박 사 학 위 논 문
Ph.D. Dissertation

뮤테이션 기법의 효용성 향상을 위한 뮤턴트와 코드, 결함, 패치 사이의 관계 분석

Exploiting Mutant's Relationship with Code, Faults, and Patches for Higher Efficacy of Mutation Analysis

2023

김 진 한 (金 鎭 漢 Kim, Jinhan)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

뮤테이션 기법의 효용성 향상을 위한 뮤턴트와 코드, 결함, 패치 사이의 관계 분석

2023

김 진 한

한 국 과 학 기 술 원

전산학부

뮤테이션 기법의 효용성 향상을 위한 뮤턴트와 코드, 결함, 패치 사이의 관계 분석

김 진 한

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2022년 12월 9일

심사위원장	유 신	(인)
심 사 위 원	배 두 환	(인)
심 사 위 원	김 문 주	(인)
심 사 위 원	Robert Feldt	(인)
심 사 위 원	Annibale Panichella	(인)

Exploiting Mutant's Relationship with Code, Faults, and Patches for Higher Efficacy of Mutation Analysis

Jinhan Kim

Advisor: Shin Yoo

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
December 9, 2022

Approved by

Shin Yoo
Associate Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 김진한. 뮤테이션 기법의 효용성 향상을 위한 뮤턴트와 코드, 결함, 패치 사이의 관계 분석. 전산학부 . 2023년. 87+vi 쪽. 지도교수: 유신. (영문 논문)
Jinhan Kim. Exploiting Mutant's Relationship with Code, Faults, and Patches for Higher Efficacy of Mutation Analysis. School of Computing . 2023. 87+vi pages. Advisor: Shin Yoo. (Text in English)

초 록

뮤테이션 테스팅 기법은 오랫동안 소프트웨어의 테스트 품질을 검증하는 데에 좋은 성능을 보였으며 이를 통해 프로그램 자체를 분석하는 데에도 널리 사용되었다. 하지만 이 기법은 수많은 뮤턴트에 대한 테스트 실행을 동반하여 많은 시간과 컴퓨팅 리소스를 소모하게 되어, 아직까지도 큰 스케일의 프로그램에 대해서는 적용될 수 없다는 문제점을 가지고 있다. 이러한 문제점을 완화하기 위해, 본 논문에서는 뮤턴트와 코드, 결함, 패치와의 관계성을 탐구한다. 첫 번째로, 뮤턴트 주변의 코드에서 자연 언어 채널을 분석하여 테스트 실행을 하지 않고 뮤턴트 탐지 여부를 예측하는 방법을 제시한다. 두 번째로, 뮤턴트의 결함과의 관계성을 분석하여 결함 위치 추정 정확도를 높이는 방법을 제시한다. 다양한 통계 추정 모델과 과거 뮤테이션 정보를 활용하여 현재 결함 위치를 뮤턴트에 대한 테스트 실행 없이 예측한다. 마지막으로 뮤턴트와 패치의 관계성을 분석한다. 본 논문에서는 뮤턴트와 패치가 구문적으로 다르지 않다고 가정하고, 서로가 얼마나 비슷하며 각자를 대체할 수 있는지를 보이는 다양한 실험을 수행한다. 이러한 실험 통해 본 논문에서 제시한 세 가지 관계 분석이 뮤테이션을 넓은 범위에서 적용될 수 있게 하고 더욱 효과적으로 할 수 있게 할 수 있다는 것을 확인한다.

핵 심 낱 말 뮤테이션 테스팅, 뮤테이션 기법, 뮤테이션 분석, 소프트웨어 테스팅, 소프트웨어 엔지니어링

Abstract

Mutation testing has long been used as a powerful testing technique to evaluate the test suite quality as well as to analyse the program under test. However, it still has issues of scalability and applicability due to the high cost of running tests against mutants. To improve its efficacy even on large programs, this dissertation explores mutant's relationship with code, faults, and patches that have not been explored much but are likely to be useful. First, we investigate the mutant's relationship with code to assist predicting the kill of a mutant without running the tests. By exploring the natural language channel of the code around the mutant, we build a deep neural network to learn and predict the mutant's killability. Second, we explore the mutant's relationship with the faults to localise the faults. We propose several statistical inference techniques that can learn the mutant-fault relationships in advance, in terms of their similarity of test executions. Lastly, we investigate the mutant's relationship with the patches. We hypothesise that the mutants (faults) and patches do not syntactically differ from each other and conduct various empirical studies to show how much they are similar. Our empirical results suggest that exploiting those relationships makes mutation analysis more viable and effective.

Keywords Mutation testing, mutation analysis, software testing, software engineering

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Mutation testing is a powerful testing technique to evaluate quality and capability to detect a fault of a given test suite in a program [1]. It generates a number of program variants called mutants by using several syntactic change operators (i.e., mutation operators). Based on the mutants and their interactions with the test executions, mutation testing produces a comprehensive measurement of test effectiveness called mutation score. It is computed as the ratio of the number of detected (i.e., killed) mutants to the total mutants. The underlying assumption of mutation testing is based on a coupling effect [2]: its simple syntactic mutants can represent complex real faults, in other words, if the test suite can detect the mutants, it can also detect the real faults. This means that even if we do not have any clue about how many of the real faults there will be in the program, the mutation testing allows us to pre-evaluate the test suite and thus prevent unknown faults. On the coupling effect, a lot of previous work has provided empirical evidence, endorsing the use of mutation testing as a proxy of the test effectiveness [2, 3].

From the beginning of its introduction, however, mutation testing has been criticized for the high costs of running the tests against all generated mutants, which is usually infeasible for large programs [1]. Several previous work has proposed to overcome this scalability issue by using random sampling of the mutants [4] or higher-order mutants [5]. These can reduce the number of mutants and test executions, but still they are not suitable for large systems like Google's on which more than ten million test executions take place everyday [3]. Another recent work utilises Continuous Integration (CI) [6]: Predictive Mutation Testing (PMT) [7, 8, 9] aims to predict mutation testing results without mutant executions by learning static and dynamic features of mutants from earlier version of the project.

Despite the recent advances, this issue in mutation testing remains unsolved. Moreover, the problem becomes even severer when the mutation testing is conducted in a test-case level, which we call a *mutation analysis* throughout this thesis. The test-suite level mutation testing is typically for computing the mutation score (i.e., how many mutants are detected by *entire test suite*), which can early stop the test executions once a mutant is killed by any test case. In contrast, the mutation analysis in a test-case level is to analyse which *test case* can detect which mutant, so it has to execute all test cases against all mutants. In addition to the original use case of mutation testing, mutation analysis can provide detailed dynamic information between each pair of test and mutant, which can be used in various applications such as fault localisation [10, 4] and automated program repair [11, 12].

This thesis aims at widening the applicability and scalability of mutation analysis by exploiting the mutant's relationship to code, faults, and patches, since we posit that much of the mutant's utility have not been exposed yet despite having a potential to provide valuable insight into the mutation analysis. We first investigate mutant's relationship with code by exploring Natural Language (NL) channel [13] of mutants to design static features used to predict the results of mutation analysis. Under the CI context, we build and train a deep model that learns those features collected from the previous versions of the program, and then use the model to predict the entire kill matrix that shows which test cases kill which mutants. Second, in order to localise the faults using mutants, we exploit the mutant's ability to represent the faults and relate to the tests. Based on the mutation analysis on the past version of the program, we train a model that learns the relationship between mutants and tests, and use it to predict the faults in the current version of the program. Lastly, we explore mutant's relationship with the patches to expand the efficacy of mutants: despite its original design principle of being artificial faults, we hypothesise that the mutants can act as the patches that help repairing the faults. Our empirical study using real-world faults and well-known mutation tools shows that 1) looking at the mutants' NL channel is helpful to learn and predict the results of mutation analysis; 2) even the mutants in the past version of the program can be effectively exploited to localise the faults in the future; 3) the faults (i.e., mutants) and patches have syntactic similarities, which opens the possibility of using both of them to learn either fix or bug-inducing patterns.

In the following three sections, we introduce the previous work that form a basis of this work. Then, we present overview of this thesis and overall organisation.

1.1 Predictive Mutation Testing

For several decades, various techniques have been proposed to reduce the high costs of mutation testing [14, 15, 16]. One such technique is to reduce the number of mutation operators [14] and the number of mutants by removing redundant (e.g., subsumed) ones [17], or accelerate the execution of the tests against the mutants [16, 18], or change the definition of killing mutants to be more relaxed [19].

Recently, Predictive Mutation Testing (PMT) [7] has been proposed to tackle the same problem but in a different way. PMT aims at *predicting* the results of mutation testing using the static and dynamic features of mutants designed based on PIE theory [8]. It does not require any test executions against mutants, but instead uses the coverage information of the tests to learn and predict the detection of the given mutant by the test suite. For example, the two most important dynamic features of PMT are *numTestCovered* and *numExecuteCovered*. Given a mutant to be predicted, PMT computes the former feature by counting the tests that cover the mutated statement, and the latter by counting the times of the mutated statement being executed by the tests. PMT's hypothesis is that the higher those features are, the more likely the mutant is to be killed. The empirical study of PMT using 163 real-world softwares shows that the predictive model trained with its features achieves 0.80 of ROC-AUC and improves the efficiency of mutation testing by up to 151x.

Although the predictive models of PMT have performed well, their features are limited to the structural static features and test-suite level dynamic features. In this thesis, we assume that those features can be less effective at predicting the results of mutation analysis, which is more fine-grained predictions in the test-case level.

1.2 Mutation Based Fault Localisation

Fault Localisation (FL) is a problem of finding the cause of the faults in the Program Under Test (PUT). As FL is often considered a bothersome and time-consuming job by the developers, many automated techniques have been proposed [20]. One promising technique is Spectrum-based Fault Localisation (SBFL). SBFL is a simple yet effective automated technique in that it only uses coverage information to compute spectra of faulty statement [21, 22]. Mutation Based Fault Localisation (MBFL) [1] is another renowned FL technique based on the mutants. Leveraging the mutants' representability of the real faults, MBFL relates the location of mutants and faults and finds the similarities of test executions of both [4, 10, 23]. Compared to SBFL, MBFL requires more resources to collect test executions per mutant, but it has been shown to be more effective than SBFL [10]. For example, given failing test cases that reveal the faults, Metallaxis [4] seeds mutants and collects kill results of each test on the mutants. Then, it uses them like a spectrum of SBFL techniques to rank the statements based on their suspiciousness scores. MUSE [10] utilises the changes of test results against the mutated statements: mutating faulty statements would lead to making the failing tests pass, and vice versa for non-faulty statements. It then uses the changes of test results to rank the statements by their scores.

While such MBFL techniques have demonstrated that mutants can be used to localise the faults, they still have the same problem with the mutation testing: they should inject the mutants and run the tests after the faults are observed, which might hinder the practical application of MBFL.

1.3 Duality of Mutation Testing

Weimer et al. [24] were the first to introduce a duality of mutation testing with respect to generate-and-validate automated program repair (G&V APR) [25], i.e., they regarded the mutants as candidate patches to repair the faults. They examined the basic hypothesis of the mutation testing, i.e., competent programmer hypothesis [11] and coupling effect, and phrased them in the context of program repair. For example, the competent programmer hypothesis states that developers tend to write correct programs but sometimes make simple and syntactic mistakes. APR can be designed to repair such mistakes by generating simple patches. Also, considering the coupling effect, it can be designed to build small patches that represent a high percentage of real-world complex patches.

Furthermore, their formulation of the duality between mutation testing and APR suggests several dualities of existing optimisation techniques used in mutation testing. Mutation testing immediately stops the test executions when a mutant is killed by one of them, and similarly, APR immediately stops when one of the candidate patches passes the tests. APR's use of a limited set of fix-patterns or repair operators is dual of selective mutation [26]. Typically many APR techniques adopt various repair operators to form a patch, and similarly, higher-order mutants are known to be hard to be killed and able to represent many first-order mutants [27].

However, the duality between mutation testing and APR that Weimer et al. [24] suggested is confined to the techniques themselves, i.e., it does not look deeper into the duality of mutants and patches that are the outputs of the techniques. If we find some evidence of such duality, we can bridge two seemingly opposite concepts and broaden the fault or patch datasets used for static bug finding or pattern-based program repair.

1.4 Overview of Thesis

This thesis aims to widen the applicability and scalability of mutation testing by exploiting mutant's relationship with code, faults, and patches. It is organised as follows.

1.4.1 Mutant's Relationship with Code (Chapter 2)

This chapter investigates NL channel in source code near the mutants to predict the killability of mutants without test executions against them. We design several static features based on NL channel such as the mutated statement, test and source method name which the mutants are related to or located in. They are collected in the past version of the program, which amortises the costs of actual mutation analysis. Then, they are preprocessed to be fed into our new model, named Seshat, and used to learn to make test-case level predictions for mutation analysis of the current version of the program. The empirical study with two mutation tools, PIT [28] and Major [29], and real-world faults in Defects4J shows that faster mutation analysis can be achieved with average F-score of 0.83 in predicting the kill matrix for the versions that are up to years apart.

1.4.2 Mutant's Relationship with Faults (Chapter 3)

This chapter introduces a new MBFL technique, named SIMFL, that exploits the mutant's relationship with the real faults for better fault localisation. This has been exploited in other MBFL techniques but our approach utilises different information source from the past codebase (i.e., CI context) to overcome the scalability issues in MBFL techniques. Based on the Bayesian inference, probabilistic coupling, and Multi-Layer Perceptron, we build various statistical inference models that learn the relationship between the locations of mutants and faults. Those relationships are based on the similar patterns of test executions between them: for example, if the faults are only killed by `t1` then we related them with the mutants only killed by `t1`. We then train the statistical model to predict the location of the faults, which can amortise the costs of running tests after faults are observed. The empirical study with real-world faults in Defects4J shows that the proposed models can locate 113 faults on the top out of 224 and outperform the state-of-the-art MBFL techniques. Moreover, We observe that the subsumed mutants disturb and inflate the method-level aggregation of FL scores by SIMFL. As a result, eliminating them significantly improves the effectiveness of SIMFL. Lastly, we investigate a hybridisation of SIMFL and Seshat, which is to use the predicted kill matrix by Seshat for SIMFL to enable a realistic scenario that is not dependent on pre-existence of failing tests. Even if we simulate SIMFL on the predicted kill matrix, it still localises 95 faults on the top out of 194 faults, supporting its practical applicability.

1.4.3 Mutant's Relationship with Patches (Chapter 4)

This chapter explores the mutant's relationship with the patches. The mutants (faults) and patches have been considered two opposed concepts, either trying to find realistic bugs to simulate and prevent future defects, or to automatically generate realistic patches to help developers. Despite the fundamental differences between them, however, we hypothesise that they do not syntactically differ from each other when considered simply as code changes. A large scale empirical evaluation reveals that up to 70% of patches and faults can be clustered together based on the similarity between their lexical patterns; further, 44% of the code changes can be abstracted into the identical change patterns. Moreover, we investigate whether code mutation tools can be used as Automated Program Repair (APR) tools, and APR tools as code mutation tools. In both cases, the inverted use of mutation and APR tools can perform surprisingly well, or even better, when compared to their original, intended uses. For example, 89% of patches found by SequenceR, a deep learning based APR tool, can also be found by its inversion, i.e., a model trained with faults and not patches. Similarly, real fault coupling study of mutants reveals that TBar, a template based APR tool, can generate 14% and 3% more fault couplings than traditional mutation tools, PIT and Major respectively, when used as a mutation tool. Our findings suggest that the valid scope of mining code changes for either mutation or APR can be wider than previously thought.

1.4.4 Conclusion (Chapter 5)

This chapter concludes the thesis and propose a summary of future directions of this work.

Chapter 2. Mutant's Relationship with Code

This chapter explores the mutant's relationship with code by using the natural language channel in the source and test code related to the mutants. We design several static features based on it and feed them into the deep neural network to predict the test-case level mutation results. In addition to test-suite level predictions by Predictive Mutation Testing (PMT), we try more fine-grained predictions that can be used to guide the test case generation or patch generation. We collect the features from the previous codebase that is obtained more than a year ago. Then we train the model that learns from them, which later predicts the mutation results of the current version of the program. An empirical evaluation on 37 subjects in DEFECTS4J and two Java mutation tools shows that our model successfully learns to predict the test-case level mutation results, suggesting that the natural language channel can be a useful source of information to predict the mutant's killability.

2.1 Introduction

A long standing issue in a mutation analysis is its limited scalability [1]. As the size of the System Under Test (SUT) grows, the number of generated mutants also increases significantly. Each mutant then has to be compiled, and executed, to check whether it is killed (i.e., detected as behaving differently) by any of the existing test cases, resulting in a significant, sometimes infeasible, amount of cost. Many different approaches have been proposed to improve the scalability of mutation testing, but they either require more complicated program instrumentation to detect internal state deviation instead of propagated external behaviour (weak mutation [30, 19]), need more sophisticated code mutation that combines multiple mutants into a single compilation (meta-mutation [31]), or simply discard some mutants (mutant sampling [32, 33]). In many of these approaches, the improvement in scalability is linearly bound to the number of mutants (not) analysed.

Recently, Predictive Mutation Testing (PMT) has been proposed to attack the scalability issue in mutation testing from a very different angle [7, 8]. Instead of reducing the number of mutants to analyse, PMT collects test suite level dynamic features that are highly relevant to whether a mutant can be killed or not (such as the number of tests that cover the mutated statement, or the number of times the mutated statement is executed by the test suite), and performs statistical inference about the probability of the mutant being killed by the given test suite. Given sufficient preceding mutation testing results, PMT trains a model that can predict whether a mutant will be killed by a test suite. While PMT can achieve cost saving that is not linearly bound to the number of mutants considered, its limitation is the fact that it can only make a test suite level prediction, which is sufficient to predict the mutation score (i.e., the ratio of killed mutants to the generated mutants) but not the relationship between a mutant and a single test case.

This chapter proposes Seshat[1], a predictive model for the relationship between mutants and individual test cases. Compared to PMT, Seshat can predict the entire kill matrix[2] that results from mutation analysis. We refer to this new type of predictive modelling as Predictive Mutation Analysis (PMA), to

[1] Seshat is an Egyptian deity responsible for writing and record keeping.

[2] Given m mutants and n test cases, a kill matrix M is an m-by-n matrix, where a_{ij} is 1 if the mutant m_i is killed by the test t_j, and 0 otherwise.

emphasise the finer granularity of the prediction, as opposed to PMT, whose outcome is the test suite level mutation analysis.

Seshat exploits the Natural Language (NL) channel in software [13]. Natural Language channel refers to the communication channel that explains the conceptual contexts of the actual executions via natural language elements in source code, such as identifier names. In comparison, the traditional mutation analysis depends on the Algorithmic (AL) channel, via which the semantics of the program (and, naturally, its mutant) is actually computed. The existence of NL channel as a human to human communication medium has been known for a long time [34], but recent advances in the application of language models to source code have revealed that the NL channel can be a rich source of information for various automated tasks, such as code completion [35] or fault localisation [36].

Consider a test case whose name contains domain specific terms, such as `testAccountBalance`. We posit that this test case has a much higher chance of killing any mutants that are generated within the scope of a method named `getAccountBalance()` than another method named `updateEmailAddress(int userID, String emailAddress)`. Given a kill matrix that has been obtained from actual mutation analysis, Seshat can learn the relationship through the similarity between the names of each test case and the mutants it can kill. Since the relationship is learnt in the NL channel, it can later be used to predict the relationship between unseen test cases and mutants, without any execution.

In addition to names, we extract the syntactic and semantic concepts in source code and test cases, using Deep Neural Network (DNN) with the word embedding layers and the bidirectional GRUs [37]. We include the change caused by the mutation, and the type of mutation, as features of our model. Under the cross-version scenario, we evaluate Seshat using different versions of subject programs in the DEFECTS4J benchmark and two mutation tools PIT [28] and Major [29]. Although Seshat solely depends on the static features, the results show that Seshat can predict kill matrices with up to F-score of 0.94 and outperforms PMT and a coverage based baseline model. Notably, Seshat does more than simply memorising the relationship between test cases and mutant locations, as it shows an average prediction F-score of 0.78 for newly added test cases. Compared to generating full kill matrix by executing all individual tests against mutants, the prediction by Seshat is orders of magnitude faster: encouragingly, the bigger the target program is, the higher the speed-up becomes. The contributions of this chapter are as follows:

- We introduce Seshat, a Predictive Mutation Analysis (PMA) technique that can predict full kill matrices for unseen mutants and test cases.
- We formulate predictive modelling of mutation analysis as a machine learning problem in the Natural Language (NL) channel in source code. To our knowledge, this is the first attempt to analyse mutation results using the NL channel.
- We conduct a large scale evaluation of Seshat using multiple versions of real world Java projects in the DEFECTS4J benchmark, and two widely-used mutation tools. Seshat achieves F-score of 0.83 on average, between versions that are years apart. Moreover, Seshat outperforms an existing Predictive Mutation Testing (PMT) technique with finer granularity of the prediction and shows comparable results with PMT when it is used to predict the mutation scores.
- We evaluate whether Seshat can be applied to automatically generated test cases. Our experimentation with EvoSuite suggests that, as long as a meaningful naming convention is upheld during the

generation of the test cases, Seshat can exploit the information in the same way and predict kill matrices of EvoSuite generated test suites with F-score of up to 0.86.

2.2 Seshat: Predictive Mutation Analysis using NL Channel

This section describes how we formulate prediction of the relationship between mutants and test cases via Natural Language (NL) channel. It also presents our model architecture for prediction of kill matrices.

2.2.1 Looking at Mutation through Natural Language Channel

The essential steps of mutation analysis are as follows. First, we mutate the target program using syntactic transformations, i.e., mutation operators. Second, we execute the available test cases against the mutated program. Finally, we check whether the mutant is *killed*, i.e., whether the program behaves differently when mutated and executed by the given test cases. The details of these steps are captured by the PIE theory [38]: for a test case to kill a mutant, it should first **E**xecute the mutant; the execution should result in an **I**nfected internal state, which should be **P**ropagated to the observable output.

All three stages of PIE take place in what Casalnuovo et al. call Algorithm (AL) channel, as opposed to Natural Language (NL) channel [13]; in the source code, the AL channel represents the computational semantics and executions, whereas the NL channel represents the identifiers and comments that assist human comprehension. Most of the existing analyses operate within the AL channel that dictates whether PIE conditions are satisfied or not, while the NL channel has not been considered as an important factor.

An interesting recent advance, Predictive Mutation Testing (PMT), aims to build a predictive model using the features based on the PIE theory [7, 8]. The two dynamic features, which are known to be the most important features in PMT, are related to the test execution: *numTestCovered* (the number of tests in the whole test suite covering the mutated statement) and *numExecuteCovered* (the number of times the mutated statement is executed by the whole test suite). The higher these dynamic feature values are, the more likely it is that the PIE conditions are met. We note that these dynamic features are essentially statistical aggregation within the AL channel, and also that the static features in PMT all concern structural properties of the code that exist in the AL channel.

We propose to reconstruct the results of mutation analysis in the NL channel. Intuitively, the prediction made by PMT using the AL channel is that "if a mutant is executed frequently by many different test cases, it is more likely to be killed". Here, the code coverage is used as a surrogate measure of proximity by computational semantics, as is often the case in regression testing optimisation [39]. Our parallel intuition is as follows: *"if the properties of a mutant are syntactically/semantically similar, or closely related to, those of a given test case, it is more likely to be killed by the test case."*

2.2.2 Predictive Mutation Analysis

A clear benefit of using the NL channel is that we can make predictions about a single mutant and a single test case. On the contrary, PMT depends on the aggregation of the features over *the entire test cases* and the satisfaction of the PIE conditions. We hypothesize that this will hinder making accurate predictions for a single test case. However, the prediction within the NL channel can be made by learning one-to-one relationship between the mutants and the test cases. The one-to-one relationship allows us to predictively build the entire kill matrix, which is required by many applications of mutation analysis

```
public void testFactory_daysBetween_RPartial_MonthDay() {
    MonthDay start1 = new MonthDay(2, 1);
    MonthDay start2 = new MonthDay(2, 28);
    MonthDay end1 = new MonthDay(2, 28);
    MonthDay end2 = new MonthDay(2, 29);

    assertEquals(27, Days.daysBetween(start1, end1).getDays());
    assertEquals(28, Days.daysBetween(start1, end2).getDays());
    assertEquals(0, Days.daysBetween(start2, end1).getDays());
    assertEquals(1, Days.daysBetween(start2, end2).getDays());

    assertEquals(-27, Days.daysBetween(end1, start1).getDays());
    assertEquals(-28, Days.daysBetween(end2, start1).getDays());
    assertEquals(0, Days.daysBetween(end1, start2).getDays());
    assertEquals(-1, Days.daysBetween(end2, start2).getDays());
}
```

Figure 2.1: Example test method of joda-time

```
public static double[] nullToEmpty(final double[] array) {
    if (array == null || array.length == 0) {
        return EMPTY_DOUBLE_ARRAY;
    }
    return array;
}
```

Figure 2.2: Example source method of commons-lang

such as fault localisation [4, 10, 40, 41], test data generation [42, 32, 27], and automated program repair [11, 12, 24, 43]. To distinguish the difference in prediction granularity, we call the one-to-one predictive modelling of mutation results as a Predictive Mutation Analysis (PMA).

2.2.3 Input Features of Seshat

Seshat uses the following features to perform PMA via the NL channel in the source code.

Test and Source Method Name

Test cases are written to target specific parts of the source code: any mutants generated in the corresponding part are, by default, more likely to be killed by those test cases. We exploit the fact that developers often put meaningful names to both source and test code [44]. By exploiting the linguistic links between them, we define two input features for the name of the test and the source method, respectively.

Figure 2.1 shows a test method of joda-time. From its method name, `testFactory_daysBetween_RPartial_MonthDay`, we can deduce that it is likely to test a source method named `daysBetween`. In addition, the name of the class that this test method belongs to is `TestDays`, which indicates that the tests in this class are related to the `Days` class in the source code. Based on this observation, we build new input features for the test method, and their target source method, by concatenating their method and class names, respectively. To handle these features, we use a Deep

Neural Network (DNN) model consisting of the word embedding layers and GRUs [37]. See Section 2.2.5 for more details of our model architecture.

Note that such a linguistic link may not always exist. For example, consider one of the test methods in commons-lang named `testLang865`, which is specifically designed to handle the bug report whose unique identification number is 865.[3] The test actually checks whether `LocaleUtils.toLocale` can parse strings starting with an underscore, but there is no clue of this goal in the name of the test method.

