



PhD-FSTC-2019-62
The Faculty of Science, Technology and Communication

DISSERTATION

Defense held on the 13th September 2019 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Thierry TITCHEU CHEKAM
Born on 22nd September 1988 in Yaoundé (Cameroon)

ASSESSMENT AND IMPROVEMENT OF THE PRACTICAL USE OF MUTATION FOR AUTOMATED SOFTWARE TESTING

Dissertation Defense Committee

Dr. Yves LE TRAON, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Jacques KLEIN, Chairman
Assistant Professor, University of Luxembourg, Luxembourg

Dr. Mike PAPADAKIS, Vice Chairman
Research Scientist, University of Luxembourg, Luxembourg

Dr. Serge DEMEYER
Professor, University of Antwerp, Belgium

Dr. Paolo TONELLA
Professor, Università della Svizzera Italiana, Switzerland

To my Lord Jesus Christ and my family.

ABSTRACT

Software testing is the main quality assurance technique used in software engineering. In fact, companies that develop software and open-source communities alike actively integrate testing into their software development life cycle. In order to guide and give objectives for the software testing process, researchers have designed test adequacy criteria (TAC) which, define the properties of a software that must be covered in order to constitute a thorough test suite. Many TACs have been designed in the literature, among which, the widely used statement and branch TAC, as well as the fault-based TAC named mutation. It has been shown in the literature that mutation is effective at revealing fault in software, nevertheless, mutation adoption in practice is still lagging due to its cost.

Ideally, TACs that are most likely to lead to higher fault revelation are desired for testing and, the fault-revelation of test suites is expected to increase as their coverage of TACs test objectives increase. However, the question of which TAC best guides software testing towards fault revelation remains controversial and open, and, the relationship between TACs test objectives' coverage and fault-revelation remains unknown. In order to increase knowledge and provide answers about these issues, we conducted, in this dissertation, an empirical study that evaluates the relationship between test objectives' coverage and fault-revelation for four TACs (statement, branch coverage and, weak and strong mutation). The study showed that fault-revelation increase with coverage only beyond some coverage threshold and, strong mutation TAC has highest fault revelation.

Despite the benefit of higher fault-revelation that strong mutation TAC provide for software testing, software practitioners are still reluctant to integrate strong mutation into their software testing activities. This happens mainly because of the high cost of mutation analysis, which is related to the large number of mutants and the limitation in the automation of test generation for strong mutation.

Several approaches have been proposed, in the literature, to tackle the analysis' cost issue of strong mutation. Mutant selection (reduction) approaches aim to reduce the number of mutants used for testing by selecting a small subset of mutation operator to apply during mutants generation, thus, reducing the number of analyzed mutants. Nevertheless, those approaches are not more effective, w.r.t. fault-revelation, than random mutant sampling (which leads to a high loss in fault revelation). Moreover, there is not much work in the literature that regards cost-effective automated test generation for strong mutation. This dissertation proposes two techniques, *FaRM* and *SEMu*, to reduce the cost of mutation testing. *FaRM* statically selects and prioritizes mutants that lead to faults (fault-revealing mutants), in order to reduce the number of mutants (fault-revealing mutants represent a very small proportion of the generated mutants). *SEMu* automatically generates tests that strongly kill mutants and thus, increase the mutation score and improve the test suites.

First, this dissertation makes an empirical study that evaluates the fault-revelation (ability to lead to tests that have high fault-revelation) of four TACs, namely statement, branch, weak mutation and strong mutation. The outcome of the study show evidence that for all four studied TACs, the fault-revelation increases with TAC test objectives' coverage only beyond a certain threshold of coverage. This suggests the need to attain higher coverage during testing. Moreover, the study shows that strong mutation is the only studied TAC that leads to tests that have, significantly,

the highest fault-revelation.

Second, in line with mutant reduction, we study the different mutant quality indicators (used to qualify "useful" mutants) proposed in the literature, including fault-revealing mutants. Our study shows that there is a large disagreement between the indicators suggesting that the fault-revealing mutant set is unique and differs from other mutant sets. Thus, given that testing aims to reveal faults, one should directly target fault-revealing mutants for mutant reduction. We also do so in this dissertation.

Third, this dissertation proposes *FaRM*, a mutant reduction technique based on supervised machine learning. In order to automatically discriminate, before test execution, between useful (valuable) and useless mutants, *FaRM* build a mutants classification machine learning model. The features for the classification model are static program features of mutants categorized as mutant types and mutant context (abstract syntax tree, control flow graph and data/control dependency information). *FaRM*'s classification model successfully predicted fault-revealing mutants and killable mutants. Then, in order to reduce the number of analyzed mutants, *FaRM* selects and prioritizes fault-revealing mutants based of the aforementioned mutants classification model. An empirical evaluation shows that *FaRM* outperforms (w.r.t. the accuracy of fault-revealing mutant selection) random mutants sampling and existing mutation operators-based mutant selection techniques.

Fourth, this dissertation proposes *SEMu*, an automated test input generation technique aiming to increase strong mutation coverage score of test suites. *SEMu* is based on symbolic execution and leverages multiple cost reduction heuristics for the symbolic execution. An empirical evaluation shows that, for limited time budget, the *SEMu* generates tests that successfully increase strong mutation coverage score and, kill more mutants than test generated by state-of-the-art techniques.

Finally, this dissertation proposes *Muteria* a framework that enables the integration of *FaRM* and *SEMu* into the automated software testing process.

Overall, this dissertation provides insights on how to effectively use TACs to test software, shows that strong mutation is the most effective TAC for software testing. It also provides techniques that effectively facilitate the practical use of strong mutation and, an extensive tooling to support the proposed techniques while enabling their extensions for the practical adoption of strong mutation in software testing.



ACKNOWLEDGEMENTS

Throughout this thesis, I have received a great deal of support and assistance.

First of all, I am deeply grateful to my supervisor, Prof. Yves Le Traon, for his kindness and support throughout my thesis. He trusted in my research and gave me valuable feedback for this dissertation. I am also grateful for the opportunity he gave me to be involved in a few teaching activities that taught me many things.

I am equally grateful to my daily supervisor, Dr. Mike Papadakis, for his patience, and his advice, training, guidance, and encouragement. He helped me to have a good direction for my thesis and showed me how to perform research and present to others. The frequent discussions we had helped me to get a deeper understanding of software testing.

Of all the people involved in my thesis, I am thankful to my co-authors for their efforts and feedbacks that contributed to making this thesis stronger. I also want to thank all jury members for their interest in my research and the time invested for my dissertation.

I am thankful to the National Research Fund (FNR) of Luxembourg for trusting in the initial ideas that lead to this dissertation by granting the fundings that supported my thesis.

I would like to express my gratitude to all my colleagues from SERVAL (SnT) for all the good discussions we had and the interesting reading group sessions.

Finally and more personally, I thank my wife, Mrs. Titchou Videline, for her love, constant support and patience during my thesis. I also thank both my parents and my brothers and sisters for their constant encouragement and help. They have been encouraging me ever since I can remember, and getting to this level of studies has a lot to do with their love and kindness. I am particularly thankful to my Lord and Savior Jesus Christ who gave me life, peace, health and strength to start and finish this thesis with joy all through.

Thierry Titchou Chekam

Luxembourg, Luxembourg, August 2019



CONTENTS

Abstract	v
Contents	viii
1 Introduction	1
1.1 Context	2
1.1.1 Software Testing and Mutation Testing	2
1.1.2 State of Mutation Testing in Research and Adoption in Practice	3
1.2 Challenges of Mutation Testing	4
1.2.1 The added Value of Mutation	4
1.2.2 The Large Number of Mutants	5
1.2.3 Mutation Testing Test Automation	5
1.3 Overview of the Contribution and Organization of the Dissertation	6
1.3.1 Contributions	6
1.3.2 Organization of the Dissertation	8
2 Technical Background and Definitions	11
2.1 Test Adequacy Criteria-based Software Testing	12
2.2 Mutation Testing	12
2.2.1 General Information	12
2.2.2 Mutant Killing	13
2.2.3 Some Definitions	14
2.2.4 Mutant Quality Indicators	14
2.3 Symbolic Execution	14
2.4 Supervised Machine Learning	15
2.4.1 Binary Classification problem	16
2.4.2 Some Classification Models	16
2.4.3 Evaluation techniques	18
2.4.4 Performance evaluation	19
2.5 Additional Definitions	20
2.5.1 Average Percentage of Fault Detected	20
2.5.2 Statistical Test	20
2.5.3 Effect Size	21
2.6 Summary	21
3 Related Work	23
3.1 Evaluation of Testing Criteria	25
3.2 Mutant Selection and Prioritization	25

3.2.1	Useful Mutants	26
3.2.2	Static Mutant Selection	26
3.2.3	Approaches Based on Static and Dynamic Ananlysis	27
3.2.4	Machine learning based approaches	27
3.2.5	Mutant Reduction Summary	29
3.3	Automated Tests Input Generation For Mutation Testing	29
3.4	Summary	31
4	An Empirical Evaluation of Test Adequacy Criteria	33
4.1	Introduction	34
4.2	Test Adequacy Criteria	35
4.2.1	Statement and Branch Adequacy Criteria	36
4.2.2	Mutation-Based Adequacy Criteria	36
4.3	Research Questions	36
4.4	Research Protocol	37
4.4.1	Programs Used	38
4.4.2	CoREBench: realistic, complex faults	38
4.4.3	Test Suites Used	39
4.4.4	Tools for Mutation Testing and Coverage Measurement	40
4.4.5	Analyses Performed on the Test Suites	41
4.5	Experimental Results	43
4.5.1	RQ1: Clean Program Assumption	43
4.5.2	RQ2: Fault revelation at higher levels of coverage	44
4.5.3	RQ3: Fault Revelation of Statement, Branch, Weak and Strong Mutation	45
4.6	Threats to Validity	47
4.7	Conclusions	48
5	Mutant Quality Indicators	51
5.1	Introduction	52
5.2	Mutant Quality Indicators	53
5.2.1	Unit-based MQIs	53
5.2.2	Set-based MQIs	54
5.3	Experiment Setup	54
5.3.1	Programs and Faults	54
5.3.2	Automated Tools	55
5.3.3	Experimental Procedure	56
5.4	Results	56
5.4.1	Prevalence of mutant quality indicator categories	56
5.4.2	Relations between mutant quality indicators	56
5.4.3	Mutant types and quality indicators	58
5.4.4	Fault classes with no fault revealing mutants	59
5.4.5	Links between mutant types and fault classes	60
5.5	Conclusion	60
6	Selecting Fault Revealing Mutants	65
6.1	Introduction	67
6.2	Context	69

6.2.1	Problem Definition	69
6.2.2	Mutant Selection	71
6.2.3	Mutant Prioritization	72
6.3	Approach	72
6.3.1	Implementation	75
6.3.2	Demonstrating Example	75
6.4	Research Questions	77
6.5	Experimental Setup	79
6.5.1	Benchmarks: Programs and Fault(s)	79
6.5.2	Automated Tools Used	81
6.5.3	Experimental Procedure	82
6.5.4	Mutant Selection and Effort Metrics	83
6.6	Results	84
6.6.1	Assessment of killable mutant prediction (RQ1 and RQ2)	84
6.6.2	Assessment of fault revelation prediction	85
6.6.3	Mutant selection	87
6.6.4	Mutant prioritization	90
6.6.5	Experiments with large programs (RQ7)	96
6.7	Discussion	100
6.7.1	Working Assumptions	100
6.7.2	Threats to Validity	101
6.7.3	Representativeness of test subjects	102
6.7.4	Redundancy between the considered faults	103
6.7.5	Other Attempts	104
6.8	Conclusions	105
7	Killing Stubborn Mutants Via Symbolic Execution	107
7.1	Introduction	109
7.2	Context	111
7.2.1	Symbolic Encoding of Programs	111
7.2.2	Symbolic Encoding of Mutants	112
7.2.3	Example	112
7.3	Symbolic Execution	113
7.4	Killing Mutants	114
7.4.1	Exhaustive Exploration	114
7.4.2	Conservative Pruning of the Search Space	115
7.4.3	Heuristic Search	116
7.5	<i>SEMu</i> Cost-Control Heuristics	117
7.5.1	Pre Mutation Point: Controlling for Reachability	117
7.5.2	Post Mutation Point: Controlling for Propagation	118
7.5.3	Controlling the Cost of Constraint Solving	119
7.5.4	Controlling the Number of Attempts	119
7.6	Empirical Evaluation	119
7.6.1	Research Questions	119
7.6.2	Test Subjects	120
7.6.3	Employed Tools	121
7.6.4	Experimental Setup	121

7.6.5	Experimental Settings and Procedure	123
7.6.6	Threats to Validity	124
7.7	Empirical Results	124
7.7.1	Killing ability of <i>SEMu</i>	124
7.7.2	Comparing <i>SEMu</i> with KLEE	125
7.7.3	Comparing <i>SEMu</i> with infection-only	125
7.8	Conclusion	127
8	Built Tools And Frameworks	129
8.1	<i>Mart</i> : A Mutant Generation tool for LLVM Bitcode	130
8.1.1	Overview	130
8.1.2	<i>Mart</i> Mutants Generation	130
8.1.3	Implementation and Usage	135
8.2	<i>Muteria</i> : An Extensible and Flexible Multi-Criteria Software Analysis Framework	136
8.2.1	Overview	136
8.2.2	Background and Motivation	137
8.2.3	<i>Muteria</i> Framework Overview	137
8.2.4	Case Study	141
8.2.5	Related Tools	142
8.3	Summary	142
9	Conclusion	143
9.1	Summary	144
9.2	Future Research Directions	145
	Bibliography	149



LIST OF ABBREVIATIONS

APFD Average Percentage of Faults Detected

AUC Area Under Curve

MQI Mutant Quality Indicator

MS Mutation Score

PUT Program Under Test

ROC Receiver Operation Characteristic

RQ Research Question

SDL Statement Deletion

TAC Test Adequacy Criterion

TCE Trivial Compiler Equivalence

LIST OF FIGURES

1.1	Process of Test Adequacy Criteria-based Software testing (adapted from <i>Introduction to Software Testing</i> [AO16])	3
1.2	Issues addressed in this dissertation.	4
1.3	Organization of the dissertation.	8
2.1	Program Mutation.	13
2.2	Symbolic execution and test input generation.	16
2.3	Supervised machine learning binary classification workflow. The top sub-figure illustrates the training phase and the bottom sub-figure illustrates the testing phase (class prediction).	17
4.1	The test pool with overall coverage score values.	40
4.2	RQ1: Comparing the “Faulty” with the “Clean” (‘Fixed’) programs. Our results show that there is statistically significant difference between the coverage values attained in the “Faulty” and “Clean” programs (subfigure 4.2a) with effect sizes that can be significant (subfigure 4.2b).	41
4.3	Fault coupling in the ‘Faulty’ and ‘Clean’ versions.	45
4.4	Fault Revelation of the studied criteria for the highest 5% coverage threshold and test suite size of 7.5% of the test pool.	46
4.5	Fault coupling between the studied criteria.	47
5.1	Relations between different mutant quality indicators.	61
5.2	Types of mutants involved in the mutant quality indicator categories.	62
5.3	Ratio of mutants involved in quality indicator categories per mutant type.	63
5.4	Ratio of faulty versions with fault revealing mutants (among all faults of the same type) per fault class and mutant type.	64
5.5	Faulty versions (see Table 5.1) without fault revealing mutants (ratios)	64
6.1	Fault revealing mutant selection. Contrast between sufficient mutant set selection and fault revealing mutant selection. Sufficient mutant set selection aims at selecting a minimal subset of mutants that is killed by tests that also kill the whole set of mutants. Fault revealing mutant selection aims at selecting a minimal subset of mutants that is killed by tests that reveal the same underlying faults as the tests that kill the whole set of mutants.	69

6.2	Overview of the <i>FaRM</i> approach. Initially, <i>FaRM</i> applies supervised learning on the mutants generated from a corpus of faulty program versions, and builds a prediction model that learns the fault revealing mutant characteristics. This model is then used to predict the mutants that should be used to test other program versions. This means that at the time of testing and prior to any mutant execution, testers can use and focus only on the most important mutants.	70
6.3	Example of mutant ranking procedure by <i>FaRM*</i> . the ranking is a concatenation of the ranked <i>predicted killable</i> mutants and the ranked <i>predicted equivalent</i> mutants.	75
6.4	Example program where mutation is applied. The C language comments on each line show the number of mutants generated on the line.	78
6.5	(a) An example of mutant <i>M</i> from the example program from Figure 6.4, (b) the abstract syntax tree of the mutated statement and (c) the control flow graph of the function containing the mutated statement.	78
6.6	Distribution of Codeflaws Benchmark problems by number of implementations.	80
6.7	Distribution of Codeflaws Benchmark faulty programs by number of lines of code changed to fix the fault.	81
6.8	Receiver Operating Characteristic For Killable Mutants Prediction on Codeflaws	85
6.9	Receiver Operating Characteristic For Fault Revealing Mutants Prediction on Codeflaws	86
6.10	Information Gain distributions of ML features on Codeflaws	86
6.11	Fault revelation of the mutant selection strategies on Codeflaws. All three <i>FaRM</i> and <i>FaRM*</i> sets outperform the random baselines.	88
6.12	Proportion of SDL and E-SELECTIVE mutants among all mutants for Codeflaws subjects.	88
6.13	Fault revelation of <i>FaRM</i> compared with SDL on Codeflaws. <i>FaRM</i> sets outperform the SDL selection. Approximately 2% (number of SDL mutants) of all the mutants are selected.	89
6.14	Fault revelation of <i>FaRM</i> compared with E-Selective on Codeflaws. Approximately 38% (number of E-Selective mutants) of all the mutants are selected.	90
6.15	Fault revelation of <i>FaRM</i> compared with E-Selective for selection size 5% of all mutants. <i>FaRM</i> and <i>FaRM*</i> sets outperform E-Selective selection.	90
6.16	Fault revelation of <i>FaRM</i> compared with E-Selective for selection size 15% of all mutants. <i>FaRM</i> sets outperform E-Selective selection.	90
6.17	APFD measurements considering all mutants for the selected mutants cost metric for Codeflaws. The <i>FaRM</i> prioritization outperform the random baselines.	91
6.18	APFD measurements considering only killable mutants for the selected mutants cost metric on Codeflaws. The <i>FaRM</i> prioritization outperform the random baselines, independent of non-killable mutants.	92
6.19	Mutant prioritization performance in terms of faults revealed (median case) for the selected mutants cost metric on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies.	92
6.20	Execution cost of prioritization schemes	93
6.21	APFD measurements for the required tests cost metric on Codeflaws. The <i>FaRM</i> prioritization outperform the random baselines.	94

6.22	Required tests prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies.	94
6.23	APFD measurements considering all mutants. The <i>FaRM</i> prioritization outperform the defect prediction.	95
6.24	Mutant prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies.	96
6.25	<i>FaRM</i> performance in terms of faults revealed (median case) on CoREBench considering all mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies.	97
6.26	<i>FaRM</i> performance in terms of faults revealed (median case) on CoREBench considering only killable mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies.	97
6.27	Mutation score (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the mutation score attained by the strategies.	98
6.28	Subsuming Mutation score (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the subsuming mutation score attained by the strategies.	98
6.29	Ratio of equivalents (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the proportion of equivalent mutants selected by the strategies.	99
6.30	APFD measurements on CoREBench for the required tests cost metric. The <i>FaRM</i> prioritization outperform the random baselines.	99
6.31	Required tests prioritization performance in terms of faults revealed (median case) on CoREBench. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies.	100
6.32	Correlations between mutants and faults in three defect datasets. Similar correlations are observed in all three cases suggesting that Codeflaws provides good indications on the fault revealing ability of the mutants.	102
6.33	CoREBench results on similar (repeatedIDs) and dissimilar (Non-RepeatedIDs) implementation. We observe similar trend in both cases suggesting a minor or no influence of code similarity on <i>FaRM</i> performance.	103
7.1	Example. The rounded control locations represent conditionals (at least 2 possible transition from them).	113
7.2	Example of Symbolic execution for mutant test generation. After control location 9, the symbolic execution on the original program contains transition $9 \rightarrow 10$ with $n = x$ while the symbolic execution of the mutant M_2 contains transition $9 \rightarrow 10$ with $n = x + 1$	115

