27)	4.00	8.00	10.00	**13.00**	**91.92**	0.00	0.2441
	Closure	132 (133)	**12.00**	**37.00**	46.00	**66.00**	173.77	0.00	**0.1831**
	Math	102 (106)	20.00	**44.00**	**56.00**	**74.00**	111.88	0.00	**0.3583**

show degraded performance because we use less recent mutation analysis results.

4.3 Sampling Impact (RQ3)

To investigate how the mutation sampling rates affect the performance of SIMFL, we attempt to localise the studied faults using mutants sampled with different rates. For this study, we use the PM^+(F) model, and compute the same set of evaluation metrics.

Table 6 shows the uniform sampling results with rates of 0.1, 0.3, 0.5, and 0.7: all metric values are averages across 20 different samples. The column "wef mean" contains the average of 20 mean

wef values, each of which is a mean wef across all faults in the corresponding subject. Table 6 also includes the results obtained without sampling (Full). The best results are typeset in bold.

As expected, the Full configuration often shows the best performance, followed by sampling rates of 0.7 and 0.5. Since we expect different mutants to contribute different amounts of information to

localisation, we do not find it surprising that sampling rates down to 0.5 can sometimes perform as well as the Full configuration. However, the performance does not degrade at the same rate as the sampling rate, as can be seen from the results obtained using the sampling rate of 0.1.

Table 7: Stratified Random Sampling Results

N	Project	Total Studied	Sample Ratio	acc @1	@3	@5	@10	wef mean mean	std	MAP
5	Lang	62 (65)	0.26	18.65	35.40	41.70	47.50	290.71	54.18	0.4419
	Chart	26 (26)	0.23	2.95	7.00	8.70	12.40	659.49	208.85	0.2310
	Time	26 (27)	0.32	1.00	2.20	3.20	5.15	188.51	64.31	0.0919
	Closure	132 (133)	0.30	2.85	5.90	8.10	14.50	1043.36	133.09	0.0462
	Math	102 (106)	0.14	9.85	23.25	34.25	52.50	411.71	94.78	0.2028
10	Lang	62 (65)	0.42	22.95	38.75	46.90	50.50	201.91	31.16	0.5057
	Chart	26 (26)	0.36	4.15	7.80	10.10	15.15	354.06	119.40	0.2838
	Time	26 (27)	0.47	1.95	3.00	4.00	6.50	106.08	2.50	0.1304
	Closure	132 (133)	0.46	5.50	11.20	15.30	26.90	522.53	100.51	0.0802
	Math	102 (106)	0.23	12.90	31.10	43.55	63.00	221.61	32.84	0.2580
15	Lang	62 (65)	0.53	26.05	42.35	48.75	52.25	166.19	21.18	0.5571
	Chart	26 (26)	0.44	4.70	7.75	10.85	15.90	224.57	84.11	0.3027
	Time	26 (27)	0.57	1.80	3.85	5.20	7.75	100.19	2.28	0.1439
	Closure	132 (133)	0.56	8.50	19.90	25.30	37.35	363.72	54.78	0.1203
	Math	102 (106)	0.30	14.65	35.30	48.50	67.65	175.70	31.93	0.2893
20	Lang	62 (65)	0.61	27.70	43.80	49.20	52.95	149.88	20.69	0.5795
	Chart	26 (26)	0.51	**5.35**	8.90	12.10	16.20	203.40	1.14	0.3268
	Time	26 (27)	0.65	1.55	3.15	7.15	9.90	96.06	1.59	0.1478
	Closure	132 (133)	0.63	**13.00**	24.45	32.45	43.70	291.28	46.07	0.1548
	Math	102 (106)	0.36	17.55	37.50	52.00	70.60	147.92	26.04	0.3133
Full	Lang	62 (65)	1.00	**38.00**	**46.00**	**51.00**	**54.00**	**125.11**	0.00	**0.6807**
	Chart	26 (26)	1.00	5.00	**11.00**	**14.00**	**17.00**	**195.08**	0.00	**0.3566**
	Time	26 (27)	1.00	**4.00**	**8.00**	**10.00**	**13.00**	**91.92**	0.00	**0.2441**
	Closure	132 (133)	1.00	12.00	**37.00**	**46.00**	**66.00**	**173.77**	0.00	**0.1831**
	Math	102 (106)	1.00	**20.00**	**44.00**	**56.00**	**74.00**	**111.88**	0.00	**0.3583**

Since larger methods are likely to produce more mutants, uniform sampling will effectively sample more mutants for larger methods. We investigate whether this is disadvantageous for relatively smaller methods by evaluating stratified sampling: given the threshold parameter N, stratified sampling randomly chooses only N mutants from methods with more than N mutants, and chooses all mutants if their number is below N. Table 7 contains the results obtained using stratified mutant sampling with $N \in \{5, 10, 15, 20\}$. The column "Sample Ratio" is the ratio of the number of mutants sampled by stratified sampling to the number of all mutants.