Code Tokens of Mutated Statement

Given a source and a test method, their names as features remain identical for all the mutants that are generated within the source method. However, they do not survive, or get killed by, the same test method collectively. We need additional features that allow us to distinguish individual mutants. To capture the characteristic of each mutant, we take the code snippet of the line in which the mutant is generated, as well as the actual token(s) before and after the mutation. Note that this triplet of information is often provided by the mutation tool themselves.[4]

For example, consider the source method in Figure 2.2. Major mutation tool mutates Line 2 by changing `array.length == 0` to `array.length >= 0`. We use the entire if-statement as well as before/after code fragments:

- Mutated Statement: `if (array == null || array.length == 0)`
- Before: `array.length == 0`
- After: `array.length >= 0`

Similar to the method name features, code tokens of mutated statements are processed by the DNN model. In particular, the before and after code tokens are compared to each other using comparison layers (see Section 2.2.5 for more details).

Mutation Operator

Lastly, we use mutation operator as a categorical input feature to explicitly represent how destructive the operator is. We posit that it may be easier to kill the mutants generated using Return Values mutator than Binary Arithmetic Operation mutator. We use one-hot encoding to represent the type of mutation operator. Note that the dimension of the one-hot encoding depends on the number of mutation operators supported by the mutation tool. PIT provides 11 operators in its default configuration and Major provides nine operators in our setting.

2.2.4 Preprocessing

All textual inputs go through the following preprocessing.

- Word Filtering: The numeric or string literals may exhibit local features that would be difficult to generalise. To avoid overfitting to the local features, we filter numeric and string literals by replacing them with special tokens. Also, we use two more special tokens for unknown word and for removed words due to the mutation.

[3] https://issues.apache.org/jira/projects/LANG/issues/LANG-865

[4] Since PIT does not provide before and after tokens in the report, we exclude them when training models for PIT.

- Subword Splitting: Compound words tend to convey several concepts. For instance, `ConvertToAUTF8String` can be seen as a compound of 'Convert', 'To', 'A', 'UTF8', and 'String'. The compound words not only increase the vocabulary size of the corpus, but also present challenges for effective learning of the word embedding due to their rareness [45]. To address this issue, we employ the state-of-the-art subword splitter Spiral [46] and segment the compound words into subwords.

Finally, all preprocessed tokens for all input features are aggregated to build a dataset and vocabulary. Each element in the dataset represents an one-to-one mapping between a mutant and a test, labelled 0 if the mutant survives the test, and 1 if killed. The dataset is made up of only tests that cover the mutant: label 0 means that the mutant is covered but not killed by the test. This will reduce training and prediction time, as collecting coverage is relatively inexpensive compared to the actual mutation analysis. In the study, we only consider cross-version scenarios, and not cross-project scenarios. This means that we will be less affected by the Out-of-Vocabulary (OOV) problem [47].

2.2.5 Model Architecture

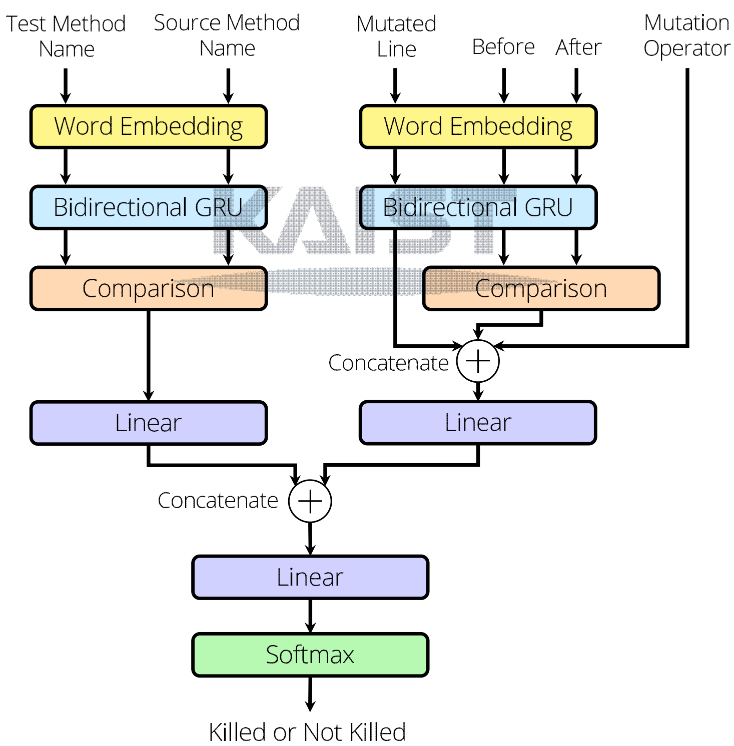


Figure 2.3: Model architecture of Seshat

Figure 2.3 illustrates a model architecture of Seshat. It consists of the word embedding layers to convert words to vector representations, the bidirectional GRU layers to extract sequential context in names and code tokens, the comparison layers to quantify differences between two vector representations, and the linear layer and softmax for the final classification. We detail each component in the following subsections.

Input Layer

The input layer consists of a name-based and a mutation-specific part. We argue that the name-based input features mostly reflect the NL channel in the code, whereas the mutation-specific input features reflect more of the AL channel in the code, as the code tokens contain some parts of program logic. Consequently, we use independent word embedding layer, as well as the bidirectional GRU, for each part.

Word Embedding and Encoding Layer

The word embedding layer maps each word to a numerical representation that captures the relative relationship between words. We use $\mathbf{E_p} \in \mathbb{R}^{|V_p|\times d}$ for the name-based features, and $\mathbf{E_q} \in \mathbb{R}^{|V_q|\times d}$ for the mutant-specific features: V_p and V_q denote vocabularies of the names and code tokens respectively, and d denotes a dimension of the word embedding. We train the embedding layer from scratch as part of the model training, instead of using pre-trained weights.

The words in the test method name $\{w_{t,1}, \ldots, w_{t,n}\}$ and the source method name $\{w_{s,1}, \ldots, w_{s,m}\}$ are passed to the word embedding layer and converted to $x_{t,i} = \mathbf{E_p}(w_{t,i}) \in \mathbb{R}^d$, $1 \leq i \leq n$ and $x_{s,j} = \mathbf{E_p}(w_{s,j}) \in \mathbb{R}^d$, $1 \leq j \leq m$. Subsequently, the bidirectional GRU is used to extract hidden context between words in two directions. For the test method name:

$$\overrightarrow{h_{t,i}} = \overrightarrow{\mathbf{GRU}}(x_{t,i}) \tag{2.1}$$

$$\overleftarrow{h_{t,i}} = \overleftarrow{\mathbf{GRU}}(x_{t,i}) \tag{2.2}$$

$$h_{t,i} = \overrightarrow{h_{t,i}} \oplus \overleftarrow{h_{t,i}} \tag{2.3}$$

The hidden representations of a forward GRU and a backward GRU are concatenated, composing one representations $h_{t,i}$. Next, we adopt an attention mechanism to reward those words that are deemed to be important.

$$u_{t,i} = \tanh\left(W_{att}h_{t,i} + b_{att}\right) \tag{2.4}$$

$$\alpha_{t,i} = \frac{\exp\left(u_{t,i}^{\top}u\right)}{\sum_i \exp\left(u_{t,i}^{\top}u\right)} \tag{2.5}$$

$$v_t = \sum_i \alpha_{t,i}h_{t,i} \tag{2.6}$$

W_{att} denotes a learnable weight matrix and b_{att} is a corresponding bias. The attention vector is normalised by softmax (Equation 2.5) and makes resulting embedding vector v_t through the weighted sum with $h_{t,i}$ (Equation 2.6). The same bidirectional GRU and attention mechanism are applied to $h_{s,j}$, computing v_s. As a result, the v_t and v_s are embedding vectors of test and source method name, respectively, after the bidirectional GRU layer.

The mutation-specific part of the input uses an independent embedding layer $\mathbf{E_q}$ and a bidirectional GRU layer. Except the mutation operator feature that is categorical data and thus one-hot encoded, mutation-specific features are fed into $\mathbf{E_q}$, the bidirectional GRU, and attention mechanism in the same way as the name-based features. At the end of this process, we get three embedding vectors: v_l (mutated line), v_b (before), and v_a (after).

Comparison Layer

Seshat is based on the intuition that the semantic similarity between the names of the source and test method will reflect, and eventually allow us to learn, the mutant-test relationship. To capture the semantic similarity between names, we use a comparison layer [48, 49] to measure similarity between two embedding vectors, v_t and v_s. The comparison layer includes various comparison functions: Neural Tensor Network (Bilinear layer), Neural Network (Linear layer), Cosine and Euclidean similarity, element-wise subtraction and multiplication. For example, Neural Network is a simple network with one linear layer and Neural Tensor Network is similar but with bilinear layer, each of which has learnable parameters W_{NN}, b_{NN}, W_{NT}, b_{NT}:

$$c_{NN} = ReLU(W_{NN}(v_t \oplus v_s) + b_{NN}) \tag{2.7}$$

$$c_{NT} = ReLU(v_t^T W_{NT} v_s + b_{NT}) \tag{2.8}$$

Other comparison functions do not have learnable parameters, and simply compute distances between two input vectors: c_{cos} (cosine similarity), c_{euc} (Euclidean distance), c_{sub} (element-wise subtraction), and c_{mul} (element-wise multiplication). The comparison vectors are concatenated and form an embedding vector v_{ts}:

$$v_{ts} = c_{NN} \oplus c_{NT} \oplus c_{cos} \oplus c_{euc} \oplus c_{sub} \oplus c_{mul} \tag{2.9}$$

With the mutant-specific inputs, before and after represent the changes of the same part of the code. Therefore, we apply the comparison functions to the embedding vector of the before (v_b) and after (v_a): v_{ba}. Note that some comparison functions such as Neural Tensor Network contain learnable parameters, we use separate comparison functions for before and after. To group and reduce the dimensionality of each embedding vector, v_{ts} and v_{ba}, we use two linear layers and concatenate them:

$$v_{tsba} = W_{ts}(v_{ts}) \oplus W_{ba}(v_{ba}) \tag{2.10}$$

The embedding vector v_{tsba} is passed to the final linear layer and the softmax function to produce the probability of the input belonging to each class (i.e., killed or not killed).

$$p = softmax(W(v_{tsba})) \tag{2.11}$$

During training, all learnable weights such as the word embeddings, the weight matrices, or the biases are learnt by minimizing the cross entropy loss.

2.3 Experimental Setup

We design an empirical study that evaluates Seshat on seven different Java projects consisting of 37 program versions in Defects4J.

2.3.1 Research Question

We ask the following five research questions to evaluate Seshat:

RQ1. Effectiveness

Can Seshat learn and predict a kill matrix using the NL channel in the source code and tests? How does Seshat perform against PMT and a coverage based heuristic? RQ1 concerns the effectiveness of Seshat under a cross-version scenario with the comparisons to other models. We answer RQ1 by training Seshat using the actual kill matrices of earlier versions of a subject system, and using it to predict the kill matrices of subsequent versions. We also train a test case level PMT and a coverage based baseline to compare the F-score for each subject.

RQ2. Efficiency

How efficient is Seshat compared to actually performing mutation analysis to obtain an entire kill matrix? RQ2 concerns how much execution time can be saved by Seshat, compared to the execution time of the traditional mutation analysis. We report the time required for preprocessing and predicting the full kill matrix.

RQ3. Generalisation

How well does Seshat generalise to test cases that are newly added to a subject system? If Seshat truly learns through the NL channel, it should retain its predictive power for the new and unseen test cases, as well as the unseen mutants. If Seshat simply memorises the features, on the other hand, its prediction accuracy for newly introduced test cases will be dramatically lower than that for existing test cases. We answer RQ3 by evaluating Seshat separately, for the old and new test cases.

RQ4. Mutation Score

How well does Seshat predict whether a mutant is killed or survives by a given test suite (i.e., mutation score) compared to PMT? RQ4 concerns whether the full kill matrix predicted by Seshat actually produces an accurate aggregated result over the test suite as well. If it does, the results would support that Seshat incorporates the coarser grained PMT. We answer RQ4 by computing the mutation score and F-score using predicted kill matrix by Seshat, and comparing them to the actual mutation score and F-score of PMT.

RQ5. Naming Convention and Automated Test Generation

How sensitive is Seshat when applied to automatically generated test cases and their naming convention? With RQ5, we evaluate Seshat using a test data automatically generated by EvoSuite [50]. We begin by training Seshat with a kill matrix of EvoSuite generated test suite, and predicting the kill matrix of another, independently generated EvoSuite test suite for a subsequent version. In addition, we also cross-evaluate the performance of Seshat between the developer written test suites and the EvoSuite generated test suites. The use of the machine generated test cases has multiple implications. First, unlike the developer written test suites that gradually evolve with the SUT, EvoSuite test suites are generated for each version in our experimental protocol, getting rid of the continuity in the test suite contents. Second, the machine generated test cases follow different naming conventions from those used by developers. We adopt the descriptive naming strategy implemented in EvoSuite [51] and see how much information can be obtained from the NL channel. Finally, we expect the test adequacy of the machine generated test suites to be different from those written by developers in terms of the mutation testing.

In the evaluation, we investigate how much the predictive power of Seshat is affected by these factors when we replace the developer written tests with the automatically generated tests using EvoSuite.

2.3.2 Subject Program

We select 37 subject program versions on seven different projects in DEFECTS4J v2.0.0, which has been widely used in the software testing research, and provides an infrastructure that makes tests run, coverage analysis, and mutation analysis to be easily performed and reproduced. Table 2.1 lists the latest and the oldest subject program versions for each project. Column 1 shows project name, and the number of program versions in each project we use in the study. We denote the version number by the identifier number used in DEFECTS4J, and denote the two adjacent versions, or a version and immediately preceding version by the two versions whose version numbers are right next or before to each other.

To facilitate cross-version scenario, we select programs whose version number is a multiple of five or ten. Note that smaller version number does not always represent more recent codebase; Gson, Cli, JC, and Csv assign smaller version number to older codebase. Chart is the largest subject with 96k LoC and Time has 4k tests which is the most and 74 times larger than tests of Csv which contains only 54 tests. We exclude subjects of Time under the Major mutation tool because their mutation analyses for entire kill matrix have not been completed within 48 hours.

2.3.3 Mutation Tool

Since the result of mutation analysis is highly dependent on the mutation tool and its configuration, we evaluate Seshat using two widely studied Java mutation tools, PIT ver. 1.5.2 [28] and Major ver. 1.3.4 [29]. PIT uses bytecode mutator that is known to be efficient and well integrated with various development environments. Meanwhile, Major adopts compiler-integrated mutator that transforms the abstract syntax tree (AST) and provides in-depth mutation report. Table 2.1 lists the number of generated mutants, killed mutants, and kill percentage of PIT and Major. It shows that large LoC leads to the large number of mutants: Major generates up to 81k mutants for Chart whereas it generates 11k mutants for Csv.

In the study, we filter mutants of PIT that are not captured by our preprocess steps, or killed due to implicit oracles such as time out or uncaught out of memory exceptions, because PIT does not report the tests that caused the kill. Therefore, in Section 2.4.4, we have computed the mutation score ourselves after excluding all those mutants, not using the score reported by the mutation tool.

2.3.4 PMT and Coverage based Baseline Model

To compare the performance of Seshat against other models, we choose two models: PMT and a simple coverage based baseline model. Following the recent study that has investigated the model choices for PMT, we implement a Random Forest classifier using 12 features that has shown the best performance [8]. Most of the features of PMT are collected in the same way, but the test suite level features such as *numExecuteCovered* are collected in the test case level for PMA. In addition, as a sanity check, we also include a coverage based model that predicts all mutants covered by any test will be killed. Coverage information for this heuristic has been collected using Cobertura.

Table 2.1: Subject Program

Project	Identifier	Version	LoC	# Tests	Date	Major		PIT	
						# Mut. Gen.	Killed %	# Mut. Gen.	Killed %
commons-lang	Lang	1	21,788	2,291	2013-07-26	22,793	74.2%	10,546	85.4%
	Lang	10	20,433	2,198	2012-09-27	19,767	74.7%	9,477	85.6%
	Lang	20	18,967	1,876	2011-07-03	19,073	74.5%	8,994	84.3%
	Lang	30	17,660	1,733	2010-03-16	18,144	74.7%	8,234	85.5%
	Lang	40	17,435	1,643	2009-10-22	17,972	74.1%	8,138	73.8%
	Lang	50	17,760	1,720	2007-10-31	18,151	73.0%	8,744	81.3%
	Lang	60	16,920	1,590	2006-10-31	17,819	72.6%	8,506	81.5%
joda-time	Time	1	27,801	4,041	2013-12-02	20,257	65.1%	9,706	78.4%
	Time	5	27,664	4,013	2013-11-01	20,144	65.1%	9,658	78.6%
	Time	10	27,341	3,954	2013-06-16	19,991	65.2%	9,634	78.0%
	Time	15	27,215	3,894	2012-04-30	19,756	65.3%	9,604	77.9%
	Time	20	27,156	3,868	2011-10-23	19,618	65.1%	9,556	77.9%
	Time	25	26,805	3,810	2010-12-05	19,488	65.5%	9,416	77.7%
jfreechart	Chart	1	96,382	2,193	2010-02-09	81,006	23.6%	35,690	33.9%
	Chart	5	89,347	2,033	2008-11-24	75,024	23.6%	33,157	33.7%
	Chart	10	84,482	1,805	2008-06-10	71,052	22.8%	31,183	33.1%
	Chart	15	84,134	1,782	2008-03-19	70,647	22.6%	30,977	32.8%
	Chart	20	80,508	1,651	2007-10-08	67,479	22.2%	30,471	31.4%
	Chart	25	79,823	1,617	2007-08-28	66,766	22.1%	30,213	31.0%
gson	Gson	15	7,826	1,029	2017-05-31	5,044	64.2%	2,670	76.1%
	Gson	10	7,693	996	2016-05-17	4,775	65.4%	2,576	75.7%
	Gson	5	7,630	984	2016-02-02	4,722	64.6%	2,546	75.8%
	Gson	1	5,418	720	2010-11-02	2,295	61.8%	1,564	70.3%
commons-cli	Cli	30	2,497	354	2010-06-17	1,592	81.3%	710	89.2%
	Cli	20	1,989	148	2008-07-28	1,118	77.3%	509	83.5%
	Cli	10	2,002	112	2008-05-29	1,151	68.4%	515	77.9%
	Cli	1	1,937	94	2007-05-15	1,118	66.3%	499	73.3%
jackson-core	JC	25	25,218	573	2019-01-16	30,010	53.8%	13,198	63.5%
	JC	20	21,480	384	2016-09-01	25,257	49.8%	11,115	58.5%
	JC	15	18,652	346	2016-03-21	21,599	48.1%	9,632	56.1%
	JC	10	18,930	330	2015-07-31	22,089	48.6%	9,773	56.6%
	JC	5	15,687	240	2014-12-07	18,610	47.3%	8,227	54.6%
	JC	1	15,882	206	2013-08-28	16,982	46.5%	7,561	53.0%
commons-csv	Csv	15	1,619	290	2017-12-11	1,173	71.4%	599	84.0%
	Csv	10	1,276	200	2014-06-09	1,043	71.1%	493	81.9%
	Csv	5	1,236	189	2014-03-13	996	72.3%	469	81.9%
	Csv	1	806	54	2012-03-27	695	68.1%	280	80.4%

2.3.5 Evaluation Metric and Protocol

For all RQs, we use the standard evaluation metric for binary classification, as PMA is essentially a prediction of binary classes, i.e., killed or not killed. We compute precision, recall, and F-score, but only report F-score for the sake of brevity, as they show similar trends.

To simulate cross-version scenarios, we set the base versions used for training the model and apply them to predict the kill matrices of later versions. For all the two adjacent versions in our subjects, on average, the number of elapsed days is 445, the number of added lines is 14,469, and the number of removed lines is 14,826.[5]

2.3.6 Configuration and Environment

We use a default set of mutation operators for PIT and default template file (`.mml`) specified in DEFECTS4J for Major. We run mutation analyses using PIT with two threads and `fullMutationMatrix` options. For Major, we use `sort_methods` and `exportKillMap` options to enable it to compute the entire kill matrix in a test case level. For the versions in Chart, we run Major within the `killmap` tool to reliably manage stack overflow errors involving JVM crashes.[6]

In the preprocessing step, we perform lexical analysis using a Java lexer implemented in SLP library.[7] In training model, the dimension of word embedding is set to 50, the number of features in the hidden state of GRU is set to 100, and dropout rate is set to 0.5. The maximum training epoch is 10.

The mutation analysis using `killmap` tool was performed on four machines each of which runs Ubuntu 18.04.4 LTS on Intel i5-10600 CPU @ 3.30GHz and 16GB RAM. All remaining experiments were performed on machines running Ubuntu 16.04.4 LTS, Intel Xeon E5-2630 v4 CPU, Nvidia TITAN Xp GPU, and 256GB RAM.

2.4 Results

This section answers the research questions using the results from the empirical evaluation of Seshat.

2.4.1 Effectiveness (RQ1)

Figures 2.4 and 2.5 show how the F-score changes when we train a model using a kill matrix of a base version and predict kill matrices of subsequent versions that are gradually farther away. The x-axis shows the relative time interval between versions: the longer the time interval between versions, the further the distance in x-axis between them. We also specify the exact number of elapsed days between the first two versions in the x-axis for the sake of understanding. The colours in Figures 2.4 and 2.5 represent the models we use: the blue circle marker represents Seshat, the green X marker represents PMT, and the orange triangle marker represents the coverage based baseline. In the same colour variation, the model trained with the older program is marked as darker colour, and the connected line between markers represent that the models are trained using the same base version.

Figure 2.5 shows that Seshat performs the best for versions of Chart on PIT: the average F-score from all pairs of versions of Chart is 0.92. For all subjects in PIT, Seshat achieves the average F-score 0.84. Compared to PIT, as shown in Figure 2.4, Seshat shows slightly worse prediction performance on Major, with the overall average F-score 0.81.

Also, Seshat outperforms PMT and the coverage based model. PMT and the coverage based model produce average F-scores of 0.70 and 0.39, respectively, when predicting the kill matrices of PIT, and 0.68 and 0.19 for the kill matrices of Major. This supports our hypothesis that PMT lacks fine-grained features needed for test case level prediction. It is not surprising that the coverage based heuristic performs the worst, since the coverage is not a sufficient condition for killing a mutant.

[5] We use `git-diff` tool to extract changed lines in `.java` files.
[6] We obtained the tool from https://bitbucket.org/rjust/fault-localization-data/src/master/killmap (commit ea1741dbb) and then modified it to run all given test cases.
[7] https://github.com/SLP-team/SLP-Core

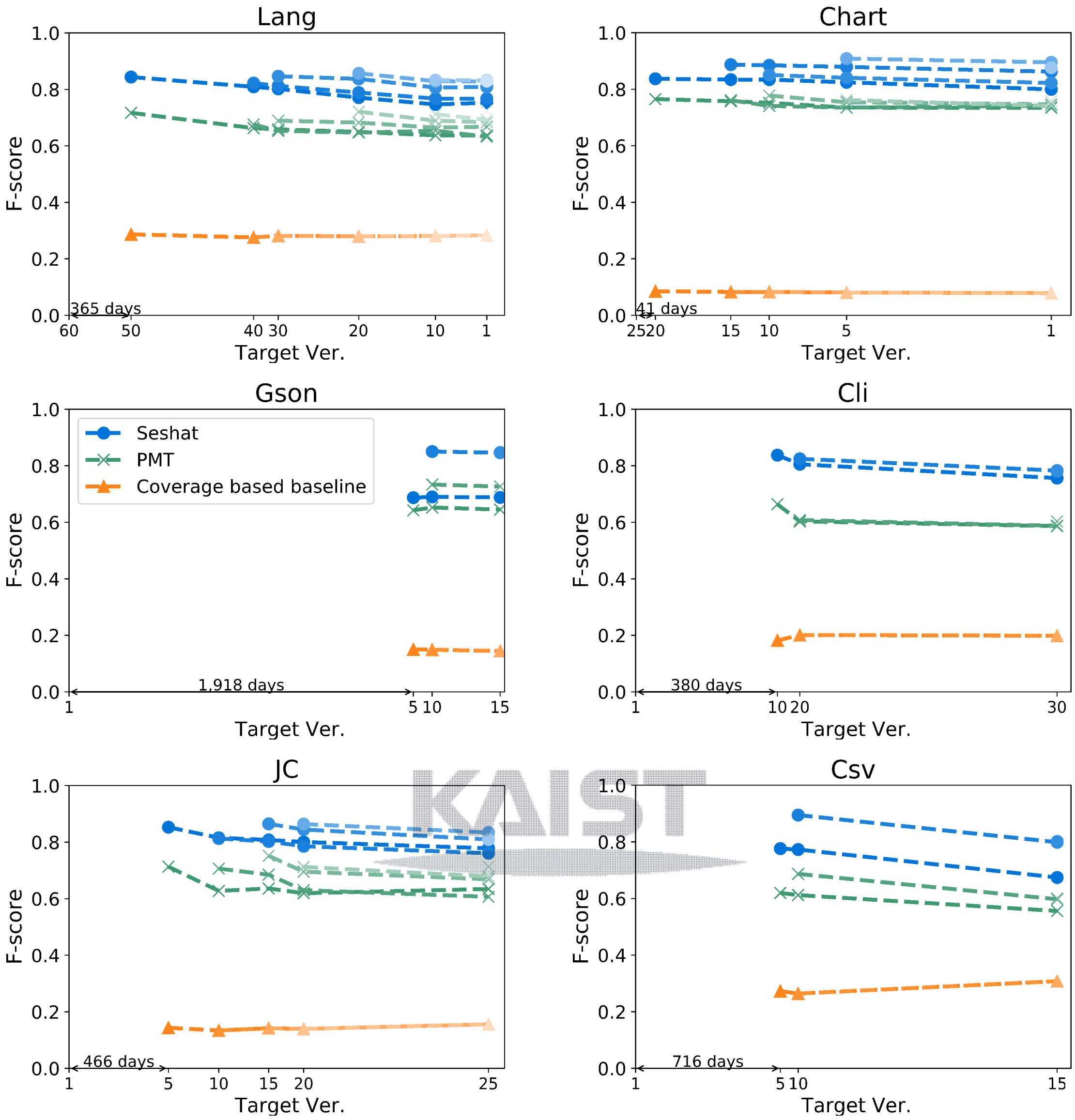


Figure 2.4: Prediction of the full kill matrix on Major

By comparing lines with different tints in the same colour variation, we can observe how the time between two versions affects the prediction accuracy of Seshat. The older the model used for training is, the worse the prediction becomes. When predicting for the latest version, models trained on the immediately preceding studied version shows 0.08 and 0.09 point higher F-scores when compared to models trained on the oldest versions, for Major and PIT respectively. However, we also note that the accuracy degradation is relatively slow: the F-scores decrease by 0.019 and 0.013 point per year on average for Major and PIT, respectively. PMT also follows this trend but the coverage based model does not.