7.3	Illustration of <i>SEMu</i> cost-control parameters. Subfigure (a) illustrates the Precondition Length where the green subtree represents the candidate paths constrained by the precondition (the thick green path prefix is explored using seeded symbolic execution). Subfigure (b) illustrates the Checkpoint Window (here CW is 2). Subfigure (c) illustrates the Propagation Proportion (here PP is 0.5) and the Minimum Propagation Depth (here if MPD is 1 the first test is generated, for unterminated paths, from <i>Checkpoint 1</i>).	117
7.4	Size of the test subjects.	120
7.5	Comparing the stubborn mutant killing ability of <i>SEMu</i> , KLEE and the <i>infection-only</i>	125
7.6	Comparing the mutant killing ability of <i>SEMu</i> and KLEE in per program basis. .	126
7.7	Comparing the mutant killing ability of <i>SEMu</i> and <i>infection-only</i> in per program basis.	126
8.1	LLVM bitcode mutation process of <i>Mart</i> . The rounded edges rectangles with double border lines represent LLVM bitcode files. The square edge rectangles represent the steps of the mutation process. Each step is implemented by a component of <i>Mart</i>	131
8.2	Example of bitcode mutation by <i>Mart</i> . Sub-figure (a) is an example of a mutation operators configuration description in a simple description language. Sub-figure (b) illustrates an example of code mutation; the second <i>fragment</i> in the original code is replaced by a mutant <i>fragment</i> . Sub-figure (c) presents an example of the mutation using the configuration of (a).	132
8.3	Mutant operators description language syntax diagram	135
8.4	Test Adequacy Criteria (TAC) based software analysis process (adapted from Offut's "Two mutation processes" [Off11]). The <i>Process 1</i> is adapted from the "Traditional process" and <i>Process 2</i> from the "Post-Mothra Process"	138
8.5	Architecture of <i>Muteria</i> framework. The components with black rectangle provide interfaces for corresponding tools to connect to the framework. The controller enable the integration. All components are accessible by the users through the framework API.	140
8.6	report of software analysis with <i>Muteria</i>	141

LIST OF TABLES

2.1	Confusion Matrix	18
3.1	Summary of previous studies on the relationship of test criteria and faults.	24
4.1	The subject programs used in the experiments. For each of them, the number of test cases (TC), their size in lines of code and number of considered faults are presented.	38
4.2	The influence of coverage thresholds on fault revelation for test suite size 7.5% of the test pool. All the coverage levels below the highest 20% are not significant. Sub-table (a) records fault revelation at highest x% coverage levels and sub-table (b) the results of a comparison of the form “rand” (randomly selected test suites) VS “highest x%” (test suites achieving the highest x% of coverage), e.g., for Branch and highest 20% the \hat{A}_{12} suggests that Branch provides a higher fault revelation in its last 20% coverage levels in 53% of the cases with average fault revelation difference of 1.4%.	43
4.3	Comparing fault revelation for the highest 5% coverage threshold and test suite size of 7.5% of the test pool.	46
5.1	Fault Classes	55
5.2	Prevalence of mutant categories.	57
6.1	Description of the static code features	74
8.1	Mutant Types	133
8.2	<i>Muteria</i> in Practice: Some drivers sizes (Python LOC)	141
8.3	<i>Muteria</i> in Practice: Some User Configurations	141

INTRODUCTION

This Chapter presents the context and challenges of this dissertation. First, the general principles of software testing and mutation testing in particular are set, then, the challenges of using mutation testing in practice, addressed in this dissertation, are presented.

Chapter content

1.1	Context	2
1.1.1	Software Testing and Mutation Testing	2
1.1.2	State of Mutation Testing in Research and Adoption in Practice	3
1.2	Challenges of Mutation Testing	4
1.2.1	The added Value of Mutation	4
1.2.2	The Large Number of Mutants	5
1.2.3	Mutation Testing Test Automation	5
1.3	Overview of the Contribution and Organization of the Dissertation	6
1.3.1	Contributions	6
1.3.2	Organization of the Dissertation	8

1.1 Context

This dissertation focuses on software functional white-box testing. In white-box testing, the internal structures of the program under test are available and used to design the tests. Functional testing verifies specific functionalities of the programs under test.

1.1.1 Software Testing and Mutation Testing

Software testing. Similarly to most engineering fields, software engineering makes use of quality control practices to guarantee the quality of developed software. The most widely used software quality control activity is software testing. The main goals of software testing are to find faults in the developed software and give confidence on the correctness of the developed software.

Software testing involves, among others, the creation of tests (test suite), the execution of the created tests against the developed software, and the observation of the program behavior after the tests' execution in order to determine its correctness. One key part of software testing is the creation (design) of the test suites. The quality of software testing depends on the quality of the created test suites.

Many studies suggest using coverage concepts, such as statement coverage, for testing [Gli+13; GJG14; IH14; And+06; FI98a]. These coverages concepts are criteria (termed *test adequacy criteria* (TAC)) used to measure the quality of test suites and guide testers to design new tests (by targeting uncovered elements or test objectives).

The general process followed when using TACs for software testing is illustrated in Figure 1.1. First, The targeted software (program) component is selected and tests are generated to exercise it. Then, the tests are executed on the program and the TAC coverage of the tests is measured. If the expected coverage level is not reached, the process execution jump back to the test generation phase to create additional tests and cover the uncovered test objectives. This is repeated until the pre-defined coverage threshold is reached. When the pre-defined threshold is reached, the program is checked for functional correctness and, if the tests reveal faults in the program (tests "fail"), the faults are fixed (program repaired) and the process is restarted. The whole process described so far is repeated until the tests do not reveal any fault. When the tests do not reveal faults or the time budget allocated has elapsed, the process ends and the software under test is considered tested. We note that this process is the standard way of using TACs to improve the test suites of software. However, recently, TACs are also used in different ways, as presented by Petrovic *et al.* [Pet+18], where TAC test objectives are used, for instance, to help developers understand the code. In this dissertation we focus on the traditional use of TACs.

Most TACs are based on the structure of the program. A few examples are statement, branch, block, function, and paths coverage. However, in 1971, Richard Lipton originally proposed a different TAC based on artificial faults and named *Mutation* [OU01]. Industrial studies show that software practitioners find mutation useful to uncover faults in their software [SW09; PI18; Pet+18].

Mutation testing. Mutation testing refers to the use of Mutation as TAC for software testing. Mutation attempts to simulate real faults by inserting artificial faults into a program under test. Those faults are simple syntactic alterations of the program syntax derived from a predefined

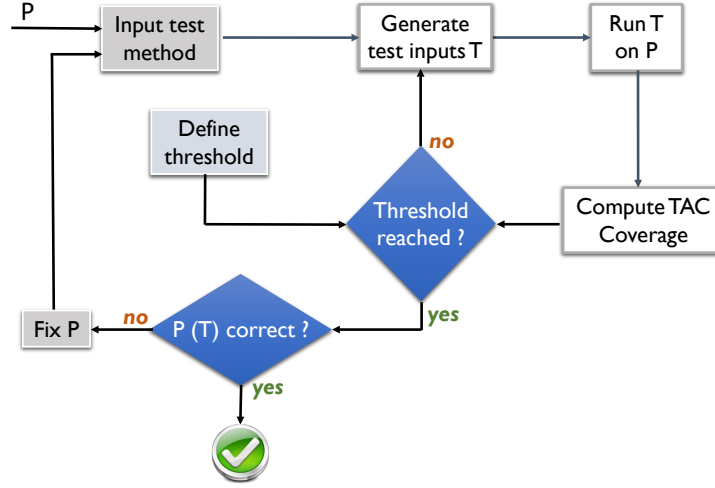


Figure 1.1: *Process of Test Adequacy Criteria-based Software testing (adapted from Introduction to Software Testing [AO16])*

set of rules called *mutation operators*. An example of mutation operator transforms the addition operation into subtraction operation (In a program, a statement such as $sum = a + b$ will be mutated into $sum = a - b$). The application of a mutant operator on a compatible code element results in a new program called *mutant* (mutant of the program under test; the program under tests is also called *original program*). Mutants are the test objectives for Mutation TAC. In order to create mutants used to test a program, a predefined set of mutation operators are applied throughout the code of the program under test to create multiple mutants. The creation of mutants is performed, in an automated manner, by a mutant generation tool, which is a software that inputs a program and applies a set of mutation operators to create mutants of the input program.

The mutation, as previously described is called *first-order mutation* and the mutants are called *first order mutants*. When two or more simples syntactic changes are simultaneously induced into the program under test, the mutation is called *higher order mutation* and the mutants are called a *higher order mutants*. In this dissertation, we focus on *first order mutants* and use the terms *mutant* and *mutation* to refer to *first order mutant* and *first order mutation*, respectively.

When a test execution on the original program differs (at the output) from the test execution on a mutant program, we say that the mutant is *killed* by the test (the test objective is covered). Otherwise, we say that the mutant *survives* the test. Similarly, when a test execution creates a difference in program state between the original program and a mutant right after the execution of the mutated statement, we say that the mutant infects the program state under that test execution.

1.1.2 State of Mutation Testing in Research and Adoption in Practice

Mutation testing has gained a lot of attention from the software engineering and testing research community [Pap+19; JH11]. There is an increasing number of research papers related to mutation and many address issues related to the adoption of mutation testing in practice (by software development companies) [PI18; Pet+18; Del+18]. However, the software industry is somehow slow to adopt mutation testing into the testing process. This is partially due to (1) the lack of

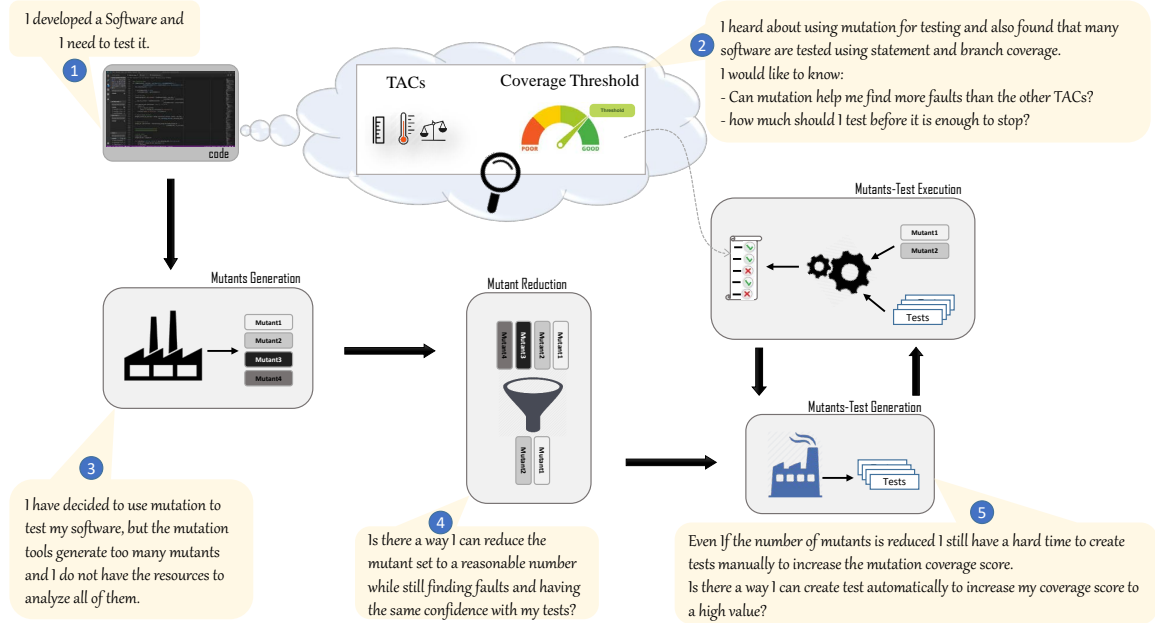


Figure 1.2: Issues addressed in this dissertation.

familiarity with mutation testing and the lack of effective, fully integrated mutation toolkits; (2) the satisfaction with popular and less effective test adequacy criteria; and (3) the challenges of using mutation testing in practice [SW09; MR16].

1.2 Challenges of Mutation Testing

There are several issues that hinder the adoption of mutation in practice, most of which result from the challenges of using mutation testing in practice. This section presents the challenges of mutation testing that we identified as well as the particular problems that we address in this dissertation. We show in Figure 1.2 an overview of the challenges of mutation testing that we identified.

1.2.1 The added Value of Mutation

The aim for software testing is to ensure the quality of the software, in other words, good test suites should reveal potential faults in a program. One concern about relying on TAC for software testing is the relationship between the fault revelation and the coverage of test suites. It is important to know the achieved level of confidence on software quality (or the quantity of potential fault that can be revealed) when a certain level of coverage is reached by the test suites. These information would guide practitioners to set the optimal (cost-effective) coverage level for their testing practices. However, the relationship between attained coverage and fault-revelation of test suites remains unknown for mutation and other TACs. Furthermore, the question of which TACs best guide software testing towards fault revelation remains controversial and open, thus, the benefit of mutation over other TACs is not well established. Most of the studies conducted in line of these problems make an assumption that we evaluate and prove to be

unreliable. Therefore, the confidence on the partial knowledge drawn from existing studies is further reduced. These challenge are addressed in this dissertation.

1.2.2 The Large Number of Mutants

Using mutation testing is a very costly activity due to the large number of mutants generated. In fact, the number of mutants may easily reach 1,000 in a program having less than 50 lines of code. Moreover, generating tests to kill mutants is a tedious task for software developers [SW09] and mutant-test execution can be expensive. The large number of mutants leads to an overhead, during the test generation and the test execution, for which software companies might not be willing to spend extra resources. Thus, it is important to reduce the number of mutants early (prior test generation or execution phase). Within the large number of mutants generated for mutation testing, a large portion is semantically equivalent to the original program and thus, need to be removed early. This form a challenge for mutation testing. Among the mutants that are not equivalent, only a few are valuable for testing purposes. Reducing the generated mutants set to the valuable mutants is another challenge of mutation testing.

Equivalent mutants. The problem of the equivalent mutants have been partly addressed by recent approaches such as the Trivial Compiler Equivalence (TCE) [Pap+15] (which reduces the mutants set by roughly 30%). Nevertheless, the number of non-equivalent mutants is still very large and there is need for further mutant reduction to ensure the practical use of mutation testing. Moreover, a technique that successfully selects valuable mutants (regardless of the mutant quality indicator considered [Pap+15]) also tackles the equivalent mutants problem. Therefore, the focus now goes to the valuable mutants, which are few.

Valuable mutants. Since the early days of mutation testing, researchers formed many mutant reduction strategies, such as selective mutation [ORZ93; WM95a] and random mutant selection [T A+79] to select valuable mutants. However, the lost in fault revelation by those mutant reduction techniques is still very high. This is partly due to the mutant quality indicator ("usefulness" or value metric) used by those techniques to select valuable mutants and the inability of those approaches to properly make the selection. Thus, there is a need for an approach that can reduce the number of mutants while retaining a reasonable fault revelation. This challenge is addressed in this dissertation.

1.2.3 Mutation Testing Test Automation

Manual software testing is tedious. Software companies are leaning toward test automation which mainly involves automated test generation and automated tests execution. For mutation testing to receive increasing adoption, there is a need for efficient test automation. Regarding tests execution, the main objective is to execute the tests, on mutants, in a cost effective manner. Thus, to efficiently execute multiple tests with multiple mutants. A challenge of mutation testing regards the efficiency of mutant-tests execution. Regarding the test generation, currently, most of the test generation is manual and particularly costly for mutation testing [SW09]. In order to reap the benefits of mutation, practitioners need to aim for tests that kill the mutants that are not killed by tests generated using testing methods other than mutation testing. These mutants are harder to kill but important for deep testing. Without a technique that eases the test generation to kill such mutants, many practitioners will not use mutation testing. Therefore,

another challenge of mutation testing is automated test generation to increase the mutation coverage score.

Test execution Tackling the challenge of the large number of mutants automatically reduce the cost of test execution. However, this cost is still very high if all tests need to be executed against the mutants, separately. Several approaches have been proposed in the literature to reduce the cost of mutant-test execution [Piz+19]. Kim *et al.* [KMK13] suggested executing only the tests that make the mutation infect the program state. Wang *et al.* [Wan+17] used the concept of mutant schemata [UOH93] to further reduce the cost of mutant test execution by enabling shared execution between the mutants. Vercammen *et al.* [Ver+18] proposed an approach that reduces the number of tests that need to be executed on each mutant by filtering-out the tests that are not intended for the method (function) under test. These approaches, coupled with the use of parallel execution give a satisfactory performance for mutant-test execution in practice. However, before test execution, the tests need to be designed with the aim of killing mutants, as long as the expected coverage score threshold is not reached. This is a tedious activity and needs to be automated.

Test generation Many techniques targeting mutation-based test generation have been proposed [Ana+13; Pap+19]. However, most of these focus on generating test suites from scratch, by maximizing the number of mutants killed, mainly by either covering the mutants or by targeting mutant infection. These techniques often still fail to kill many mutants. Therefore, manual effort is still highly required in order to build test suites that achieve high mutation coverage score, which is a costly task for practitioners. The challenge here is to automatically generate tests that kill additional mutants to improve the test suites for higher coverage. This challenge addressed in this dissertation.

1.3 Overview of the Contribution and Organization of the Dissertation

This section presents the contributions of this dissertation to address the aforementioned challenges on mutation testing, and the organization of this dissertation as depicted in Figure 1.3.

1.3.1 Contributions

Following are the contributions of this dissertation.

- **An empirical study on mutation and widely used TACs (Chapter 4).** We make an empirical study on the relationship between TAC coverage and fault revelation of test suites for mutation, statement and branch TACs. The empirical study is conducted on real-world programs and in a real-world setting. The study shows that the fault revelation increases with coverage only beyond a certain threshold of coverage and that (strong) mutation has the highest fault revelation. This suggests to target very high coverage scores when using mutation testing (and other studied TACs). Furthermore, we study an assumption (called "clean program assumption") made in previous similar studies and found that the assumption does not always hold, creating a potential threat to the validity of those studies. Therefore, based on our empirical study, there is an even stronger motivation to improve

the practicality of mutation testing (by reducing its cost), given that it is superior in effectiveness to the most widely used TACs (namely statement and branch coverage) and proven useful to reveal faults in software. The first step that we take in this cost reduction quest is to reduce the number of mutant.

- **An empirical study on mutant quality indicators for valuable mutants (Chapter 5).** We make an empirical study on the agreement between mutant quality indicators (MQI) found in the literature, including fault-revealing MQI. The aim of the study is to give an insight into whether there are important differences between the MQIs and what are their links with fault revelation. Knowing this would guide mutant reduction techniques to target the most appropriate set of mutants. The empirical study is conducted on a large benchmark of programs and faults and, the results show that all studied MQIs identify a small subset of mutants (less than 10% of the whole mutants set), suggesting that targeting a particular MQI mutant set can greatly reduce the number of mutants. Moreover, there is a large disagreement between the studied MQIs. This suggests that considering MQIs mutants other than fault-revealing ones as valuable would incur a fault revelation loss. Therefore, this study suggests to directly target fault revealing mutants for mutant reduction.
- **A mutant reduction technique with high fault revelation (Chapter 6).** We present *FaRM*, a mutant reduction technique that learns to select and prioritize fault-revealing mutants. Fault revealing mutants are mutants whose killing lead to the revelation of potential faults in the program under test. Our studies show that fault revealing mutants account to less than 5% of the whole mutants set. Successfully selecting fault revealing mutants can drastically reduce the number of mutants needed to be analyzed (in term of test generation and execution) and thus reduce the cost of mutation testing. *FaRM* is based on supervised machine learning and defines a set of mutant features that are used to learn the characteristics of fault revealing mutants on existing faulty programs and predict the fault revealing mutants on (new) programs under test. An evaluation of *FaRM* on two benchmarks of faulty C programs show that *FaRM* outperforms all the existing mutant selection approaches in term of faults revealed by the selected mutants. The machine learning classification technique built in *FaRM* is usable to select killable mutants and valuable mutants of other MQIs (such as subsuming mutants).