Compared to the Full configuration, the performance degradation as N decreases is notably worse than what has been observed from the results of uniform random sampling. However, even with $N = 5$, the sample ratio is about 0.25 on average, higher than the smallest sampling rate for the uniform sampling. The comparison suggests that, contrary to our concern for a potential bias against smaller methods, stratified sampling is actually harmful to SIMFL. One interpretation of the result is that, if we assume that the location of a fault is a random variable, larger methods are by definition more likely to contain it.

We answer RQ3 that the impact of mutation sampling is observable but not too disruptive: uniform sampling can take sampling rate down to 0.1 and still localise on average, at the top, more than 80% of the faults placed at the top when using all mutants. However, stratified sampling actually harms SIMFL: larger methods need to be represented by more mutants.

5 DISCUSSION

In this section, we discuss the results in Section 4 with a comparison to a state-of-the-art SBFL technique, and detailed analysis on the performance for Commons Lang.

5.1 Comparison to Other FL Techniques

There are a number of factors that present challenges for comparing SIMFL with existing MBFL techniques. All existing MBFL techniques are designed to mutate the faulty version of the program, whereas the design focus of SIMFL is to perform the mutation analysis beforehand. Given the high cost of MBFL techniques [30], we could not perform two sets of mutation analysis for all studied faults. Moreover, existing empirical evaluation of MBFL techniques has either focused on C benchmarks [24, 27, 28] or used EXAM scores at the statement level as the evaluation metric [30], making an indirect comparison of published results also difficult.

To gain some insights into the trade-off between amortised modelling efforts and localisation accuracy, we instead present a comparison of method level fault localisation results with a state-of-the-art Spectrum Based Fault Localisation (SBFL) technique, FLUCCS [32], which aggregates multiple SBFL techniques as well as static code and change metrics using learning-to-rank techniques. After reproducing the results of FLUCCS using the publicly available version[5], we have extracted the intersection of studied faults using DEFECTS4J fault ids, and computed evaluation metrics for the common faults. Table 8 contains $acc@1$, $acc@5$, and $acc@10$ results of SIMFL with MLP(F+P) configuration, as well as the corresponding results of FLUCCS with GP^A_{med} configuration.

Table 8: Comparison to FLUCCS [32]

Project	Common Faults	SIMFL acc@1	acc@5	acc@10	FLUCCS acc@1	acc@5	acc@10
Lang	60	**42**	**53**	53	30	47	**55**
Chart	24	13	17	19	**15**	**23**	23
Time	26	**10**	**19**	**22**	9	16	19
Math	89	38	63	70	**43**	63	**71**
Total	199	**103**	**152**	164	97	149	**168**

Figure 4 gives a closer look at how individual faults are ranked differently by SIMFL and FLUCCS respectively. Figure 4a plots each individual fault at (x, y), in which x is its rank by SIMFL and y its rank by FLUCCS. Any data points near the $y = x$ line represent faults that are ranked similarly by both SIMFL and FLUCCS; data points far away from the line represent faults that are ranked very differently by two techniques.

While there is a high-density cluster of data points near the $y = x$ line in the high-rank region (many faults are ranked at the top by both techniques), we note that there is also a long tail dispersion of data points far from the line. This suggests that there are clusters of faults that one technique is significantly more effective than the other, and vice versa. Figure 4b and 4c magnify these long tails by focusing on faults that are ranked within the top 20 by one technique but not by the other. These faults suggest that two techniques can be complementary to each other for certain faults, and there may exist an effective hybridisation of two.