The worst F-scores are observed in the models that are trained with the oldest versions of Gson and Csv. With both projects, the first interval is much longer than those in others: over five years for Gson, and almost two years for Csv. Combined with the relatively smaller size of these two projects, we suspect that the codebases for these two projects have changed more than other projects, which makes

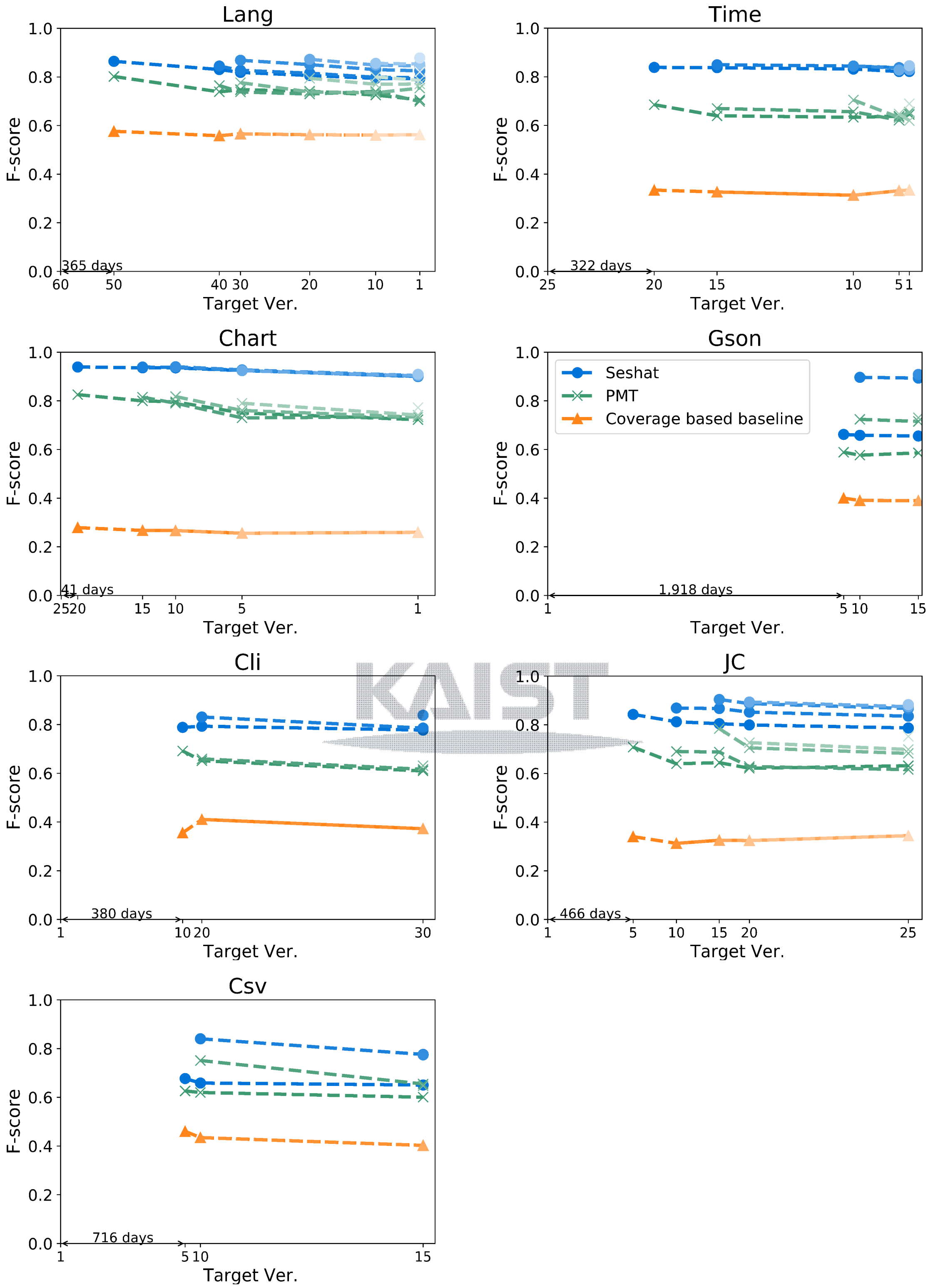


Figure 2.5: Prediction of the full kill matrix on PIT

the prediction harder. For example, the oldest version of Gson only produces about half the mutants then those produced from the next analysed version, resulting in F-scores consistently lower than 0.7 for all predictions based on itself.

> **Answer to RQ1:** Seshat successfully learns the features in the NL channel that are effective in predicting the full kill matrices: Seshat achieves average F-score of 0.81 and 0.84 for Major and PIT, respectively, which significantly outperforms PMT and the coverage based model.

Table 2.2: Efficiency of Seshat

Project	Ver.	Major	Seshat	Speed-up	PIT	Seshat	Speed-up
Lang	1	12,924s	**267s**	48.34X	1,472s	**128s**	11.46X
	10	13,185s	**244s**	54.10X	1,385s	**116s**	11.95X
	20	5,395s	**214s**	25.15X	970s	**104s**	9.30X
	30	5,220s	**212s**	24.61X	900s	**97s**	9.24X
	40	4,756s	**206s**	23.08X	836s	**93s**	9.02X
	50	6,793s	**196s**	34.72X	865s	**93s**	9.29X
Time	1	-	-	-	2,294s	**616s**	3.73X
	5	-	-	-	2,170s	**592s**	3.67X
	10	-	-	-	1,957s	**614s**	3.19X
	15	-	-	-	2,138s	**564s**	3.79X
	20	-	-	-	2,309s	**557s**	4.15X
Chart	1	64,719s	**1,248s**	51.87X	2,295s	**369s**	6.22X
	5	53,986s	**1,093s**	49.40X	2,014s	**338s**	5.95X
	10	46,983s	**998s**	47.07X	1,542s	**293s**	5.26X
	15	46,429s	**962s**	48.27X	1,520s	**289s**	5.26X
	20	42,475s	**873s**	48.68X	1,466s	**258s**	5.68X
Gson	15	16,738s	**347s**	48.27X	376s	**118s**	3.19X
	10	15,986s	**339s**	47.13X	351s	**120s**	2.94X
	5	15,253s	**348s**	43.82X	345s	**113s**	3.04X
Cli	30	1,290s	**35s**	36.98X	74s	**19s**	3.81X
	20	498s	**37s**	13.35X	43s	**20s**	2.12X
	10	408s	**32s**	12.94X	42s	**18s**	2.30X
JC	25	113,343s	**538s**	210.53X	1,391s	**204s**	6.81X
	20	88,075s	**392s**	224.44X	1,013s	**142s**	7.16X
	15	45,069s	**317s**	142.17X	669s	**115s**	5.82X
	10	44,110s	**305s**	144.85X	676s	**113s**	5.99X
	5	31,257s	**209s**	149.49X	557s	**83s**	6.72X
Csv	15	5,289s	**52s**	101.00X	1,781s	**23s**	77.91X
	10	1,317s	**33s**	40.30X	1,359s	**16s**	83.12X
	5	1,179s	**31s**	37.44X	1,493s	**16s**	93.90X

2.4.2 Efficiency (RQ2)

The main drawback of building a full kill matrix comes from its huge computational cost. It requires even more time than traditional mutation testing, because even if a mutant is killed by some test earlier, it does not skip running other tests. For example, considering Lang 10, traditional mutation testing using PIT was 4.3x faster than those with full kill matrix option. Therefore, we investigate how much execution time can be saved by Seshat, compared to that of traditional mutation analysis with full kill matrix option.

Table 2.2 shows the execution time of two mutation tools and Seshat to compute the full kill matrix. Columns 3 and 4 list the execution time of Major and Seshat that predicts Major's kill matrix of the corresponding version, respectively. Similarly, Columns 6 and 7 list the execution time of PIT and Seshat. Overall, the results show that PIT can compute the full kill matrix in a reasonable time: it takes up to 38 minutes for Time and up to 74 seconds for Cli. Moreover, PIT is on average 28x faster than Major since Major has generated more mutants than PIT in our study. Still, Seshat can speed up from 2x to 94x, compared to PIT. In particular, Seshat predicts kill matrices of Csv exceptionally faster than others; although all Csv versions have fewer than 300 tests, they are executed relatively slow, resulting in PIT running more than 22 minutes. In contrast, Seshat is not affected by the test executions, but only the number of tests, so it can predict kill matrices in less than 23 seconds. Seshat also requires significantly less time than Major for all subjects, with the average speed-up of 68x. Major runs more than 31 hours for JC 25, but Seshat takes only nine minutes.

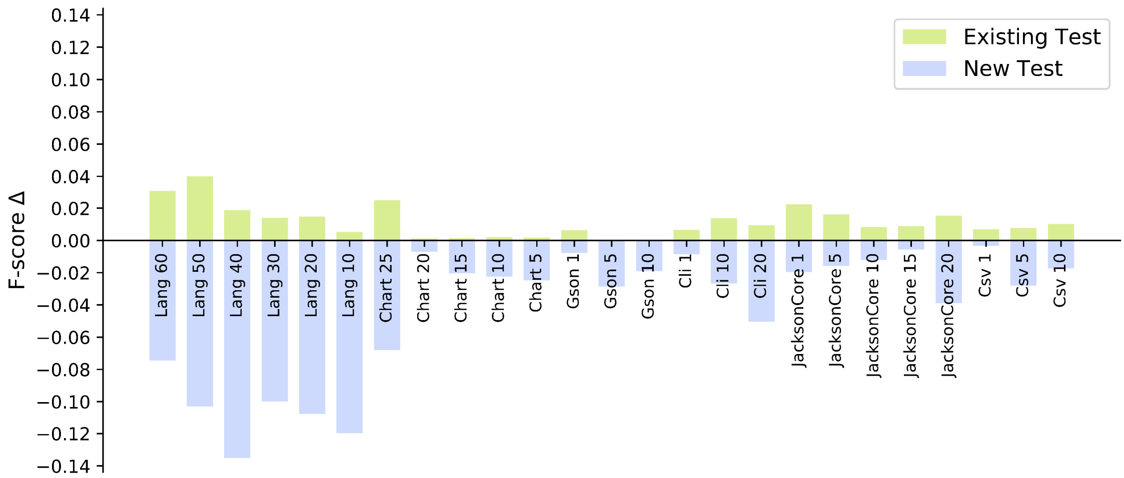
Note that the reported speed-ups are specifically in the context of predicting the entire kill matrices. It is theoretically possible that Seshat can be even more efficient if we only wanted to predict the mutation scores like PMT: we simply need to stop the prediction for a mutant once it is predicted to be killed by any test case. However, as Seshat maps one-to-one relation between multiple mutants and test cases, the inference phase is fully parallelised using a GPU across the elements in the kill matrices, making mutation score level evaluation of efficiency difficult. Even if we serialise the computation, the ordering of prediction would affect when the ideal early stopping point should be. Consequently, we limit our evaluation of efficiency to the prediction of entire kill matrices.

Answer to RQ2: Seshat is much more efficient in predicting the full kill matrix than traditional mutation analysis, with the average speed-up of 68x against Major and 14x against PIT.

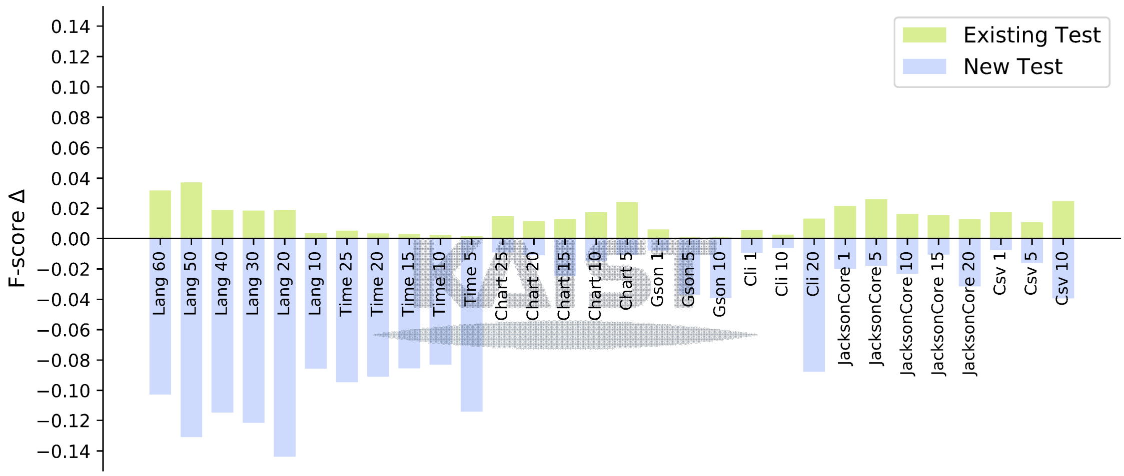
2.4.3 Generalisation (RQ3)

In RQ3, we investigate whether Seshat can generalise to the unseen test cases that did not exist in the test suite of the version used for training. Before prediction, we separate the test cases into two categories, existing and new. Subsequently, we report the prediction F-score for each group independently. While we only categorise test cases, we posit that the new test cases are likely to cover newly introduced code in the version that we apply Seshat to.

Figure 2.6 shows how F-score changes between the old and the new test categories per projects. The x-axis represents the version used for training, and the y-axis represents the averaged difference in the F-scores when both categories are compared to the F-score computed using all tests: the F-score Δ of 0 means that the category produces the same F-score as that of the entire test suite.



(a) Major



(b) PIT

Figure 2.6: Changes of F-score of existing tests and new tests

For all subjects, the existing tests show F-scores that are higher by 0.01 and 0.01 on average, while new tests show F-scores that are lower by 0.04 and 0.05 on average, Major and PIT respectively. The difference suggests that Seshat does perform better for the existing test cases. The notable outliers are from the versions in Lang, for both Major and PIT: the average F-score Δ is -0.11. We suspect that this is related to the test case granularity: test cases in Lang tend to cover fewer methods compared to other projects. In our analysis, a test case in Lang covers about 14 methods on average, whereas a test case in all the other projects covers about 127 methods on average. The finer granularity may drive Seshat to learn stronger one-to-one mappings between the test cases and the source methods, with higher risk of overfitting. In turn, this will make it more difficult for Seshat to make an accurate prediction for the new test cases. Another outlier is the versions in Time analysed with PIT; however, the reason may be the opposite because Time has the highest average number of methods covered by a test case, which is 360. It is more difficult to learn mappings between the source and the test methods if the relationship is dominantly one-to-many, resulting in low performance of Seshat.

> **Answer to RQ3:** Seshat can generalise to the new source and test code: the F-score decreases only by 0.04 and 0.05 with Major and PIT, respectively, for new and unseen test cases.

2.4.4 Mutation Score (RQ4)

The test case level of PMA is finer-grained than the test suite level of PMT: once we predict a kill matrix, we can easily compute the Mutation Score (MS) from the matrix. We expect a conversion in the opposite direction to be more challenging. This is because, intuitively, PMA is more challenging as a learning problem than PMT, since it needs to learn and produce more information (an entire kill matrix) compared to PMT (which essentially predicts an aggregation of a kill matrix).

Table 2.3: Mutation Score Prediction by Seshat and PMT

Project	Major				PIT			
	F-Score		MS Error		F-Score		MS Error	
	Seshat	PMT	Seshat	PMT	Seshat	PMT	Seshat	PMT
Lang	**0.825**	0.715	**0.848**	17.547	0.908	**0.912**	**2.112**	4.503
Time	-	-	-	-	0.894	**0.903**	**0.254**	6.091
Chart	**0.818**	0.747	**3.047**	4.641	**0.896**	0.827	**1.785**	6.306
Gson	0.625	**0.644**	10.837	**9.590**	**0.817**	0.795	**4.281**	6.445
Cli	0.669	**0.781**	23.943	**7.922**	0.847	**0.905**	10.813	**3.583**
JC	**0.775**	0.693	**4.370**	9.755	**0.851**	0.832	**0.595**	5.862
Csv	**0.672**	0.668	17.286	**15.007**	0.844	**0.856**	**1.166**	7.149

Table 2.3 shows the F-score and MS error (i.e., $|MS_{Real} - MS_{Pred}|$) when Seshat and PMT predict whether a mutant is killed by the entire test suite. In general, it is hard to determine which model performs better because the differences of both F-score and MS error vary considerably between the subjects. However, despite the aggregation over the predicted kill matrix to compute MS, Seshat shows comparable and sometimes better results than PMT.

In addition to the analysis of the absolute error, we further investigate the actual difference (i.e., $MS_{Real} - MS_{Pred}$) for all predictions by Seshat and PMT. In total, out of 55 predictions, 11 MS_{Pred} by Seshat are higher than MS_{Real}. In contrast, all 55 MS_{Pred} by PMT are higher than MS_{Real}. This indicates that PMT tends to overestimate mutation scores, while the errors of Seshat are both over and underestimations. For Seshat, the observed trend is plausible, because some of the kill relations may be very unique, and therefore harder to learn properly. In comparison, considering the importance of dynamic features for PMT, the overestimation by PMT is in line with the widely known limitations of structural coverage: it relies significantly on features related to the execution of the mutant, but the coverage does not necessarily result in killing it.

A hybrid model that combines the features of Seshat and PMT may improve the prediction performance for the MS, as well as the contents of the kill matrix. However, the focus of this chapter was to evaluate the feasibility of Seshat based on the NL channel. We leave further investigation of hybridisation as future work.

> **Answer to RQ4:** Seshat can predict the mutation results with average F-score of 0.80, and average mutation score error of 6.26, producing comparable results with PMT that achieves 0.79 and 8.03 respectively.

2.4.5 Naming Convention and Automated Test Generation (RQ5)

Table 2.4: Predicting kill matrix by replacing developer-written tests (Dev) with EvoSuite tests (Evo)

Project	Target Vesrion	Dev_{train} Evo_{test}	Evo_{train} Dev_{test}	Evo_{train} Evo_{test}	Dev_{train} Dev_{test}
Lang	60	0.61	0.68	0.74	0.86
	50	0.61	0.67	0.72	0.84
	40	0.58	0.64	0.76	0.87
	30	0.62	0.62	0.71	0.87
	20	0.64	0.65	0.77	0.86
	10	0.65	0.67	0.76	0.88
Time	25	0.65	0.66	0.77	0.84
	20	0.66	0.64	0.76	0.85
	15	0.66	0.65	0.74	0.84
	10	0.65	0.64	0.73	0.83
	5	0.66	0.66	0.76	0.85
Gson	1	0.56	0.60	0.67	0.66
	5	0.72	0.64	0.82	0.90
	10	0.71	0.66	0.81	0.91
Cli	1	0.57	0.59	0.52	0.79
	10	0.56	0.60	0.87	0.83
	20	0.57	0.60	0.86	0.84
Csv	1	0.57	0.58	0.66	0.68
	5	0.68	0.71	0.76	0.84
	10	0.57	0.68	0.70	0.77

We evaluate Seshat with EvoSuite [50] generated test cases in RQ5: the aim is to investigate whether the NL channel still holds useful information when test cases are automatically generated. Instead of relying on naming conventions of human developers, we use the descriptive naming strategy of EvoSuite [51]. We hereafter denote developer-written test suites with Dev, and EvoSuite generated test suites with Evo. We first collect kill matrices of all Evo test suites. Subsequently, we train Seshat model on Evo (i.e., Evo_{train}) and test on Dev (i.e., Dev_{test}), or train Seshat model on Dev (i.e., Dev_{train}) and infer kill matrix for Evo (i.e., Evo_{test}). In addition, we provide the results of Evo_{train} & Evo_{test} and Dev_{train} & Dev_{test}. Note that we only conduct the experiment using PIT because running Major on EvoSuite tests times out after 48 hours.

Table 2.4 presents the F-scores for each train-test pair. Overall, Dev_{train} & Dev_{test} pair achieves the highest F-scores, followed by Evo_{train} & Evo_{test} pair. We posit that two factors contribute to the performance of Seshat with Dev test suites. First, human written tests may provide richer semantic information in the NL channel. Second, Dev is a regression test suite, whereas Evo test suites are obtained by independent and separate runs of EvoSuite for each version. Since Dev test suites are more stable (i.e., many test cases overlap between versions), the prediction of kill matrices may be easier for Dev test suites. However, the descriptive naming strategy of EvoSuite does provide some information in the NL channel, allowing Seshat to achieve F-score of up to 0.86 under the configuration of Evo_{train} & Evo_{test}.

In contrast, replacing Dev with Evo results in deterioration of the effectiveness of Seshat. Dev_{train} & Evo_{test} and Evo_{train} & Dev_{test} show similar results, achieving average F-scores 0.62 and 0.64 respectively. The differences in naming convention between training and testing make the kill matrix prediction more difficult.

> **Answer to RQ5:** As long as a meaningful naming convention is applied during test generation, Seshat can also predict kill matrices for automatically generated test cases.

2.5 Discussion

This section raises a few issues that we have identified while performing the empirical evaluation. We believe that these can lead into interesting future work.

2.5.1 Test Quality and Coding Convention

Suppose we take any test suite for which Seshat can successfully perform PMA, and remove all assertions from its test cases. This will render any prediction useless. What Seshat does is to reconstruct the relationship between test cases and mutants in the NL channel, based on the training data. Consequently, it is vulnerable to such manipulation. In contrast, if a test case without any assertion always kills a mutant using an implicit oracle (such as crashes), Seshat will duly learn this relationship. In practice, we expect Seshat to be used to reduce and amortise the cost of mutation analysis, and not to completely replace it. If there is a continuity in the coding and naming convention in both source and test code, Seshat is likely to maintain its prediction accuracy. If there is a meaningful change in test quality, either improvement or decrease, the prediction accuracy will degrade. We expect this deviation to be picked up by the regular application of concrete mutation testing.

2.5.2 Data Imbalance in PMA

The balance between the number of mutant-test pairs that result in kills and non-kills cannot be known in advance, and likely not to be perfectly 1:1. As Seshat is essentially solving a classification problem, here we consider the implications of data imbalance.

The text-based nature of Seshat prevents the use of resampling approaches such as SMOTE [52] or ADASYN [53]. Therefore, we investigate the impact of class imbalance on Seshat using random over- and undersampling (allowing duplicates). We over- and undersample the training data to make the class ratio 1:1. Subsequently, we train new models and evaluate them on the same subjects. Compared to the results of RQ1, with Major, the average F-score increases by 0.026 and 0.025 when we use over-and undersampling, respectively. With PIT, however, F-score decreases by 0.001 with oversampling, and increases by 0.010 with undersampling.

The different responses to over- and undersampling between Major and PIT can be attributed to the status of their initial imbalance. The initial class ratio between killed and not killed mutants generated by Major across all studied versions is 0.396 on average. With PIT, the initial imbalance ratio is much minor at 0.985 on average: there are actually more killed mutants than not killed ones in some projects. We suspect that this leads to over- and undersampling having more random effects on the results.

We note that the difference in F-score between initial and resampled results are not significant. Data imbalance will have more significant consequences if it only appears in the training data. However, due to the continuity in development, projects with imbalanced kill matrices are likely to exhibit the same imbalance in the future, resulting in relatively minor consequences in our cross-version evaluation.

Overall, we conclude that addressing the class imbalance issue can improve the performance of Seshat in general, but the implications of the imbalance can be subtly different depending on the choice of the mutation tool, its configuration, and the contents of the kill matrix itself.

Table 2.5: Results of Ablation Study on Major

Project	Decrease in F-score			
	Method Name	Mutated Line	Before&After	Mutation Op.
Lang	0.009	-0.003	0.041	-0.011
Chart	-0.056	-0.057	-0.030	-0.072
Gson	-0.010	0.028	0.025	-0.014
Cli	0.018	-0.003	0.078	0.014
JC	-0.006	0.016	0.018	-0.015
Csv	0.050	-0.015	0.142	0.045

2.5.3 Ablation Study

To investigate which feature in Seshat contributes most to the prediction, we conduct an ablation study by removing each feature one by one and training the model. We then repeat the RQ1 study with Major to compare the F-scores from the ablated models to those from the original model. Table 2.5 reports the decrease of F-score for each removed feature. The column "Method Name" refers to the model that omits "Source Method Name", and the column "Before & After" refers to the model that omits both "Before" and "After" in the model architecture, shown in Figure 2.3. The columns "Mutated Line" and "Mutation Operator" refer to the models with corresponding input components removed, respectively.

Interestingly, the results show that the model performance varies across the subjects but do not always deteriorate: for the versions of Chart, the ablated models perform better than the original model. However, the average difference of the F-score is 0.008, which may be too small to precisely assess the relative feature importance when considering the stochastic nature of training DNN models using hyperparameters. We presume that our features share some common information and complement each other, allowing the ablated model to retain its predictive power. We leave the design of more destructive study to measure the relative feature importance as future work.

Table 2.6: The changes between versions reported by cloc, only considering changed Java files.

Project	Versions	# Commits	Added	Modified	Removed
Lang	1 ↔ 10	189	3,932	6,436	244
	10 ↔ 20	413	7,829	2,836	1,456
	20 ↔ 30	552	4,965	1,120	1,834
	30 ↔ 40	286	45,689	2	44,357
	40 ↔ 50	444	7,819	1,250	10,540
	50 ↔ 60	166	2,333	435	464
Gson	15 ↔ 10	89	637	222	74
	10 ↔ 5	79	377	21	137
	5 ↔ 1	823	14,474	950	8,220
Time	1 ↔ 5	16	583	873	21
	5 ↔ 10	62	1,306	229	548
	10 ↔ 15	68	623	81	11
	15 ↔ 20	31	396	71	11
	20 ↔ 25	65	855	136	99,260
Chart	1 ↔ 5	186	13,081	3,205	3,350
	5 ↔ 10	349	9,598	9,122	1,181
	10 ↔ 15	50	1,046	917	196
	15 ↔ 20	163	7,087	1,218	3,842
	20 ↔ 25	31	1,527	652	110
Cli	30 ↔ 20	252	5,678	0	4,566
	20 ↔ 10	51	941	597	1,222
	10 ↔ 1	61	520	72	141
Csv	15 ↔ 10	410	2,453	1,209	383
	10 ↔ 5	53	435	219	221
	5 ↔ 1	489	2,350	497	675
JC	25 ↔ 20	304	10,684	459	725
	20 ↔ 15	150	7,522	755	3,387
	15 ↔ 10	142	1,092	194	1,037
	10 ↔ 5	260	6,387	360	1,130
	5 ↔ 1	263	3,383	2,127	2,096

2.5.4 Relation to Regression Mutation Testing

We introduce and discuss the differences between Seshat and Regression Mutation Testing (ReMT) [54], as ReMT has a similar goal as Seshat, namely to reduce the cost of mutation analysis in the context of evolving programs. When considering successive commits of evolving systems, ReMT reuses mutation testing results from the previous version and selects only the subset of tests that are affected by the latest change to rerun and update the kill matrix. If the underlying Regression Test Selection (RTS) technique is sound and complete, ReMT can output the correctly updated kill matrix with the minimum effort required.

ReMT is the most efficient when successively applied to each and every version: if intervals become longer, changes will accumulate, making ReMT increasingly inefficient (i.e., it will have to execute more and more tests and mutants). This phenomenon is known to exist in the context of regression test case selection [55]. ReMT also involves other cost and assumptions. For example, to precisely select mutants whose kill outcome may change in the new version, some techniques adopt static analysis techniques that may incur additional cost; to determine the mutant kill accurately, we also need to assume that the current version is correct.