Despite the mutant reduction achieved by *FaRM*, practitioners also need to generate tests to kill those mutants (this is a tedious task, especially when thorough testing is needed). The next step that we take in our quest to reduce the cost of mutation is to automate the test generation to improve the test suite for higher mutation coverage score.

- **An automated test generation technique that targets stubborn mutants (Chapter 7).** We present *SEMu*, an automated test generation tool aiming at killing stubborn mutants (mutants that survives existing test suites). We target stubborn mutants because they enable thorough testing of specific software components and killing them would usually improve the test suites beyond the level that can be achieved by other TACs such as statement and branch coverage. The stubborn mutants are often harder to kill and therefore require more effort for manual analysis. Automating the test creation to kill stubborn mutant would reduce to effort to apply mutation in practice. *SEMu* enables such automation by leveraging symbolic execution to kill stubborn mutant. The main idea is to propagate the program state infection (difference between the original and mutant

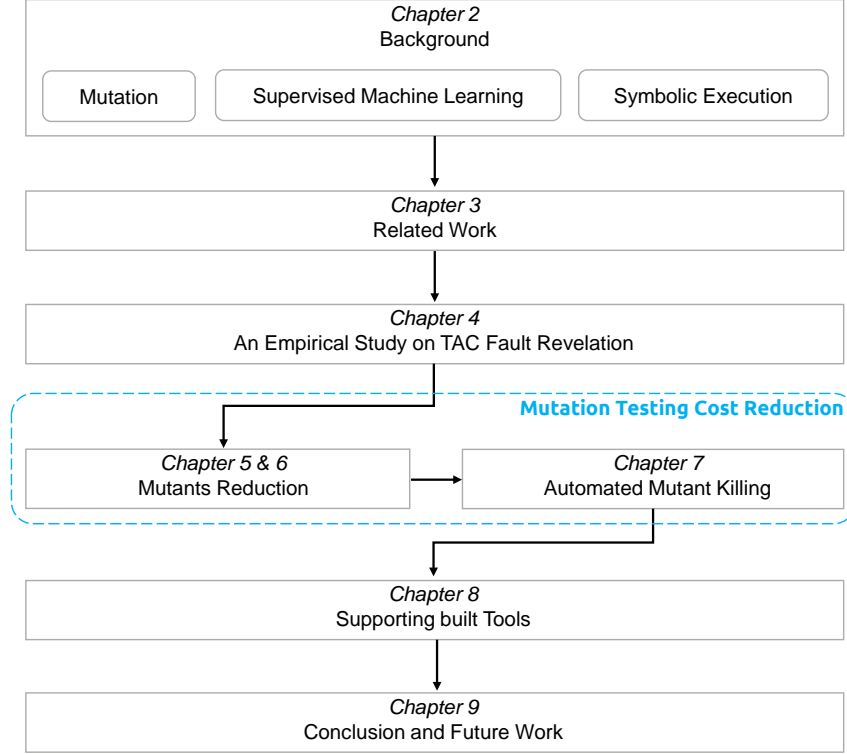


Figure 1.3: Organization of the dissertation.

program) in a cost effective manner to generate tests that kill the stubborn mutants. An empirical evaluation of *SEMu* on real-world programs show that *SEMu* successfully kills stubborn mutants and outperform existing test generation techniques in number of killed stubborn mutants.

- **A (mutation) TAC testing tool-set (Chapter 8).** We present the tools and frameworks that are built to implement and support the techniques presented in this dissertation, and enable their use for research and in practice. The built tools are: *Mart*, a mutant generation tool for the LLVM [LA04] bitcode, and *Muteria*, a framework that automate the mutation testing process by enabling an easy integration of the techniques implemented to improve the cost of mutation testing.

1.3.2 Organization of the Dissertation

In the remaining of this dissertation, Chapter 2 presents an overview on the concepts and techniques used in this dissertation. Chapter 3 presents the previous work that are related to the contributions presented in this dissertation. Chapter 4 presents our empirical study that evaluates the relation between test adequacy criteria coverage and fault revelation of test suites, and shows that mutation is superior to widely used test adequacy criteria. Chapter 5 presents our empirical study on mutant quality indicators that should be used to define valuable mutants for mutant reduction. Chapter 6 presents and evaluates *FaRM*, our approach to tackle the mutant reduction challenge of mutation testing. Chapter 7 presents and evaluates *SEMu*, our approach to tackle the automated test generation challenge of mutation testing. Chapter 8 describes the tools and frameworks built as the result of this work, to contribute for the adoption of mutation

testing in practice. Finally, Chapter 9 concludes this dissertation and presents the future work.

TECHNICAL BACKGROUND AND DEFINITIONS

This Chapter presents the technical background and definitions used in this dissertation.

Chapter content

2.1	Test Adequacy Criteria-based Software Testing	12
2.2	Mutation Testing	12
2.2.1	General Information	12
2.2.2	Mutant Killing	13
2.2.3	Some Definitions	14
2.2.4	Mutant Quality Indicators	14
2.3	Symbolic Execution	14
2.4	Supervised Machine Learning	15
2.4.1	Binary Classification problem	16
2.4.2	Some Classification Models	16
2.4.3	Evaluation techniques	18
2.4.4	Performance evaluation	19
2.5	Additional Definitions	20
2.5.1	Average Percentage of Fault Detected	20
2.5.2	Statistical Test	20
2.5.3	Effect Size	21
2.6	Summary	21

2.1 Test Adequacy Criteria-based Software Testing

Test adequacy criteria, or simply test criteria, define the requirements of the testing process. Thus, they form a set of elements (requirements or objectives) that should be exercised by test suites [ZHM97]. Goodenough and Gerhart [GG75] define test adequacy criteria as predicates stating that a criterion captures “what properties of a program must be exercised to constitute a thorough test, i.e., one whose successful execution implies no errors in a tested program”. As a result, they guide testers in three distinct ways [ZHM97]; by pointing out the elements (test objectives) that should be exercised when designing tests, by providing criteria for terminating testing (when coverage is attained), and by quantifying test suite thoroughness.

The test thoroughness, or test adequacy of a test suite is quantified by measuring the number of test objectives exercised by the test suite. In particular, given a set of test objectives, the ratio of those that are exercised by a test suite is called the *coverage* (see Equation 2.1). A test suite that manages to exercise all the objectives of a given test adequacy criterion is adequate with regards to the criterion.

$$Coverage = \frac{|Exercised\ Test\ Objectives|}{|All\ Test\ Objectives|} \quad (2.1)$$

2.2 Mutation Testing

2.2.1 General Information

Mutation testing [DLS78] is a test adequacy criterion that sets the revelation of artificial defects, called mutants, as the requirements of testing. As every test adequacy criteria, mutation assists the testing process by defining test requirement that should be fulfilled by the designed test cases, i.e., defining when to stop testing.

Software testing research (including Chapter 4 of this dissertation) has shown that designing tests that are capable of revealing mutant-faults results in strong test suites that in turn reveal real faults [FWH97; LPO09; Pap+19; Jus+14] and are capable of subsuming or almost subsuming all other structural testing criteria [Off+96b; FWH97; AO08].

Mutants form artificially-generated defects that are introduced by making changes to the program syntax. The changes are introduced based on specific syntactic transformation rules, called *mutation operators*. The syntactically changed program versions form the mutant-faults and pose the requirement of distinguishing their observable behavior from that of the original program. A mutant is said to be *killed*, if its execution distinguishes it from the original program. In the opposite case it is said to be *alive*.

Figure 2.1 illustrates the creation of mutants from a sample C program. Two mutation operators are applied on the original program to generate two mutants by mutating the statements at lines 3 and 2. The execution of a test input $x = 0$ will print "Even - Pos" on the original, "Odd - Pos" on mutant 1 and "Neg" on mutant 2. Both mutants have an output that is different from the original program's output, thus, the test input kills both mutant 1 and mutant 2. However, the test input $x = -1$ does not kill mutant 1 but kills mutant 2.

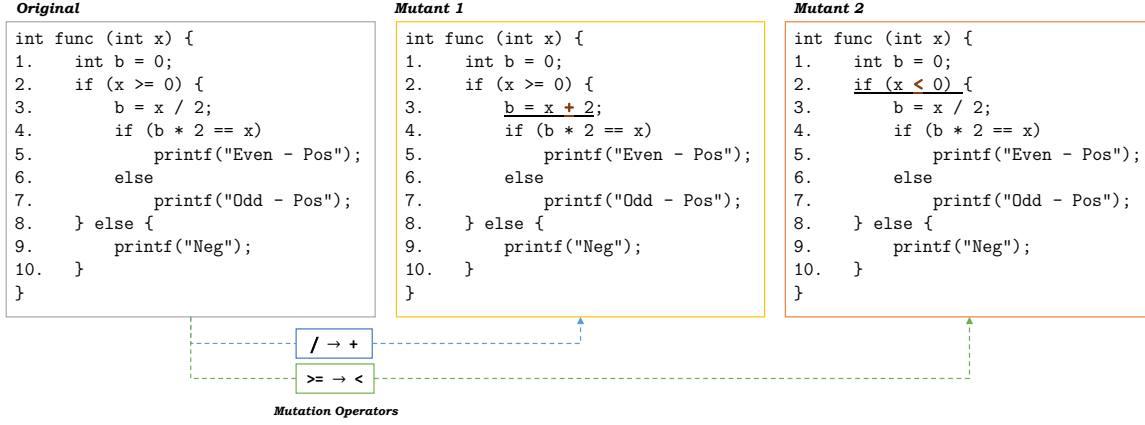


Figure 2.1: Program Mutation.

Mutation quantifies test thoroughness, or test adequacy [DLS78; DO91; FI98b], by measuring the number of mutants killed by the candidate test suites. In particular, given a set of mutants, the ratio of those that are killed by a test suite is called *mutation score*. Although all mutants differ syntactically from the original program, they do not always differ semantically. This means that there are some mutants that are semantically equivalent to the original program, while being syntactically different [OC94; Pap+15]. These mutants are called *equivalent* mutants [DLS78; OC94] and have to be removed from the test requirement set. The mutation score is computed using the Equation 2.2.

$$\text{Mutation Score} = \frac{|Killed\ Mutants|}{|All\ Mutants| - |Equivalent\ Mutants|} \quad (2.2)$$

Mutation score denotes the degree of achievement of the mutation testing requirements [AO08]. Intuitively, the score measures the confidence on the test suites (in the sense that mutation score reflects the fault revelation ability). Unfortunately, previous research [FWH97] and the study presented in Chapter 4 of this dissertation show that the relation between killed mutants and fault revelation is not linear as fault revelation improves significantly only when test suites reach high mutation score levels.

2.2.2 Mutant Killing

In order to strongly kill the mutant \mathcal{M} (created by mutating a program statement s of a program \mathcal{P}), a concrete test input t must satisfy the following 3 criteria (referred in the literature as RIP model [AO08; DO91; Mor90]):

- *Reachability* (R). The execution of t on \mathcal{P} must reach the program location s that is mutated to create \mathcal{M} .
- *Infection* (I). The execution of the input t must cause a difference in internal state of \mathcal{P} and \mathcal{M} right after the execution of s .
- *Propagation* (P). The difference of internal states must be propagated through the executions to the programs outputs.

2.2.3 Some Definitions

Definition 2.2.1. (*Equivalent mutant*). A mutant M of a program P is equivalent if and only if M is semantically equivalent to P .

Definition 2.2.2. (*Duplicate mutants*). Two mutants M_1 and M_2 of a program P are duplicate if and only if they are semantically equivalent.

Definition 2.2.3. (*Killable*). A mutant is killable if there exist (it is possible to create) a test that kills it.

Definition 2.2.4. (*indistinguished mutants*). Two mutants M_1 and M_2 of a program P are indistinguished if and only if any test that kills M_1 also kills M_2 and vice-versa. Note that this is not necessarily semantical equivalence as a test may kill the two mutants differently (M_1 and M_2 have different outputs which also differ from P ' output).

Definition 2.2.5. (*Subsumption*). A mutant M_1 subsumes another mutant M_2 if and only if M_1 is killable and any test that kills M_1 also kills M_2 .

2.2.4 Mutant Quality Indicators

The mutants generated during mutation testing have various characteristics. Mutants may be considered as valuable based on specific mutant characteristics. These characteristics are quality indicators for mutants (more details in Chapter 5). The 3 main mutant quality indicators emphasized in this dissertation are fault-revealing, subsuming and stubborn mutant quality indicators.

Fault Revealing are the mutants that are killed only by test cases that reveal a fault. *Subsuming* are the mutants that are subsumed only by indistinguished mutant. *Stubborn* are the mutants that are resistant to killing by test cases that execute them. Thus hard to infect or propagate.

Testing Requirement. Similarly to the mutation score defined in Equation 2.2, there exist a mutation score for each mutant quality indicator, which represents the improved test thoroughness of mutation with regards to the mutant quality indicator. We represent the mutant quality indicators mutation scores with the formula in Equation 2.3, where MQI is a mutant quality indicator (e.g. *Subsuming Mutation Score*).

$$MQI \text{ Mutation Score} = \frac{|Killed \ MQI \ Mutants|}{|All \ MQI \ Mutants|} \quad (2.3)$$

2.3 Symbolic Execution

Symbolic execution replaces concrete input with symbolic representation of the input domain and executes the program. When the execution begins, the path condition of the single exploration path is $\phi = True$. Each non branching statement is symbolically executed by using symbols (proceeding from the input) in place of concrete values. When a branching statement is encountered, the branching condition g is evaluated. In case $g \rightarrow True$, the *then* branch is followed by the execution. In case $g \rightarrow False$, the *else* branch is followed by the execution. If $g \nrightarrow True$ and $g \nrightarrow false$, both the *then* and *else* branches are feasible, thus, the execution is split (forks) into 2 executions (one execution follows each branch). The execution following the

then branch has the path condition updated as $\phi := \phi \wedge g$, while the execution following the *else* branch has $\phi := \phi \wedge \neg g$.

As the symbolic execution advances, the number of paths explored increase. At any point of the symbolic execution, the paths explored form a tree [Kin76] whose nodes represent control locations and, each node has an execution state $\langle \phi, \sigma \rangle$ made of its path condition ϕ and symbolic program state σ (made of control location (representing program counter value) and the symbolic valuation of variables).

Figure 2.2 illustrates the symbolic execution of a program, followed by a test generation. The left hand side sub-figure shows a sample C program and, the right hand side sub-figure shows the complete symbolic execution tree of that program. Each node (box), on the tree, represents the symbolic execution state (path condition and program state) at the point just before the line of code represented in the gray circle. The execution goes from the top with a single execution state. Each execution state is updated after the execution of each line of code and duplicated at branching statements. Here we have 3 paths. The test input that follow each path can be generated by solving (with a constraint solver like Z3 [DB08]) the path condition (ϕ) of the paths, at the completion of the symbolic execution.

Path explosion. Symbolic execution suffers from the problem of path explosion, which is a scalability bottleneck of symbolic execution caused by the huge or infinite number of paths that a program may have. In order to handle the path explosion issue of symbolic execution, several heuristic have been proposed to reduce the paths explored by the symbolic execution (guide the symbolic execution).

Any set Π of program paths form a tree that could result from a (guided) symbolic execution. Such a symbolic execution is restricted at branching statements to follow only the branches leading to paths contained in Π .

Preconditioned Symbolic execution [Mec+18] is a form of guided symbolic execution where the initial path condition (at the beginning of the symbolic execution) is set to a specific condition (pre-condition). The precondition restricts the symbolic execution to only follow a subset of program paths. The pre-condition can be derived from pre-existing tests and used to reduce the scalability (path explosion) problem [CS13] of symbolic execution.

Seeded symbolic execution is a form of pre-conditioned symbolic execution that constrain symbolic execution to explore the path followed by some concrete executions called *seeds*.

2.4 Supervised Machine Learning

Machine learning refers to techniques that use complex models to extract knowledge from existing data (training) and apply to new data (testing) to make predictions.

Supervised machine learning is a from of machine learning where the data used to train the models are labeled and the models are used to predict the labels of new data. There exist two main classes of supervised machine learning, classification models and regression models. Regression models are used to predict numerical values (e.g. predicting the house price in a region) while classification models are used to predict the category of the data (e.g. classifying

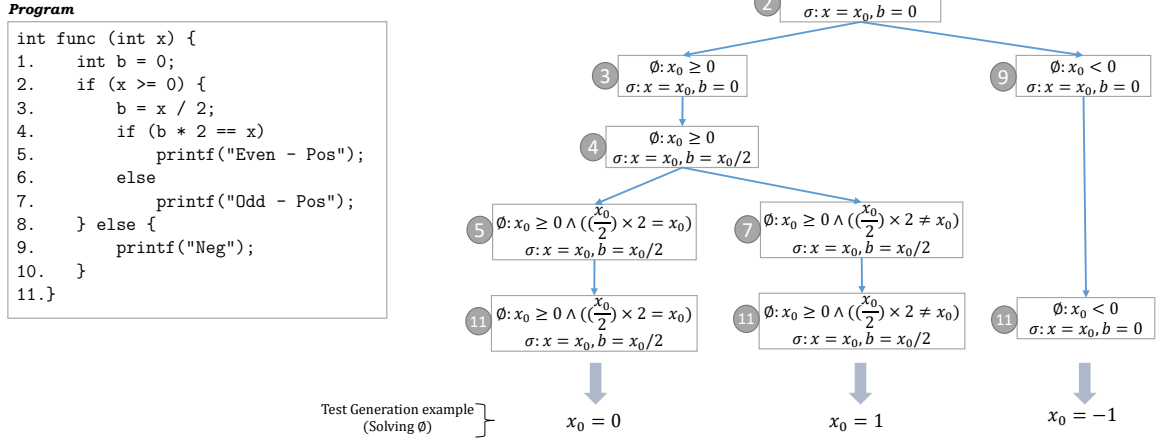


Figure 2.2: Symbolic execution and test input generation.

an email as span or not). This dissertation makes use of classification based supervised learning models.

2.4.1 Binary Classification problem

A k dimension classification problem is a problem that consists in classifying (labeling) data into k classes ($k \geq 2$). A binary classification problem is a classification problem where $k = 2$. The problems tackled, in this dissertation, using supervised machine learning, are binary classification problems.

Figure 2.3 illustrates the use of supervised machine learning for binary classification. At the training phase, the training data is used to train the classifier. The training data is made of a feature matrix and a class label vector. In the feature matrix each row represents a data point and each column represents a feature. The cell (i, j) represents the value of feature j for the data point i . The class label vector represents the class label of each data point (positive or negative class). At the class prediction phase, the class label of new data points are predicted using the feature matrix of the new data points. The feature vector (row of the feature matrix) of each data point is input to the (previously trained) classifier which outputs the corresponding predicted class label.

2.4.2 Some Classification Models

The two classification models evaluated in this dissertation are presented in this subsection.