---

[5]https://bitbucket.org/teamcoinse/fluccs

(a) Faults in Top 80   (b) Top 2,000 (SIMFL) vs. 20 (FLUCCS)   (c) Top 20 (SIMFL) vs. 2,000 (FLUCCS)
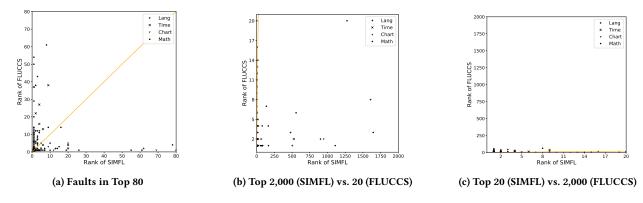
Figure 4: One-to-one comparison of Fault Rankings by SIMFL and FLUCCS

Considering that SIMFL does not use any information from the actual failing executions, we think that its results are comparable to those of FLUCCS, and that the trade-off is manageable. SIMFL tends to outperform FLUCCS for Lang and Time, but FLUCCS outperforms SIMFL for other projects. This is discussed in the following section (Section 5.2). A closer analysis of individual faults suggests that two techniques can be complementary to each other.

5.2 Test Case Granularity

A common pattern observed in all configurations of SIMFL is that it performs the best for Commons Lang. Following Laghari and Demeyer [20], we hypothesise that this may be related to the test case granularity: if each test case covers only a small number of methods, SIMFL can benefit from this as failures of each test case will be tightly coupled with a few candidate locations.

Figure 5 contains the results of test granularity analysis. The left boxplots show the number of methods whose mutants are killed by the failing tests per projects, while the right boxplots show the same numbers but only for faults that are successfully localised within the top three places.
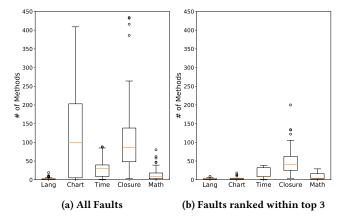

(a) All Faults   (b) Faults ranked within top 3

Figure 5: The number of methods whose mutants are killed by the failing test cases

The results show that successfully localised faults are revealed by test cases that have finer granularity. Test cases of Commons

Lang have significantly finer granularity when compared to other subjects, leading us to conjecture that test case granularity is why SIMFL performs more effectively against Lang than others. However, the results also show that SIMFL is not simply reflecting a one-to-one mapping between methods and their unit tests. A failing test case of Closure covers mutants in 50 methods on average, but SIMFL can still localise 46 out of 132 faults within the top five if we use $PM^+(F)$ (see Table 4).

We need more experiments to fully understand the conditions under which SIMFL performs well but the analysis above indicates that granularity is not the whole story. We hypothesize that SIMFL should also be able to work well with higher-level, system test cases if there are many such test cases and that they are diverse [7]. If such a test suite is combined with extensive mutation analysis the kill matrix should encapsulate diverse enough failure patterns so that statistical inference can still pinpoint individual faults.

5.3 Statistical Inference Techniques

Given the problem formulation we have introduced in this paper, there is a plethora of statistical inference techniques that can be potentially used for fault localisation. We have used three of the most basic techniques. Even though more advanced, but then typically also more complex, techniques could be used there is support in the literature for using more basic methods. For example, Hall et al. [10] found that fault prediction models that performed well were often based on simple modelling techniques such as Naive Bayes or Logistic Regression. Still, future work should investigate the use of modern machine learning alternatives since the extra accuracy can save precious developer time.

Since the results from fault localisation techniques are typically presented to developers as a ranked list of code locations to be investigated alternative modelling techniques such as Learning to Rank [4, 32] should also be investigated.

We also note that formulating the fault localisation problem as statistical inference naturally opens up to the inclusion of other features during modelling. For example, SIMFL can be hybridised with other fault localisation techniques, either by incorporating existing features, e.g. code coverage information, into its own models, or by being used as a feature in models built or learnt for other localisation techniques.

6 THREATS TO VALIDITY

Given the controlled setting for our experiments and the clearly defined objective measures, there are few threats to the internal validity of our study. There are some threats to internal validity that are inherent to any mutation analysis and hard to completely avoid, such as non-determinism due to mutation and equivalent mutants, which have been discussed in Section 4. Similarly, we see few threats to the construct and conclusion validity. The metrics we used are standard in the fault localisation literature. To more definitively compare different statistical inference techniques we could use statistical analysis. However, establishing one best technique is not our main goal here and we would likely need more study subjects for such a comparison to be meaningful.

Rather, the main threat of our study is to its external validity. Even though we studied five different subjects from the real-world DEFECTS4J benchmark to mitigate this threat, this does not allow us to generalize to many, other programs and test suite contexts. Still, there was enough variation among the five subjects for us to identify SIMFL's dependence on the granularity of the test cases (as discussed in Section 5.2 above). However, only further work on more subjects can help gauge the generality of our results.