Seshat can be complementary to ReMT, as the results show that it can provide reasonable predictions of kill matrices across much longer intervals, incurring very little cost, without assuming anything about the correctness of the current version. As an extreme example, suppose we want to update the kill matrix of Lang 60 to get the kill matrix of Lang 1, two versions with 1,590 and 2,291 tests respectively (see Table 2.1). Even if we do not consider any modified tests between two versions, at least 701 *newly introduced tests* (44%) will have to be run. We do not think that this is a reasonable scenario for ReMT. Table 2.6 shows the magnitudes of accumulated changes between versions, listing the number of commits, added, modified, and removed lines between two adjacent versions. The number of commits between adjacent versions, on average, is 216, which is a wider interval than ReMT's setting that only considers the consecutive commits.

2.6 Threats to Validity

The major threats to internal validity lie in the implementation of Seshat as well as the correctness of actual mutation analysis that provides training data to Seshat. The models are implemented using widely used frameworks that withstood public scrutiny, such as Scipy and PyTorch. For training data, we depend on the publicly available mutation analysis script provided by Defects4J.

The threats to external validity concern the choice of subjects, programming language, and the mutation tools. Due to the nature of our technique, the results are dependent on the coding conventions adopted by the studied projects. We tried to use the most recent version of Defects4J with more diverse projects to avoid overfitting to a specific coding style. Since writing informative names is regarded as best practice, we expect the NL channel in source code written in other programming languages will provide similarly rich information. We adopt two mutation tools, Major and PIT, to reduce the threats related to the choice of a mutation tool.

The threats to construct validity include evaluation metrics we used to draw the conclusions. All evaluation metrics are standard measures for classification tasks: precision, recall, and F-score.

2.7 Related Work

Existing approaches for mutation testing cost reduction include mutant selection, mutant execution acceleration, and mutant score prediction [56]. Offutt et al. [14] suggested to use a small set of sufficient mutation operators instead of using a vast number of mutation operators. Namin et al. [15] advanced mutation operator selection by leveraging a multi-objective linear regression which can learn an optimal mutation operator set according to given corpus of mutation testing results. To reduce the runtime of mutant executions, Zhang et al. [16] applied the idea of test case prioritization to improving the rate of mutant kills. AccMon [57] prevented redundant mutant executions by monitoring internal states and cutting off a mutant execution if it fails to induce different infection states. Regression Testing Selection (RTS) [18] presented a practical Regression Mutation Testing [54], by selecting relevant tests that should be rerun to determine whether the new mutants are killed. Despite having a similar regression assumption with ours, we note that the RTS approach is specifically aimed at reducing the mutation execution cost, while Seshat takes one step further to provide a predictive model. Consequently, the RTS approach can be combined with Seshat when Seshat needs to periodically update its reference kill matrix by (re)running test cases. Recently introduced Predictive Mutation Testing (PMT) [7] opened up a new dimension in mutation testing by seeking to predict whether a mutant will be killed or not by the entire test suite based on the structural information on the target program and the coverage information on the given test suite: consequently, it does not require any mutant execution. Seshat, whoever, aims for a finer granularity in its prediction by attempting to infer the entire kill matrices, instead of only predicting whether a mutant is killed by the test suite or not.

The finer-grained, kill matrix level prediction may be utilised in any application that depends on the one-to-one relationship between mutants and individual test cases. Test generation techniques that use the mutants as a guidance are one of such applications. For example, to create test cases that can kill Subsuming Higher Order Mutants (SHOM) [27], one needs to start from the entire kill matrices of First Order Mutants, which Seshat can approximate. DEMiner [42] proposed to improve a concolic testing based on the information of which mutation affects which test execution: if there exists an NL channel in test cases, Seshat can be used to approximately capture the relationship at a lower cost. Some existing Automated Program Repair (APR) techniques have taken advantage the mutation analysis. Weimer et al. [24] highlighted Generate-and-Validate program repair as a dual of the mutation testing, while PraPR [43] explicitly exploited mutation tools for APR. The high cost of patch validation in APR suggests that, in both cases, Seshat may partially replace concrete mutation analysis to find an attractive trade-off between analysis cost and accuracy.

A downstream task that can directly benefit from PMA in particular is Mutation Based Fault Localisation (MBFL) [20]. MBFL techniques have exploited a mutation analysis that relates source code and tests through mutants [41, 4, 10, 40]. By introducing the syntactic modifications (i.e., mutants) to the buggy program, MBFL techniques observe the changes of test results then assign a suspiciousness score to each statement to identify the possible location of the given faults. For instance, MUSE [10] and MUSEUM [40] are based on the two conjectures: mutating correct statements is likely to make passed tests fail, and mutating faulty statements is likely to make failed tests pass. Then they compute the suspiciousness scores using the ratio of fail-to-pass tests and pass-to-fail tests. Using Spectrum based Fault Localisation (SBFL)-like formulas, Metallaxis [4] mutates the buggy program and see whether the test results show a similar pattern with the patterns of the faults. SIMFL utilises a statistical inference on the kill matrix to find the mutant that makes tests fail in a similar pattern with the faults, then it suspects that the faults would be close to the location of the mutant [41]. In all cases, the cost of mutation analysis is directly added to the cost of debugging, as these techniques tend to perform mutation analysis once a bug is detected. Seshat can be easily applied to them to reduce the cost of the mutation analysis.

Even though the mutation testing has been considered a robust measurement of test effectiveness, code coverage has been widely used as an alternative proxy due to its efficiency. It measures how well the specific code structures are exercised by the tests, e.g., a statement coverage checks whether the line is executed by at least one test case. Several studies have shown that there is a correlation between code coverage and test effectiveness [58, 59], which supports the use of the coverage as one of our baselines. However, as it does not convey the intent of the given test suite, further studies have found that the test size and the assertion should be taken into account to quantify test effectiveness [60, 61]. Checked coverage [62] is one such example that attempted to incorporate the information of the assertions in the coverage using backwards slicing on the assertions in the test. However, its dependence on the static analysis may limit the applicability of checked coverage.

Moreover, our work is also different from the existing cost reduction techniques in that Seshat utilises the natural language channel between the target program and the test cases to construct the prediction model. Recent improvements of neural language models have enabled various techniques to exploit the semantic information inferred from the natural language channel in programs [35, 36, 13]. Since the natural language channel is both related to and constrained by the algorithm channel [13], a hybrid approach would be promising to overcome the limitation of any existing analysis that aims to reason about program semantics. For example, neural word embedding has been used to discover semantically similar code snippets [63], as well as specifications [64]. Recurrent Neural Network (RNN) models have been used to predict type signatures of JavaScript functions based on function and parameter names and comments [65]. Similarly, Seshat aims to approximate the relationship between mutants and test cases that kill them based on the similarity between their names in the embedding space.

2.8 Summary of this Chapter

This chapter investigates the mutant's relationship with code by looking at its NL channel. To exploit this, we proposes Seshat, a Predictive Mutation Analysis (PMA) technique that can learn and predict *an entire kill matrix*, as opposed to Predictive Mutation Testing (PMT) whose aim is to predict the mutation score. The source code and test related to the mutant, as well as the mutant-specific code fragments are used to organise input features through Deep Neural Network (DNN). Empirical evaluation on 37 subjects in DEFECTS4J and two widely used Java mutation tools, Major and PIT, demonstrates that Seshat predicts kill matrices with average F-score of 0.83 and is 39 times faster than traditional mutation analysis. In addition to predicting existing tests precisely, Seshat generalises to the new tests, with degradation of the prediction accuracy by 0.05 in F-score. Seshat outperforms PMT and a coverage based baseline model in predicting of the entire kill matrix by 0.14 and 0.45 point in F-score, and performs as well as PMT in predicting of the mutation score only.

Chapter 3. Mutant's Relationship with Faults

This chapter investigates the mutant's relationship with faults that shows how much the mutants can represent the real faults. Specifically, we target Mutation Based Fault Localisation (MBFL), a task that exploits this relationship to localise the faults. We present several statistical inference models named SIMFL, which can learn the relationships between the mutants and faults based on the test executions on them. Using the mutation results of the previous version of the program, SIMFL can predict the faults in the current version without executing the tests. This not only amortises the costs of test execution after the fault is detected but also shows better localisation performance compared to other SBFL and MBFL techniques.

3.1 Introduction

Fault Localisation (FL) is the problem of diagnosing the root cause of the fault by highlighting a few program elements likely to be responsible for the fault. It is an expensive debugging activity as it involves a human inspection to point out the suspicious locations in Program Under Test (PUT) based on the understanding of the root cause. As such, the necessity of developing automated localisation techniques has received increasing attention [22, 66, 67, 68, 69, 70], not only to help human developers but also to be adopted as a prerequisite for Automated Program Repair (APR) [71, 72, 73].

Mutation Based Fault Localisation (MBFL) [10, 23, 74, 4, 75] is one such way that utilises mutation analysis. Mutants are program variants made up of a single or multiple syntactic operator(s), designed to simulate the real faults. Various empirical studies [2, 76, 77] have provided evidence of a correlation between detection of the mutants and detection of the faults. Exploiting this, Metallaxis [74, 4] considers the location of the mutants that have similar failing patterns with the faults as candidate locations of the fault. MUSE [10] and MUSEUM [23] have taken different approaches that reward the mutants causing partial fixes and devalue the mutants making an additional failure.

Although the usefulness of MBFL techniques has long been demonstrated, the substantial cost of mutation analysis has hampered their practical applications. The cost reduction techniques for MBFL are in line with the cost reduction techniques for traditional mutation testing: random sampling of the mutants [4] and higher-order mutants [5]. They have reduced the number of mutants and test executions, but still they are not suitable for large systems like Google's on which more than ten million test executions take place everyday [3]. To avoid this, recent advances employ the context of Continuous Integration (CI) [6]: predictive mutation testing techniques [7, 8, 9] aim to predict mutation testing results without mutant executions by learning static and dynamic features of mutants from earlier version of the project.

In this work, we propose SIMFL (Statistical Inference for Mutation-based Fault Localisation), an MBFL technique that operates in the CI context. Unlike other FL techniques that undertake work on locating faults *a posteriori*, SIMFL uses a kill matrix computed on earlier version of the PUT, which therefore amortises the costs of mutation analysis that has been done after the actual faults are observed. The kill matrix records the test results of each mutant, and SIMFL uses it to relate the locations of mutants and faults: if the mutants were killed by the similar set of tests that detects the faults, SIMFL values such mutants and presumes their locations as likely locations of the faults. To model this kind of predictive inferences, SIMFL makes use of several statistical inference techniques such as Bayesian inference, probabilistic coupling, and more sophisticated models such as Logistic Regression or Multi-Layer Perceptron. Once building a predictive model in advance, SIMFL produces a rank of suspicious program elements by feeding the test results of the fault into the model.

We conduct extensive empirical experiments on SIMFL in three different use cases. Using the real-world fault dataset, DEFECTS4J [78], and Major mutation tool [79], SIMFL places 113 faults on the top out of 224 faults and outperformed existing MBFL techniques. Even if SIMFL is built on the predicted kill matrix by Seshat [9], it localises 95 faults on the top out of 194 faults, showing its practical value. We also investigate how much SIMFL is affected by mutant sampling and the results show that SIMFL retains 80% of localisation accuracy when it uses only 10% of mutants. Lastly, we demonstrate the assumption that subsumed mutants [80, 81] would disrupt the scoring functions of SIMFL. The evaluation of SIMFL models after eliminating the subsumed mutants reveals that the number of faults localised at the top increases up to 51.

The contributions of the chapter are as follows:

- We present SIMFL, an MBFL technique that infers the location of faults based on the mutation analysis performed before the faults are observed. This allows SIMFL to significantly amortise the costs of mutation analysis, and the experiments on DEFECTS4J dataset showed that SIMFL achieves better localisation accuracy than existing MBFL techniques.

- We introduce several modelling schemes of SIMFL using statistical inferences and machine learning techniques. We not only compare their effectiveness but also provide the empirical evidence of the impacts of model viability and mutant sampling.

- We further investigate the impact of test granularity and mutant filtering based on the kill reason (e.g., exception). The in-depth observation on the relationship with other FL techniques open up a positive hybridisation of SIMFL and other techniques.

- We have added a new use scenario of SIMFL using a predicted kill matrix of Seshat, which is not dependent on pre-existence of failing tests.

- We have proposed a new ranking model based on probabilistic coupling. We apply a definition of mutant-fault coupling that values the mutants tightly coupled to the fault.

- We have investigated the effects of removing the subsumed mutants. We observed that the subsumed mutants disturb and inflate the method-level aggregation of FL scores by SIMFL. The new experiment showed that eliminating them significantly improves the effectiveness of SIMFL.

3.2 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 3.1 depicts the expected use case scenario of SIMFL, which includes four stages:

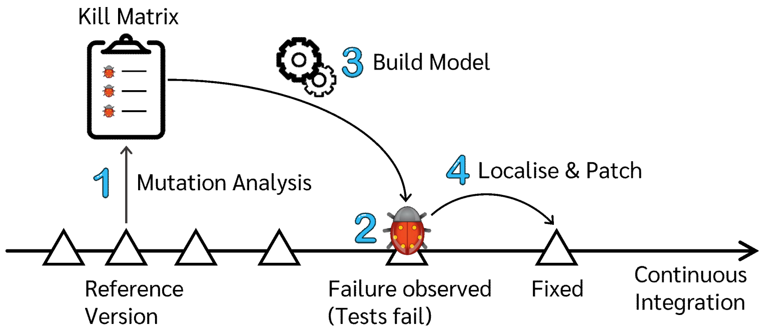


Figure 3.1: Expected use case scenario of SIMFL

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.

Table 3.1: An Example Kill Matrix

Class	Method	Mutants	Test Results t_1	t_2	t_3	t_4	0-1 Vectors of t_1, t_2, t_3, t_4
com. acme. Foo	getType	n.isName() $\mapsto$ true (m_1)	✗	✗	✓	✓	$(1, 1, 0, 0)$
		n.isName() $\mapsto$ false (m_2)	✓	✓	✗	✓	$(0, 0, 1, 0)$
		bFlag or isInferred $\mapsto$ isInferred (m_3)	✓	✗	✗	✓	$(0, 1, 1, 0)$
	resolveType	param.isTemplateType() $\mapsto$ true (m_5)	✓	✗	✓	✓	$(0, 1, 0, 0)$
		resolvedType() $\mapsto$ deleted (m_6)	✗	✗	✗	✓	$(1, 1, 1, 0)$

3.2.1 Mutation Analysis

We perform mutation analysis on the reference version of a program $\mathbf{P}$ with a test suite $\mathbf{T}$, and compute a kill matrix $\mathbf{K}$, which contains a complete report of all tests executed on all mutants. An example kill matrix is shown in Table 3.1: the mutant m_1 located in the method `getType` is killed by the test cases t_1 and t_2, whereas m_2 is only killed by t_3.

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 $\mathbf{K}_m$, we can approximate the probability of test case t killing the mutants located on the program element e as follows:

$$Pr\left(F_t \mid M_e\right) \simeq \frac{|\{m \in \mathbf{X}_e \mid t \in \mathbf{K}_m\}|}{|\mathbf{X}_e|} \tag{3.1}$$

Note that this is strictly an *approximation* based on the observed kill matrix because it is impossible to produce and evaluate all possible mutants. The exact value of $Pr(M_e)$ is the ratio of the number of all possible mutants on e to the number of all possible mutants on $\mathbf{P}$; for $Pr(F_t \mid M_e)$, we need to calculate the number of all possible mutants in e that are killed by t. Neither is feasible. Consequently, we assume that we can analyse a *finite* set of mutants that allow us to approximate Equation 3.1.

Next, using Bayes' rule, we calculate the revised probability of the event that the program element e has been mutated, given that the test case t fails:

$$\begin{aligned} Pr(M_e \mid F_t) &= \frac{Pr(F_t \mid M_e)\, Pr(M_e)}{Pr(F_t)} \\ &\simeq Pr(\text{fault exists in } e \mid F_t) \end{aligned} \tag{3.2}$$

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.

3.2.2 Ranking Models Based on Bayes' Rule

We regard the probability in Equation 3.2 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.

Exact Matching (EM)

This model is an extension of Equation 3.2 to a set of test cases. Let $\mathbf{T} = \{t_i \mid 1 \leq i \leq n \leq n'\}$ 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, $Pr\left(M_{e_i} \mid F_{\mathbf{T}_f}\right)$. It follows that:

$$\underset{i}{\operatorname{argmax}}\, Pr\left(M_e \mid F_{\mathbf{T}_f}\right) = \underset{i}{\operatorname{argmax}} \frac{Pr\left(F_{\mathbf{T}_f} \mid M_e\right) Pr(M_e)}{Pr\left(F_{\mathbf{T}_f}\right)} \tag{3.3}$$

The denominator in Equation 3.3, $Pr\left(F_{\mathbf{T}_f}\right)$, 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:

$$\begin{aligned} &\underset{i}{\operatorname{argmax}}\, Pr\left(F_{\mathbf{T}_f} \mid M_{e_i}\right) Pr(M_{e_i}) \\ &= \underset{i}{\operatorname{argmax}}\, Pr(F_{t_1} \cap \cdots \cap F_{t_n} \mid M_{e_i})\, Pr(M_{e_i}) \\ &= \underset{i}{\operatorname{argmax}} \frac{|\{m \in \mathbf{X}_{e_i} \mid \{t_1, \ldots, t_n\} = \mathbf{K}_m\}|}{|\mathbf{X}_{e_i}|} Pr(M_{e_i}) \\ &= \underset{i}{\operatorname{argmax}} \frac{|\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}|}{|\mathbf{X}_{e_i}|} \frac{|\mathbf{X}_{e_i}|}{|\mathbf{X}_{\mathbf{P}}|} \\ &= \underset{i}{\operatorname{argmax}} \frac{|\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}|}{|\mathbf{X}_{\mathbf{P}}|} \\ &= \underset{i}{\operatorname{argmax}}\, |\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m\}| \end{aligned} \tag{3.4}$$

Intuitively, Equation 3.4 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 3.4 changes as follows:

$$\begin{aligned}
&\operatorname*{argmax}_i Pr\left(F_{\mathbf{T}_f} \cap P_{\mathbf{T}_p} \mid M_{e_i}\right) Pr\left(M_{e_i}\right) \\
&= \operatorname*{argmax}_i Pr\left(F_{t_1} \cap \cdots \cap F_{t_n} \cap P_{t_{n+1}} \cap \cdots \cap P_{t_{n'}} \mid M_{e_i}\right) Pr\left(M_{e_i}\right) \\
&= \operatorname*{argmax}_i \left|\{m \in \mathbf{X}_{e_i} \mid \mathbf{T}_f = \mathbf{K}_m \wedge \mathbf{T}_p = \mathbf{T} \setminus \mathbf{K}_m\}\right|
\end{aligned} \tag{3.5}$$

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.

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

$$\begin{aligned}
&\operatorname*{argmax}_i \prod_{t \in \mathbf{T}_f} \left(Pr\left(M_{e_i} \mid F_t\right) + \epsilon\right) \\
&= \operatorname*{argmax}_i \prod_{t \in \mathbf{T}_f} \left(\left|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{K}_m\}\right| + \epsilon\right)
\end{aligned} \tag{3.6}$$

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

$$\begin{aligned}
&\operatorname*{argmax}_i \sum_{t \in \mathbf{T}_f} Pr\left(M_{e_i} \mid F_t\right) \\
&= \operatorname*{argmax}_i \sum_{t \in \mathbf{T}_f} \left|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{K}_m\}\right|
\end{aligned} \tag{3.7}$$

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: the value of ϵ does not affect the ranking.

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

$$\begin{aligned}
&\operatorname*{argmax}_i \left(\prod_{t \in \mathbf{T}} \left(Pr\left(M_{e_i} \mid F_t\right) + \epsilon\right) \prod_{t \in \mathbf{T}_p} \left(Pr\left(M_{e_i} \mid P_t\right) + \epsilon\right)\right) \\
&= \operatorname*{argmax}_i \prod_{t \in \mathbf{T}} \left(\left|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{T}_f \iff t \in \mathbf{K}_m\}\right| + \epsilon\right)
\end{aligned} \tag{3.8}$$

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

$$\begin{aligned} &\underset{i}{\operatorname{argmax}} \left(\sum_{t \in \mathbf{T}} \left(Pr\left(M_{e_i} \mid F_t\right)\right) + \sum_{t \in \mathbf{T}_p} \left(Pr\left(M_{e_i} \mid P_t\right)\right) \right) \\ &= \underset{i}{\operatorname{argmax}} \sum_{t \in \mathbf{T}} \left|\{m \in \mathbf{X}_{e_i} \mid t \in \mathbf{T}_f \iff t \in \mathbf{K}_m\}\right| \end{aligned} \tag{3.9}$$

3.2.3 Ranking Models Using Classifiers

Scores from the Bayesian inference models described in Sections 3.2.2 and 3.2.2 are 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. 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 $\alpha_{\mathbf{T_i}}$ denote a 0-1 vector of the test results of $\mathbf{T_i}$, where 0 indicates that test case fails, and 1 indicates that test case passes. We first build a training set using the kill matrix $\mathbf{K}$: test results per mutant $\mathbf{T_i}$ are transformed into $\alpha_{\mathbf{T_i}}$, and the class is labelled based on the method where the mutant is located.

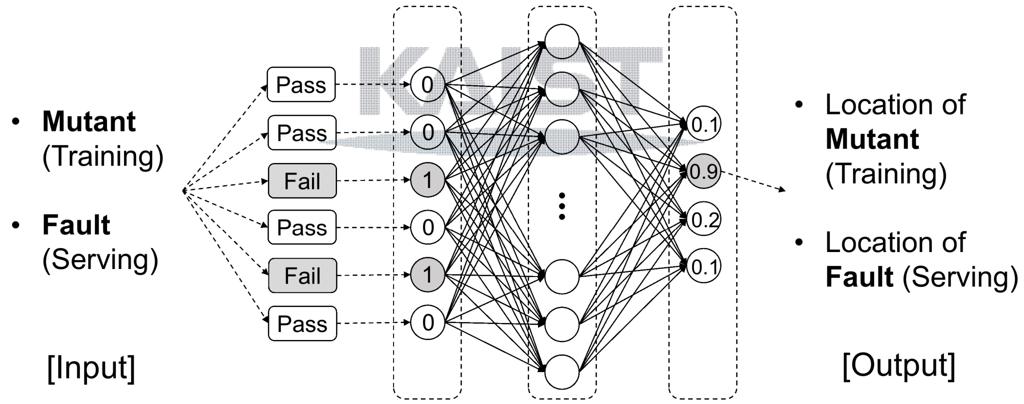


Figure 3.2: Illustration of MLP training and serving

We train representative linear and non-linear classifiers using Logistic Regression (LR) and Multi-Layer Perceptron (MLP) [82, 83]. For our study, we use a vanilla MLP that consists of one input layer, one hidden layer with 50 neurons, and one output layer, as shown in Figure 3.2. In the serving phase, we use the suspiciousness score of each program element, which is obtained before the model computes the most suspicious method. 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 Sections 3.2.2 and 3.2.2, training these classifiers requires additional analysis cost to SIMFL, although the training cost of these models is much lower than the cost of mutation analysis.

Ranking Model Based on Probabilistic Coupling

All SIMFL models are intuitive based on the concept of coupling effect. We now exploit the strict definition of the mutant-fault coupling to build a new model based on probabilistic coupling [77], which aims to quantify the degree of coupling between mutants and faults. A mutant is said to be *coupled* to a fault if the tests that kill the mutant can also kill (i.e., detect) the fault. That is, if the mutant *dynamically subsumes* the fault, we call the mutant is perfectly coupled to the fault [17]. Adopting this subsumption relationship for FL, we focus on the mutants coupled with the faults: the stronger mutants are coupled to a fault, the more likely they are to be closer to the fault.

Probabilistic coupling extends the notion of perfect coupling by considering a probabilistic detection of faults in order to precisely estimate the fault-revealing capability of mutants [77]. Even if the mutant does not subsume the fault, it can be *probabilistically* coupled to the faults if at least *some* tests that kill the mutant are also able to detect the faults. In the following, we illustrate the example cases of the perfect coupling and probabilistic coupling, as well as how we model them as scores to rank the likely fault locations.

1. Perfect coupling: Let us assume that t_2 and t_3 in Table 3.1 are fault revealing tests (i.e., failing tests). From the view of the subsumption, m_2 is perfectly coupled to the fault because it is detected by t_3 but the fault is detected by both t_2 and t_3.

2. Probabilistic coupling: On the other hand, mutant m_1 does not subsume the fault (i.e., not perfectly coupled). However, as it is detected by t_2, there is a possibility that m_1 is coupled to the fault if m_1 is detected by t_2 and also the fault if detected by t_2.

Based on these two notions, we define a new model called PC that rewards mutants coupled to the faults. The score of each mutant is computed as follows:

$$pc(m) = \begin{cases} 1, & \text{if } \mathbf{K}_m \subseteq \mathbf{T}_f \\ \frac{|\mathbf{K}_m \cap \mathbf{T}_f|}{|\mathbf{K}_m|}, & \text{if } \mathbf{K}_m \nsubseteq \mathbf{T}_f \text{ and } |\mathbf{K}_m \cap \mathbf{T}_f| > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.10}$$

The full score (i.e., 1) is given to the perfectly coupled mutant, the partial score is given to the probabilistically coupled mutant, and no score is given to the decoupled mutant (i.e., 0). We aggregate the mutants in the same method by summing up their scores to compute a method-level FL score. Note that the notion of probabilistic coupling cannot be defined if we only consider the failing tests. Therefore we only present PC(F+P) model as follows:

$$\text{PC(F+P)} = \underset{i}{\text{argmax}} \sum_{m \in \mathbf{X}_{e_i}} pc(m) \tag{3.11}$$

3.3 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.

3.3.1 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 [84]. 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 [78]. This presents a challenge for the realistic evaluation of SIMFL in the context it was designed for. To address this issue, we introduce three experimental protocols, Faulty Commit Emulation (FCE), Test Existence Emulation (TEE), and Kill Matrix Prediction (KMP).

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.

We argue that this is more realistic than injecting mutation faults artificially to evaluate SIMFL. Since mutants are exactly what SIMFL uses to build its models, SIMFL may unfairly benefit if evaluated using mutants as faults. Instead, we emulate faulty commits using faults that some developers actually had introduced in real-world software. Existing work on test data generation has also used the fixed version as the reference version, against which a test generation tool is applied. The reversed fix patch is then used to emulate regression faults for the evaluation of the generated tests [85, 76]. Our approach with FCE is similar in the sense that we analyse the fixed version first, then use the outcome to localise the emulated regression fault.

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 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. Compared to FCE, TEE follows the ground truth code changes, and only assumes the earlier existence of fault revealing test cases. We

[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 and thus that the proposed approach can be useful.

use TEE to complement the FCE scenario. Specifically, TEE can evaluate whether training SIMFL models with kill matrices of earlier versions degrades its localisation accuracy.