2.4.2.1 Gradient Boosted Decision Trees

Decision trees are a form of supervised learning that builds models in form of a tree structure. It breaks down the dataset into smaller subsets while developing an associated decision tree incrementally. Each non leaf node of the tree represents a decision node. In the process of constructing decision trees, two metrics are mainly used:

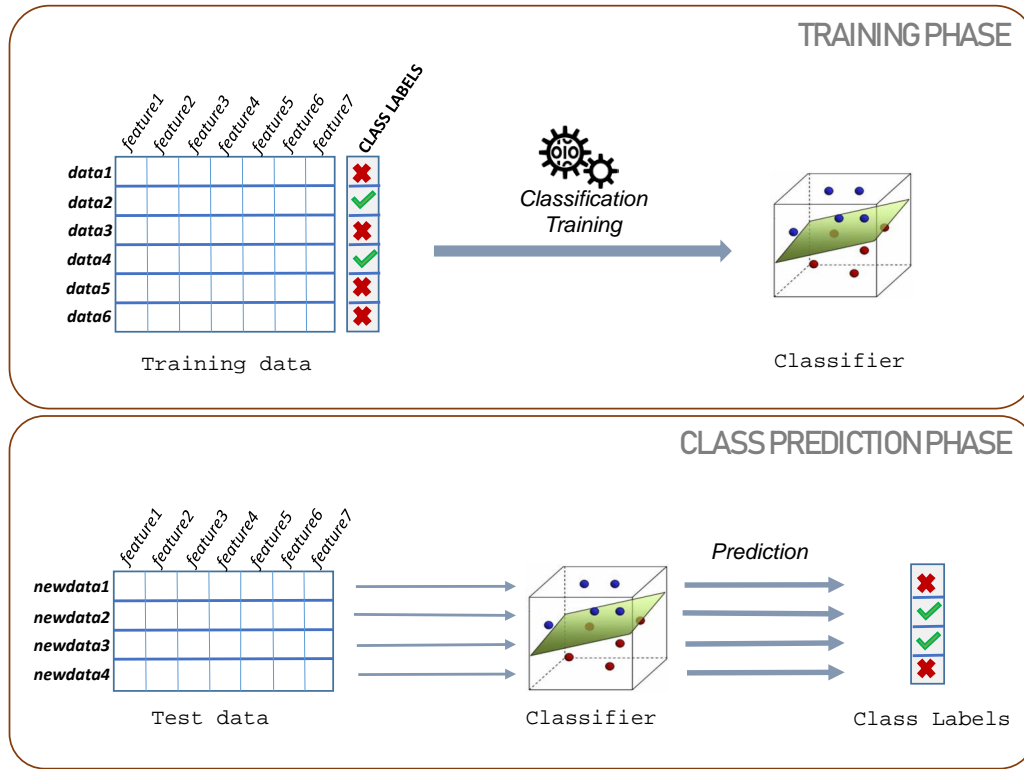


Figure 2.3: Supervised machine learning binary classification workflow. The top sub-figure illustrates the training phase and the bottom sub-figure illustrates the testing phase (class prediction).

The Entropy calculates the homogeneity of a sample. It decreases as the homogeneity increase. A sample that is equally divided has the highest entropy.

The Information Gain estimates the information contained in each of the data attributes by measuring the relative change in entropy with respect to the independent attributes. During the construction of a decision tree, the attributes are ranked according to their information gain and the ranking is used for filtering at every node of the tree.

Decision trees easily over-fit to the training data as the number of data attributes (features) increase and the decision tree becomes deeper.

Ensemble models based on decision tree help to leverage the overfitting of decision trees. They use multiple learners to solve the classification problem. Gradient Boosted Decision Trees is an example of ensemble based model that is shown to perform well in practice [CN06].

Gradient boosted decision trees combines multiple small decision trees (weak learners) in order to reduce bias. the weak learners are organized in a cascading manner where the output of a weak learner is the input of another. Thus, during prediction, each decision tree predicts the error of the preceding decision tree to improve (boosting) the error gradient.

2.4.2.2 Deep Neural Networks

Neural networks or more precisely artificial neural networks are a type of machine learning models inspired by the brain. The models are made of nodes, called neurons, that make computations. Similar to brain neurons which are activated with electric signals, artificial neural

Table 2.1: *Confusion Matrix*

		Prediction Outcome		
		Positive	Negative	Total
Actual Value	Positive	<i>True Positive (TP)</i>	<i>False Negative (FN)</i>	TP+FN
	Negative	<i>False Positive (FP)</i>	<i>True Negative (TN)</i>	FP+TN
	Total	TP+FP	FN+TN	

network neurons are activated when their computed values reach a certain threshold.

A neural network is organized into sequential layers which are a set of neurons. neurons of each layer are connected to all neurons of the next layer and a weight is assigned to each connection between neurons. The inputs data is represented as a vector of numerical values corresponding to the dimension of the first layer. The inputs are processed from the first to the last layer and the computation results are propagated, throughout the network, to the last layer. The learning (training) phase of a neural network consist in using the training data to assign the weights between connected neurons. The learned weights are used during testing to compute the predicted label of new data.

Neural networks that have more than 2 layers are called deep neural networks. Deep neural network have a broad usage spectrum, including image recognition, text analysis, etc.

2.4.3 Evaluation techniques

It is important to evaluate machine learning classification models to estimate the generalization of the models on future data. In this regard, there are two categories of model evaluation techniques:

Holdout. This technique aims to test the model on different data than it was trained on. Typically, the data is randomly divided into three subset:

- (1) The *training set* which is used to build the classification models.
- (2) The *validation set* which is used to asses the performance of the models during the training phase. This enable fine tuning of the models' parameters in order to find the best performing model.
- (3) The *test set* which is used to asses the likely performance of the models on future (unseen) data.

Cross validation. This technique randomly partition the data into two subsets: one subset used to train the models and another one used to evaluate the models. The most common form of cross validation is *k-fold* cross validation where the original dataset is partitioned into (user specified number) k equal size subsamples. k models are trained by iteratively training on $k - 1$ of the k subsamples and evaluating on the one remaining subsample. Each round of training and evaluation is called fold. At the end, the model that has median performance is selected for use. The most used values of k are 5 and 10.

2.4.4 Performance evaluation

Supervised machine learning classifiers are evaluated by their ability to correctly classify new data. Binary classifiers often predict the probability of the test data to belong to one of the two classes called the *positive class*. In the mutant selection problem addressed in this dissertation, the mutants of the targeted mutant quality indicator form the *positive class*. The prediction of a binary classifier falls into one of the 4 situations presented in the *confusion matrix* (Table 2.1). True positive (TP) represents the number of *positive class* data elements that the classifier classifies in the *positive class*. False positive (FP) represents the number of *negative class* data elements that the classifier classifies in the *positive class*. True negative (TN) represents the number of *negative class* data elements that the classifier classifies in the *negative class*. False negative (FN) represents the number of *positive class* data elements that the classifier classifies in the *negative class*.

The metrics used in this dissertation for the evaluation of classification techniques are the following widely used metrics:

2.4.4.1 Precision, recall and F-Measure

The precision of a binary classifier is the probability that the a new data classified in the *positive class* is actually a *positive class* data. In the evaluation, the precision is approximated statistically based on the test data with the formula:

$$Precision = TP / (TP + FP)$$

The Recall of a binary classifier is the probability that the classifier correctly classify a new *positive class*'s data as positive. The recall is also referred to as the sensitivity or true positive rate (TPR). In the evaluation, the recall is approximated statistically based on the test data with the formula:

$$Recall = TP / (TP + FN)$$

The F-measure or F-score or *F1* score of a binary classifier is the harmonic average of the precision and recall. The F-measure takes values between 0 and 1. higher F-measure means better classification performance. In the evaluation, the F-measure is calculated as following:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

2.4.4.2 Receiver Operating Characteristics

The fall-out or false positive rate (FPR) of a classifier is computed using the formula $FPR = FP / (FP + TN)$.

A well known mean to visualize the performance of a classifier is to plot the FPR against the TPR at various threshold settings (as binary classifiers returns probabilities, the threshold setting is the cut-off point to classify data in either class). The resulting curve is called the Receiver Operating Characteristics (ROC) curve. Each point of the ROC curve has coordinates (FPR_i, TPR_i) for

each threshold setting i , where FPR_i and TPR_i are the FPR and TPR at the threshold setting i , respectively.

The goal of the ROC curve is to visualize the separability of a model which is summarized by the area under the ROC curve.

2.4.4.3 Area Under The Curve

The Area Under the ROC Curve (referred in this dissertation as AUC) represents the separability of a binary classifier. In other words, the AUC is the probability that the classifier ranks a randomly chosen *positive class* data higher than a randomly chosen *negative class* data. AUC value range between 0 and 1. Higher AUC means better classifier performance.

2.5 Additional Definitions

2.5.1 Average Percentage of Fault Detected

Given a set of artifacts and a set of objectives that should be covered by the artifacts, where each artifact may cover 0 or many objectives and an objective may be covered by 0 or many artifact, artifacts prioritization consist in ordering the artifacts to reach higher objectives coverage when selecting the top k artifacts. In order to evaluate the prioritizations, the cumulated artifacts coverage curve is plotted. Each point (x, y) of the cumulated artifacts coverage curve is such that y is the objective coverage of the top- x set of artifacts. The area under the cumulated artifacts coverage curve gives a value that "scores" an artifacts prioritization.

In the context of software testing, test case prioritization aims to prioritize tests to find potential faults early. Here, the artifacts are the tests and the objectives are the faults. The cumulated artifacts coverage curve is called Average Percentage of Faults Detected (APFD) [Hen+16]. Similarly, this dissertation makes use of APFD to score mutants prioritizations. The artifacts are the mutants and the objectives are faults. A mutant covers an objective (fault) if the mutant can lead to the fault.

2.5.2 Statistical Test

The empirical studies conducted in this dissertation make use of statistical tests to check that the values of one sample are higher than the values of another sample (validate or refute the hypothesis that two sample are not statistically different). The statistical test used in this dissertation is the Wilcoxon test which is a non-parametric test. The test gives a value (called *p-value*) that decides whether the two samples are different based on a specified confidence level threshold. Lower *p-value* increase the confidence in the difference between the two sample. The *p-value* range from 0 to 1 and the two sample are considered different with statistical significance when the *p-value* is less or equal to the confidence level threshold. A commonly used confidence level threshold for difference is *p-value* = 0.05.

2.5.3 Effect Size

In this dissertation, we use the Vargha Delaney effect size \hat{A}_{12} as metric for effect size. Vargha Delaney effect size [VD00] quantifies the size of the differences (statistical effect size) [AB11; Woh+00]. The \hat{A}_{12} effect size is simple and intuitive. It measures the probability that values drawn from one set of data will have a different value to those drawn from another. $\hat{A}_{12} = 0.5$ suggests that the data of the two samples tend to be the same. Values of \hat{A}_{12} higher than 0.5 indicate that the first dataset tends to have higher values, while values of \hat{A}_{12} lower than 0.5 indicate that the second data set tends to have higher values. Note that the \hat{A}_{12} value is meaningful only when there is statistical significant difference in the data samples.

2.6 Summary

This chapter presented an overview and the definitions of the the main concepts used in this dissertation. The next chapter will present the work, from the literature, that are in line with the problems tackled in this dissertation and, when needed, will contrast them with the contributions of this dissertation.

RELATED WORK

This chapter discusses the existing work related to the contributions of this dissertation. Special care was taken to exhaustively cover all the published studies until the time of writing.

Chapter content

3.1	Evaluation of Testing Criteria	25
3.2	Mutant Selection and Prioritization	25
3.2.1	Useful Mutants	26
3.2.2	Static Mutant Selection	26
3.2.3	Approaches Based on Static and Dynamic Ananalysis	27
3.2.4	Machine learning based approaches	27
3.2.5	Mutant Reduction Summary	29
3.3	Automated Tests Input Generation For Mutation Testing	29
3.4	Summary	31

Table 3.1: *Summary of previous studies on the relationship of test criteria and faults.*

Author(s) [Reference]	Year	Largest Sub- ject	Language	Test Criterion	Fault Types	Summary of Primary Scientific Findings
Frankl & Weiss [FW91; FW93]	'91, '93	78	Pascal	branch, all-uses	real faults	Coverage correlates with test effectiveness. All-uses correlates with test effectiveness, while branch does not.
Offutt <i>et al.</i> [Off+96b]	'96	29	Fortran, C	all-uses, mutation	seeded faults	Both all-uses and mutation criteria are effective but mutation detects more faults.
Frankl <i>et al.</i> [FWH97]	'97	78	Fortran, Pascal	all-uses, mutation	real faults	Test effectiveness (for both all-uses and mutation criteria) is increasing at higher coverage levels. Mutation performs better.
Frankl & Iakounenko [FI98a]	'98	5,000	C	all-uses, branch	real faults	Test effectiveness increases rapidly at higher levels of coverage (for both all-uses and branch criteria). Both criteria have similar test effectiveness.
Briand & Pfahl [BP00]	'00	4,000	C	block, c-uses, p-uses, branch	simulation	There is no relation (independent of test suite size) between any of the four criteria and effectiveness
Andrews <i>et al.</i> [And+06]	'06	5,000	C	block, c-uses, p-uses, branch	real faults	Block, c-uses, p-uses and branch coverage criteria correlate with test effectiveness.
Namin & Andrews [NA09]	'09	5,680	C	block, c-uses, p-uses, branch	seeded faults	Both test suite size and coverage influence (independently) the test effectiveness
Li <i>et al.</i> [LPO09]	'09	618	Java	prime path, branch, all-uses, mutation	seeded faults	Mutation testing finds more faults than prime path, branch and all-uses test criteria.
Papadakis & Malevris [PM10a]	'10	5,000	C	Mutant sampling, 1 st & 2 nd order mutation	seeded faults	1 st order mutation is more effective than 2 nd order and mutant sampling. There are significantly less equivalent 2nd order mutants than 1 st order ones.
Ciupa <i>et al.</i> [Ciu+11]	'09	2,600	Eiffel	Random testing	real faults	Random testing is effective and has predictable performance.
Wei <i>et al.</i> [WMO12]	'12	2,603	Eiffel	Branch	real faults	Branch coverage has a weak correlation with test effectiveness.
Hassan & Andrews [HA13]	'13	16,800	C, C++, Java	multi-Point Stride, data flow, branch	mutants	Def-uses is (strongly) correlated with test effectiveness and has almost the same prediction power as branch coverage. Multi-Point Stride provides better prediction of effectiveness than branch coverage.
Gligoric <i>et al.</i> [Gli+13; Gli+15]	'13, '15	72,490	Java, C	AIMP, DBB, branch, IMP, PCC, statement	mutants	There is a correlation between coverage and test effectiveness. Branch coverage is the best measure for predicting the quality of test suites.
Inozemtseva & Holmes [IH14]	'14	724,089	Java	statement, branch, modified condition	mutants	There is a correlation between coverage and test effectiveness when ignoring the influence of test suite size. This is low when test size is controlled. Also, different criteria have little impact on the strength of the correlation.
Just <i>et al.</i> [Jus+14]	'14	96,000	Java	statement, mutation	real faults	Both mutation and statement coverage correlate with fault detection, with mutants having higher correlation.
Gopinath <i>et al.</i> [GJG14]	'14	1,000,000	Java	statement, branch, block, path	mutants	There is a correlation between coverage and test effectiveness. Statement coverage is the best measure for predicting the quality of test suites.
This Dissertation (Chapter 4)	'17	83,100	C	statement, branch, weak & strong mutation	real faults	There is a strong connection between coverage attainment and fault revelation for strong mutation but weak for statement, branch and weak mutation. Fault revelation improves significantly at higher coverage levels.

3.1 Evaluation of Testing Criteria

This section discusses empirical studies related to the evaluation of test adequacy criteria and is mainly related to Chapter 4.

Table 3.1 summarizes the characteristics and primary scientific findings of previous studies on the relationship between test criteria and fault revelation. As can be seen, there are three types of studies, those that use real faults, seeded faults and mutants. Mutants refer to machine-generated faults, typically introduced using syntactic transformations, while seeded faults refer to faults placed by humans.

One important concern regards the Clean Program Assumption when using seeded or mutant faults (see Chapter 4). In principle most of the previous studies that used seeded or mutant faults assume the Clean Program Assumption as their experiments were performed on the original (clean) version and not on the faulty versions. This is based on the intuitive assumption that as artificial faults denote small syntactic changes they introduce small semantic deviations. Our work (Chapter 4) shows that this assumption does not hold in the case of real faults and thus, leaves the case of artificial faults open for future research. Though, previous research has shown that higher order (complex) mutants [PM10a; Off92] are generally weaker than first order (simple) ones and that they exhibit distinct behaviors [GJG17], which implies that the assumption plays an important role in the case of artificial faults.

Only the studies of Frankl and Weiss [FW91; FW93], Frankl *et al.* [FWH97], Frankl and Iakounenko [FI98a], Ciupa *et al.* [Ciu+11] and Wei *et al.* [WMO12] does not assume the Clean Program Assumption. Unfortunately, all these studies have a limited size and scope of their empirical analysis and only the work of Frankl *et al.* [FWH97] investigates mutation. Generally, only three studies (Offutt *et al.* [Off+96b], Frankl *et al.* [FWH97] and Li *et al.* [LPO09]) investigate the fault revelation question of mutation, but all of them use relatively small programs and only the work of Frankl *et al.* [FWH97], uses real faults, leaving open the questions about the generalizability of their findings.

The studies of Andrews *et al.* [And+06] and Just *et al.* [Jus+14] used real faults to investigate whether mutants or other criteria can form substitutes of faults when conducting test experiments. This question differs from the fault revelation one as it does not provide any answer on the ability of the test criteria to uncover faults. Also, both these studies make the Clean Program Assumption and do not control for test suite size.

Overall, although the literature contains results covering a considerable number of test adequacy criteria, including the most popular (branch, statement and mutation-based criteria), our current understanding of these relationships is limited and rests critically upon the Clean Program Assumption.

3.2 Mutant Selection and Prioritization

Since the early days of mutation testing, researchers realised that the number of mutants is one of the most important problems of the method. Therefore, several approaches have been proposed to address this problem. This section discusses work related to mutant reduction and mainly related to Chapter 6.

3.2.1 Useful Mutants

Although effective, mutation requires too many mutants making the cost of generating, analysing and executing them particularly high. Recent studies have shown that only a small number of mutants is sufficient to represent them [KPM10; ADO14; Pap+16] and that the majority of the mutants are somehow “irrelevant” to the underlying faults (faults that testers seek for) [Pap+18]. Along these lines, Natella *et al.* [Nat+13] experimented with fault injection and demonstrated that up to 72% of injected faults are non representative.

3.2.2 Static Mutant Selection

Static Mutant Selection approaches typically fall into 2 categories: mutant sampling and selective mutation.

3.2.2.1 Mutant Sampling

Mutant random sampling was one of the first attempts to mutant reduction [Bud80; Acr80]. Random sampling randomly selects a fixed proportion of the generated mutants for analysis, non-selected mutants are discarded. Random sampling was evaluated by Wong [Won93] who found that a sampling ratio of 10% results in a test effectiveness loss of approximately 16% (evaluated on Fortran programs using the Mothra mutation testing system [DeM+88]). More recently, Papadakis and Malevris [PM10a], using the Proteum mutation testing tool [DMV01], reported a fault loss on C operators of approximately 26%, 16%, 13%, 10%, 7% and 6% for sampling ratios of 10%, 20% ..., 60% respectively.

3.2.2.2 Selective Mutation

An alternative approach to reduce the number of mutants is to select them based on their types, i.e., according to the mutation operators. Mathur [Mat91] introduced the idea of constrained mutation (also called selective mutation), using only two mutation operators. Wong *et al.* [WM95a] experimented with sets of operators and found that two operators alone have a test effectiveness loss of approximately 5%. Offutt *et al.* [ORZ93; Off+96a] extended this idea and proposed a set of 5 operators, which had almost no loss on its test effectiveness. This 5 mutation operator set is considered as the current standard of mutation as it has been adopted by most of the modern mutation testing tools and used in most of the recent studies [Pap+19].