7 RELATED WORK

SIMFL is an MBFL technique that allows ahead-of-time mutation analysis. Existing MBFL techniques, Metallaxis [28, 29], MUSE [24] and its variation MUSEUM [12], all require the faulty program to be mutated, incurring significant analysis cost after the observation of failure. Metallaxis uses SBFL-like formulas to measure the similarity between failure patterns of the actual fault and mutants. MUSE and MUSEUM both focus on two principles: first, if we mutate already faulty parts of the program, it is unlikely that we will observe more failing test cases, and we may even observe partial fixes, and second, if we mutate non-faulty parts of the program, it is likely that passing tests will now fail. MUSE and MUSEUM define their suspiciousness scores using the ratios of fail-become-pass and pass-become-fail tests. Pearson et al [30] showed that these MBFL techniques can be improved by hybridising mutation-based scores with other features.

As a fault localisation technique, SIMFL can be seen as predicting where faults are most likely to be located. This is different from, but related to, other forms of software defect prediction [10, 21] which typically uses software metrics and code attributes, but rarely information from the dynamic testing, to predict defect-proneness. A recent exception is the study by Xiao et al. [38], which uses test results to dynamically tune traditional defect prediction models. Still, they predict and rank modules based on their overall defect proneness, with the goal of directing quality assurance efforts, rather than for localising actually observed faults.

SIMFL was initially formulated based on Bayesian analysis to infer likely fault locations given test information. In the context of fault localisation, Abreau et al. [1] have introduced BARINEL, an SBFL technique that adopts Bayesian reasoning to generate candidate sets of multiple fault locations. To the best of our knowledge, SIMFL is the first MBFL technique that uses Bayesian inference as well as other statistical inference techniques. Bayesian inference has been used in other forms of defect modelling. Fenton et al. [9] proposed and later summarised [8] work on using Bayesian Belief Networks for software defect prediction. Similarly, Okutan and Yildiz [26] used Bayesian modelling to predict defect proneness based on software metrics. However, like the other work on software defect prediction, the focus is on software quality, reliability, and the number of remaining defects, not, as for SIMFL, on helping to locate specific faults based on test information.

Mutation analysis has been used to improve statistical software fault prediction models, by Bowes et al. [3]. By adding 40 mutation-based metrics calculated based on information collected by running the publicly available PITest mutation testing tool to 39 traditional, static source code metrics the study showed improvements in predictive performance. The models predicted which classes where faulty rather than helping rank and locate specific faults at the method level as we do here.

Like other fault localisation (FL) techniques, SIMFL tries to locate faults in a SUT based on dynamic information [35]. The most widely studied FL approach is Spectrum-based Fault Localisation (SBFL) [5, 14, 15] which combines code coverage and test outcomes to rank code locations by their *suspiciousness score*. FLUCCS [32] is a state-of-the-art FL technique which aggregates multiple SBFL techniques and improves predictions with software defect prediction metrics, both static code as well as change metrics. Like other SBFL techniques, FLUCCS is more costly than SIMFL since it requires code coverage information for the SUT version on which the tests failed. While SIMFL also depends on dynamic information, it not only eliminates the need for code instrumentations, but can perform the dynamic (i.e., mutation) analysis ahead-of-time. This allows the cost to be amortized over multiple development iterations and allows faster feedback when a failure is observed.

Xuan and Monperrus [39] improve SBFL by creating more fine-grained test cases based on individual assertions in the existing test suite. As discussed above SIMFL performs better when test cases are of higher granularity and can better point to a specific or small set of source code locations. Future work should investigate if we could thus combine SIMFL with the fine-graining techniques of Xuan and Monperrus.

Overall, SIMFL differs from software defect prediction methods since it has different goals, modeling inputs, and prediction targets. It is less costly than SBFL techniques in that it requires neither code coverage nor static code metric information. Also, in comparison to other mutation-based FL techniques, it can amortize the cost of mutation analysis over multiple development iterations.

8 CONCLUSION AND FUTURE WORK

This paper introduces SIMFL, a Mutation Based Fault Localisation (MBFL) technique that allows users to perform the mutation analysis in advance, before the actual failure is observed. SIMFL relies on statistical inference techniques to train predictive models that can be used with the information of which test cases have actually failed. This allows us to use the concrete and precise dependencies between source code and test cases for fault localisation, without having to expend the large cost of mutation analysis only after the failures are observed. We have empirically evaluated SIMFL using 357 real world faults from DEFECTS4J benchmark. SIMFL can localise 103 faults at the top, and is capable of retaining over 80%

of its localisation accuracy at the top when we sample only 10% of all generated mutants.