Kill Matrix Prediction (KMP)

Lastly, we present a new scenario that is free from the pre-existence of failing tests. The basic idea is to utilise a predicted kill matrix inferred by Seshat [9], a predictive mutation analysis technique producing an entire kill matrix. Having learnt from the kill matrix computed from a past version, Seshat can predict the kill matrix of the current version based on the syntactic and semantic features in source code, test code, and mutant. Most importantly, Seshat can predict the kill results of test cases that are introduced after the reference version from whose kill matrix Seshat learnt from. That is, this scenario using Seshat does not care about the pre-existence of failing tests. For example, to locate the faults of Lang programs, we first take the oldest version of them as the reference version, i.e., V_{fix} of `Lang 65`[2]. Then using Seshat, we predict the kill matrix of the faulty version which is more recent version than `Lang 65`, e.g., V_{bug} of `Lang 30`. Next, we train the SIMFL models based on the predicted kill matrix of V_{bug} of `Lang 30`, and produce a ranked list of likely faulty locations. Note that we do use V_{fix} of `Lang 65` to predict the faults of V_{bug} of `Lang 65`, which is what FCE does.

Building Kill Matricies for FCE and TEE

Building a full kill matrix requires huge computational cost: mutation analysis on all versions of Closure using Major exceeded our 24 hours timeout, and other subject programs also required significant amounts of analysis time. To address this practical concern, for empirical evaluation, we have constructed the kill matrix using only the *relevant test cases* as defined by DEFECTS4J[3], 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 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.

Building Kill Matricies for KMP

Inputs for training Seshat include the full kill matrix and the tokens of source and test code, meaning that Seshat has the same problem raised in Section 3.3.1 that requires substantial costs for computing full kill matrix. Thus we exclude all versions of Closure that did not halt within a given timeout. Also, we exclude all versions of Time to obtain the full kill matrix using *all tests*. Compared to FCE that uses *relevant test cases* by DEFECTS4J, we believe that it is important for Seshat to be trained on a diverse and large set of tests to retain its predictive power on unseen source code and tests.

[2] We denote version numbers of `Lang` programs by the identifier numbers used in DEFECTS4J.

[3] See https://github.com/rjust/defects4j/tree/v1.3.1#export-version-specific-properties

We select the reference model as the oldest fixed version of each project, e.g., V_{fix} of `Lang 65` or `Math 106`. They are used to train Seshat and later used to predict the full kill matrix of target buggy version.

Using Test Runtime Information

The use case of SIMFL assumes that, while the actual mutation analysis can be performed in advance, the inference models are trained after the observation of a failure (see 3.1). In practice, the observation of the behaviour of the failing test cases can provide information that is beyond the mutation analysis. Consequently, we exploit this additional information by collecting coverage reports of failing test cases using `Cobertura`. We then exclude any methods and mutants that are not covered by the failing test cases from model training and the final ranking.

Table 3.2: Subject Programs in DEFECTS4J

Subject	# Faults	kLoC	# Methods	# Mutants	# Test cases
Commons-lang (Lang)	65	50	1,527	21,178	2,245
JFreeChart (Chart)	26	132	4,903	75,985	2,205
Joda-Time (Time)	27	105	1,946	21,689	4,130
Closure compiler (Closure)	133	216	5,038	58,515	7,927
Commons-math (Math)	106	104	2,713	79,428	3,602
Total	357	607	16,126	256,792	20,109

3.3.2 Subject Programs

In our study, we use 357 versions of five different programs from the DEFECTS4J version 1.3.1. They provide reproducible and isolated faults of real-world programs. Table 3.2 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 containers.

3.3.3 Research Questions

RQ1. Localisation Effectiveness on FCE: *Does the models of SIMFL produce accurate fault localisation compared to the state-of-the-art FL techniques?* RQ1 is answered by computing the standard evaluation metrics on the eight models of SIMFL under the FCE scenario outlined in Section 3.3.1. We compare SIMFL with two MBFL techniques (MUSE and Metallaxis), two SBFL techniques (Ochiai and DStar), and two learning-to-rank based FL techniques (TraPT and FLUCCS).

RQ2. Model Viability on TEE: *How well does SIMFL hold up when applied using models built earlier?* RQ2 is answered by computing the standard evaluation metrics using prior models built under the TEE scenario outlined in Section 3.3.1.

RQ3. Localisation Effectiveness on KMP: *How well does SIMFL localise faults when it uses the predicted kill matrix inferred by Seshat?* To answer RQ3, we conduct the same experiment with RQ1, but with the kill matrices predicted by Seshat. We report standard evaluation metrics and investigate the effects of using predicted kill matrices.

RQ4. Effects of Sampling: *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 the equal number of mutants from each method as possible.

RQ5. Effects of Subsumed Mutants: *What is the impact of subsumed mutants to SIMFL models?* As we suspect that the subsumed mutants would inflate the method-level aggregation of SIMFL scores, we study the impact of removing the subsumed mutants by comparing $acc@n$ scores with RQ1 results.

3.3.4 Evaluation Metrics and Tie Breaking

We use three standard evaluation metrics:

- $acc@n$: 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.

- 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.

3.3.5 Mutation Tool and Operators

In the study, we use Major version 1.3.4 [79] as our mutation analysis tool, and choose all mutation operators in Major. Note that some operators had to be turned off for specific classes so that Major does not generate an exceptionally large number of mutants.[4]

3.4 Results

3.4.1 Localisation Effectiveness on FCE (RQ1)

We start by comparing different SIMFL models. Subsequently, using the best SIMFL model, we compare SIMFL to the state-of-the-art fault localisation techniques.

[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.

Table 3.3: Effectiveness of SIMFL models on FCE scenario.

Model	Project	Total Studied	*acc* @1	@3	@5	@10	*wef* med	MAP	Model	Project	Total Studied	*acc* @1	@3	@5	@10	*wef* med	MAP
EM (F)	Lang	65	35	45	47	48	**0.0**	0.6176	EM (F+P)	Lang	65	36	41	43	44	**0.0**	0.5922
	Chart	26	**6**	**11**	**13**	15	5.0	0.3294		Chart	26	6	9	10	11	27.0	0.2917
	Time	27	**4**	9	9	13	8.5	0.2451		Time	27	10	13	14	15	3.0	0.3819
	Closure	133	10	31	41	57	17.0	0.1753		Closure	-	-	-	-	-	-	-
	Math	106	22	43	53	71	4.0	0.3404		Math	106	32	45	47	49	3.0	0.4098
	Total	357	77	139	163	204				Total	224	84	108	114	119		
PM* (F)	Lang	65	38	47	51	53	**0.0**	0.6732	PM* (F+P)	Lang	65	27	36	37	42	1.0	0.5264
	Chart	26	**6**	**11**	**13**	16	5.0	0.3562		Chart	26	7	9	12	14	6.0	0.3598
	Time	27	**4**	**10**	10	13	8.0	0.2549		Time	27	1	3	4	12	16.0	0.1172
	Closure	133	11	36	50	66	9.5	0.1982		Closure	-	-	-	-	-	-	-
	Math	106	23	**47**	59	77	3.5	0.3753		Math	106	14	26	33	42	12.0	0.2460
	Total	357	82	151	183	225				Total	224	49	74	86	110		
PM+ (F)	Lang	65	40	48	52	53	**0.0**	0.6977	PM+ (F+P)	Lang	65	19	31	33	37	2.0	0.4291
	Chart	26	**6**	10	**13**	**19**	**4.0**	**0.3697**		Chart	26	5	9	12	13	8.0	0.2712
	Time	27	**4**	**10**	10	13	8.0	0.2564		Time	27	0	2	3	5	40.5	0.0616
	Closure	133	**12**	**41**	**52**	65	11.0	0.2005		Closure	-	-	-	-	-	-	-
	Math	106	24	46	59	77	4.0	0.3845		Math	106	9	15	20	29	23.0	0.1574
	Total	357	86	**155**	**186**	**227**				Total	224	33	57	68	84		
LR (F)	Lang	65	**41**	49	**53**	**55**	**0.0**	**0.7179**	LR (F+P)	Lang	65	40	49	51	53	**0.0**	0.7017
	Chart	26	5	9	12	14	6.0	0.3175		Chart	26	8	14	14	16	**2.0**	0.4194
	Time	27	**4**	**10**	**12**	**14**	5.5	0.2668		Time	27	8	14	17	19	2.0	0.4094
	Closure	133	**12**	37	50	**68**	**9.0**	**0.2074**		Closure	-	-	-	-	-	-	-
	Math	106	**28**	**47**	59	75	**3.0**	**0.3976**		Math	106	32	43	47	51	3.0	0.4066
	Total	357	**90**	152	**186**	226				Total	224	88	120	129	139		
MLP (F)	Lang	65	39	**51**	**53**	**55**	**0.0**	0.7052	MLP (F+P)	Lang	65	**48**	**55**	**56**	**56**	**0.0**	**0.7882**
	Chart	26	5	10	12	15	6.0	0.3319		Chart	26	9	13	15	19	2.0	0.4477
	Time	27	**4**	**10**	**12**	**14**	**5.0**	**0.2710**		Time	27	**11**	**16**	**18**	**24**	**1.0**	**0.4847**
	Closure	133	11	33	41	60	12.0	0.1888		Closure	-	-	-	-	-	-	-
	Math	106	26	46	**62**	**79**	**3.0**	0.3941		Math	106	**45**	**61**	70	**82**	**1.0**	**0.5194**
	Total	357	85	150	180	223				Total	224	**113**	**145**	**159**	**181**		
									PC (F+P)	Lang	65	42	51	51	54	**0.0**	0.7323
										Chart	26	**10**	**16**	**17**	**22**	**1.0**	**0.5389**
										Time	27	7	15	**18**	21	2.0	0.4133
										Closure	-	-	-	-	-	-	-
										Math	106	37	**61**	**71**	81	1.5	0.4934
										Total	224	96	143	157	178		

Comparison Between SIMFL Models

Table 3.3 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 3.3.2 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 48 out of 61 faults at the first place for Lang, and 45 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 the additional information: MLP(F+P) places 28 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. In such a case, the EM(F) model will lose the information, while the 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.

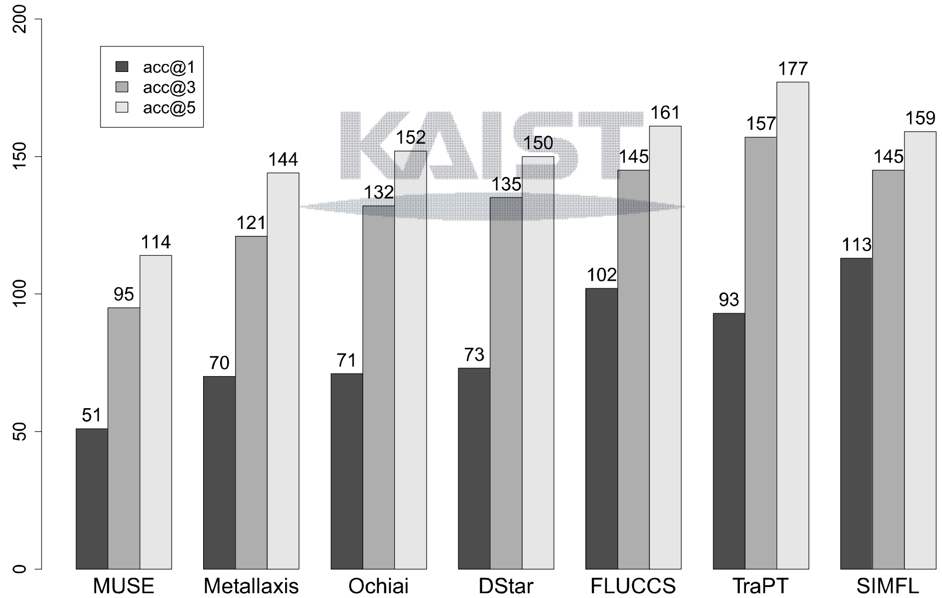


Figure 3.3: Comparison to other FL techniques: *acc@n* metric values without counts of Closure

Comparison to Other FL Techniques

To gain some insights into the trade-off between amortised modelling efforts and localisation accuracy, we compare the method-level fault localisation results of the state-of-the-art MBFL and SBFL techniques, the result of which is shown in 3.3. We obtained the performance of the each model (i.e., *acc@n*) on Defects4J from the literatures, and artefact of Zou et al. [86]. Based on the results of the comparison between SIMFL models, we choose MLP(F+P) to represent SIMFL. However, since Closure has been excluded from the evaluation of F+P models, we have excluded Closure from the results of other techniques for a fair comparison.

3.3 shows that MLP(F+P) is better than other techniques in terms of *acc*@1, but TraPT performs better in terms of *acc*@3 and *acc*@5. Although SIMFL does not make use of learning-to-rank technique to boost performance by fully including runtime information or suspiciousness scores of other FL techniques, SIMFL localises faults at the top better than others, and shows comparable results to the learning-to-rank techniques: FLUCCS and TraPT.

> **Answer to RQ1**: SIMFL can localise faults accurately compared to the existing techniques: SIMFL places up to 25.86% (90 of 348 for LR(F)) of studied faults at the top using F models, and 55.67% (113 of 203 for MLP(F+P)) of studied faults at the top using F+P models.

Table 3.4: Viability of F models using TEE scenario. The ranks that do not have the same ranks with FCE are typeset in bold.

	Fault	Commit (Δrev.)	Rank EM	PM*	PM+	LR	MLP
Closure	21	FCE Rank	2	2	2	2	2
		32a12ba (2)	2	2	2	2	2
		43a5523 (3)	2	2	2	2	2
	61	FCE Rank	7	4	5	5	6
		f5529dd (3)	7	4	5	5	6
		b12d1d6 (4)	7	4	5	5	6
		245362a (7)	7	4	5	5	**7**
		8abd1d9 (8)	7	4	5	5	**8**
		37b0e1b (9)	7	4	5	5	6
	62	FCE Rank	1	1	1	1	1
		245362a (2)	1	1	1	1	1
		8abd1d9 (3)	1	1	1	1	1
		37b0e1b (4)	1	1	1	1	1
	115	FCE Rank	14	22	24	19	11
		b9262dc (5)	14	**19**	**22**	**18**	**13**
		911b2d6 (6)	14	22	24	19	**12**
	120	FCE Rank	7	7	7	6	6
		2aee36e (3)	**24**	**24**	**24**	**15**	**16**
Math	46	FCE Rank	188	188	188	47	85
		bbb5e1e (1)	188	188	188	**35**	**47**
		37680e2 (2)	188	188	188	**35**	**47**
		1861674 (3)	188	188	188	**35**	**27**
		f0b12de (4)	188	**1**	**1**	**3**	**1**
		8581b76 (5)	188	188	188	**35**	**41**
	89	FCE Rank	13	13	13	8	7
		43336b0 (1)	**12**	**12**	**12**	**2**	**3**
		cdd62a0 (2)	**14**	**14**	**14**	**2**	**5**
		90439e5 (3)	13	13	13	8	**11**
		36a8485 (4)	13	13	13	8	**13**
		dbe7842 (5)	13	13	13	8	7
		d84a587 (6)	13	13	13	8	**12**
		d27e072 (7)	13	13	13	8	**10**
		3590bdc (8)	13	13	13	8	**8**
		6b108c0 (9)	13	13	13	8	**13**
		9c55428 (10)	13	13	13	8	**12**

3.4.2 Model Viability on TEE (RQ2)

Following the TEE scenario described in Section 3.3.1, 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 trained five F models on each of the 28 reference versions to localise the fault in the faulty version, resulting in 140 rankings based on TEE scenario. Note that we did not consider F+P models on these reference versions because they require more than 24 hours for mutation analysis, as described in Section 3.3.2.

Table 3.4 shows the rank of the faulty method for each F model built on each preceding reference version. Out of 140 TEE based rankings produced by F models, 103 are identical to the corresponding FCE ranking. One notable exception is Math 46 (f0b12de) that shows a significant improvement over the FCE scenario rank. We have manually 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 has been removed for the original program. We study the impact of different kill reasons in Section 3.5.3, and furthermore discuss this as one of the threats to internal validity in 3.6.

Answer to RQ2: Performances of SIMFL using models built with preceding reference versions tend to be stable when compared to the FCE results: only 19 out of 140 cases show degraded performance since we used less recent mutation analysis results.

Table 3.5: Effectiveness of SIMFL models on KMP scenario, using all tests.

Model	Total	*acc*@1		*acc*@3		*acc*@5		*acc*@10	
EM(F)	194	57	−20	95	−44	108	−55	122	−82
PM*(F)	194	61	−21	101	−50	114	−69	131	−94
PM$^+$(F)	194	60	−26	104	−51	119	−67	134	−93
LR(F)	194	63	−27	104	−48	116	−70	134	−92
MLP(F)	194	66	−19	106	−44	120	−60	140	−83
EM(F+P)	194	35	−49	57	−51	65	−49	82	−37
PM*(F+P)	194	53	+4	76	+2	90	+4	107	−3
PM$^+$(F+P)	194	66	+33	96	+39	119	+51	138	+54
LR(F+P)	194	96	+8	122	+2	133	+4	145	+6
MLP(F+P)	194	95	−18	125	−20	132	−27	150	−31
PC(F+P)	194	93	−3	132	−11	137	−20	148	−30

Table 3.6: Effectiveness of SIMFL models on KMP scenario, using relevant tests.

Model	Total	*acc*@1		*acc*@3		*acc*@5		*acc*@10	
EM(F+P)	194	39	−45	64	−44	77	−37	94	−25
PM*(F+P)	194	61	+12	91	+17	108	+22	131	+21
PM$^+$(F+P)	194	66	+33	99	+42	114	+46	139	+55
LR(F+P)	194	63	−25	94	−26	112	−17	128	−11
MLP(F+P)	194	81	−32	109	−36	121	−38	143	−38
PC(F+P)	194	71	−25	109	−34	121	−36	137	−41

3.4.3 Localisation Effectiveness on KMP (RQ3)

We compute the scores of SIMFL models by following the KMP scenario that uses a predictive kill matrix by Seshat. Table 3.5 shows the aggregated numbers of each *acc@n* metric as well as the difference with the results of the FCE scenario shown in Table 3.3. If the KMP scenario was better than the FCE scenario, we mark the differences with a plus sign, and if not, we mark them with a minus sign.

The most noticeable results in Table 3.5 is that, in case of some models, SIMFL with predicted kill matrices outperforms SIMFL with actual kill matrices. This seemingly counter-intuitive result is due to the way inaccurate predictions of Seshat affects SIMFL. Suppose Seshat predicts the kill matrix with the bias of more kills: this may result in an increased number of killed mutants in the faulty method, increasing its ranking score in turn. On the other hand, if Seshat predicts the kill matrix with the bias of fewer kills, it will result in the decreased number of killed mutants in the non-faulty methods, having a similar effect as that of passing test reducing the suspiciousness of the parts of the program they cover. Notably, this effect occurs strongly in partial models. In contrast, the performance of EM model degrades significantly. This is because, if SIMFL is to perform well, Seshat has to make a perfectly accurate prediction for the mutants in both the faulty method (to match F) and the non-faulty methods (to match P). Interestingly, the performance improvement from Seshat only applies to F+P models, suggesting that the support from P matches has a significant effect on the performance of SIMFL, as confirmed in RQ1.

Note that, in RQ1, SIMFL uses only relevant tests as defined by DEFECTS4J. However, the results in Table 3.5 is produced by predicting the entire kill matrices including all tests, meaning that SIMFL with Seshat has more information to localise faults. To investigate the impact of this difference, we also conducted the experiment for RQ3 by only predicting the kill matrix for relevant test cases using Seshat, the results of which are listed in Table 3.6. We can only use F+P models for this experiment, as the set of relevant tests includes both failing and passing tests. The *acc@n* metrics produced with relevant tests are slightly degraded, but the overall trend remains the same: the performance of partial matching models either remains the same or slightly improved, but the performance of others degraded. Interestingly, the performance of EM(F+P) model is better when using relevant tests: since there are fewer tests to achieve exact match, it becomes easier to score high with EM(F+P) model.

Answer to RQ3: Kill matrix prediction can be successfully used with SIMFL to handle test cases that are introduced after the reference version.

3.4.4 Effects of Sampling (RQ4)

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. Table 3.7 (left side) shows the uniform sampling results with rates of 0.1, 0.3, 0.5, and 0.7: all metric values are averaged across 20 different samples. Table 3.7 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 show comparable results with the Full configuration. However, the performance does not degrade at the same rate as the sampling

Table 3.7: Uniform and stratified random sampling

Ratio	Model	Total Studied	*acc* @1	@3	@5	@10	N (Ratio)	*acc* @1	@3	@5	@10
0.1	EM(F)	357	59.80	95.35	111.05	136.70	5 (0.27)	36.50	65.05	83.70	116.30
	PM*(F)	357	66.55	107.85	126.55	159.35		40.95	74.75	94.95	133.20
	PM+(F)	357	68.40	108.50	128.50	162.25		40.05	75.00	97.25	134.20
	LR(F)	357	71.75	121.50	146.85	180.20		49.15	91.10	116.80	156.55
	MLP(F)	357	76.70	120.05	142.40	177.75		49.90	88.85	115.40	151.05
	EM(F+P)	224	46.35	56.55	62.10	70.75		39.50	57.05	64.20	69.10
	PM*(F+P)	224	45.45	66.65	78.90	95.30		65.70	100.25	111.70	127.95
	PM+(F+P)	224	29.35	52.45	64.10	78.80		39.55	52.45	62.70	81.95
	LR(F+P)	224	70.70	93.30	104.30	117.10		75.95	114.40	128.30	146.05
	MLP(F+P)	224	83.60	111.30	122.10	139.60		78.15	118.20	134.50	152.00
	PC(F+P)	224	82.40	112.30	123.50	139.60		70.20	113.25	129.65	147.10
0.3	EM(F)	357	72.25	118.55	137.40	172.75	10 (0.41)	45.60	80.95	106.15	145.50
	PM*(F)	357	78.90	132.15	156.25	196.20		49.65	93.50	122.15	161.40
	PM+(F)	357	83.45	133.70	159.40	200.40		53.65	93.00	120.20	162.60
	LR(F)	357	84.85	142.15	166.65	204.90		59.95	106.60	136.50	179.05
	MLP(F)	357	82.45	139.40	164.10	202.50		56.90	102.60	132.55	174.10
	EM(F+P)	224	66.70	82.35	89.25	95.40		50.60	74.00	80.85	85.50
	PM*(F+P)	224	47.80	71.40	83.35	104.75		74.45	106.35	120.15	140.65
	PM+(F+P)	224	31.75	55.45	67.70	82.75		39.20	55.05	67.20	81.65
	LR(F+P)	224	81.95	107.60	116.40	131.50		82.15	115.90	128.80	146.70
	MLP(F+P)	224	101.55	132.20	143.40	159.85		89.70	126.35	141.60	161.45
	PC(F+P)	224	95.20	133.30	144.95	162.90		84.15	127.90	144.40	163.25
0.5	EM(F)	357	75.90	128.30	149.35	188.55	15 (0.50)	53.60	96.70	122.20	160.65
	PM*(F)	357	82.90	141.80	168.85	211.15		58.70	110.10	139.65	177.95
	PM+(F)	357	86.30	143.80	173.60	215.20		62.20	109.15	142.75	182.70
	LR(F)	357	88.75	145.90	175.75	216.65		66.70	118.30	152.35	190.90
	MLP(F)	357	84.70	146.00	170.95	212.80		63.05	114.25	146.00	187.10
	EM(F+P)	224	73.90	93.10	100.05	106.50		57.35	82.65	89.45	94.75
	PM*(F+P)	224	47.55	72.70	84.20	108.00		78.75	109.90	123.30	141.50
	PM+(F+P)	224	32.70	57.00	**68.65**	83.30		**42.15**	61.45	**71.60**	82.15
	LR(F+P)	224	86.35	112.15	121.75	135.20		86.95	116.95	129.70	148.35
	MLP(F+P)	224	104.90	138.65	151.35	166.70		94.80	134.60	147.95	163.60
	PC(F+P)	224	98.10	138.85	151.80	170.05		91.85	133.85	151.30	169.95
0.7	EM(F)	357	**78.05**	133.55	156.30	194.60	20 (0.56)	55.80	105.80	133.25	172.90
	PM*(F)	357	**84.80**	147.75	175.60	216.90		64.95	121.45	153.75	193.75
	PM+(F)	357	**88.15**	150.05	177.85	219.30		70.05	124.10	156.30	195.10
	LR(F)	357	89.65	148.50	179.85	221.25		74.35	127.05	162.95	200.15
	MLP(F)	357	84.30	144.60	173.40	214.85		69.70	124.70	158.15	196.60
	EM(F+P)	224	78.05	98.85	106.05	112.15		62.50	88.05	94.05	100.55
	PM*(F+P)	224	48.30	73.60	83.95	108.35		**81.55**	**110.15**	**125.50**	**142.60**
	PM+(F+P)	224	32.90	**57.20**	68.10	83.40		40.55	**62.40**	69.05	81.65
	LR(F+P)	224	87.20	115.20	125.15	137.85		**90.70**	118.60	**131.60**	**148.90**
	MLP(F+P)	224	108.45	142.55	155.30	170.20		98.25	138.20	150.00	165.90
	PC(F+P)	224	**99.75**	142.30	155.30	174.35		**97.25**	137.75	152.75	172.95
Full	EM(F)	357	77.00	**139.00**	**164.00**	**204.00**	Full	**77.00**	**139.00**	**164.00**	**204.00**
	PM*(F)	357	82.00	**151.00**	**183.00**	**224.00**		**82.00**	**151.00**	**183.00**	**224.00**
	PM+(F)	357	86.00	**155.00**	**186.00**	**227.00**		**86.00**	**155.00**	**186.00**	**227.00**
	LR(F)	357	**90.00**	**152.00**	**186.00**	**226.00**		**90.00**	**152.00**	**186.00**	**226.00**
	MLP(F)	357	**85.00**	**150.00**	**180.00**	**223.00**		**85.00**	**150.00**	**180.00**	**223.00**
	EM(F+P)	224	**84.00**	**108.00**	**114.00**	**119.00**		**84.00**	**108.00**	**114.00**	**119.00**
	PM*(F+P)	224	**49.00**	**74.00**	**86.00**	**110.00**		49.00	74.00	86.00	110.00
	PM+(F+P)	224	**33.00**	57.00	68.00	**84.00**		33.00	57.00	68.00	**84.00**
	LR(F+P)	224	**88.00**	**120.00**	**129.00**	**139.00**		88.00	**120.00**	129.00	139.00
	MLP(F+P)	224	**113.00**	**145.00**	**159.00**	**181.00**		**113.00**	**145.00**	**159.00**	**181.00**
	PC(F+P)	224	96.00	**143.00**	**157.00**	**178.00**		96.00	**143.00**	**157.00**	**178.00**

rate, as can be seen from the results obtained using the sampling rate of 0.1.