Many additional selective mutation approaches have been proposed. Mresa and Bottaci [MB99] defined a selective mutation procedure focused on reducing the number of equivalent mutants, instead of the number of mutants alone, as done by the studies of Mathur [Mat91] and Offutt *et al.* [Off+96a; ORZ93]. They report significant reductions on the numbers of equivalent mutants produced by the selected operators, with marginal effectiveness loss (evaluated on Fortran with Mothra). Later, Barbosa *et al.* [BMV01] defined a selective mutation procedure aimed at reducing the computational cost of mutation testing of C programs. They found that a set of 10 operators could give almost the same results with the whole set of C operators supported by Proteum (78 operators). Namin *et al.* [NAM08] used regression analysis techniques and found

that a set of 13 mutation operators of Proteum could provide substantial cost execution savings without any significant effectiveness loss (mutant reductions of approximately 93% are reported).

More recently, researchers have experimented with mutations that involve only mutants deletion [Unt09]. Deng *et al.* [DOL13] experimented with Java programs and the MuJava mutation operators [MOK06] and reported reductions of 80% on the number of mutants with marginal effectiveness losses. Delamaro *et al.* [DOA14] defined deletion operators for C and reported that they significantly reduce the number of equivalent mutants, with again marginal effectiveness losses.

After several years of development of various selective mutation approaches, recent research has established that literature approaches perform similarly to random mutant sampling. Zhang *et al.* [Zha+10b] compared random mutant selection and selective mutation (using C programs and the Proteum mutation operators) and found that there are no significant differences between the two approaches. The most recent approach is that of Kurtz *et al.* [Kur+16a] (using C programs and the Proteum mutation operators), which also reached the same conclusion (reporting that mutant reduction approaches, both selective mutation and random sampling, perform similarly).

3.2.3 Approaches Based on Static and Dynamic Analysis

Other attempts have explored the identification of the program locations to be mutated. The key argument in these research directions is that program location is among the most important factor that determines the utility of the mutants. Sun *et al.* [Sun+17] suggested selecting mutants that are diverse in terms of static control flow graph paths that cover them. Gong *et al.* [Gon+17] used code dominator analysis in order to select mutants that, when they are covered, maximize the coverage of other mutants. This work applies weak mutation and attempts to identify dominance relations between the mutants in a static way.

Petrovic and Ivankovic [PI18] identified the arid nodes (special AST nodes) as a source of information related to utility of the mutants. Their work uses dynamic analysis (test execution) combined with static analysis (based on AST) in order to identify mutants that are helpful during code reviews. We include such features in our study in Chapter 6 with the hope that they can also capture the properties of fault revealing mutants. Nevertheless, still as part of future work it is interesting to see how our features can fit within the objectives of code reviews [PI18].

Mirshokraie *et al.* [MMP15] used static (complexity) and dynamic (number of executions) analysis features to select mutants, for JavaScript programs, that reside on code parts that have low failed error propagation (they are likely to propagate to the program output). Their results show that more than 93% of the selected mutants are killable, and that more than 75% of the non-trivial mutants resided in the top 30% ranked code parts.

3.2.4 Machine learning based approaches

Recently, there have been a handful work that use machine learning techniques to reduce the cost of mutation.

3.2.4.1 Mutant Selection

The closest studies to ours are the “predictive mutation”, by Zhang *et al.* [Zha+16; Zha+18; MCZ19], and the “fault representativeness” of software fault injection by Natella *et al.* [Nat+13]. Predictive mutation testing attempts to predict the mutants killed for a given test suite without any mutant execution. It employs a classification model using both static and dynamic features (both on test suite and the mutants) and achieves remarkable results with an overall 10% error on the predicted mutation scores. Predictive mutation has a similar goal with our killable mutant prediction method. Though, predictive mutation assumes the existence of test suites, while our killable mutant prediction method does not. Nevertheless, our method targets a different problem, the prediction and prioritization of the important mutants prior to any test execution. To do so, we use only static features (on the code under test), while predictive mutation heavily relies on test code and dynamic features [MCZ19], and evaluate our approach using real faults (instead of mutants).

Natella *et al.* [Nat+13] proposed removing injected faults to achieve meaningful ‘representative’ results and reduce the application cost of fault injection. This was achieved by employing classification algorithms that use complexity metrics. This approach has a similar goal with our fault revealing mutant selection, but in a different context, i.e., it targets emulating fault behaviour and not fault revelation. Nevertheless, Natella *et al.* rely on complexity metrics, which in our case do not seem to be adequate (as we show in RQ6). Still it is interesting to see how our approach performs in the fault injection context.

Another similar line of work is Evolutionary Mutation Testing (EMT) [DM18]. EMT is a technique that attempts to select useful mutants based on dynamic features (test execution traces) and uses them to support test augmentation. EMT learns the most interesting mutation operators and locations in the code under analysis using a search algorithm and mutant execution results. Overall, EMT achieve a 45% reduction on the number of mutants. Although EMT aims at the typical mutant reduction problem (while we aim at the fault revealing one), it can complement our method. Since EMT performs mutant selections after the mutant-test executions, our method can provide a much better starting point. Another way to combine the two techniques is to use the search engine of EMT, together with our features, to refine the mutant rankings.

3.2.4.2 Mutants Prioritization

A different way to reduce the mutants’ number is to rank the live mutants according to their importance, so that testers can apply customised analysis according to their available budget. Along these lines, Schuler *et al.* [SZ13] used the mutants’ impact to rank mutants according to their likelihood of being killable. Namin *et al.* [Nam+15] introduced the MuRanker approach. MuRanker uses three features: the differences that mutants introduce (a) on the control-flow-graph representation (Hamming distance between the graphs), (b) on the Jimple representation (Hamming distance between the Jimple codes) and (c) on the code coverage differences produced by a given set of test cases (Hamming distance between the traces of the programs). Although our mutant prioritization scheme is similar to these approaches, we target a different problem, the static detection of valuable mutants. Thus, we do not assume the existence of test suites and mutants executions. The benefit of not making any such assumptions is that we can reduce the number of mutants to be analysed by testers, to be generated and executed by mutation testing

tools.

3.2.5 Mutant Reduction Summary

From the above discussion it should be clear that despite the plethora of the selective mutation testing approaches, random sampling remains one of the most effective ones. This motivated our work, which used machine learning techniques and source code features in order to effectively tackle the problem. Moreover, as most of the methods use only one features, the mutant type, which according to our information gain results does not have relatively good prediction power, they should perform poorly. More importantly, our approach differs from the previous work in the evaluation metrics used. All previous work measured test effectiveness in terms of artificial faults (i.e., mutant kills or seeded faults found), while we used real faults. We believe that this is an important difference as our target (dependent variable) is the actual measurement of interest, i.e., the real fault revelation, and not a proxy, i.e., the number of mutants killed.

3.3 Automated Tests Input Generation For Mutation Testing

Many techniques targeting mutation-based test generation have been proposed [Ana+13; Pap+19]. However, most of these focus on generating test suites from scratch, by maximizing the number of mutants killed, mainly by either covering the mutants or by targeting mutant infection. In contrast we aim at deep testing of specific program areas by targeting stubborn mutants that are hard to propagate.

Papadakis and Malevris [PM11] and Zhang et al. [Zha+10a] proposed embedding mutant related constraints, called infection conditions, within the meta-programs that inject and control the mutants in order to force symbolic execution to cover them. As a result, symbolic execution modules can produce test cases that satisfy the infection conditions and have good chances to kill the mutants. Although, simple and to some extent effective, these approaches only target mutant infection, which makes them relatively weak as our results show, i.e., they are similar to the 'no-propagation' strategy that we investigate in this Chapter 7.

To bypass the abovementioned problem Papadakis and Malevris [PM10b] and Harman et al. [HJL11] aimed at indirectly handling mutant propagation. The former technique searches symbolically the path space of the mutant programs (after the mutation point), while the later one searches the input program space defined by the path conditions in order to bypass constraints not handled by the used solver and to indirectly make the mutants propagate. In contrast our approach aims at incrementally differentially exploring the path space by considering the symbolic states and making a relevant exploration.

Panichella *et al.* [PKT18b] applied search-based testing in order to generate tests for any TAC by aggregating the multiple test objectives into a single fitness function for single-objective search. Regarding mutation, they estimate, for each mutant, the state infection and propagation using distances from the test execution trace to the mutant. Due to the information loss from the aggregation of multiple objectives, they also propose an approach [PKT18a] based on multiple-objectives search. However, these approaches do not guarantee the infection and in case of stubborn mutants, the initial tests might even already maximize the fitness function (because the initial test suite execute the mutants). Fraser and Zeller [FZ12] and Fraser and Arcuri

[FA15] also applied search-based testing in order to generate mutation-based tests. Their key advancement was to guide the search by measuring the differences between the test traces of the original and mutant programs. While such an attempt is potentially powerful, it still fails to provide the guidance needed in order to trigger such differences. Moreover, search techniques rely on the ability to execute test cases fast (applied at the unit level), making them less effective in cases of slow test execution (such as system level testing that we target here). Nevertheless, a comparison between search-based test generation and symbolic execution falls out of the scope of this dissertation.

Much of work on testing software patches has also been performed the recent years [Tan+11; MC12; MC13]. However, most of these methods aim at covering patches and not the program semantics (behavioural changes). Moreover, these techniques target the general patch testing problem, which in a sense assume very few patches with many changes. The mutation case though involves many mutants, which are small syntactic deviations, facts that our method takes advantage in order to optimize the mutant killings.

Differential symbolic execution [Per+08] aims at reasoning about semantic differences of program versions, but since it performs a whole program analysis it can experience significant scalability issues when considering large programs and multiple mutants. Directed incremental symbolic execution [Per+11] guides the symbolic exploration through static program slicing. Such a method can be imprecise due to the static nature of slicing and expensive when used with many mutants. Nevertheless, program slicing could be used to further guide our approach towards the relevant mutant exploration space.

Partition-based regression verification [BSR13] employs random testing and dynamic symbolic execution to identify partitions of the input space that when sampled can reach and propagating version differences on the program output. Similarly to the directed incremental symbolic execution, partition-based verification relies on expensive program whole program executions and static slicing, which is often imprecise, leading to large execution overheads. Our approach in contrast does not execute multiple prefixes to reach the mutation points and controls the exploration on a relatively small/manageable space.

Shadow symbolic execution [PKC16] applies a combined execution on both program versions under analysis. It relies on analysis a meta-program that is similar to the mutant's meta-program in order to take advantage of the common program parts. The major difference with our method is that we specifically target multiple mutants at the same time, limit the program exploration through data state comparisons in order to optimize performance. Since shadow targets single patches and exhaustively searches the path space (after the mutation point) it can experience scalability issues.

Overall, while many related techniques have been proposed, they have not been investigated in the context of mutation testing and particularly to target stubborn mutants. Stubborn mutants are hard to kill and their killing results in test inputs that are linked with corner cases and increase fault revelation (see Chapter 4).

3.4 Summary

This chapter presented the work, in the literature, that are related to the contributions of this dissertation. Overall, this chapter shows that the work in the literature are inconclusive about the coverage criteria that best guide the testing of software, and the relation between coverage and fault revelation. Furthermore, the problems of the large number of mutants and automated test generation for mutation testing remains unhandled. The next chapters present our contributions to fill the gap in the literature regarding those problems.

AN EMPIRICAL EVALUATION OF TEST ADEQUACY CRITERIA

The relationship between coverage and fault-revelation is not well established. Most previous studies on coverage criteria rely on the Clean Program Assumption, that a test suite will obtain similar coverage for both faulty and fixed (‘clean’) program versions. This chapter presents evidence that the Clean Program Assumption does not always hold. Also, a study is conducted using a robust experimental methodology that avoids this threat to validity, suggesting that strong mutation testing has the highest fault revelation of four widely-used criteria. The findings also reveal surprisingly high non-linearity, with the result that fault revelation only starts to increase once relatively high levels of coverage are attained.

This chapter is based on the work published in the following paper:

- Thierry Titchou Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. *In Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*, 2017

Chapter content

4.1	Introduction	34
4.2	Test Adequacy Criteria	35
4.2.1	Statement and Branch Adequacy Criteria	36
4.2.2	Mutation-Based Adequacy Criteria	36
4.3	Research Questions	36
4.4	Research Protocol	37
4.4.1	Programs Used	38
4.4.2	CoREBench: realistic, complex faults	38
4.4.3	Test Suites Used	39
4.4.4	Tools for Mutation Testing and Coverage Measurement	40
4.4.5	Analyses Performed on the Test Suites	41
4.5	Experimental Results	43
4.5.1	RQ1: Clean Program Assumption	43
4.5.2	RQ2: Fault revelation at higher levels of coverage	44
4.5.3	RQ3: Fault Revelation of Statement, Branch, Weak and Strong Mutation	45
4.6	Threats to Validity	47
4.7	Conclusions	48

4.1 Introduction

The question of which coverage criterion best guides software testing towards fault revelation remains controversial and open [Gli+13; GJG14; IH14; And+06; FI98a]. Previous research has investigated the correlation between various forms of structural coverage and fault revelation using both real and simulated faults (seeded into the program under test as mutants). Determining the answer to the test coverage question is important because many software testing approaches are guided by coverage [CS13; HJZ15; FZ12; PM10a], and the industry standards used by practising software engineers mandate the achievement of coverage [Rad92; Rei95]. Nevertheless, the findings of the studies hitherto reported in the literature have been inconclusive, with the overall result that this important question remains unanswered.

Most previous studies make an important assumption, the veracity of which has not been previously investigated. We call this assumption the ‘Clean Program Assumption’. The assumption is that test suites are assessed based on the coverage they achieve on ‘clean’ programs, which do not contain any known faults. This practice might be problematic when using faulty versions (in order to check the fault-revealing potential of the test suites) since test suites are assessed on each of the faulty versions and not the clean program from which (and for which) the coverage was measured.

Of course, it is comparatively inexpensive (and therefore attractive to experimenters) to use a single test suite for the clean program, rather than using separate test suites for each of the faulty versions. However, a test suite that is adequate for the clean program may be inadequate for some of the faulty versions, while test suites that have been rejected as inadequate for the clean program may turn out to be adequate for some of the faulty versions. Furthermore, the coverage achieved by inadequate test suites may differ between the clean version of the program and each of its faulty versions.

These differences have not previously been investigated and reported upon; if they prove to be significant, then that would raise a potential threat to the scientific validity of previous findings that assume the Clean Program Assumption. We investigated this assumption and found strong empirical evidence that, it does *not* always hold; there are statistically significant differences between the coverage measurements for clean and faulty versions of the programs we studied.

Given that we found that we cannot rely on the Clean Program Assumption, we then implemented a robust methodology, in which the test suite for each test adequacy criteria is recomputed for each of the faulty versions of the program under test. We studied statement, branch, strong and weak mutation criteria, using a set of real-world faults, recently made available [BR14], located in 145,000 lines of C code spread over four different real-world systems.

We used systems with mature test suites, which are augmented by using the popular test data generation tool KLEE [CDE08] and by hand, to ensure the availability of a high quality pool of test data, from which to draw test suites. Unfortunately, such a high quality test pool cannot yet be guaranteed using automated test data generation tools alone, partly because of the inherent undecidability of the problem, and partly because of the limitations of current tools [LMH09; Ana+13]. Nevertheless, it is important for us to have such a high quality test pool in order to allow us to sample multiple test suites related to the faults studied and to achieve different experimentally-determined levels of coverage we choose, while controlling for test suite size. Using randomised sampling from the augmented high-quality test pool, we were thus able to

generate test suites that achieve many different coverage levels, thereby placing coverage level under experimental control.

Perhaps the most surprising result from our study is that we find evidence for a connection only between coverage attainment and fault revelation for one of the four coverage criteria: strong mutation testing. For statement, branch and weak mutation testing, we found no evidence that increased coverage is connected to increased fault revelation. This is a potentially important finding, notwithstanding the ‘Threats to Validity’ of generalisation discussed at the end of this chapter, especially given the emphasis placed on branch coverage by software tools and industrial standards. While some previous studies have made similar claims (for branch and block coverage [BP00]), these conclusions were subsequently contradicted [And+06; NA09; Gli+13; Gli+15].

One of the other interesting (and perhaps surprising) findings of our study is that the relationship between strong mutation and fault revelation is not only non-linear, but exhibits a form of ‘threshold behaviour’. That is, above a certain threshold, we observed a strong connection between increased coverage and increased fault revelation. However, below this threshold level, the coverage achieved by a test suite is simply irrelevant to its fault-revealing potential. This ‘threshold observation’ and the apparent lack of connection between fault revelation and statement/branch/weak mutation coverage may go some way to explaining some of the dissimilar findings from previous studies (and may partially reduce the apparent controversy). According to our results, any attempt to compare inadequate test suites that fail to reach threshold coverage may be vulnerable to ‘noise effects’; two studies with below-threshold coverage may yield different findings, even when the experimenters follow identical experimental procedures.

More research is needed in this important area to fully understand this fundamental aspect of software testing, and we certainly do not claim to have completely answered all questions in this chapter. We do, however, believe our findings significantly improve our understanding of coverage criteria, their relationship to each other and to fault revelation. Our primary contributions are to expose and refute the Clean Program Assumption, and to present the results of a larger-scale empirical study that does not rest on this assumption. The most important finding from this robust empirical study is the evidence for the apparent superiority of strong mutation testing and the observation of threshold behaviour, below which improved coverage has little or no effect on fault revelation.

4.2 Test Adequacy Criteria

Although there is a large body of work that crucially relies upon test adequacy criteria [Ana+13; Ber07], there remain comparatively few studies in the literature that address questions related to actual fault revelation (using real faults) to reliably confirm the coverage-based assessment of test thoroughness. We therefore, empirically examine the ability of criteria-guided testing in uncovering faults. We investigate four popular test adequacy criteria; the main two structural criteria (namely statement and branch testing), and the main two fault-based criteria (namely weak and strong mutation testing).

4.2.1 Statement and Branch Adequacy Criteria

Statement testing (aka statement coverage) relies on the idea that we cannot be confident in our testing if we do not, at least, exercise (execute) every reachable program statement at least once. This practice is intuitive and is widely-regarded as a (very) minimal requirement for testing. However, programs contain many different types of elements, such as predicates, so faults may be exposed only under specific conditions, that leave them undetected by statement adequate test suites. Therefore, stronger forms of coverage have been defined [ZHM97]. One such widely-used criteria, commonly mandated in industrial testing standards [Rad92; Rei95] is branch coverage (or branch testing). Branch testing asks for a test suite that exercises every reachable branch of the Control Flow Graph of the program. Branch testing subsumes statement testing, which only asks for a test suite that exercises every node of the graph.

4.2.2 Mutation-Based Adequacy Criteria

Mutation testing deliberately introduces artificially-generated defects, which are called ‘mutants’. A test case that distinguishes the behavior of the original program and its mutant is said to ‘kill’ the mutant. A mutant is said to be weakly killed [VM97; AO08; PM10a], if the state of computation immediately after the execution of the mutant differs from the corresponding state in the original program. A mutant is strongly killed [VM97; AO08; PM10a] if the original program and the mutant exhibit some observable difference in their output behaviour. Strong mutation does not subsume weak mutation because failed error propagation [PM10a; And+14] may cause state differences to be over-written by subsequent computation.

For a given set of mutants, M , mutation coverage entails finding a test suite that kills all mutants in M . The proportion of mutants in M killed by a test suite T is called the mutation score of T . It denotes the degree of achievement of mutation coverage by T , in the same way that the proportion of branches or statements covered by T denotes its degree of branch or statement adequacy respectively.