Future experiments on more programs should investigate the effects that test case granularity, test suite strength, and test case diversity have on SIMFL's performance. Future work will also consider both improving the accuracy and lowering the cost of analysis even further. To improve accuracy, we will consider a wider range of statistical inference techniques, as well as hybridisation with other feature sets and fault localisation techniques. To lower the cost, we will investigate whether past code change and test result history can be used in tandem, or instead of, mutation analysis. Test execution information collected whenever developers locally run tests against their recent changes could potentially alleviate the need for mutation analysis altogether.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*. 88–99. https://doi.org/10.1109/ASE.2009.25

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2011. Simultaneous Debugging of Software Faults. *J. Syst. Softw.* 84, 4 (April 2011), 573–586.

[3] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. 2016. Mutation-aware Fault Prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 330–341.

[4] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. ACM, 129–136.

[5] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight bug localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05)*. ACM, New York, NY, USA, 99–104. https://doi.org/10.1145/1085130.1085143

[6] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[7] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST 2016)*. 223–233.

[8] NE Fenton and Martin Neil. 2018. Improving Software Testing with Causal Modeling. *Analytic Methods in Systems and Software Testing* (2018), 27–64.

[9] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, David Marquez, Paul Krause, and Rajat Mishra. 2007. Predicting software defects in varying development lifecycles using Bayesian nets. *Information and Software Technology* 49, 1 (2007), 32–43.

[10] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[11] Geoffrey E Hinton. 1990. Connectionist learning procedures. In *Machine learning*. Elsevier, 555–610.

[12] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging Real-World Multilingual Programs Using Mutation Analysis. *Information and Software Technology* 82 (2017), 80–95.

[13] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.

[14] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE2005)*. ACM Press, 273–282.

[15] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, USA, 467–477.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[17] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 612–615.

[18] R. Just, F. Schweiggert, and G. M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 612–615. https://doi.org/10.1109/ASE.2011.6100138

[19] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 165–176.

[20] Gulsher Laghari and Serge Demeyer. 2018. Unit Tests and Component Tests Do Make a Difference on Fault Localisation Effectiveness. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, New York, NY, USA, 280–281. https://doi.org/10.1145/3183440.3194970

[21] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[22] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 688–698. https://doi.org/10.1145/3180155.3180213

[23] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE software* 31, 3 (2014), 14–16.

[24] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST 2014)*. 153–162.

[25] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology* 20, 3, Article 11 (August 2011), 32 pages.

[26] Ahmet Okutan and Olcay Taner Yıldız. 2014. Software defect prediction using Bayesian networks. *Empirical Software Engineering* 19, 1 (2014), 154–181.

[27] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 936–946.

[28] M. Papadakis and Y. Le-Traon. 2012. Using Mutants to Locate "Unknown" Faults. In *Proceedings of the 5th IEEE Fifth International Conference on Software Testing, Verification and Validation (Mutation 2012)*. 691–700. https://doi.org/10.1109/ICST.2012.159

[29] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test., Verif. Reliab.* 25, 5-7 (2015), 605–628. https://doi.org/10.1002/stvr.1509

[30] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[31] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2483760.2483785

[32] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localisation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2017)*. 273–283.

[33] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*. IEEE, Vancouver, Canada, 364–374.

[34] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[35] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (August 2016), 707.

[36] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '07)*. IEEE Computer Society, Washington, DC, USA, 449–456. https://doi.org/10.1109/COMPSAC.2007.109

[37] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 267–278.

[38] Peng Xiao, Bin Liu, and Shihai Wang. 2018. Feedback-based integrated prediction: Defect prediction based on feedback from software testing process. *Journal of*

Systems and Software 143 (2018), 159–171.

[39] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 52–63.

[40] Shin Yoo. 2012. Evolving Human Competitive Spectra-Based Fault Localisation Techniques. In *Search Based Software Engineering*, Gordon Fraser and Jerffeson Teixeira de Souza (Eds.). Lecture Notes in Computer Science, Vol. 7515. Springer Berlin Heidelberg, 244–258.

[41] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human Competitiveness of Genetic Programming in SBFL: Theoretical and Empirical Analysis. *ACM Transactions on Software Engineering and Methodology* 26, 1 (July 2017), 4:1–4:30.

[42] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. 2011. Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning* 85, 1-2 (2011), 41–75.