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 3.7 (right side) contains the results obtained using stratified mutant sampling with $N \in \{5, 10, 15, 20\}$. The value in the parenthesis, i.e., "Ratio", is the average 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 0.27 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.

Answer to RQ4: The impact of mutation sampling is observable but not too disruptive. Using uniform sampling, on average 80% of the faults ranked at the top without sampling can still be localised at the top. However, stratified sampling actually harms SIMFL: larger methods need to be represented by more mutants.

Table 3.8: The effects of removing subsumed mutants.

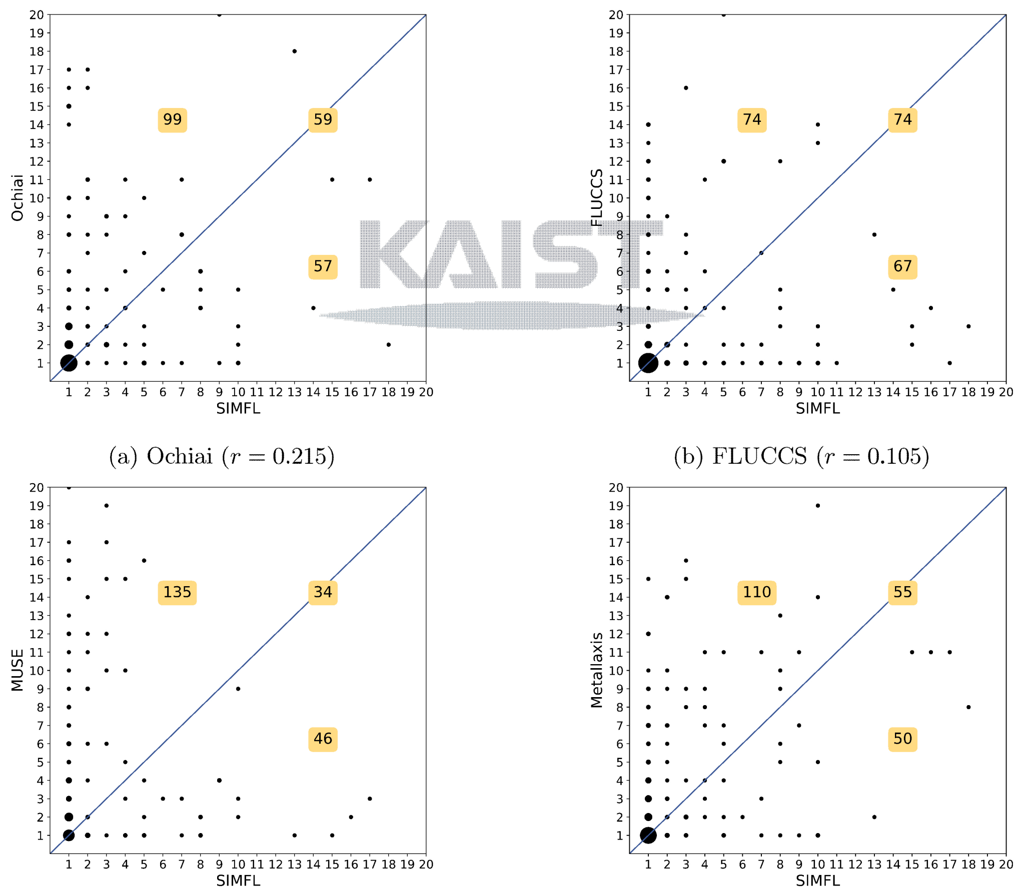
Model	Total	*acc*@1		*acc*@3		*acc*@5		*acc*@10	
EM(F)	357	92	+15	129	−10	144	−19	164	−40
PM*(F)	357	126	+44	173	+22	191	+8	211	−14
PM+(F)	357	123	+37	170	+15	189	+3	208	−19
LR(F)	357	122	+32	170	+18	184	−2	216	−10
MLP(F)	357	136	+51	183	+33	206	+26	230	+7
EM(F+P)	224	75	−9	96	−12	106	−8	115	−4
PM*(F+P)	224	71	+22	110	+36	123	+37	143	+33
PM+(F+P)	224	44	+11	91	+34	107	+39	139	+55
LR(F+P)	224	103	+15	134	+14	145	+16	156	+17
MLP(F+P)	224	119	+6	150	+5	157	−2	168	−13
PC(F+P)	224	102	+6	134	−9	145	−12	155	−23

3.4.5 Effects of Subsumed Mutants (RQ5)

It has been pointed out that mutation tools generate numerous subsumed and redundant mutants, resulting in a mutation score to be easily misunderstood [80, 81]. As the killing of the subsuming mutants always accompany the killing of their subsumed mutants, this problem not only inflates the mutation score but also inflate our method-level FL scores by SIMFL models. We investigate the effects of removing subsumed mutants. First, we identify the most subsuming mutants [87], which are located at the top of DMSG (Dynamic Mutant Subsumption Graph) [81] and are not subsumed by other mutants except their indistinguishable mutants.[5] Next, we conduct RQ1 study using only subsuming mutants and compare *acc@n* with the results of the original RQ1 study that uses all mutants.

As shown in Table 3.8, *acc*@1 of all models except EM(F+P) largely improved: MLP(F) localised 51 more faults at the top compared to the model using all mutants. The reason EM(F+P) worsened for all *acc@n* is that the subsuming mutants are likely to be killed by a few tests and they eliminate subsumed mutants needed for exact matching. This reason also explains why the overall *acc@n* of F models improved more than F+P models.

Answer to RQ5: Although we used fewer mutants, removing subsumed mutants brings positive effects to SIMFL models by increasing the number of faults located at the top up to 51, compared to using all mutants.



(a) Ochiai ($r = 0.215$)

(b) FLUCCS ($r = 0.105$)

(c) MUSE ($r = 0.072$)

(d) Metallaxis ($r = 0.130$)

Figure 3.4: Comparison of MLP(F+P) and other FL techniques.

[5] The two mutants are said to be indistinguishable if they have the same test results for all tests.

3.5 Discussion

3.5.1 Relation with Other FL Techniques

Two FL techniques can be complementary to each other if there is little overlap between faults ranked highly by each technique. To investigate whether the contribution of SIMFL is uniquely different from others, we investigate how individual faults are ranked differently by SIMFL and other FL techniques. SIMFL is represented by the MLP(F+P) model. We omit TraPT from this comparison as the individual rank information was not available from the paper; DStar is also excluded as its results are very similar to that of Ochiai.

Figure 3.4 plots each individual fault according to its rank by MLP(F+P) of SIMFL (x-axis) and its rank by the other FL technique (y-axis). Data points on the line $y = x$ represent faults that are ranked at the same place by both techniques, whereas the farther from the line a point is, the more differently it is ranked by two techniques. Plots only contain faults that are ranked within the top 20 places by at least one technique: the size of the dots corresponds to the number of faults plotted at the location of the dot. The numbers on the $y = x$ line as well as above and below the line show the total number of faults that belong to the corresponding parts, regardless of being ranked within the top 20 or not. For example, SIMFL ranks 135 faults higher than MUSE. The agreement between two techniques are measured using Pearson correlation coefficient (r): value 0 implies no correlation and, therefore, no agreement, whereas value 1 implies perfect correlation and, therefore, two identical rankings.

While there exist dense clusters of points near the top ranks around the $y = x$ line, there is no clear relationship between FL techniques. SIMFL shows low Pearson correlation coefficients against all compared techniques. Notably, SIMFL is significantly different from two existing MBFL techniques, MUSE and Metallaxis, suggesting that the way SIMFL captures the relationship between faults and tests differs significantly from existing MBFL techniques. SIMFL also ranks the most faults identically to FLUCCS, a technique that uses multiple SBFL scores as well as code and change metric, suggesting that mutation analysis can be a rich source of information for fault localisation. Overall, the results provide evidence that there exist faults that SIMFL can localise much more effectively than the other, and vice versa. The complementary nature also suggests the possibility of an effective hybridisation of SIMFL and other techniques, as recent work that combine multiple FL techniques suggest [86, 88, 66]. We leave the hybridisation as future work.

3.5.2 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 [89], we hypothesise that this may be related to the test case granularity: if each test case kills mutants that exist in only a few methods, SIMFL can benefit from this because failures of each test case will be tightly coupled with a few candidate locations.

To investigate the impact of test case granularity, we check whether the number of the methods that are relevant to failures caused by highly ranked faults is lower than the number of methods relevant to faults that are not ranked near the top. We define a method m to be relevant to the failure of a test case t if t kills a mutant in m. A finer granularity test case t is expected to be relevant to fewer methods. We categorise faults into those ranked in the top three places (set $W3$), and those that are not (set $O3$), and compare the number of relevant methods between $W3$ and $O3$.

Table 3.9 reports the results of one-tailed t-test on the number of relevant methods between $W3$ and

Table 3.9: The result of one-tailed t-test on the number of methods whose mutants are killed by the failing test cases, between the faults localised within ($W3$) and out of top three places ($O3$). The significant level is 0.05: if p-value is less then 0.05 (typeset in bold), then we can *reject* null hypothesis that is the mean of $W3$ is significantly greater than $O3$.

Model	Project	$W3$		$O3$		p	Model	Project	$W3$		$O3$		p
		mean	std	mean	std				mean	std	mean	std	
EM (F)	Lang	5.5	5.2	10.0	12.9	**0.029**	EM (F+P)	Lang	6.2	5.7	7.9	12.0	0.238
	Chart	10.0	6.8	144.9	260.5	0.056		Chart	11.4	7.3	136.1	254.5	0.086
	Time	69.2	36.1	129.6	46.7	**0.002**		Time	103.7	54.2	113.7	49.2	0.320
	Closure	167.2	117.3	322.7	182.2	**0.000**		Closure	-	-	-	-	-
	Math	15.5	19.1	34.8	26.6	**0.000**		Math	20.5	25.1	28.5	21.7	0.056
PM* (F)	Lang	5.1	3.8	11.9	14.3	**0.002**	PM* (F+P)	Lang	4.7	3.7	9.8	11.6	**0.009**
	Chart	10.0	6.8	144.9	260.5	0.056		Chart	96.3	242.1	88.4	193.4	0.534
	Time	78.4	43.9	127.6	47.4	**0.009**		Time	20.0	27.6	120.3	42.4	**0.000**
	Closure	175.7	113.8	327.6	184.9	**0.000**		Closure	-	-	-	-	-
	Math	15.4	18.8	36.3	26.6	**0.000**		Math	15.2	19.5	28.2	24.3	**0.009**
PM$^+$ (F)	Lang	5.1	3.8	12.4	14.7	**0.002**	PM$^+$ (F+P)	Lang	4.4	3.5	9.2	10.9	**0.012**
	Chart	9.8	7.1	136.6	254.3	0.072		Chart	96.9	242.0	88.1	193.5	0.538
	Time	78.4	43.9	127.6	47.4	**0.009**		Time	0.5	0.5	117.7	43.2	**0.001**
	Closure	203.2	160.8	323.5	178.0	**0.000**		Closure	-	-	-	-	-
	Math	15.5	19.1	35.8	26.5	**0.000**		Math	11.5	9.9	27.1	24.9	**0.010**
LR (F)	Lang	4.9	3.9	13.5	14.7	**0.000**	LR (F+P)	Lang	5.5	5.0	12.2	14.8	**0.006**
	Chart	11.8	6.2	135.9	254.6	0.087		Chart	70.3	200.9	117.9	223.1	0.298
	Time	78.4	43.9	127.6	47.4	**0.009**		Time	90.8	56.3	129.6	36.6	**0.031**
	Closure	193.3	136.7	322.3	184.0	**0.000**		Closure	-	-	-	-	-
	Math	16.5	19.1	35.3	27.1	**0.000**		Math	17.3	18.1	31.0	26.3	**0.003**
MLP (F)	Lang	5.4	5.1	12.9	15.0	**0.003**	MLP (F+P)	Lang	5.9	5.3	13.3	18.4	**0.013**
	Chart	10.8	6.6	144.9	260.5	0.066		Chart	14.6	10.1	174.2	283.7	**0.032**
	Time	62.3	40.0	137.7	34.7	**0.000**		Time	95.1	54.9	130.5	37.7	**0.049**
	Closure	168.6	114.0	325.3	183.0	**0.000**		Closure	-	-	-	-	-
	Math	15.6	18.9	35.8	26.7	**0.000**		Math	17.0	16.6	40.9	29.6	**0.000**
							PC (F+P)	Lang	5.4	4.0	12.7	16.5	**0.004**
								Chart	76.6	193.5	105.7	230.1	0.371
								Time	91.4	43.7	132.3	53.0	**0.025**
								Closure	-	-	-	-	-
								Math	16.5	17.4	41.8	28.1	**0.000**

$O3$: for 33 out of 45 cases, we accept the alternative hypothesis that the mean of $O3$ is significantly greater than $W3$. In other words, the faults in $W3$ are likely to be revealed by test cases with finer-granularity than the faults in $O3$. The test cases of Commons Lang have 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 (mutants) and their unit tests: failing test cases of Closure kill mutants in 203 methods on average, but PM$^+$(F) can still localise 41 out of 132 faults within the top three places (see Table 3.3).

Table 3.10: The $acc@n$ metric values after filtering mutants based on their kill reasons.

Model	Total Studied	All			Assertion			Timeout			Exception		
		@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5
EM(F)	357	77	139	163	**100**	**163**	**179**	21	30	38	53	90	107
PM*(F)	357	82	151	183	**108**	**185**	**206**	23	35	44	60	97	117
PM$^+$(F)	357	86	155	186	**114**	**183**	**206**	22	36	45	61	99	119
LR(F)	357	90	152	186	**118**	**181**	**213**	40	79	88	66	109	131
MLP(F)	357	85	150	180	**121**	**189**	**210**	43	76	91	60	106	129
EM(F+P)	224	**84**	**108**	**114**	72	89	96	7	11	16	55	64	68
PM*(F+P)	224	49	74	86	**50**	**76**	**90**	23	42	57	49	71	80
PM$^+$(F+P)	224	33	57	**68**	**34**	57	**68**	20	34	50	**34**	**59**	**68**
LR(F+P)	224	88	**120**	**129**	**89**	117	128	23	41	51	77	104	115
MLP(F+P)	224	**113**	145	159	112	**147**	**160**	29	50	55	91	120	135
PC(F+P)	224	96	143	157	**100**	**145**	**160**	22	36	43	78	113	121

3.5.3 Kill Reason Filtering

A mutated program can cause a test failure due to many different reasons, such as assertion (i.e., test oracle) violation, uncaught exception, or timeout. All these reasons are normally marked as a kill. While all three reasons do reveal some dependency between the mutated location and the test outcome (otherwise the mutant would not be killed), we suspect that different kill reasons may have varying degrees of importance for fault localisation. Assertion violations would imply that the test oracles actually capture the correct program behaviour. Uncaught exceptions and timeouts, however, may only show coincidental impacts of the mutation.

Considering the relative importance of different kill reasons, we investigate whether filtering out the kill matrix based on the exact reason of test failure has any impact on the localisation effectiveness. This is partly motivated by the use of failure messages by TraPT [90]. We train SIMFL models using one of three kill reasons, and compare their results to those of models trained using all three reasons. Kill reasons supported by Major are: assertion violations ("Assertion"), timeouts ("Timeout"), and uncaught exceptions ("Exception").

Table 3.10 shows the results of $acc@n$ metrics for SIMFL models of three different kill reasons. For all F models, using only mutants killed due to the assertion failures shows the best performance in terms of $acc@1$ and $acc@3$, adding support to our assumption that assertion violations reflect test oracles of correct program behaviour better than others. Timeouts appear to be the weakest signal.

However, for F+P models, the unfiltered original results ("All") often show the best performance. This trend reveals a seemingly counter-intuitive, yet fundamental intuition about SIMFL: test cases in $\mathbf{T}_f$ and $\mathbf{T}_p$ contribute to localisation in different ways. If a test case t is in $\mathbf{T}_f$, all mutants killed by t earlier suggest that their locations may contain the fault. However, if $t \in \mathbf{T}_p$, all mutants killed by t earlier suggests that their locations may *not* contain the fault that is detected by $t' \in \mathbf{T}_f$. Consequently, kill reason filtering can make the contributions from tests in $\mathbf{T}_f$ more precise (i.e., to only reflect real fault detection), but may also reduce the total amount of contributions from tests in $\mathbf{T}_p$ because it removes potential locations that could have been *excluded* by being associated with a test in $\mathbf{T}_p$. This explains why, for F+P models, using only Assertion as the kill reason cannot dominate the results. Note that the distribution of kills between Assertion, Timeout, and Exception is likely not uniform, which we also think contributes to the mixed results of F+P models, combined with program semantics.

3.6 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 caused by mutation and equivalent mutants, which have been discussed in Section 3.4. Similarly, we see few threats to the construct and conclusion validity. The metrics we used are standard in the fault localisation literature. We note that 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 generalise 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.

3.7 Related Work

3.7.1 Mutation-based Fault localisation

A number of MBFL techniques have been proposed in the literature. Metallaxis uses SBFL-like formulas to measure the similarity between failure patterns of the actual fault and mutants [4, 74]. MUSE [10], and its variation MUSEUM [23], depend 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, tests that used to pass are now likely to fail. MUSE and MUSEUM define their suspiciousness scores using the ratios of fail-become-pass and pass-become-fail tests. TraPT is similar to MUSE and MUSEUM in nature, but transforms both the output messages of failing tests, to distinguish different types of exceptions, and the test code itself, to prevent early program termination due to the assertion violation that precludes collecting information of other assertions [90].

MBFL techniques suffer from the cost of mutation analysis, as it requires a massive number of test executions against the generated mutants. To reduce the number of mutants needed to be inspected, Metallaxis [4] adopts mutant sampling while HOTFUZ [5] combines first-order mutants to build higher-order mutants and computes suspicious score only on them. Their results showed that this reduces the number of total mutants as well as equivalent mutants that contribute nothing to the fault localisation. Despite such efforts to reduce the costs, all existing MBFL techniques inevitably mutate the faulty program and execute tests once testing is finished. In contrast, SIMFL allows the mutation analysis to be performed ahead of time.

3.7.2 Bayesian Inference for Fault Localisation

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. [21] have introduced BARINEL, a 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. While SIMFL also uses dynamic information from mutation, the mutation analysis can be performed ahead of time, which allows the cost to be amortised over multiple development iterations, and provides faster feedback.

3.7.3 Mutant Subsumption

The subsumption relationship between two mutants appears when both are killed by the tests and all tests that kill the one also kill the other. Therefore, this indicates a redundancy of the subsumed mutant [17], allowing the mutation score to be inflated. As a result, building a minimal set of mutants can enhance the validity and efficacy of mutation analysis [81]. We showed that SIMFL can exploit this by removing the subsumed mutants in the kill matrix and achieve considerably better results than the normal setting using all mutants.

3.7.4 Predictive Mutation Testing

A recent advance to deal with the high costs of mutation testing is Predictive Mutation Testing (PMT) [7, 8]. PMT predicts the mutation score as well as test results of each mutant. Since PMT uses historical mutation testing results as its features to train the predictive model, PMT and SIMFL have a similar scenario that amortises the cost of mutation testing. However, PMT is originally designed to predict whether a test suite can kill the mutants, so it is hard to apply to SIMFL that requires the test-case level results of mutant kills. Seshat [9] provides a test-case level prediction of mutant kills using the tokens in the source code and test as well as the dynamic information of the mutants. The finer-grained prediction of Seshat enables its combined use with SIMFL, which helps SIMFL to deal with newly introduced test cases by simply predicting which mutants they can kill.

3.8 Summary of this Chapter

This chapter explores the mutant's relationship with faults and tries to solve the problem of fault localisation based on it. By exploiting its similarity in locality and patterns of test execution to the faults, we introduce new technique called SIMFL, a Mutation Based Fault Localisation (MBFL) technique utilising mutation analysis results of the earlier version. SIMFL models the relationship between the location of the artificial faults (i.e., mutants) and their test results via several modelling schemes such as statistical inference or machine learning techniques. After the faults are observed, SIMFL infers their locations without any mutant executions. We evaluate SIMFL on the real-faults in Defects4J dataset and the results show that SIMFL can locate 113 faults on the top out of 224 faults. Even if we simulate SIMFL on the predicted kill matrix, SIMFL still localises 95 faults on the top out of 194 faults, supporting its practical applicability. Moreover, we demonstrate that removing the subsumed mutants significantly improves the localisation accuracy of SIMFL.

Chapter 4. Mutant's Relationship with Patches

This chapter investigates the mutant's relationship with patches by looking at them as code changes. We hypothesise that the mutants (faults) and patches are syntactically indistinguishable from each other. We conduct three experiments to examine this hypothesis: a cluster analysis to see their syntactic similarity, Mutation-to-APR and APR-to-Mutation studies that use the mutants to repair the faults and the patches to seed artificial faults, respectively. This can allow us to integrate the two seemingly opposite approaches as well as provide a larger set of the faults or patches to be used for static bug finders or patch generation tools.

4.1 Introduction

Software bugs are prevalent, long-lived, and expensive to fix, thus they have been a major concern in software maintenance [91, 92]. The high cost of bug fixes mainly stems from the need for the developers to deeply engage with the bugs by following such steps as 1) reading the bug report, 2) inspecting the possible buggy locations, and 3) making several attempts to write patches (which involve compiling and running tests). All of these steps require a deep understanding of both the bug and the Program Under Test (PUT). In addition, developers should be aware of the possibility that, despite fixing the initial buggy symptoms, they may have introduced a new bug with their patch [93, 94], further adding to the complexity of bug fixes.

On the contrary, we note that it is trivially easy to *introduce* bugs. Given access to the source code, bugs can be introduced effortlessly, mainly because there exist infinitely more incorrect programs than the ones that satisfy the given specification in the space of all programs. This fundamental difference between writing patches and introducing bugs also can be seen in two types of automated software testing and debugging techniques: Automated Program Repair (APR) [25], which aims to automate the process of writing patches, and Mutation Testing [1], which aims to automate the process of introducing bugs. To successfully repair a bug, the APR technique needs to overcome many different challenges: it first has to accurately localise the bug, then find a specific combination of existing ingredients to compose a patch, and finally apply and validate the patch by executing test cases. On the contrary, mutation testing is context insensitive and is allowed to make small syntactic changes to an arbitrary location in the source code.

However, if we look closely at how APR tools operate, they act like mutation tools. While trying to find a patch, it is natural for them to produce many *faults*, which are the by-product of the search algorithms adopted by APR techniques. On the relationship between mutation testing and APR, Weimer et al. observed that Generate & Validate (G&V) APR techniques form a dual of mutation testing [24]: APR aims to find mutants that pass the tests, while mutation testing aims to find mutants that fail the tests. One example that exploits this duality is PraPR [43], a recently proposed APR tool that directly augments the mutation operators of a Java mutation testing tool, PIT [28]. The evaluation of PraPR shows that the repair attempts using only PIT mutation operators perform surprisingly well: for Defects4J v1.2.0 subjects, it produced the plausible patches for 106 buggy versions, correct patches for 17 buggy versions out of 395 versions.

Given that APR and mutation testing tools have exactly opposite purposes, how can one be successfully used as another? We hypothesise that it is mainly because the patches and faults, when seen simply as changes made to the code, are not that different from each other. For example, consider a code change, from `a + b` to `a - b`. Although this is a widely used arithmetic mutation operator, it can equally be a bug fixing patch. Similarly, even a code change that appears explicitly to be a patch (such as the insertion of a null-check for a pointer variable), may eventually turn out to be a fault if made to an incorrect location.

Based on this observation, we design three empirical studies to investigate the relationship between the patches and faults from different angles. First, we compare their lexical and structural similarities by leveraging code change clustering and pattern inferring algorithms, then evaluate whether the patches and faults are grouped together. Second, we evaluate two mutation tools as APR tools and evaluate their effectiveness. Finally, turning the table, we directly convert APR tools into mutation tools and evaluate their ability to generate mutants that are coupled with real faults. The results of our empirical evaluation suggest that buggy and fixing code changes are in fact more similar to each other than we expect. The implication of this finding is that, when mining a particular type of code changes (i.e., either bug inducing commits, or bug fixing commits), we can consider wider range of code changes than we thought before. Further, it suggests the possibility of cross-purpose uses of both APR and mutation testing tools, while hinting at a potential unified technique.

The main contributions of this chapter are as follows:

- We empirically evaluate the similarities between patches and faults using 6k code changes in C projects and 7k code changes in JavaScript projects, mined from various open source repositories. The results suggest that patches and faults are in fact similar to each other, as they can be grouped together by the code change clustering and pattern inferring algorithms.
- We present an empirical evaluation of mutation tools as APR techniques. We employ mutation tools that are inversions of existing APR tools, and evaluate these variations by applying them to the buggy programs in DEFECTS4J. The results shows that IBIR [95] (i.e., inverted TBar [96]) and inverted SequenceR [97] can still successfully generate 42 and 17 plausible patches, which are only eight and two fewer than their original forms.
- We also demonstrate that existing APR tools can be converted into mutation tools. A mutation coupling analysis using real faults in DEFECTS4J shows that TBar, a template-based APR tool, can successfully generate 14% and 3% more fault couplings than widely studied mutation tools, PIT and Major, respectively.

4.2 A Motivating Example

Figure 4.1 contains actual code changes from DEFECTS4J. Let us begin with a simple question: are these patches, or bug inducing changes? Figure 4.1a contains an example fix change from jfreechart that modifies != to ==, while Figure 4.1b shows the inversion of Figure 4.1a. Without the captions, it is not easy to tell them apart. We hypothesise that a code change itself is context insensitive, which is why it is hard to distinguish a patch from a bug inducing change, and vice versa. The difference becomes clearer only when we consider the surrounding context of the change, as shown in Figure 4.1c. An earlier conditional statement `if (this.plot == null)` has a similar predicate and the same return statement; we also observe that the variable `dataset` is subsequently used in Line 13, suggesting that

```
- if (dataset != null) {
+ if (dataset == null) {
```

(a) Fix change

```
- if (dataset == null) {
+ if (dataset != null) {
```

(b) Inverted fix change (i.e., bug inducing change)

```
  public LegendItemCollection getLegendItems() {
          ...
          if (this.plot == null) {
              return result;
          }
          int index = this.plot.getIndexOf(this);
          CategoryDataset dataset =
      ↪   this.plot.getDataset(index);
  -       if (dataset != null) {
  +       if (dataset == null) {
              return result;
          }
          int seriesCount = dataset.getRowCount();
          ...
```


(c) Fix change with surrounding context

Figure 4.1: A fix commit sof jfreechart (Chart 1 in DEFECTS4J)

it is more natural to return when it is **null**. With the help from the context, we can guess that the change in Figure 4.1a is likely to be a fixing change, and not a bug inducing one. This example raises the questions of whether mining and interpreting the given code changes only as either fix or bug inducing is desirable, as well as how actually different they are from each other. In the following three sections, we investigate and discuss those observations with three empirical studies, respectively.