Previous research has demonstrated that mutation testing results in strong test suites, which have been empirically observed to subsume other test adequacy criteria [AO08]. There is also empirical evidence that mutation score correlates with actual failure rates [And+06; Jus+14] indicating that, if suitable experimental care is taken, then these artificially-seeded faults can be used to assess the fault revealing-potential of test suites.

4.3 Research Questions

Our first aim is to investigate the validity of the ‘Clean Program Assumption’, since much of our understanding of the relationships between test adequacy criteria rests upon the validity of this assumption. Therefore, a natural first question to ask is the extent to which experiments with faults, when performed on the “clean” (fixed) program versions, provide results that are representative of those that would have been observed if the experiments had been performed on the “faulty” program versions. Hence we ask:

RQ1: *Does the ‘Clean Program Assumption’ hold?*

Given that we did, indeed, find evidence to reject the Clean Program Assumption, we go on to investigate the relationship between achievement of coverage and fault revelation, using a more robust experimental methodology that does not rely upon this assumption. Therefore, we investigate:

RQ2: *How does the level of fault revelation vary as the degree of the coverage attained increases?*

Finally, having rejected the Clean Program Assumption, and investigated the relationship between fault revelation for adequate and partially adequate coverage criteria, we are in a position to compare the different coverage criteria to each other. Therefore we conclude by asking:

RQ3: *How do the four coverage criteria compare to each other, in terms of fault revelation, at varying levels of coverage?*

The answers to these questions will place our overall understanding of the fault-revealing potential of these four widely-used coverage criteria on a firmer scientific footing, because they use real-world faults and do not rely on the Clean Program Assumption.

4.4 Research Protocol

Our study involves experiments on mature real-world projects, with complex real faults, developer, machine-generated and manually-written tests. All these tests yields a pool from which we sample, to experimentally select different coverage levels, while controlling for test suite size (number of test cases). Our experimental procedure follows the following five steps:

1. We used CoREBench, a set of real faults that have been manually identified and isolated, using version control and bug tracking systems in the previous work by Böhme and Roychoudhury [BR14]. Böhme and Roychoudhury with the introduction of CoREBench have created a publicly available set of real-world bugs on which others, like ourselves, can experiment.
2. We extracted the developer tests for each of the faults in CoREBench.
3. We generated test cases covering (at least partially) all the faults using the state-of-the-art dynamic symbolic execution test generation tool, KLEE [CDE08; PKC16].
4. We manually augmented the developer and automatically generated test suites that were obtained in the previous steps. To do so we used the bug reports of the faults and generated additional test cases to ensure that each fault can potentially be revealed by multiple test cases from the test pool. The combined effect of Steps 2, 3 and 4 is to yield an overall test pool that achieves both high quality and diversity, thereby facilitating the subsequent selection step.
5. We perform statement, branch, weak and strong mutation testing, using multiple subsets selected from the test pool (constructed in the Steps 2, 3 and 4), using sampling with uniform probability. Test suites for varying degrees of coverage according to each one of the four criteria were constructed for all faulty programs, one per fault in CoREBench, in order to avoid the Clean Program Assumption.

Table 4.1: *The subject programs used in the experiments. For each of them, the number of test cases (TC), their size in lines of code and number of considered faults are presented.*

Program	Size	Developer TC	KLEE TC	Manual TC	Faults
Coreutils	83,100	4,772	13,920	27	22
Findutils	18,000	1,054	3,870	7	15
Grep	9,400	1,582	4,280	37	15
Make	35,300	528	138	25	18

4.4.1 Programs Used

To conduct our experiments it is important to use real-world programs that are accompanied by relatively good and mature test suites. Thus, we selected the programs composing the CoREBench [BR14] benchmark: “Make”, “Grep”, “Findutils”, and “Coreutils”. Their standardized program interfaces were helpful in our augmentation of the developers’ initial test suites, using automated test data generation. Furthermore, the available bug reports for these programs were helpful to us in the laborious manual task of generating additional test cases.

Table 4.1 records details regarding our test subjects. The size of these programs range from 9 KLoC to 83KLoC and all are accompanied by developer test suites composed of numerous test cases (ranging from 528 to 4,772 test cases). All of the subjects are GNU programs, included in GNU operating systems and typically invoked from the command line (through piped commands). Grep is a tool that processes regular expressions, which are used for text matching and searching. The Make program automates the source code building process. Findutils and Coreutils are each collections of utilities for, respectively, searching file directories and manipulating files and text for the UNIX shell.

4.4.2 CoREBench: realistic, complex faults

To conduct this study we need a benchmark with real-world complex faults that can be reliably used to evaluate and compare the four coverage criteria we wish to study. Unfortunately benchmarks with real errors are scarce. CoREBench [BR14] is a collection of 70 systematically isolated faults, carefully extracted from the source code repositories and bug reports of the projects we study.

The most commonly-used benchmarks are the Siemens Suite and the Software Infrastructure Repository SIR [Hut+94; DER05], but sadly neither can help us to answer our particular chosen research questions. While the Siemens suite has been widely used in previous studies, the degree to which generalisation is possible remains limited, because the programs are small, and cannot truly be said to be representative of real-world systems. The SIR repository overcomes this limitation, because it contains real-world programs, and is a very valuable resource. Nevertheless, many of the faults collected for the SIR programs are artificially seeded faults. This repository is thus less relevant to our study, because we seek to study the relationship between such artificially seeded faults and real faults as part of our set of research questions.

The CoREBench benchmark we chose to use was built by analysing 4,000 commits, which led to the isolation and validation (through test cases) of 70 faults [BR14]. Every fault was identified

by exercising the project commits with validating test cases that reveal the faults. Thus, the test cases pass on the versions before the bug-introducing commit and fail after the commit. Also, the test cases pass again after the fixing commit. Further details regarding the benchmark can be found in the CoREBench paper by Böhme and Roychoudhury [BR14] and also on its accompanying website¹.

When conducting our analyses, we also verified the faulty and fixed versions using both the developer and additionally generated (either manually or automatically) test cases (details regarding the test suites we used can be found in Section 4.4.3). As the “faulty” and “fixed” program versions were mined from project repositories by analysing commits, they had differences that were irrelevant to the faults we study. While, these were only a few, they could potentially bias our results because they might arbitrarily elevate the number of program elements to be covered (due to altered code unrelated to the fault). Thus, we removed this irrelevant code, using the test suites, as behaviour-preserving indicators, using delta debugging [ZH02] to minimise the differences between the “faulty” and “fixed” versions.

Finally, we excluded nine faults from our analysis due to technical problems. Faults with CoREBench identifiers 57 and 58 for the Make program failed to compile in our environment. Also we had technical problems forming the annotations for (Make) faults with identifiers 64 and 65 and thus, KLEE could not create additional test suites for these faults. Fault 42 of Grep, 33 and 37 of the Findutils and 60, 62 of Make took us so much execution time that we were forced to terminate their execution after 15 days.

4.4.3 Test Suites Used

The developer test suites for all the projects we studied were composed of approximately 58,131 tests in total. As these were not always able to find the faults (because in this case bugs would have been noticed before being reported), the authors of CoREBench designed test cases that reveal them (typically only one test to expose each bug). However, we not only need to expose the bugs, but also to expose them multiple times in multiple different ways in order to allow our uniform test suite selection phase to benefit from a larger and more diverse pool from which to select.

Therefore, to further strengthen the test suites used in our study, we augment them in a two-phase procedure. In the first phase we used KLEE, with a relatively robust timeout limit of 600 seconds per test case, to perform a form of differential testing [ES07] called shadow symbolic execution [PKC16], which generates 22,208 test cases. Shadow symbolic execution generates tests that exercise the behavioural differences between two different versions of a program, in our case the faulty and the fixed program versions. We guided shadow symbolic execution by manual annotations to the subject programs that have no side-effects.

Unfortunately, the current publicly available version of the tool KLEE does not yet handle calls to system directories, i.e., test cases involving system directories, rendering it inapplicable to many cases of the Findutils and Make programs. Also, due to the inherent difficulty and challenge of the test data generation problem, we could not expect, and did not find, that KLEE was able to expose differences between every one of the pairs of original and faulty programs. Therefore, in a second phase we manually augment the test suites, using the bug reports (following the process

¹<http://www.comp.nus.edu.sg/~release/corebench/>

of Böhme and Roychoudhury [BR14]), designing 96 additional test cases that reveal (and that fail to reveal) the bugs. We manually generate tests in all situations where there are either fewer than five test cases that reveal a given fault or fewer than five that cover the fault but fail to reveal it, thereby ensuring that all faults have at least five revealing and five non-revealing test cases.

Our experiments were performed at the system level and involved 323,631 mutants, 53,716 branches and 77,151 statements. Every test exercises the entire program as invoked through the command line (rather than unit testing, which is less demanding, but vulnerable to false positives [GFZ12]). As a result, both automated and manual test generation were expensive. For example, the machine time that was spent on symbolic execution took approximately 1 day, on average, for each studied bug. All the test execution needed for our experiment took approximately 480 days of computation time to complete (of single-threaded analysis).

Following the recommendations of Xuan *et al.* [Xua+16] we refactored² the test cases we used to improve the accuracy of our analysis. This practice also helps to elevate the performance of symbolic execution [Gro+16]. Finally, each test ‘case’ is essentially a test *input* that needs a test oracle [Bar+15b], in order to determine its corresponding output. Fortunately, in our case, we have a reliable and complete test oracle: the output differences between the fixed and the faulty versions.

Overall, the coverage scores levels achieved by the whole test pool are presented in Figure 4.1.

4.4.4 Tools for Mutation Testing and Coverage Measurement

To conduct our experiment we used several tools in addition to the shadow symbolic execution [PKC16] feature implemented³ on top of KLEE [CDE08]. To measure statement and branch coverage we used the GNU Gcov utility. To perform mutation, we built a new tool on top of the Frama-C framework [Kir+15] as existing tools are not robust and scalable enough to be applied on our subjects. This tool supports both weak and strong mutation, by encoding all the mutants as additional program branches [PM11; PM12; Bar+15a] (for weak mutation testing), and uses

²Many test cases form a composition of independent (valid) test cases. We split these tests and formed multiple smaller and independent ones, which preserve their semantics.

³<http://srg.doc.ic.ac.uk/projects/shadow/>

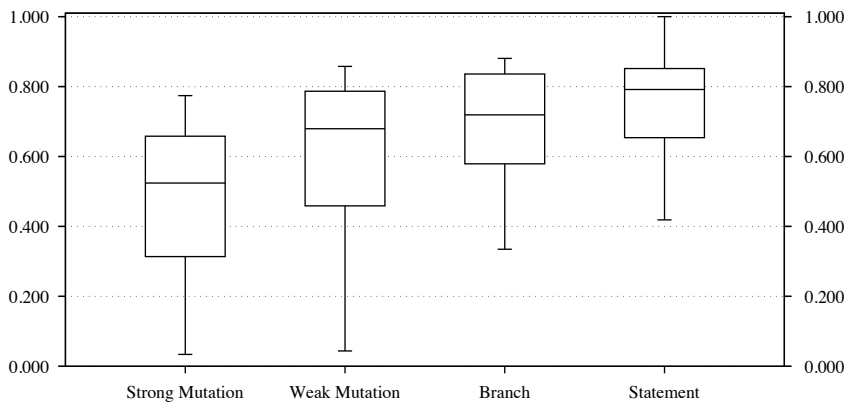


Figure 4.1: The test pool with overall coverage score values.

program wrappers, similar to those used by shadow symbolic execution, that automatically and precisely record the program outputs (for strong mutation testing).

Our mutation tool reduces the execution cost of strong mutation by checking for strong death, only those mutants that were already weakly killed [PM10a], since any mutant that is not weakly killed by a test case cannot be strongly killed, by definition. We also used the recently-published TCE (Trivial Compiler Equivalence) method [Pap+15] to identify and remove strongly equivalent and duplicated mutants, detected by TCE.

We use a timeout in order to avoid the infinite loop problem: a mutant may lead to an infinite loop, which evidently cannot be detectable in general, due to the undecidability of the halting problem. In this way, we are treating (sufficient difference of) execution time as an observable output for the purpose of strong mutation testing. Thus, a mutant is deemed to be distinct from the original program if its execution differs by more than two times the execution of the original program.

The mutation tool includes the (large and varied) set of mutant operators used in previous research [Pap+15; And+06; Jus+14]. Specifically, we used mutants related to arithmetic, relational, conditional, logical, bitwise, shift, pointers and unary operators. We also used statement deletion, variable and constant replacement.

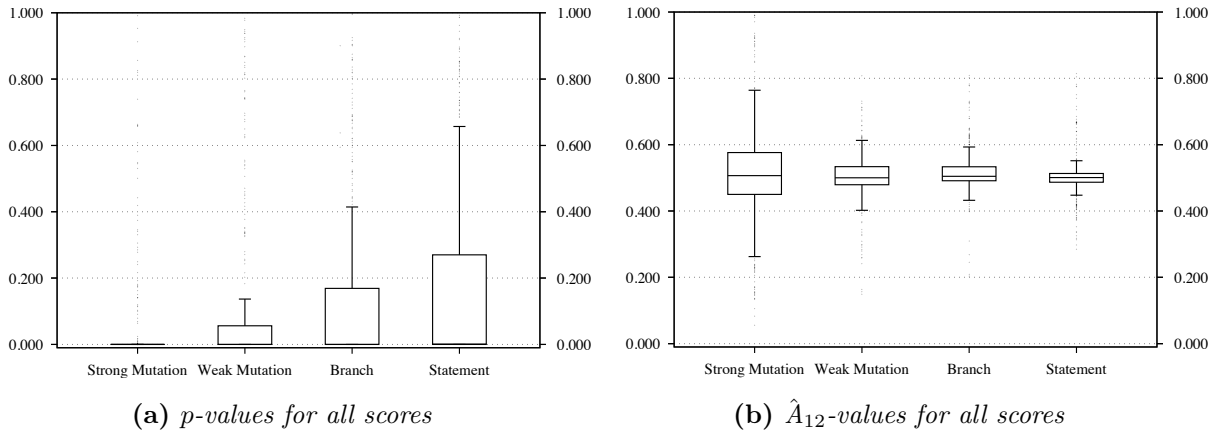


Figure 4.2: RQ1: Comparing the “Faulty” with the “Clean” (‘Fixed’) programs. Our results show that there is statistically significant difference between the coverage values attained in the “Faulty” and “Clean” programs (subfigure 4.2a) with effect sizes that can be significant (subfigure 4.2b).

4.4.5 Analyses Performed on the Test Suites

To answer our research questions we performed the following analysis procedure. We constructed a coverage-mutation matrix that records the statements and branches covered and mutants killed by each test case of the test pool.

For RQ1 we select arbitrary test sets, execute them in both the fixed (clean) and faulty versions and measure their coverage and mutation scores. We used the Wilcoxon test to compare these values. In order to facilitate inferential statistical testing, we repeat the sampling process 10,000 times so that, for each fault and for each coverage criterion, we perform 10,000 testing experiments, each with a different sampled test suite. The Wilcoxon test is a non-parametric test and thus, it is suitable for samples having unknown distribution [AB11; Woh+00]. The statistical test

allows us to determine whether or not the Null Hypothesis (that there is no difference between the test coverage achieved for the clean and faulty versions of the program) can be rejected. If the Null Hypothesis is rejected, then this provides evidence that the Clean Program Assumption does not hold.

However, statistical significance does not imply practical significance; even when the assumption does not hold, if the effect of assuming it is found to be always small, then the pernicious effects (on previous and potential future experiments) may also be small. Therefore, we also measured the Vargha Delaney effect size \hat{A}_{12} (Section 2.5.3).

To study further the differences between the faulty and the fixed program versions, we use the notion of coupling [Jus+14; Off92; PT15]. A fault is coupled with a mutant, statement or branch if every test that kills the mutant (respectively covers the statement or branch) also reveals the fault. Thus, if, for example, a statement is coupled with a fault, then every test set that covers this statement will also reveal this fault. Unfortunately, computing the exact coupling relations is infeasible since this would require exhaustive testing (to consider every possible test set). However, should we find that a fault, f remains uncoupled with all mutants, statements or branches then this provides evidence that the adequacy criterion is not particularly good at uncovering f . Based on the coupled faults we can provide further evidence related to the Clean Program Assumption. If we observe many cases where faults are coupled in one version (either faulty or fixed) while not in the other, then we have evidence against the assumption.

To answer RQ2 and RQ3 we examined the relation between coverage score and fault revelation by selecting test sets of equal size (number of tests). We thus, select 10,000 suites of sizes 2.5%, 5%, 7.5%, 10%, 12.5%, and 15% of the test pool (composed of all developer, machine and manually generated test cases). Then, for every score, c_i , in the range $[0, \text{maximum recorded score}]$, we estimate the average fault revelation rate for all the tests that have coverage values at least c_i . This rate estimates the probability that an arbitrary $c_i\%$ -adequate test suite detects a fault.

We then compare these fault revelation probabilities for different levels of minimal coverage attainment. Ideally, we would like to control both test size and coverage across the whole spectrum of theoretically possible coverage levels (0-100%). However, since coverage and size are dependent it proved impossible to do this, i.e., large test sizes achieve high coverage, but not lower, while smaller sizes achieve lower coverage but not higher. Therefore, to perform our comparisons we record the highest achieved scores per fault we study. For RQ2 we compared the fault revelation of scores for arbitrary selected test suites with those of the highest 20%, 10%, and 5% coverage attainment (of same size). For RQ3 we compared the fault revelation of the criteria when reaching each level of coverage in turn. To perform the comparisons we used three metrics: a Wilcoxon test to compare whether the observed differences are statistically significant, the Vargha Delaney \hat{A}_{12} for the statistical effect size of the difference and the average fault revelation differences. Finally, to further investigate RQ3, we also compare the number of faults that are coupled with the studied criteria according to our test pool.

Table 4.2: The influence of coverage thresholds on fault revelation for test suite size 7.5% of the test pool. All the coverage levels below the highest 20% are not significant. Sub-table (a) records fault revelation at highest $x\%$ coverage levels and sub-table (b) the results of a comparison of the form “rand” (randomly selected test suites) VS “highest $x\%$ ” (test suites achieving the highest $x\%$ of coverage), e.g., for Branch and highest 20% the \hat{A}_{12} suggests that Branch provides a higher fault revelation in its last 20% coverage levels in 53% of the cases with average fault revelation difference of 1.4%.

Test Criterion	Av Fault Revelation		
	highest 20%	highest 10%	highest 5%
Statement	0.507	0.523	0.541
Branch	0.512	0.530	0.553
Weak Mutation	0.501	0.523	0.541
Strong Mutation	0.551	0.625	0.674

(a) Fault Revelation for Higher Coverage Test-suites

Test Criterion	rand vs. highest 20%			rand vs. highest 10%			rand vs. highest 5%		
	$p - value$	\hat{A}_{12}	Av diff	$p - value$	\hat{A}_{12}	Av diff	$p - value$	\hat{A}_{12}	Av diff
Statement	0.673	0.478	-0.009	0.408	0.456	-0.024	0.230	0.437	-0.041
Branch	0.527	0.467	-0.014	0.374	0.453	-0.031	0.123	0.419	-0.054
Weak Mutation	0.978	0.498	-0.001	0.619	0.474	-0.023	0.388	0.455	-0.041
Strong Mutation	0.163	0.427	-0.054	0.009	0.364	-0.128	0.001	0.334	-0.176

(b) Fault Revelation of Random vs Higher Coverage Test-suites

4.5 Experimental Results

4.5.1 RQ1: Clean Program Assumption

The Clean Program Assumption relies on the belief that the influence of faults on the program behaviour is small. However, white-box adequacy criteria depend on the elements to be tested [VM97]. Thus, faulty and clean programs have many different test elements simply because their code differs. Unfortunately, applying experiments to the clean version does not tell us what would happen on the program execution (of the same test) of the faulty program versions. Therefore, we seek to investigate the differences in the coverage scores of test suites when applied to the clean and the faulty programs.