4.3 Similarity Study (RQ1)

RQ1. How similar are patches and faults to each other? We answer RQ1 using two approaches: clustering both patches and faults, and abstracting both as change patterns. With clustering, we investigate whether patches and faults can belong to the same cluster, whereas with pattern inferring, we investigate whether patches and faults can be abstracted into the same pattern. Answers to these questions would provide evidence of how much the patches and faults have similar structures and patterns. We perform the clustering analysis using FlexiRepair [98], and the pattern inferring analysis using SemSeed [99].

4.3.1 Cluster Analysis with FlexiRepair

FlexiRepair [98] is an extension and combination of FixMiner [100] and Spinfer [101]. Below, we will briefly introduce them with the procedure of how the clusters are formed.

FixMiner

To represent a code change, FixMiner [100] constructs a tree representation with three types of information: `Shape`, `Action`, and `Token`. Each of the information type corresponds to an abstraction level of the resulting representation. FixMiner starts the clustering at the highest abstraction level, which is `Shape`, and successively applies clustering to the results from the previous abstract level with `Action` and `Token`.[1] First, at abstraction level of `Shape`, FixMiner groups the code changes based on the type of the root node of AST and its depth (e.g., IfStatement/7), and identifies their clusters using algorithms of GumTree [102]. Next, the abstraction level of `Action` further divides the clusters generated by `Shape`, based on the change operations identified by GumTree. These clusters are labelled as node/depth/ShapeTreeClusterId (e.g., IfStatement/7/2).

```
@@
expression E0;
@@
- if (isupper((int )*E0))
+ if (isupper(*E0))
  {
-   *E0 = tolower((int )E0);
+   *E0 = tolower(*E0);
  }
```

(a) IfStatement/21/1/0

```
@@
expression E0;
@@
- if (isdigit((int )*E0))
+ if (isdigit(*E0))
  {
   ...
  }
```

(b) IfStatement/21/1/1

Figure 4.2: Example clusters resulted from FlexiRepair

Spinfer

Spinfer [101] aims to infer semantic patches of Linux kernel by identifying the similar code fragments and control flows across code changes. Following FlexiRepair [98], we use Spinfer to finalise the clustering process on each cluster by FixMiner. As a result, the clusters generated by Spinfer have

[1] As FlexiRepair only considered `Shape` and `Action` for clustering, we exclude `Token` from the clustering process.

the lowest abstraction level and indexed as node/depth/ShapeTreeClusterId/ActionTreeClusterId (e.g., IfStatement/7/2/0).

The final output of Spinfer is the clusters of generic patches in the form of Coccinelle [103] transformation rules: Figure 4.2 shows two of generated clusters. They are assigned to the same cluster of IfStatement/21/1 at the `Action` abstraction level by FixMiner, but have been further divided into two smaller clusters, IFStatement/21/1/0 and IFStatement/21/1/1, by Spinfer because they have different control flow structures.

Table 4.1: C subject programs for cluster analysis using FlexiRepair

Repository	# Commits	Repository	# Commits
libtiff	3,570	openssh-portable	11,004
cmake	38,414	gmp	16,782
redis	8,770	lighttpd1.4	3,882
gzip	604	lighttpd2	1,551
libarchive	5,472	git	46,715
cairo	11,724	MonetDBLite-C	48,461
curl	26,967	freeradius-server	37,535
tcl	17,145	bind9	31,286
nginx	6,858	tmux	7,790
apr	8,945	gstreamer	19,016

Dataset

We reuse the dataset presented by FlexiRepair: it contains mined commits in C projects from Github, Gitlab, and Savannah. However, it was not possible for us to process all repositories provided by the FlexiRepair dataset due to the limited computational resources. Instead, we randomly selected 20 repositories, as listed in Table 4.1, and collect the code changes with the inverted changes, resulting in 720,000 changes. After applying the filtering rules of FlexiRepair to extract only fix commits, we have 3,000 fix changes that result in total 6,000 code changes considering inverted changes (i.e., bug inducing changes).

Evaluation

For the evaluation of FlexiRepair clusters, we use three levels of clusters from `Shape`, `Action`, and Spinfer, respectively. For the sake of simplicity, we hereafter denote the three levels as 'Level1', 'Level2', and 'Level3'. We report the number of clusters that include both the patches and faults for each level. Note that we cannot report other evaluation metrics such as cluster membership accuracy, as there is no ground truth of correct cluster membership for code changes.

4.3.2 Pattern Analysis with SemSeed

SemSeed [99] presents an efficient algorithm for seeding realistic bugs by abstracting and matching bug inducing patterns. SemSeed selects an AST subtree of the changed line and extracts two token sequences (before and after the change) from the subtree. From them, the bug seeding patterns are inferred by abstracting all identifiers and literals into the placeholders, resulting in a pair of abstracted token sequences, $t = \langle t_x, t_y \rangle$. The tokens in t_x and t_y are either identifier or literals, or non-identifiers and non-literals.

As SemSeed only learns its patterns from faults, we modify it to consider both patches and faults, and see how many same $\langle t_x, t_y \rangle$ pairs are found in both of them. Let $t \in T_p$ be the pairs from the patches and $t \in T_f$ be the pairs from the faults. We report the number of pairs that exist in both T_p and T_f, which would be the code changes that represent both patches and faults. Based on the dataset presented by SemSeed that contains 3,600 inverted fix changes from 100 JavaScript projects in Github, we build our own that includes both fix and inverted changes, resulting in total 7,200 code changes.

Table 4.2: Cluster analysis with FlexiRepair. x refers to the number of clusters having both patches and faults and y refers to the number of total clusters.

Cluster level	Level1 (node/depth)	Level2 (ShapeTreeClusterId)	Level3 (SpinferClusterId)
x / y	222 / 316 (70%)	569 / 1,800 (31%)	227 / 3,859 (7%)

```
@@
identifier I0;
expression E1;
@@
- char *I0 = rad_malloc(E1 + 1);
+ char *I0 = rad_malloc(E1);
```

(a) DeclStmt/10/0/0

```
@@
identifier I0;
expression E1;
@@
- const char *I0 = E1;
+ char *I0 = (char *)E1;
```

(b) DeclStmt/10/18/2

Figure 4.3: Example Level1 cluster (DeclStmt/10)

4.3.3 Results of Cluster Analysis with FlexiRepair

Table 4.2 shows the number of clusters that have both patches and faults (x) and the total number of clusters (y). Among all clusters, 70% of Level1 clusters contain both patches and faults, followed by 31% of Level2 clusters, and 7% of the Level3 clusters: the trend confirms our expectation that the less we abstract the code changes, the more separated they would be. Figures 4.3 and 4.4 present a closer look at how clusters are formed at Level1 and Level2: Figure 4.3a and Figure 4.3b are in same cluster of DeclStmt/10, and Figure 4.4a and Figure 4.4b are in same cluster of ExprStmt/5/0. Both levels

```
@@
identifier I1 = {getch,strequal};
expression list E2;
expression E0;
@@
- E0 = I1(E2);
+ E0 = (char )I1(E2);
```

(a) ExprStmt/5/0/1

```
@@
assignment operator A1;
expression E2, E0, E3;
@@
- E0 A1 strlen(*E2) + E3;
+ E0 A1 (int )strlen(*E2) + E3;
```

(b) ExprStmt/5/0/2

Figure 4.4: Example Level2 cluster (ExprStmt/5/0)

have placeholder identifiers or expressions that can be replaced, but Level2 clusters tend to show closer resemblance.

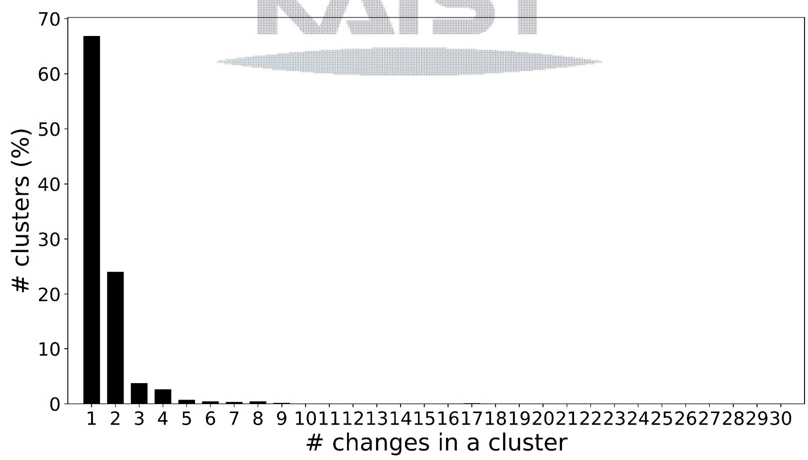


Figure 4.5: The number of code changes for each Level3 cluster.

In addition, we examine how many code changes are in each Level3 cluster, since the overlaps of the patches and faults in Level3 clusters are only 227 out of 3,859 cases (7%). Figure 4.5 shows that 67% of Level3 clusters contain only one code change, failing to be grouped with any others due to the low abstraction level. Once we exclude such singletons, the proportion of the clusters that have both the patches and faults becomes 22%.

```
- reject(error);
+ reject(convertImapError(error));
```

(a) Fix commit of Adobe Brackets @a460c47

```
- callback(_err);
+ callback(mapError(_err));
```

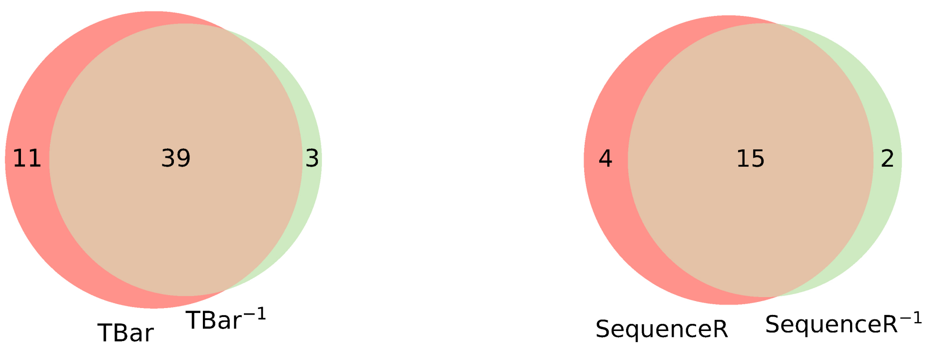
(b) Inverted fix commit of nylas-mail @93942e7

Figure 4.6: Two code changes having same SemSeed patterns

4.3.4 Results of Pattern Analysis with SemSeed

Next, we analyse the patterns of the patches and faults using SemSeed. We count the code changes whose patterns (t) are found in both patches (T_p) and faults (T_f). Out of 3,951 code changes for each group, 1,752 code changes (44%) have patterns found in both T_p and T_f. For example, Figure 4.6 shows two code changes from different projects: Figure 4.6a is a fix change from Adobe Brackets which adds a method call `convertImapError` around `error`, and Figure 4.6b is a bug inducing change from nylas-mail which does similar modifications to Figure 4.6a. Both changes are abstracted into the pattern from $Idf_1(Idf_3)$; to $Idf_1(Idf_2(Idf_3))$; where Idf represents a placeholder identifier.

Answer to RQ1: up to 70% of patches and faults can be clustered together by FlexiRepair; 44% of them can be abstracted into the same pattern by SemSeed. We conclude that patches and faults are not mutually exclusive and can be similar to each other.



(a) TBar and TBar^{-1} (b) SequenceR and SequenceR^{-1}

Figure 4.7: The number of bugs for which mutation or APR can generate plausible patches.

4.4 Mutation-to-APR Study (RQ2)

RQ2. Can mutation tools be APR tools? Given that the patches and faults have similar forms, we now move on to the cross-evaluation between them by investigating the patch effectiveness of the mutation. We treat the generated faults as patches and see whether they are actually plausible patches. If the patches and faults are fundamentally different and mutually exclusive, mutation tools will fail to generate any plausible patches.

Ideally, we would use existing mutation tools to perform this study. However, among the available and widely used Java mutation tools, PIT [28] has already been evaluated as an APR tool by PraPR [43], whereas Major [29] does not allow any modifications that are necessary for our experimentation as its source code is not available. As a result, we use IBIR [95], which mutates source code using inverted templates of TBar [96], and SequenceR^{-1}, which is originally the Neural Machine Translation (NMT) based APR tool, SequenceR [97], but trained with inverted dataset so that it 'translates' correct code to buggy code. Essentially, these two tools are direct inversions of APR tools, meaning that they are now designed to introduce bugs. Consequently, if these inversions can still patch bugs, it would show that our hypothesis about the similarities between patches and bugs is correct.

4.4.1 Modification Details

For this study, we introduce the following modifications to the original tools.

- *IBIR:* To use IBIR as an APR tool, we take the source code of TBar and replace the fix templates of TBar with the templates of IBIR. We switch templates instead of directly using IBIR, in order to take advantage of APR related support functionalities that already reside in TBar. We denote this modified tool TBar^{-1} to differentiate it from the original TBar as well as IBIR.

- *SequenceR:* SequenceR learns to translate a buggy code into a correct code using the fix changes. We follow the same training procedure, but we place the buggy code in the position of the correct code, and vice versa: this model will learn how to translate the correct code to the buggy code. We denote this model SequenceR^{-1}.

4.4.2 Evaluation Metrics and Configurations

We report and compare the number of bugs for which mutation and APR tools generate plausible patches. We use Defects4J v1.2.0, which includes four project and 231 bugs.[2] We use the old version of Defects4J, v1.2.0, as some of the studied tools are specifically designed to target Defects4J v1.2.0. Note that we compare SequenceR and SequenceR^{-1} using only 75 bugs, as the original tool is designed to target only one line patches.

4.4.3 Results

We run two mutation tools, TBar^{-1} and SequenceR^{-1}, and two corresponding APR tools, TBar and SequenceR, on the buggy programs in Defects4J. Figure 4.7 depicts two Venn diagrams that show the number of bugs for which each tool can generate plausible patches. TBar generates plausible patches for 50 bugs, which is eight more than those fixed by TBar^{-1}; SequenceR generates plausible patches for 19 bugs, which is two more than those fixed by SequenceR^{-1}. There are overlaps of 73% (39) between TBar and TBar^{-1}, and 71% (15) between SequenceR and SequenceR^{-1}, respectively. Although, APR tools are better than mutation tools at generating plausible patches in general, the results suggest that mutation tools can successfully perform as APR tools even though they learnt to operate in the opposite direction.

We also perform a qualitative analysis of the bugs that either APR or mutation tool exclusively fixes. Of 11 bugs fixed only by TBar, eight are fixed with the insertion operators, five of which are

[2]See details at https://github.com/rjust/defects4j/tree/v1.2.0.

related to adding null pointer checkers. On the other hand, out of three bugs fixed only by TBar^{-1}, two are fixed with the deletions operators. We suspect that this is due to the difference in the number of operators in these tools: TBar includes 12 insertion operators but only two deletion operators, meaning that TBar^{-1} has 12 deletion operators and two insertion operators.

SequenceR exclusively fixes four bugs, three of which are fixed with the variable replacements. The two exclusive fixes by SequenceR^{-1} are also related to the variable replacement and changing the condition of a if statement. Since SequenceR is an NMT based tool, it is difficult to analyse and interpret its operations. Further analysis would require improved explainability of the underlying NMT models.

> **Answer to RQ2**: despite having learnt from faults, mutation tools can successfully find plausible patches for 42 and 17 buggy programs in DEFECTS4J, compared to their counterpart APR tools that find plausible patches for 50 and 19 buggy programs.

4.5 APR-to-Mutation Study (RQ3)

RQ3. Can APR tools be mutation tools? We investigate whether the code changes produced by APR tools can be used as bug injections, despite the original intention of being patches. Conducting a real fault coupling study by following Just et al. [76], we evaluate the effectiveness of mutants produced by APR tools. As there are inherent differences between APR tools and mutation tools, we describe the challenges we faced, as well as how we dealt with them.

4.5.1 What kinds of APR tools can we modify?

Table 4.3 presents a list of Java APR tools we have considered. Our final selection criteria are as follows:

- *Availability*: we exclude the tools that are not publicly available or do not make their source code publicly available. Hercules [104], CapGen [73], and CURE [105] are excluded for this reason.
- *Executability*: We exclude ssFix [106] because it fails to connect to the private code search engine, and CoCoNut [107] because it has unresolved issues in preprocessing of training data as well as the model training.[3]
- *Failing tests*: We exclude the tools that require failing tests for patch generation, because we will apply them to correct programs for injecting faults. We exclude APR tools based on genetic programming, such as GenProg [108] and ARJA [109], because they use fitness functions that check whether tests that originally failed subsequently pass.

After filtering, we are left with four APR tools: SimFix [110], PraPR [43], TBar [96], and SequenceR [97]. These have all been recently published and open sourced; further, they do not require failing tests.

4.5.2 How to modify them?

Due to the differences in design goals between APR and mutation tools, we are forced to make a few modifications to the chosen APR tools:

[3] https://github.com/lin-tan/CoCoNut-Artifact/issues/2

Table 4.3: Considered Java APR tools

Tool	Selected?	Public?	Working?	Failing tests	Published year
GenProg [108]	No	Yes	Yes	Yes	2011
Angelix [111]	No	Yes	Yes	Yes	2016
Nopol [112]	No	Yes	Yes	Yes	2016
ssFix [106]	No	Yes	No	No	2017
CapGen [73]	No	No	-	No	2018
ARJA [109]	No	Yes	Yes	Yes	2018
SketchFix [113]	No	Yes	Yes	Yes	2018
SimFix [110]	Yes	Yes	Yes	No	2018
Hercules [104]	No	No	-	No	2019
PraPR [43]	Yes	Yes	Yes	No	2019
TBar [96]	Yes	Yes	Yes	No	2019
SequenceR [97]	Yes	Yes	Yes	No	2019
CoCoNut [107]	No	Yes	No	No	2020
CURE [105]	No	No	-	No	2021

Where to fix (i.e., mutate)

APR tools employ FL techniques to locate the buggy statements. In contrast, mutation tools usually have manual options for specifying the files to be mutated. Thus, we make APR tools to target the locations to be mutated by directly manipulating the FL results.

When to terminate

for mutation testing, we assume that PUT has no defects, i.e., it has a green test suite. However, APR tools assume that PUT has defects, so they terminate when the candidate patch passes all tests. As our mutation goal is to simulate all possible mutants, we modify the termination criterion of APR tools so that they generate all target mutants without considering test results.

Filtering some pre-defined patterns

TBar is based on the templates that have been collected from the fix patterns in the repositories, as well as patches generated by other APR tools. Among the collected patterns, there are some patterns that are highly likely fix-patterns, e.g., inserting a null pointer checker. Even if it is possible that the developers would insert a wrong or inappropriate null pointer checker, we exclude such patterns by default, as we posit that those cases are rare. The two APR tools, TBar and PraPR, are modified in this way. However, to evaluate the effect of this filtering strategy, we also include the versions without such filtering, which we denote TBar_α and PraPR_α.

Sampling

APR tools are generally designed to focus their efforts into a single location that is believed to be the location of the fault (which is why their results are highly dependent on the FL results [114]). In contrast, mutation tools aim to evenly spread their efforts across the entire program, with ways to sample mutants to control their numbers. In this regard, we modify APR tools so that we randomly sample only five mutations per location.

4.5.3 Coupling Study

Coupling Effect Hypothesis (CEH) states that, if a test suite can detect and kill simple mutants, it will also be able to detect larger and more complex faults [2]. This is why mutation testing is supposed to work. As such, the effectiveness of a mutation testing tool can be precisely measured if we can measure the degree of coupling with real faults.

We follow the same procedure of the coupling study adopted by Just et al. [76]. The mutants are said to be *coupled* with real faults, if they are killed only by the test cases that reveal the real faults. Given that there are failing test cases, $ft_i \in FT$, let T_{pass} be a set of passing tests and T_{fail} be a set of tests that includes all tests in T_{pass} and a single failing test case, i.e., $T_{fail} = T_{pass} \cup \{ft_i\}$. We then compose the pairs of T_{pass} and T_{fail} denoted by $\langle T_{pass}, T_{fail}\rangle$ using the bugs and the corresponding failing test cases in DEFECTS4J. For each pair, if there is at least one mutant that survives T_{pass} but is killed by T_{fail}, we mark the pair as *coupled*. Generating mutants using the modified APR tools, we compare the number of *coupled pairs* with the results from the two baseline mutation tools, PIT and Major.

Although we limit the mutant generation using random sampling at runtime, the number of generated mutants varies significantly between the studied tools. It would not be a fair comparison if there is a tool that generates much more mutants than others, as it will by definition have a higher chance of being coupled with real faults. Therefore, for any tool that generates more mutants than our reference mutation tool, Major, we further take a random sample out of those tools so that we consider the same number of mutants as Major. We repeat the experiment 30 times and report numbers on average. We will also report the unsampled total couplings, to see the impact of this additional sampling.

4.5.4 Configurations

We use the default set of mutation operators provided by DEFECTS4J for Major, and the mutators in 'old defaults group' for PIT.[4] To compose $\langle T_{pass}, T_{fail}\rangle$ pairs from bug benchmark, we use DEFECTS4J v1.2.0 and exclude some subjects that we fail to run all subject tools on, resulting in 493 pairs. We ignore any equivalent mutants, as we are only interested in coupled mutants, while equivalent mutants cannot be coupled by definition. For all APR tools, we follow the settings specified in their original study. To alleviate a huge cost of running tests against mutants, we generate mutants on the files that the target faults reside in.

4.5.5 Results

We begin by presenting the results of the fixed-size fault coupling study, for which we sample the same number of mutants as Major from all studied mutation tools. Subsequently, we present the results of unrestricted coupling study, for which we use all mutants generated by each tool. Finally, we present a qualitative analysis of the new types of mutations introduced by APR tools.

[4] https://pitest.org/quickstart/mutators/

Table 4.4: Results of a fixed-size fault coupling study

Tool	# Coupled Pairs	# Total Pairs	# Total Mutants
Major	300	493	31,877
PIT	249	493	29,855
TBar	316	493	31,679
TBar^{-1}	314	493	31,679
TBar$_{\alpha}$	306	493	31,731
SequenceR	90	493	29,291
SequenceR^{-1}	132	493	24,720
SimFix	99	493	29,411
PraPR	204	493	31,874
PraPR$_{\alpha}$	175	493	31,877

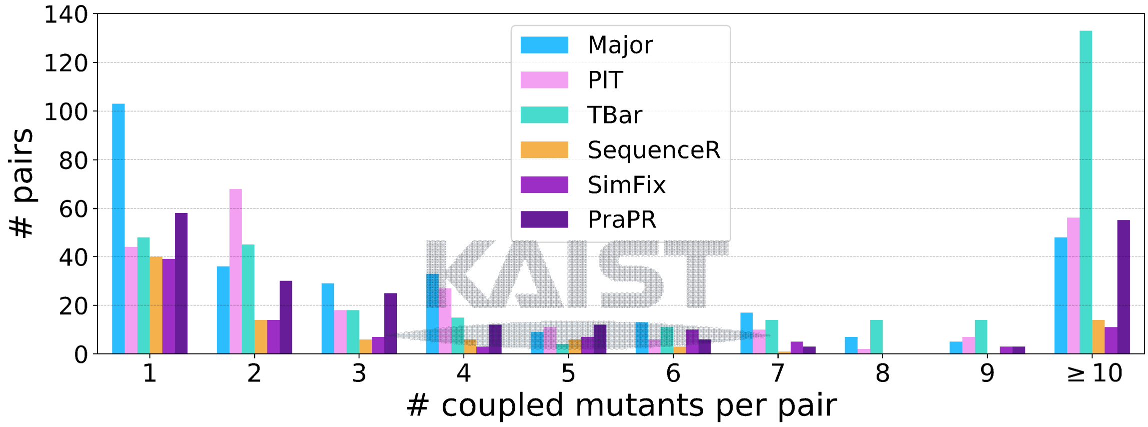


Figure 4.8: A distribution of the number of coupled mutants for each coupled pair.

Fixed-Size Coupling Study Results

Table 4.4 presents the results of the fixed-size coupling study: it shows the total number of pairs, the number of coupled pairs, and the total number of mutants considered for each tool. TBar and its variants perform better than two traditional mutation tools by making up to 316 coupled pairs out of total 493 pairs (64%). In addition, we examine the number of coupled *mutants* for each coupled pair as shown in Figure 4.8. On the pairs that have more than ten coupled *mutants*, TBar makes 133 coupled paired whereas Major and PIT make 48 and 56 coupled pairs.[5] It is well known that the fix templates of TBar have been a powerful baseline for APR, and its effectiveness is shown to be valid for mutant generation.

APR tools other than TBar, on the other hand, are outperformed by the two mutation tools, Major and PIT. A closer inspection of the *mutants* generated by SequenceR and SimFix suggests that the degree of freedom these tools have sometimes results in mostly frivolous code changes. Mutations

[5]Note that the duplicated mutants [115], i.e., mutants that are semantically different from the original, but equivalent to each other, may inflate these numbers. However, since mutant equivalence is in general undecidable, we simply report the raw results. We expect TBar to be largely free from this effect, as each mutant it generates correspond to a unique template. We have excluded syntactic duplicates.

created by these tools include insertion of duplicate logical clauses (e.g., changing `if (a != 0)` to `if ((a != 1) && (a != 0))`) or insertion of pointless parentheses (e.g., changing `return m != null && m.containsKey(value)` to `return (m != null) && (m.containsKey(value))`). We hypothesise that these kinds of changes lack purposes, both as a patch and as a mutation: compared to the carefully curated change templates of TBar, the changes made by SequenceR is much more difficult to interpret as it depends on a sequence-to-sequence NMT model. Since the NMT model does not concern semantics of the produced token squences, some of the *translations* may lack purposes. Similarly, SimFix depends mostly on structural similarity to source the change to be applied as mutation. Without considering the surrounding semantic context, the changes are also likely to lack focus.

Interestingly, PraPR performs worse than PIT, producing 204 coupled pairs compared to 249 made by PIT, although it is built based on PIT by augmenting nine mutation operators of PIT with six additional operators. However, it is the six newly introduced operators that mainly contribute to the performance deterioration. These operators perform either addition of field and method guards, or addition of pre/post conditions, which we think are too specialised for the purpose of program repair to perform as generic fault injection.

Finally, we investigate whether filtering pre-defined patterns has any merits by comparing TBar with TBar_α, and PraPR with PraPR_α, respectively. Both comparisons show that the filtering helps generation of coupled mutants: TBar makes ten more coupled pairs, while PraPR makes 29 more.

Table 4.5: Results of an unrestricted fault coupling study

Tool	# Coupled Pairs	# Total Pairs	# Total Mutants
Major	300	493	31,877
PIT	306	493	48,133
TBar	416	493	83,185
TBar^{-1}	413	493	82,943
TBar_α	414	493	103,603
SequenceR	154	493	50,072
SequenceR^{-1}	164	493	30,773
SimFix	166	493	64,887
PraPR	326	493	186,640
PraPR_α	328	493	242,258

Unrestricted Coupling Study Results

Table 4.5 shows the results of a coupling study using all generated mutants. TBar and its variants again outperform others by making up to 416 coupled pairs out of 493 total pairs (84%), showing that pre-defined fix patterns are capable of generating effective mutants. SequenceR and SimFix also perform better when we do not restrict the number of mutants: they generate 64 and 67 more coupled pairs compared to the fixed-size coupling study results. However, they are still significantly less effective than PIT and Major.

While unrestricted PraPR and PraPR_α outperform PIT and Major, this is mainly thanks to the huge number of mutants they generate: PraPR_α generates 200k mutants, compared to 40k generated by PIT. The operators newly introduced to PraPR are responsible for these extra 160k mutants, but they only contribute 20 additional coupled pairs.

In the unrestricted coupling study, TBar$_\alpha$ and PraPR$_\alpha$ subsume TBar and PraPR, respectively. Therefore, the effect of filtering some of the pre-defined patterns can be clearly observed by comparing them: both TBar$_\alpha$ and PraPR$_\alpha$ only make two more coupled pairs than TBar and PraPR, respectively, despite generating 1.2 times more mutants. Consequently, we conclude that the advantages of using all pre-defined patterns are negligible.

New and Stronger Mutation Operators

```
+ for (int i = 0; i < listSize; i++) {
+     if (!ShapeUtilities.equal ...
```

(a) Chart 6 (fix)

```
- ... Partial(iChronology, newTypes, newValues);
+ ... Partial(newTypes, newValues, iChronology);
```

(b) Time 4 (fix)

```
- ... convertLocalToUTC(localInstant, false);
+ ... convertLocalToUTC(localInstant, false, instant);
```

(c) Time 26 (fix)

```
- return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
+ return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
```

(d) Closure 10 (fix)

```
+ removeDuplicateDeclarations(root);
...
- removeDuplicateDeclarations(root);
```

(e) Closure 102 (fix)

Figure 4.9: The real faults in DEFECTS4J that are exclusively coupled with the mutants of APR tools.

Just et al. [76] reported that 27% of real faults are not coupled to the mutants generated by Major, and categorised the types of those uncoupled real faults. Based on it, we investigate the real faults to which PIT and Major cannot couple any mutants, but APR tools can, and report the operators used by the APR tools.

- *Statement or code deletion*: TBar can delete a single or multiple lines of code (see Figure 4.9a), whereas both PIT and Major lack the Statement Deletion (SDL) mutation operator to avoid compilation failures.
- *Argument swapping*: TBar and PraPR can generate mutants that swap the arguments to the method call (see Figure 4.9b), as they anticipate swapped arguments as a potential developer mistake.
- *Argument omission*: TBar, PraPR, and SequenceR can remove extra arguments from a method call (see Figure 4.9c), anticipating such omissions as a potential mistake.

- *Similar method called*: PraPR can change the method called to a similar method since PraPR has a method replacement operator that is not a part of PIT operators (see Figure 4.9d), in anticipation of developer mistakes.
- *Statement shifting*[6]: TBar can change the location of a statement to the other line thanks to its move statement operator (see Figure 4.9e).

> **Answer to RQ3**: APR tools can successfully generate the mutants coupled with real faults, revealing new and stronger mutation operators.

Figure 4.10: A web interface displaying the tasks given to the developers. In the left side, we present a code change with three surrounding lines. In the right side, participants can select either 'Break the program' or 'Fix the program' based on their understanding of the given code change.

4.6 Discussion

This section presents a preliminary human study of the perceived similarities between patches and faults, and subsequently discusses the differences in results obtained from algorithmic experiments in RQs 1 to 3 and from human perceptions.

4.6.1 Developers' Perspective: A Preliminary Study

Our answers to the three RQs provide evidence of similarity between patches and faults. However, they are based either only on lexical patterns (RQ1), or on the results of test executions (test based patch validation for RQ2, and fault coupling for RQ3). Neither reflects the semantics of the underlying code properly. To examine the similarity between patches and faults semantically, we have conducted an IRB approved (IRB-21-305) preliminary human study with five developers with an average of six years of industry experience.

We prepared 20 code changes, ten patches and ten faults (i.e., inverted patches), from Defects4J, and asked the participants to label them as patches or faults after looking at the code change itself as well as three lines of surrounding code before and after the changes. Figure 4.10 depicts a web interface

[6]This type was not originally listed by Just et al. [76].

```
for (int i = 0; i < searchList.length; i++) {
+   if (searchList[i] == null  replacementList[i] == null) {
+       continue;
+   }
    ...
```

(a) Lang 39 (fix)

```
    stepSize = hNew;
-   if (forward) {
-       if (stepStart + stepSize >= t) {
-           stepSize = t - stepStart;
-        }
-    } else {
-       if (stepStart + stepSize <= t) {
-        stepSize = t - stepStart;
-       }
-  }
   ...
```

(b) Math 39 (bug inducing)

```
if (this.maxMiddleIndex >= 0) {
-   ... (this.maxMiddleIndex) ...
+   ... (this.minMiddleIndex) ...
    long maxMiddle = s + (e - s) / 2;
    if (middle > maxMiddle) {
    ...
```

(c) Chart 7 (bug inducing)

```
while (days < 0) {
-   end.add(Calendar.MONTH, -1);
-   days += end.getActualMaximum(Calendar.DAY_OF_MONTH);
+   days += 31;
    ...
```

(d) Lang 63 (bug inducing)

Figure 4.11: Example code changes in DEFECTS4J presented to the participants of the preliminary human study.

displaying the binary labelling question, as well as the code change. In addition, we also conducted an exit interview and questioned about their decision.

Overall, the developers correctly label 16 out of 20 code changes on average, which we interpret as better than a random guess despite the small sample size. While we need more thorough and larger scale human study, the preliminary results suggest that the similarities reported in algorithmic analyses (RQ1 to RQ3) are in fact more complicated and nuanced.

4.6.2 Discussion of Human Study Results

We categorise and summarise the results of our qualitative analysis as follows:

- **Clear fix patterns**: the developers easily identify the clear fix patterns such as adding a null pointer checker.

  "I'm sure this change is a fix commit. It adds a null checker." (see Figure 4.11a)

  "There's a new if statement that looks like a safeguard."

- **Addition vs. Deletion**: if there are only deletions in the given code change, they tend to think that it is a buggy commit. In case of only additions, however, they assume that it is a fix commit.

  "Honestly, I'm not sure what the code is meant to do, but, inserting a piece of new code seems like a fix."

  "Looks like the code should check an overflow of stepSize, but this change deletes all of that." (see Figure 4.11b)

- **Surrounding context**: although they do not have any prior knowledge about the given code, the developers always look at not only the change itself, but the surrounding context to understand the meaning of the change. They try to decode the change and find a continuity in use of variables and coding convention.

  "The surrounding if statement is for MaxMiddleIndex, so the change should've included some logic related to Max, not Min." (see Figure 4.11c)

- **Common mistakes**: they identify common mistakes based on their experience.

  "days += 31 does not make sense to me. I've written Datetime-related code many times, but adding 31 to days was always a mistake." (see Figure 4.11d)

  "(The bug was a mistake on writing a series of numbers) Order is wrong. 00E5 should be followed by 00E6."

In most cases, the developers fail to answer with confidence; they sometimes provide incorrect or unsupported reasons for their labelling, such as the type of modifications being additions or deletions only. Given that deleting code alone can produce plausible patches [116], we do not think this is a safe heuristic.

We suspect that the lack of confidence may be due to the fact that the participants are not particularly familiar with the PUTs. Had they been experts on these projects, they would have scored better. This hypothesis is partly supported by the fact that participants suggested surrounding contexts or common mistake patterns as the reason behind their labelling. We think these are specifically the qualities that algorithmic analyses of code change similarities currently lack. We leave the similarity analysis that can incorporate surrounding semantic context as future work.

4.7 Threats to Validity

Threats to internal validity concern any factor that may influence the observed effects. To mitigate such threats, we limit the APR and mutation tools we study to those that are publicly available and widely studied. We also limit any modification we introduce to the minimum.

Threats to external validity concern the degree to which our results can be generalised. We tried to incorporate as many datasets and programming languages as possible, by studying C and JavaScript (RQ1) as well as Java (RQ2 and 3). While our idea is not inherently confined to a specific programming language, only further experimentations can generalise our results to new tools and languages. We adopt DEFECTS4J as the standard benchmarks in both mutation testing and APR.

Threats to construct validity occur when the metrics we use fail to measure what we initially plan to observe. For RQ1, we simply count the number of clusters as there are no clear ground truths: we believe this is the simplest and the most direct measurement we can take. For RQ2, we only report the number of bugs for which subject tools can generate plausible patches. This is a simple count based metric that can be validated by test execution. For RQ3, we report the number of coupled pairs, following a widely accepted protocol.

4.8 Related Work

Weimer et al. [24] explored the duality between APR and mutation testing. Specifically, they formalised and characterised Generate & Validate (G&V) program repair as a dual of mutation testing: the equivalent mutant problem is related to the redundant repairs, while the coupling effect is related to the hypothesis that simple operators can fix many complex faults. Both assume the competent programmer hypothesis, meaning that APR also assumes that a relatively simple patch can repair the bug. This is what PraPR explicitly exploits using the mutation operators implemented in PIT [43]. In this work, we further investigate the approach taken by PraPR using TBar^{-1} and SequenceR^{-1}. Moreover, our work is not confined to APR and mutation testing, but also considers the similarities between human written patches and faults (RQ1). While Weimer et al. proposed the duality as a theoretical framework, we present empirical evidence of the relationship.

Brown et al. [117] proposed a mutant mining technique that essentially inverts fix changes mined from open source repositories. In one of their experiments, they ask whether 'forward' and 'backward' patches are different. The forward patches refer to the original fix changes, whereas backward patches refer to their inversions (i.e., faults). Interestingly, their results showed an overlap of 1,710 mined operators, out of 13,929 operators mined from both directions. However, all operators were mined from a single project, Space [118], limiting the scope of generalisation. We have conducted a larger empirical evaluation with multiple programs and languages. Our results also suggest that mining mutation operators only from one direction may miss some relevant code changes.

While APR techniques can successfully patch many faults, it is known that they also produce many incorrect changes during the process [108]. This partly motivates our use of APR tools as a source of code mutation. Recently proposed NMT based APR techniques seek to avoid the generation of incorrect and wasted patches by incorporating the surrounding contexts better [107, 105]. The qualitative analysis of our results for RQ2 and 3, as well as the results of the preliminary human study, also hints at the importance of contexts, calling for future work on ways to representing as well as comparing them.

4.9 Summary of this Chapter

This chapter aims to relate two seemingly opposite concepts in software testing, fixing (patch) and introducing bugs (fault). We highlight their syntactic similarities based on empirical evaluations. An analysis of 13k fix and bug inducing changes collected from open source repositories shows that, when abstracted and clustered together, it is difficult to distinguish patches from faults: up to 70% of the patches and faults are clustered together. Based on these results, we also show that mutation tools can be successfully used as APR tools, and vice versa. An evaluation using Defects4J bugs shows that mutation tools generate plausible patches for 42 and 17 bugs, only eight and two fewer than original APR tools, TBar and SequenceR, respectively. Finally, we also show that APR tools can successfully generate mutants that are coupled with real faults. Our findings suggest that the scope of code changes traditionally used to mine mutation operators, or to learn fix patterns and templates, may need to be widened to incorporate additional code changes that are relevant.

Chapter 5. Conclusion

5.1 Summary of Contributions

This thesis explores mutant's relationship with code, faults, and patches to improve the efficacy of mutation testing. In summary, the contributions of the thesis are as follows:

- We present a new predictive model named Seshat that can learn to predict the mutation results using static features extracted from the NL channel near the mutant. To the best of our knowledge, this is the first attempt to use NL features to predict the mutation result. We build a deep neural network to learn the mapping between the NL features and the kill results. The empirical results suggest that this model trained on the old version of the program successfully predicts the entire kill matrix of the current version with an average F-score of 0.83.

- We propose new statistical inference models called SIMFL that learn to predict the location of the faults using the similarities between the faults and mutants. In the CI context, SIMFL is designed to learn to predict the location of the mutants (faults) using the mutation results of previous version of the program. Our results show that the proposed models can localise 78% of the studied faults within the top five ranks. Also, we investigate a combination of SIMFL and Seshat where we make SIMFL learn on the predicted kill matrix by Seshat. Even in this case, SIMFL still localises 95 faults on the top out of 194 faults, supporting its practical applicability.

- We conduct large empirical studies to investigate the mutant's relationship with patches in terms of their syntactic similarities. The results show that up to 70% of the mutants and patches can be grouped together and even if we use the mutants as patches and vice versa, the performance of corresponding techniques (mutation or APR tools) still remains high. This implies that exploiting this relationship can widen the scope of both faults and patches, and further help to improve several testing techniques such as static bug finders, APR, and mutation tools.

5.1.1 Mutant's Relationship with Code

Mutation analysis can provide valuable insights into both System Under Test (SUT) and its test suite. However, it is not scalable due to the cost of building and testing a large number of mutants. Predictive Mutation Testing (PMT) has been proposed to reduce the cost of mutation testing, but it can only provide statistical inference about whether a mutant will be killed or not by the entire test suite. We propose Seshat, a Predictive Mutation Analysis (PMA) technique that can accurately predict the entire kill matrix, not just the mutation score of the given test suite. Seshat exploits the natural language channel in code, and learns the relationship between the syntactic and semantic concepts of each test case and the mutants it can kill, from a given kill matrix. The learnt model can later be used to predict the kill matrices for subsequent versions of the program, even after both the source and test code have changed significantly. Empirical evaluation using the programs in the Defects4J shows that Seshat can predict kill matrices with the average F-score of 0.83 for versions that are up to years apart. This is an improvement of F-score by 0.14 and 0.45 point over the state-of-the-art predictive mutation testing technique, and a simple coverage based heuristic, respectively. Seshat also performs as well as PMT for the prediction of the mutation score only. When applied to a Mutation Based Fault Localisation (MBFL) technique, the predicted kill matrix by Seshat is successfully used to locate faults within the top ten position, showing its usefulness beyond prediction of mutation scores. Once Seshat trains its model using a concrete mutation analysis, the subsequent predictions made by Seshat are on average 39 times faster than actual test-based analysis. We also show that Seshat can be successfully applied to automatically generated test cases with an experiment using EvoSuite.

5.1.2 Mutant's Relationship with Faults

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 before a failure is observed, allowing the amortisation of the analysis cost. 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 using DEFECTS4J shows that SIMFL can successfully localise up to 113 out of 203 studied faults (55%) at the top, and 159 (78%) faults within the top five, significantly outperforming existing MBFL techniques while using the results of mutation analysis that has been undertaken before the test failure. The amortised cost of mutation analysis can be further reduced by mutation sampling: SIMFL retains 80% of its localisation accuracy at the top rank when using only 10% of generated mutants, compared to results obtained without sampling.

5.1.3 Mutant's Relationship with Patches

Software bugs pose an ever-present concern for developers, and patching such bugs requires a considerable amount of costs through complex operations. In contrast, *introducing* bugs can be an effortless job, in that even a simple mutation can easily break the Program Under Test (PUT). Existing research has considered these two opposed activities largely separately, either trying to automatically generate realistic patches to help developers, or to find realistic bugs to simulate and prevent future defects. Despite the fundamental differences between them, however, we hypothesise that they do not syntactically differ from each other when considered simply as code changes. To examine this assumption systematically, we investigate the relationship between patches and buggy commits, both generated manually and automatically, using a clustering and pattern analysis. A large scale empirical evaluation reveals that up to 70% of patches and faults can be clustered together based on the similarity between their lexical patterns; further, 44% of the code changes can be abstracted into the identical change patterns. Moreover, we investigate whether code mutation tools can be used as Automated Program Repair (APR) tools, and APR tools as code mutation tools. In both cases, the inverted use of mutation and APR tools can perform surprisingly well, or even better, when compared to their original, intended uses. For example, 89% of patches found by SequenceR, a deep learning based APR tool, can also be found by its inversion, i.e., a model trained with faults and not patches. Similarly, real fault coupling study of mutants reveals that TBar, a template based APR tool, can generate 14% and 3% more fault couplings than traditional mutation tools, PIT and Major respectively, when used as a mutation tool. Our findings suggest that the valid scope of mining code changes for either mutation or APR can be wider than previously thought.

5.2 Summary of Future Work

Based on the findings and implications, we believe that several future work can be performed to further improve the efficacy of mutants:

5.2.1 Additional features for PMA (Chapter 2)

Since our focus was to investigate the NL channel of the mutants for PMA, we limited our model to use only those features. Most of them are token-based features such as the method name or mutated statement: they have not been considered as an information source for mutation testing because the mutation testing is in principle a dynamic analysis heavily relying on the test executions.

However, like PMT, we posit that dynamic features such as the number of test executions on the mutated statement would be effectively exploited for PMA and can be easily integrated into Seshat. Those dynamic features are based on the coverage information of the test suite, which is a prerequisite for mutation testing and thus can be utilised without any additional effort. Moreover, we can consider PMT's other static features based on the structure of the program such as McCabe Complexity [119] and the depth of nested blocks of the mutated statement. Plus, the other NL features such as variable names in the method body can also be considered. We believe that a hybrid model that uses Seshat's NL features and other features is expected to be a promising future direction.

5.2.2 A cross-project evaluation of PMA (Chapter 2)

A design principle of Seshat is based on the assumption that the test quality and coding convention are consistent between the versions. However, if someone intentionally removes the assertions in the tests or replaces the method names with arbitrary ones, Seshat will not be able to make correct predictions according to those manipulations. As we discussed in the previous subsection (Section 5.2.1), in that case we may need additional dynamic features to compensate for NL features since the dynamic features are sensitive to such manipulations.

In our evaluations, we checked whether Seshat can still perform well on the newly introduced tests (RQ3) and the machine-generated tests (RQ5). However, we believe that there is room for further investigation on the impact of such inconsistency between the trained program and tested program. For example, a cross-project study where training and testing are conducted on different programs can be considered for further investigation of the generalisability of Seshat.

5.2.3 Integrating into the bigger model (Chapter 3)

Recently, there have been several studies that combine all families of FL techniques into a single DNN model [120, 86]. For example, they collect the suspiciousness-based features from SBFL, MBFL, and Learning-to-Rank based FL techniques, and connect each feature group into the DNN model. Their empirical study has shown its promising localisation performance, suggesting that using various feature dimensions is highly effective. As one of the MBFL family, SIMFL can be easily integrated into such a big model. Moreover, SIMFL may compose a new feature dimension because the information source of SIMFL is a previous version of the program, which has not been exploited by any other MBFL techniques. As such, we think that using SIMFL as a part of the big DNN model is a promising future work.

5.2.4 Exploiting the widened scope of code changes (Chapter 4)

Our empirical study focuses on investigating the relationship between the mutants and patches. Specifically, at a code change level, we compare the syntactic similarity between them and conduct two application studies that show the interchangeability of mutants and patches.

More than using the mutants as patches or vice versa, our findings support that exploiting both at once can be beneficial. For example, in order to mine the real faults in the open source projects, most of the previous work had to rely on inverting the fix commits to make them into the faults. This is because finding bug-inducing commits is difficult in that the developers could not be aware of introducing a bug and it is likely to contain the changes that are not related to the bug [121, 122]. In contrast, finding fix commits is much easier as the developers intend to fix the bug with a commit message such as 'fixes the bug #3 and issue #12' or 'This commit fixes the broken behavior of the function foo'. With our findings, mining the real faults means mining other faults (inverted ones) at the same time. This is straightforward to implement as one needs to simply reverse the code changes to double the size of the existing dataset. Further application study that exploits this larger dataset would be a viable next research step. For example, we can evaluate the performance of a static bug finder that originally used a real faults dataset but now uses the larger one.

5.2.5 Strengthening the developer study (Chapter 4)

Although our preliminary human study was conducted on a small scale of five developers and 20 code changes, the qualitative analysis provides valuable insights into the developers' perception of the mutants and patches. For example, considering surrounding semantic contexts is important for them to understand the code changes. This opens up a new research direction to investigate the impact of the surrounding contexts of the faults and patches: even though our evaluation shows that they are syntactically similar but they may be semantically different. Investigation into how to distinguish them would be a promising future work. Also, we can redesign the questions given to the developers that have either fruitful or superficial information about the surrounding context. Also, further analysis on larger developers and code changes can be promising future work.

Bibliography

[1] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.

[2] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1:3–18, 1992.

[3] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 910–921. IEEE, 2021.

[4] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test., Verif. Reliab.*, 25(5-7):605–628, 2015.

[5] Jong-In Jang, Duksan Ryu, and Jongmoon Baik. Hotfuz: Cost-effective higher-order mutation-based fault localization. *Software Testing, Verification and Reliability*, page e1802, 2021.

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

[7] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45(9):898–918, 2018.

[8] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on cross-project predictive mutation testing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 160–171. IEEE, 2019.

[9] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. Predictive mutation analysis via natural language channel in source code. *ACM Trans. Softw. Eng. Methodol.*, 2021.

[10] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, ICST 2014, pages 153–162, 2014.

[11] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.

[12] Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45 – 60, 2014.

[13] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE NIER 2020, pages 25–28, 2020.

[14] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996.

[15] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 351–360, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 235–245, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Bob Kurtz, Paul Ammann, Marcio E Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185. IEEE, 2014.

[18] Lingchao Chen and Lingming Zhang. Speeding up mutation testing via regression test selection: An extensive study. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–69. IEEE, 2018.

[19] A. Jefferson Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20:337–344, 1994.

[20] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707, August 2016.

[21] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2009, pages 88–99, November 2009.

[22] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 244–258. Springer Berlin Heidelberg, 2012.

[23] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Museum: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology*, 82:80–95, 2017.

[24] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366, 2013.

[25] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[26] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of 1993 15th international conference on software engineering*, pages 100–107. IEEE, 1993.

[27] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. Association for Computing Machinery.

[29] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436, 2014.

[30] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, 1982.

[31] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In Thomas Ostrand and Elaine Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, 1993.

[32] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130, 2010.

[33] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, 2015.

[34] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, May 1984.

[35] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[36] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.

[37] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[38] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[39] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 22(2):67–120, March 2012.

[40] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 464–475, 2015.

[41] Jinhan Kim, Gabin An, Robert Feldt, and Shin Yoo. Ahead of time mutation based fault localisation using statistical inference. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 253–263, 2021.

[42] Y. Kim, S. Hong, B. Ko, D. L. Phan, and M. Kim. Invasive software testing: Mutating target programs to diversify test exploration for high test coverage. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 239–249, April 2018.

[43] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30, 2019.

[44] Andrew Trenk. Testing on the toilet: Writing descriptive test names. https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html. Accessed: 2020-08-28.

[45] Rafael Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big Code != Big Vocabulary: Open-Vocabulary Models for Source code. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20. ACM, 2020.

[46] Michael Hucka. Spiral: splitters for identifiers in source code files. *Journal of Open Source Software*, 3(24):653, 2018.

[47] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[48] Shuohang Wang and Jing Jiang. A compare-aggregate model for matching text sequences. *arXiv preprint arXiv:1611.01747*, 2016.

[49] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. Cc2vec: Distributed representations of code changes. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20. ACM, 2020.

[50] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, February 2013.

[51] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 57–67, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, June 2002.

[53] Haibo He, Yang Bai, E. A. Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1322–1328, June 2008.

[54] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 331–341, 2012.

[55] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. *Software Testing, Verification, and Reliability*, 15(4):257–279, 2005.

[56] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[57] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 295–306, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[59] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, 2014.

[60] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[61] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 214–224, 2015.

[62] David Schuler and Andreas Zeller. Checked coverage: an indicator for oracle quality. *Software testing, verification and reliability*, 23(7):531–551, 2013.

[63] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery.

[65] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 304–315. IEEE Press, 2019.

[66] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localisation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, 2017.

[67] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Simultaneous debugging of software faults. *J. Syst. Softw.*, 84(4):573–586, April 2011.

[68] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology*, 20(3):11:1–11:32, August 2011.

[69] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 26(1):4:1–4:30, July 2017.

[70] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 449–456, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 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, pages 191–201, New York, NY, USA, 2013. ACM.

[72] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*, pages 364–374, Vancouver, Canada, 16-24 May 2009. IEEE.

[73] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. ACM.

[74] M. Papadakis and Y. Le-Traon. Using mutants to locate "unknown" faults. In *Proceedings of the 5th IEEE Fifth International Conference on Software Testing, Verification and Validation*, Mutation 2012, pages 691–700, 2012.

[75] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.

[76] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[77] Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 237–249, 2020.

[78] René Just, Darioush Jalali, and Michael D. Ernst. 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, pages 437–440, New York, NY, USA, 2014. ACM.

[79] R. Just, F. Schweiggert, and G. M. Kapfhammer. 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)*, pages 612–615, Nov 2011.

[80] René Just, Gregory M Kapfhammer, and Franz Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 720–725. IEEE, 2012.

[81] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE seventh international conference on software testing, verification and validation*, pages 21–30. IEEE, 2014.

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

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

[84] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 688–698, New York, NY, USA, 2018. ACM.

[85] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211, Nov 2015.

[86] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 2019.

[87] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 354–365, 2016.

[88] Jifeng Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME 2014, pages 191–200, Sept 2014.

[89] Gulsher Laghari and Serge Demeyer. 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, pages 280–281, New York, NY, USA, 2018. ACM.

[90] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):92, 2017.

[91] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[92] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.

[93] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36, 2011.

[94] Claire Le Goues. Automatic program repair using genetic programming. *named-content content-type= ref-degree¿ Ph. D. dissertation¡/named-content¿,¡ institution content-type= institution¿ Faculty School Eng. Appl. Sci.¡/institution¿,¡ institution content-type= institution¿ Univ. Virginia¡/institution¿, Charlottesville, VA, USA*, 2013.

[95] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Ibir: Bug report driven fault injection. *arXiv preprint arXiv:2012.06506*, 2020.

[96] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019.

[97] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[98] Anil Koyuncu, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Flexirepair: Transparent program repair with generic patches. *arXiv preprint arXiv:2011.13280*, 2020.

[99] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 906–918, 2021.

[100] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020.

[101] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. Spinfer: Inferring semantic patches for the linux kernel. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 235–248, 2020.

[102] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[103] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 601–614, 2018.

[104] Seemanta Saha et al. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2019.

[105] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[106] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE, 2017.

[107] Thibaud Lutellier, Viet Hung Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[108] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[109] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2018.

[110] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309, 2018.

[111] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[112] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.

[113] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th international conference on software engineering*, pages 12–23, 2018.

[114] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[115] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering*, 44(4):308–333, 2017.

[116] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[117] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522, 2017.

[118] W Eric Wong, Joseph Robert Horgan, Aditya P Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 522–528. IEEE, 1997.

[119] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[120] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

[121] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 326–337, 2019.

[122] Gabin An and Shin Yoo. Reducing the search space of bug inducing commits using failure coverage. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE (Ideas, Visions, and Reflections Track), 2021.