The results of our statistical comparison (p -values) between the coverage scores obtained from the faulty and clean programs are depicted in Figure 4.2a. These data show that all measures differ when applied on the clean rather than the faulty program versions.

These differences are significant (at the 0.05 significance level) for all four criteria and for 86%, 74%, 66% and 60% of the cases for strong mutation, weak mutation, branch and statement coverage respectively. Strong mutation differences are more prevalent than those of the other criteria indicating that the Clean Program Assumption is particularly unreliable for this coverage criterion.

The results related to the effect sizes are depicted in Figure 4.2b, revealing that large effect sizes occur on all four criteria. Strong mutation has larger effect sizes than the other criteria, with some extreme cases having very high or low \hat{A}_{12} values.

One interesting observation from the precedent results is that the faults do not always have the same effect. Sometimes they decrease and sometimes they increase the coverage scores. It is noted that the effect sizes with \hat{A}_{12} values higher than 0.5 denote an increase of the coverage, while below 0.5 denote a decrease. Therefore, the effect of the bias is not consistent and thus, not necessarily predictable.

To further investigate the nature of the differences we measure the couplings between statements, branches and mutants with the faults. Figure 4.3 presents a Venn diagram with the number of coupled faults in the “Faulty” and the “Clean” versions. We observe that 10, 12, 6, and 4 couplings (represent 16%, 20%, 10% and 7% of the considered faults) are impacted by the version differences when performing statement, branch, weak mutation and strong mutation testing.

We also observe that for statement, branch, weak and strong mutation, 1, 2, 2, and 2 faults are coupled only to test criteria elements on the faulty versions, while 9, 10, 4 and 2 faults only coupled on the clean versions. Interestingly, in the clean versions branch coverage performs better than weak mutation (couples with 37 faults, while weak mutation with 33), while in the faulty version it performs worst (couples with 29, while weak mutation with 31). These data, provide further evidence that results drawn from the two programs can differ in important ways, casting significant doubts on the reliability of the Clean Program Assumption.

4.5.2 RQ2: Fault revelation at higher levels of coverage

The objective of RQ2 is to investigate whether test suites that reach higher levels of coverage (for the same test suite size) also exhibit higher levels of fault revelation. To answer this question we selected 10,000 arbitrary test suites (per fault considered) using uniform sampling so that they all have the same test size. We then compare their fault revelation with that of the tests suites that achieve the highest levels of coverage. Thus, we compare with test suites that lie in the top 5%, 10% and 20% of coverage, to investigate different levels of maximal coverage attainment.

Table 4.2 records the results for the controlled test size equal to 7.5% of the test pool, which are representative of those we attained with the other sizes, i.e., 2.5%, 5%, 10%, 12.5% and 15%. Overall, our data demonstrate that all criteria do not exhibit any significant improvement in their fault revelation when considering the threshold of the highest 20% (all p -values are above the 0.05). This is also true for lower coverage thresholds, i.e., when considering the highest 25%, 30% etc. The fact that the fault revelation differences are not significant indicates that test sets having coverage values lying within the highest 20% cannot be said to find significantly more faults than arbitrary test sets of the same size.

The surprising finding is that fault revelation does not improve when test suites achieve the top 20% of the levels of coverage for a given test suite size for *any* of the four criteria. However, for strong mutation, and only for strong mutation, we *do* observe more significant differences when the top 10% and the top 5% of coverage are attained. Furthermore, for strong mutation, the average fault revelation rate was 5% higher than the arbitrary test sets (for the highest 20%). This increases to approximately 13% and 18% when considering the test suites that had the highest 10% and 5% coverage attainment.

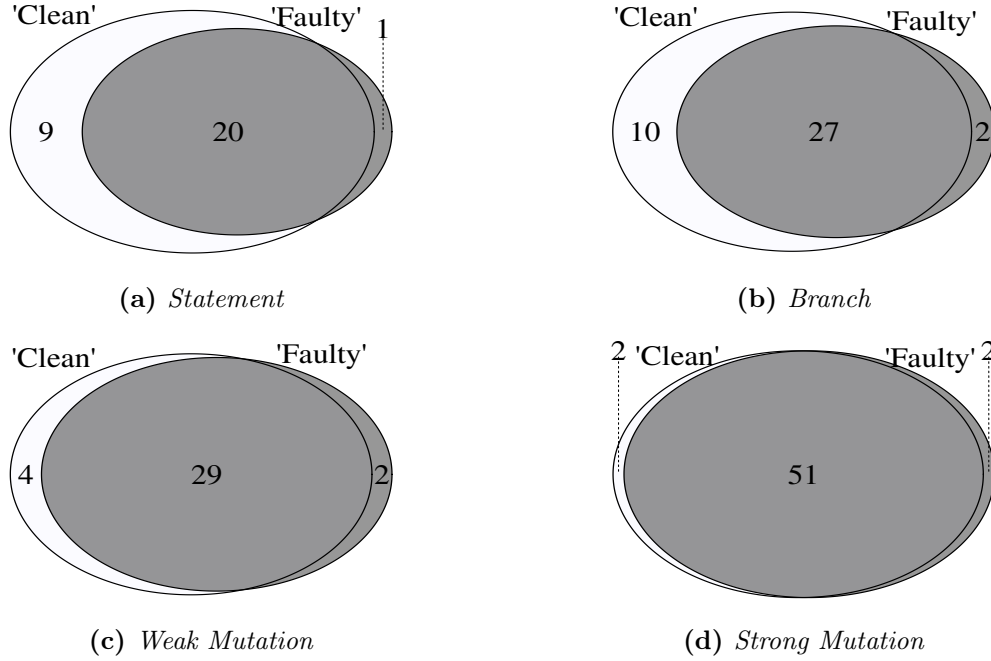


Figure 4.3: Fault coupling in the 'Faulty' and 'Clean' versions.

As can also be seen from the results, confining our attention to only the top 10% (and even the top 5%) levels of coverage attainable for a given test suite size does not produce any such improvement in fault revelation for the other three criteria. That is, test suites with the higher 10% and higher 5% of coverage attainment for statement, branch and weak mutation in Table 4.2 do not exhibit statistically significantly higher fault revelation than arbitrary test suites of the same size.

By contrast, test suites that lie within the highest 10% and 5% for strong mutation do achieve significantly higher fault revelation than arbitrary test suites of the same size. For both the 10% and 5% thresholds, the differences are statistically significant at the 0.05 level, and also exhibit relatively strong effect sizes (the Vargha Delaney effect size measures, of 0.364 and 0.334 respectively, are noticeably lower than 0.5). Furthermore, at the highest 5% the p -value is lowest and the effect size largest. This p -value remains significant at the 0.05 level after the (highly conservative) Bonferroni correction.

Taken together, these results provide evidence that test suites that achieve strong mutation coverage have higher fault revelation potential than those that do not, while there is no such evidence for statement, branch and weak mutation. Finally, we notice that relatively high levels of strong mutation are required (top 10%) for this effect to be observed; below this threshold level, differences between arbitrary and (partially) strong mutation adequate test suites are insignificant.

4.5.3 RQ3: Fault Revelation of Statement, Branch, Weak and Strong Mutation

RQ2 compared arbitrary test suites with higher adequacy test suites of the same size for each coverage criterion. This answered the *within-criteria* question, for each criterion, of whether

Table 4.3: Comparing fault revelation for the highest 5% coverage threshold and test suite size of 7.5% of the test pool.

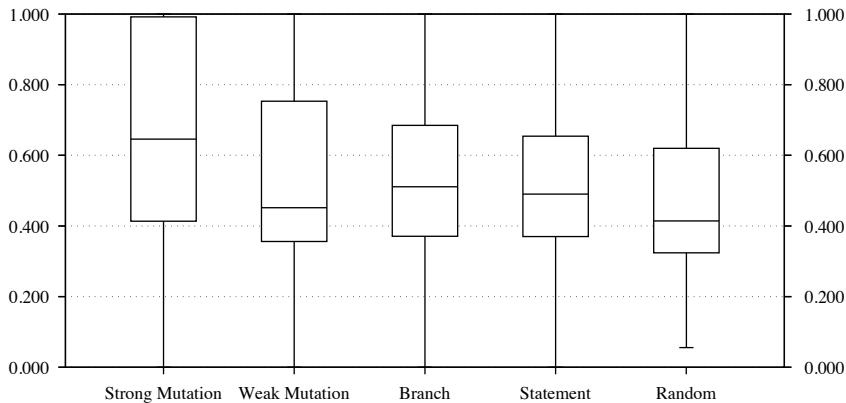
Criteria comparison	$p - val$	\hat{A}_{12}	Av Fault Revelation Diff
Strong Mut vs Statement	0.013	0.630	0.134
Strong Mut vs Weak Mut	0.025	0.618	0.122
Strong Mut vs Branch	0.012	0.631	0.134
Weak Mut vs Statement	0.836	0.489	-0.001
Weak Mut vs Branch	0.570	0.470	-0.013
Branch vs Statement	0.722	0.519	0.012

increasing coverage according to the criterion is beneficial. However, it cannot tell us anything about the differences in the faults a tester would observe *between criteria*, a question to which we now turn.

Table 4.3 reports the differences in pairwise comparisons between the four criteria, for test suites containing 7.5% of the overall pool of test cases available; the same size test suites we used to answer RQ2. Results for other sizes of test suites are similar, but space does not permit us to present them all here. We report p -values without correction for multiple statistical testing, since we are simply interested in the relative differences between each coverage criteria, rather than determining statistical significance (p -values below some predetermined α -level threshold).

The results from this analysis suggest that there are no significant differences between the fault revelation achieved by statement, branch and the weak mutation, when compared to one another. The results also indicate that fault revelation achieved by strong mutation is likely to outperform all other criteria, i.e., weak mutation, branch and statement coverage. Figure 4.4 visualises these results (fault revelation of the four criteria and randomly selected test suites) and demonstrate the superiority of strong mutation over the other criteria.

Finally, Figure 4.5 shows the faults coupled uniquely (and jointly) to each of the four adequacy criteria. This provides another view of the likely behaviour of test suites that target each of these coverage criteria with respect to the real-world faults considered. Each region of the Venn

**Figure 4.4:** Fault Revelation of the studied criteria for the highest 5% coverage threshold and test suite size of 7.5% of the test pool.

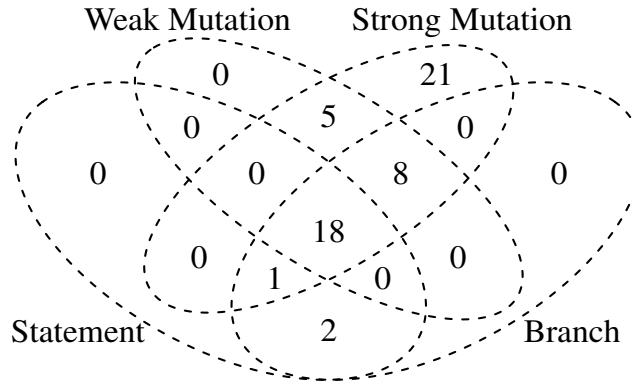


Figure 4.5: Fault coupling between the studied criteria.

diagram corresponds to an intersection of different coverage criteria, and the number recorded for each region indicates the number of faults coupled by the corresponding intersection of criteria. This allows us to investigate the faults that are uniquely coupled by each criterion, and those coupled jointly by pairs, triples and quadruples of criteria.

In the fault dataset we study there are 61 faults, 6 are not coupled to any of the criteria and 18 are coupled to all four criteria (the quadruple of criteria region depicted in the centre of the Venn diagram). It is interesting that all faults coupled by weak mutation are also coupled by strong mutation, since strong mutation does not subsume weak mutation. Branch coverage theoretically subsumes statement coverage, but only when 100% of the feasible branches and 100% of the reachable statements are covered; there is no theoretical relationship between partial branch coverage and partial statement coverage. Therefore, it is interesting that, for our partially adequate test suites, all faults are coupled with statement coverage are also coupled with branch coverage. By contrast, 3 faults coupled to branch coverage are not to weak mutation (one of which is coupled to strong mutation), while weak mutation has 5 faults coupled that are uncoupled with branch coverage.

However, differences between statement, branch and weak mutation are relatively small by comparison with the differences we observe between strong mutation and the other three criteria. Indeed, Figure 4.5 provides further compelling evidence for the superiority of strong mutation testing over the other coverage criteria. As can be seen, 21 faults are uniquely coupled to strong mutation. That is, 21 faults are coupled to strong mutation that are not coupled to *any* of the other criteria (showing that strong mutation uniquely couples to 38% of faults that are coupled to any of the four criteria). By contrast, each of the other three criteria has *no* faults uniquely coupled, and even considering all three together only have two faults that are not coupled to strong mutation. These faults are only coupled to branch and statement coverage.

4.6 Threats to Validity

As in every empirical study of programs, generalisation remains an open question, requiring replication studies. We used C utility programs. Programs written in other languages and with different characteristics may behave differently. All four programs we used are “well-specified, well-tested, well-maintained, and widely-used open source programs with standardized program interfaces” [BR14] with bug reports that are publicly accessible. Our results may generalise

to other well-specified, well-tested, well-maintained, and widely-used open source C programs, but we have little evidence to generalise beyond this. Additional work is required to replicate and extend our results, but clearly any future work should either avoid the Clean Program Assumption or first investigate its veracity for the selected pool of subjects.

Another potential threat to the validity of our findings derives from the representativeness of our fault data. We used real faults, isolated by Böhme and Roychoudhury [BR14] and used by other researchers [TR15; PKC16]. Since these faults were found on well-tested widely-used programs, we believe that they are representative of faults that are hard to find, but further research is required to test this belief.

The use of automatically generated and manually augmented test suites also poses a threat to generalisability. While we cannot guarantee the representativeness of this practice, it is desirable in order to perform experiments involving multiple comparisons that use a good mix of tests that reveal (and fail to reveal), the faults studied. We control for test suite size and different levels of achievement of test adequacy, and perform multiple samples of test suites to cater for diversity and variability. Nevertheless, we cannot claim that the test suites we used are necessarily representative of all possible test suites.

We restricted our analysis to the system level testing, since the developers' tests suites were also system level tests and we used a wide set of mutation operators, included in most of the existing mutation testing tools, as suggested by previous research [And+06; Jus+14; AO08; Pap+15]. We view this as an advantage, because, according to Gross *et al.* [GFZ12], applying testing at the system level makes robust experimentation that reduces many false alarms raised when applying testing on the unit level, while focusing on a narrower set of mutation operators would tend to increase threats to validity. However, this decision means that our results do not necessarily extend to unit level testing, nor to other sets of mutation operators.

All statements, branches and mutants that cannot be covered (or killed) by any test in our test pool are treated as infeasible (or as equivalent mutants). This is a common practice in this kind of experiment [Gli+13; Zha+13; And+06; FI98a], because of the inherent underlying decidability problem. However, it is also a potential limitation of our study, like others. Furthermore, since we observe a 'threshold' behaviour for strong mutation, it could be that similar thresholds apply to statement branch and weak mutation criteria, but these thresholds lie above our ability to generate adequate test suites.

There may be other threats related to the implementation of the tools, our data extraction and the measurements we chose to apply, that we have not considered here. To enable exploration of these potential threats and to facilitate replication and extension of our work, we make our scripts, tools and data available⁴.

4.7 Conclusions

We present evidence that the Clean Program Assumption does not always hold: there are often statistically significant differences between coverage achieved by a test suite applied to the clean (fixed) program and to each of its faulty versions, and the effect sizes of such differences can be large. These differences are important as they may change the conclusions of experimental

⁴<https://sites.google.com/site/mikepapadakis/faults-mutants>

studies. According to our data, weak mutation is more effective than branch testing in the faulty programs and less effective in the clean ones. This finding means that future empirical studies should either avoid the Clean Program Assumption, or (at least) treat it as a potential threat to the validity of their findings. Note that this affects the empirical studies that involve techniques that rely on test criteria coverage. The unreliability of the Clean Program Assumption motivated us to reconsider the relationship between four popular test adequacy criteria and their fault revelation. We thus reported empirical results based on an experimental methodology that benefits from enhanced robustness (by avoiding the Clean Program Assumption).

In this study, we provide evidence to support the claim that highest levels of strong mutation testing yield increased fault revelation, while statement, branch and weak mutation testing enjoy no such fault revealing ability. Our findings also revealed non-linearity in the relationship between strong mutation coverage attainment and fault-revealing potential. An important consequence of this non-linearity is that testers will need to have first achieved a threshold level of coverage before they can expect to receive the benefit of increasing fault revelation with further increases in coverage.

Knowing that strong mutation has the highest fault revelation among the studied test adequacy criteria, the next chapter presents an empirical study that evaluates the various mutant quality indicators in the literature and their fault revelation.

MUTANT QUALITY INDICATORS

The question of which are the valuable mutants has received little attention in mutation testing literature. Naturally, the choice of mutants impacts the quality of the performed analysis and has the potential of changing the conclusions of empirical studies. To this end, in this chapter, we collect definitions related to mutant quality indicators and analyze their relations. We identify two classes of indicators, related to individual mutants and to mutant sets. We analyse a large set of mutants from 3,902 (real) faulty program versions, belonging to 40 fault classes, collected from an on-line programming contest. Our analysis categorises mutants as valuable, according to the studied quality indicators, profiles their types and examines the relations between them. Our results suggest that there is a large disagreement between the indicators and that the connection between mutant type, its quality and its ability to reveal faults is weak. Additionally, our findings reveal that the ability of mutants to uncover faults differs significantly across the different fault classes and that some mutant types are well linked (or completely disconnected) to specific fault classes.

This chapter is based on the work published in the following paper:

- Mike Papadakis, Thierry Titchou Chekam, Yves Le Traon. 2018. Mutant Quality Indicators. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2018)*, 2018

Chapter content

5.1	Introduction	52
5.2	Mutant Quality Indicators	53
5.2.1	Unit-based MQIs	53
5.2.2	Set-based MQIs	54
5.3	Experiment Setup	54
5.3.1	Programs and Faults	54
5.3.2	Automated Tools	55
5.3.3	Experimental Procedure	56
5.4	Results	56
5.4.1	Prevalence of mutant quality indicator categories	56
5.4.2	Relations between mutant quality indicators	56
5.4.3	Mutant types and quality indicators	58
5.4.4	Fault classes with no fault revealing mutants	59
5.4.5	Links between mutant types and fault classes	60
5.5	Conclusion	60

5.1 Introduction

The question of what constitutes a good mutant remains controversial and unknown. Naturally, in mutation testing, the ‘quality’ of mutants plays a central role and can have major implications on the performed analysis. For instance, empirical studies may come to biased conclusions if they use all available mutants [Pap+16]. Similarly, the use of restrictive mutant sets may result in a much lower strength testing [Lau+17].

Previous research has investigated this problem by examining the types of mutants. Mutants of specific types are considered as more important than others as they encode test requirements not captured by other mutant types [Off+96a]. Apart from the types of mutants, some early studies set specific ‘quality’ criteria to judge mutants’ quality. Thus, they suggested that mutants quality should be measured through the ‘easiness’ (ratio of valid program inputs that kill the mutant) of killing them. The underlying idea is that easy to kill mutants (killed by most of the test inputs) are not of a particular value.

Other studies suggested that quality mutants are those that are stubborn, i.e., resistant to killing by the test cases that execute them, [HHD99]. Thus, mutants that are hard to infect or propagate, i.e., killed by few test cases that execute them are valuable. The reason is that these mutants go beyond coverage, i.e., the mutants are not killed by coverage-based test cases [YHJ14].

Another way to define mutants’ quality is based their diversity w.r.t to the program input domain. This way good mutants are a subset that is defined w.r.t to a reference set of all mutants. Thus, good mutants are those that are killed by different test cases than other mutants. This means that the selected set of mutants is as much disjoint, in terms of their killing condition, as possible [KPM10]. In other words, disjoint mutants have a minimum overlap between the mutants’ killings. The underlying idea is that a disjoint mutant set should be representative of all mutants, i.e., their killing must result in the killing of all mutants, and at the same time they are the set of the harder to kill than any other alternative set of mutants.

This dissertation (Chapter 6) suggests that quality mutants are those that guide testers towards revealing real faults (see Chapter 6). The underlying idea is that good mutants should lead to test cases that reveal frequent real faults. Thus, instead of covering the whole spectrum of mutants one should cover the mutants that are most likely to be linked with faults.

Given the plethora of the mutant quality indicators, a natural question to ask is whether there are important differences between them and what are the links with fault revelation. In other words, we are interested to see if the indicators agree on which are the valuable mutants and whether these mutants are linked with fault revelation. Answering these questions is important in order to direct future research (such as Chapter 6) and increase the understanding of the mutation testing foundations.

In this chapter we study the relatively differences between the quality indicators. We investigate the types of mutants that they involve and explore the link between different mutant types and different fault classes. We find that all quality indicators identify only a few, less than 10%, mutants as good. We also find that there is a large disagreement, between the indicators, on which are the good mutants. In particular we find that 39%, 42% and 6% of the fault revealing mutants are also subsuming, hard-to-kill and hard-to-propagate, while 17%, 60% and 4% of the subsuming mutants are fault revealing, hard-to-kill and hard-to-propagate.

Perhaps more importantly, we find a weak connection between fault revelation and quality indicators, suggesting the need for specialized approaches targeting the particular class of fault revealing mutants. We also show that the link between mutants and faults differs significantly across fault classes. We demonstrate that almost half of the faults related to missing code are weakly linked with mutants, while 90% of the faults related to OAAN (wrong arithmetic operator used) category are strongly linked with *SCALAR.BINARY* mutants (mutants created by mutation operators that mutate scalar binary expressions). These results suggest that future studies should consider the particular classes of faults targeted by the proposed approaches. Overall, our study increases the understanding on what contributes to the mutants' quality and opens several directions for future research.

5.2 Mutant Quality Indicators

We performed an expert literature review by considering the papers collected in the recent survey of mutation testing [Pap+19]. Our analysis revealed the following two classes of indicators:

5.2.1 Unit-based MQIs

Fault Revealing (F.R.) are the mutants that are killed only by test cases that reveal a fault. Note that this assume a program under test that contains unknown faults, and the testers would then look for the mutants that when killed, a fault is revealed. These mutants are fault revealing mutants and their intuition is that faults have patterns that can be captured by some mutants (the fault revealing ones). These mutants are the ones that are linked with fault revelation. Thus, one should cover only the mutants that are most likely to be linked with frequently occurring (real) faults (refer to Chapter 6). In our experiment, we approximate fault revealing mutants based on test suites (mutants killed only by tests that reveal a fault in the program under test are fault revealing).

We use the notation "F.R.-1.0" for fault revealing mutants. We consider another (more relaxed) class of fault revealing mutants, denoted as "F.R.-0.9" which involves the mutants for which at least 90% of the killing tests are fault revealing tests.

Subsuming mutants are defined based on the subsumption relation and the indistinguished mutants. According to Ammann et al. [ADO14] "one mutant subsumes another if at least one test kills the first and every test that kills the first also kills the second". Indistinguished are two mutants that are always killed by the same tests.

Subsuming are the mutants that are subsumed only by indistinguished mutants. We consider as subsuming, all mutants that are in the leaf nodes of the mutant subsumption graphs [Kur+14], built based on the employed test suites.

Hard-to-kill (Hard) are the mutants that are killed only by a small fraction of test cases. We consider two classes of hard-to-kill mutants. Those that are killed by at most 5% and 2.5% of the available test suites. We denote these as "Hard-0.050" and "Hard-0.025".

Another way to define the hardness to kill is based on the RIP model [AO08]. Thus, hardness can be defined as hardness to reach, infect and propagate. Here, we only consider mutants that are hard to propagate.

Hard-to-propagate (HardP) are the mutants that are killed only by a small fraction of test cases that infect them. We consider two classes of hard-to-propagate, those that are killed by at most 25% and 10% of the test cases that infect them. We denote these as “HardP-0.25” and “HardP-0.10”.

5.2.2 Set-based MQIs

Non-duplicated is the set of mutants that has no indistinguished mutants. We approximate mutant duplication based on the available test suites.

Disjoint/Surface - Minimal/Dominator. is the subset of mutants with the minimum number of subsuming mutants. Conceptually, there are no difference between disjoint/surface and minimal/dominator mutants. The actual differences are thin and are due to the selection procedure. Disjoint mutants are a subset with minimum joint killings, approximated through a greedy heuristic [Pap+16; KPM10]. Surface mutants [Gop+16] are also approximated by a similar heuristic. The minimal/dominator mutants form the actual minimal subset, selected bases on a systematic procedure [Kur+14].

The set-based indicators depend on the individual choice of the individual mutants and cannot be compared with the unit-based ones. Their relations are also well understood and thus, in the rest of the chapter we mainly focus on the unit-based.

5.3 Experiment Setup

5.3.1 Programs and Faults

We used the Codeflaws benchmark [Tan+17] that involves programs selected from an on-line programming contests¹. In Codeflaws, every faulty program version is unique and has two instances, the ‘faulty’ and the ‘fixed’ one. The former regards the rejected, while the later the accepted submission. In total, Codeflaws contains 3,902 faults of 40 defect classes. These programs are of 1 to 322 lines of code and are accompanied by a test suite that was used to test and judge the programs as faulty and fixed. We choose Codeflaws because it contains many, diverse, relatively hard to expose faults.

To conduct a valid experimentation, we augment the available test suites using KLEE [CDE08], a state-of-the-art test generation tool. Although, these test suites greatly increased the cost of our experiment, we considered their use of vital importance as otherwise our results could be subject to “noise effects” (see Chapter 4). Overall, our experiment involved 122,261 test cases, 3,213,543 mutants, whose execution required a total of 8,009 CPU days of computation.

We aim at investigating the link between mutants and faults. Thus, we consider important to focus at hard faults, i.e., fault not revealed by every test case. In total, approximately half of our faults are trivial ones (revealed by a large fraction of test case). Thus, we restrict our analysis on the 1,629 faults that are revealed by less than 25% of the test cases involved.

¹<http://codeforces.com/>

Table 5.1: *Fault Classes*

AST Type	Fault Class	Example
Higher Order	Expression HEXP	$\ominus if(C) \oplus if(C \parallel D)$
	Non-branch Stmt HDMS	$\ominus printf(s); \ominus x = y + 3;$
	Combination HCOM	$\ominus rep(i, n) \oplus for(...)$
	Non-branch Stmt HIMS	$\oplus printf(s); \oplus x = y + 3;$
	Branch Stmt HBRN	$\oplus if(C) \{printf(s); \}$
	Others HOTH	$\ominus g(0); \oplus for(...)\{f(i); \}$
Statement	Function Call SISF	$\oplus scanf("%d", &n);$
	Type STYP	$\ominus int a \oplus long a$
	Control Flow SRIF	$\ominus if(a > b) \oplus if(g(a) > b)$
	Data Flow SISA	$\oplus t = 0$
	Move SMOV	$\ominus f(x); \quad g(x); \quad \oplus f(x);$
Operand	Variable DRWV	$\ominus b = 0; \quad \oplus a = 0;$
	Array DCCA	$\ominus int x[2]; \quad \oplus int x[20];$
	Variable DRVA	$\ominus if(i > 0) \quad \oplus if(k > 0)$
	Constant DCCR	$\ominus if(x > 4) \quad \oplus if(x > 3)$
Operator	Control Flow ORRN	$\ominus if(a > 0) \quad \oplus if(a \geq 0)$
	Arithmetic OAAN	$\ominus v2 -= 2 \quad \oplus v2 += 2$
	Function Call OFPF	$\ominus f("%d", i); \quad \oplus f("%ld", i);$
	Control Flow OILN	$\ominus if(x) \quad \oplus if(x \& \& f(x))$
	Arithmetic OAIS	$\ominus x += y \quad \oplus x += y / 2$

Table 5.1 records the main fault classes in the dataset. It is noted that these 20 classes involve more than 10 fault instances that are revealed by less than 25% of the tests in our test suites. Following the classification scheme of Codeflaws the faults fall into 4 categories. The fault classes are related to faults in operators of expressions, faults in operands of expressions, faults in control-flow related statements (e.g., missing if conditional) and faults in function call statements or other statements. The last column of Table 5.1 records examples of the fault classes (taken from [Tan+17]). These demonstrate the way the example faults were patched, i.e., "-" denotes the statement(s) deleted/modified and "+" the statements added in order to fix the fault.

5.3.2 Automated Tools

We used KLEE to perform test augmentation with the following settings: a two hours time limit per program, a Random Path search strategy, Randomize Fork Enabled, Max Memory 2048, Symbolic Array Size 4096, Symbolic Standard input size 20 and Max Instruction Time of 30 seconds. This resulted in 26,229 test cases. Since the automatically generated test cases do not include any test oracle, we used the programs' fixed version as oracle. Thus, we considered as failing, every test case that resulted in different observable output when executed in the 'faulty' than in the 'fixed' program. Similarly, we identified the killed mutants using the program output.

We use our built mutation testing tool (*Mart*), presented in Chapter 8, that operates on LLVM bytecode. Actually all our metrics and analysis were performed on the LLVM bytecode. We use the

default configuration of *Mart*, which consists in 18 operators, composed of 816 transformation rules. Refer to Chapter 8, sub-section 8.1.2.2 for more information about the mutation operators.

To reduce the influence of redundant and equivalent mutants, we applied TCE [Pap+15] implemented in *Mart*. TCE Detected 523,097 and 934,415 equivalent and redundant mutants.

5.3.3 Experimental Procedure

We start our analysis by forming a pool of all available test cases. We then constructed a mutation-fault matrix that records the mutants killed and faults revealed by each one of the available test cases. We then applied mutation on the faulty program versions so that we are faithful to real settings and avoid making the Clean Program Assumption studied in Chapter 4. We used this matrix to categorize the mutants.

In summary, we form the population of all mutants, identify the mutants' categories and analyze their relations. We consider the relations between the indicators w.r.t to all killable mutants and to mutants of the same type. For the different fault classes, we follow the taxonomy adopted by Codeflaws [Tan+17]. Details about the considered fault classes are recorded on Tables 5.1 and details about the mutant types are recorded in Table 8.1 of Chapter 8. Each mutant type is written as a '-' separated pair of *matching code fragment* and *replacing code fragment*, e.g. "SCALAR.BINARY-DELSTMT" mutation operator matches a scalar binary expression and mutates it by deleting its statement (see Chapter 8 Section 8.1.2.2)

5.4 Results

5.4.1 Prevalence of mutant quality indicator categories

We start our analysis by measuring the prevalence of the mutants that are characterized as good by the studied quality indicators. Table 5.2 records the total number of mutants involved, the ratio and average (per program) number of them, per considered indicators. Interestingly, we can observe that only a small fraction of all mutants (less than 10%) is characterized as good, according to all categories (the only exception is the HardP-0.25).

This finding suggests that the great majority of the mutants are not good and may have undesirable effects on the interpretation of the mutation score. Thus, it is likely that one can achieve a good mutation score by simply killing bad mutants and not the good ones. Unfortunately, this fact can have serious implications on the confidence inspired by mutation testing [Pap+16]. Therefore, a first finding is that the majority of the mutants are bad ones according to every quality indicator.

5.4.2 Relations between mutant quality indicators

Up to this point, our analysis has shown that few mutants are characterized as good by every quality indicator. However, we have seen nothing about the relations between the different categories of the good mutants. In other words we would like to see whether the indicators agree between themselves on which are the good mutants and which are not.

Table 5.2: *Prevalence of mutant categories.*

Category	No. Mutants	Ratio	Program Av.
Fault Revealing (FR-1.0)	44,221	3%	27
Fault Revealing (FR-0.9)	65,809	4%	40
Subsuming	98,709	6%	61
Hard-to-kill (Hard-0.050)	111,442	7%	68
Hard-to-kill (Hard-0.025)	45,286	3%	28
Hard-to-propagate (HardP-0.25)	325,158	21%	200
Hard-to-propagate (HardP-0.10)	137,864	9%	85
Non-Duplicated	321,822	21%	80
Disjoint/Dominator	20,182	1%	12

To investigate this issue we explore the geography of the mutants' population. Thus, we characterize every mutant according to the studied indicators and measure the number of them that belong on the same and different categories. We present these results in a pairwise manner in Figure 5.1. In these diagrams the surface represent the number of mutants that belong to each category. The surfaces have been scaled so that they reflect the actual size relation between the different categories. Thus, we can see that FR-1.0 are less (in number) than the subsuming mutants.

A first observation from Figure 5.1 is that there is a large disagreement, between the indicators, on which are good mutants. In particular we observe that hard-to-propagate mutants is a distinct category, i.e., it has a very small overlap with every other category. We also observe a medium to small overlap of fault revealing with the subsuming and hard-to-kill mutants. Interestingly when relaxing the fault revealing probability to 90% (FR-0.9) results in a movement away from the subsuming or hard-to-kill mutants. These results suggests that not all the mutants are linked to faults. As subsuming mutants represent the whole spectrum of mutants they include many that are not linked with faults. On the contrary fault revealing ones belong to those parts of the spectrum that are linked with the faults and overall these two categories are not the same. Thus, future research should devise techniques to specialize mutants to the targeted domain or faults.

Another interesting result is that hard-to-kill mutants have a large overlap with subsuming mutants. Still they are not the same, but a large proportion of them is included. We continue our analysis by presenting the exact relations in terms of percentages, measured w.r.t to each category.

Fault Revealing VS. Subsuming: Our results suggests that 39% and 27% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also subsuming. Interestingly, 17% and 18% of the subsuming mutants are also fault revealing (with probability equal to 1.0 and 0.9). This results suggest that only a few of the subsuming mutants are linked with the faults and that more than half of all the mutants that are linked with the faults are subsumed (not subsuming!).

Fault Revealing VS. Hard-to-kill: We find that 42% and 29% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-kill (killed with less than 5% of the tests). On the other side, 16% and 17% of the Hard-to-kill mutants (killed with less than 5% of the tests)

are also fault revealing (with probability equal to 1.0 and 0.9). This result suggests that there is a (slightly) stronger link between faults and hard-to-kill than faults and subsuming mutants.

When consider stronger mutants, killed by less than 2.5% of the tests, we find that 18% and 12% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-kill (killed with less than 2.5% of the tests). Considering the inverse relation we find that 18% and 18% of the Hard-to-kill mutants (killed with less than 2.5% of the tests) are also fault revealing (with probability equal to 1.0 and 0.9). This suggests that stronger mutants have a weaken link with the faults than less strong ones.

Fault Revealing VS. Hard-to-propagate: 6% and 7% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 25% of the tests that infect the mutant) and 1% and 1% of the Hard-to-propagate mutants (killed by less than 25% of the tests that infect the mutant) are also fault revealing (with probability equal to 1.0 and 0.9). When consider stronger mutants, 1% and 1% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 10% of the tests that infect the mutant). The inverse relation shows that $\approx 0\%$ and 1% of the Hard-to-propagate mutants (killed by less than 10% of the tests that infect the mutant) are also fault revealing (with probability equal to 1.0 and 0.9). These show that a very weak link exists between the faults and hard-to-propagate mutants.

Subsuming VS. Hard-to-kill: 60% and 38% of the subsuming mutants are also Hard-to-kill (killed with less than 5% and 2.5% of the tests) and 53% and 83% of the Hard-to-kill mutants (killed with less than 5% and 2.5% of the tests) are also subsuming. This relation shows the relatively strong link between the subsuming and hard-to-kill mutants.

Subsuming VS. Hard-to-propagate: 4% and 1% of the subsuming mutants are also Hard-to-propagate (killed by less than 25% and 10% of the tests that infect the mutant) and 1% and 1% of the Hard-to-propagate mutants (killed by less than 25% and 10% of the tests that infect the mutant) are also subsuming. This relation shows the weak link between hard-to-propagate mutants and other categories.

Hard-to-kill VS. Hard-to-propagate: 1% and $\approx 0\%$ of the Hard-to-kill mutants, killed by less than 5% and 2.5% of the tests, are also Hard-to-propagate (killed by less than 25% of the tests that infect the mutant) and $\approx 0\%$ and $\approx 0\%$ of the Hard-to-propagate mutants (killed by less than 25% of the tests that infect the mutant) are also Hard-to-kill (killed by less than 5% and 2.5% of the tests). When consider stronger mutants, 0% and 0% of the Hard-to-kill mutants, killed by less than 5% and 2.5% of the tests, are also Hard-to-propagate (killed by less than 10% of the tests that infect the mutant). 0% and 0% of the Hard-to-propagate mutants (killed by less than 10% of the tests that infect the mutant) are also Hard-to-kill (killed by less than 5% and 2.5% of the tests). This relation also shows the weak link between hard-to-propagate mutants and other categories.

5.4.3 Mutant types and quality indicators

We investigate the link between mutant type (characterized by its syntactic transformation) and quality indicators by checking whether there are types of mutants that are more likely to generate good mutants. We thus, check the types of mutants involved in every category, i.e., the ratio of the good mutants that are of each type. We also measure the ratio of the good mutants

among those generated per considered mutant type. The former case shows the types of mutants composing the good ones, while the later shows whether the type of mutants relates to the good ones.

Figure 5.2 presents the types of mutants involved in the studied categories. For simplicity we have omitted the results of Hard-to-kill-0.025 and Hard-to-propagate-0.10, as they are quite similar to those of Hard-to-kill-0.050 and Hard-to-propagate-0.25. Interestingly, the majority of the mutants are of the same types (the top 4 most prevalent types are the same). Thus, the types of *SCALAR.BINARY-SCALAR.BINARY*, *SCALAR.BINARY-SCALAR.UNARY*, *SCALAR.ATOM-SCALAR.UNARY* and *SCALAR.ATOM-SCALAR.BINARY* cover more than 80% of all the good mutants. However, this is due to the number of mutants that are generated by these operators.

The graphs of Figure 5.3 record the ratios of mutants (of the same type) involved in the studied categories for every considered mutant type. Interestingly, the profiles of the four categories differ significantly. This shows (again) that the indicators disagree between them and characterize different mutants (and different types) as good ones. Interestingly, all types of mutants (with one exception) contributes to all the categories, indicating that all of them are of a value.

We also observe that with a few exceptions the mutant type does not seem to matter much on any category. Regarding the fault revealing mutants, 5 types seems to generate larger proportions of good mutants than the other 12, but overall all types have a similar ratio. Subsuming and hard-to-kill mutants have one type, the *SWITCH-REMOVECASES*, which generates a significantly higher ratio of good mutants. However, beside this type all other mutant types generate similar ratios. The case of hard-to-propagate mutants is a bit different as it involves 6 types with a rather low contribution, while the rest 12 types have similar ratios. Overall, by comparing the results of Figures 5.2 and 5.3 we see that some (few) mutant types are more important than others but overall, all mutant types are important.

5.4.4 Fault classes with no fault revealing mutants

Having investigated the link between mutant type and quality indicators, we turn our analysis on the different fault classes. We thus, investigate which types of faults have no fault revealing mutants. This is important as these cases are faults that are likely to be missed by mutation testing. Overall, in our dataset, we have 462 faulty program versions without any fault revealing mutant. To investigate whether there is any link between fault class and absence of fault revealing mutants we report the percentage of faults (per class of faults) without fault revealing mutants. In order to avoid coincidental results we removed from our dataset every faulty class that includes less than 10 fault instances. This resulted in 20 faulty classes, out of the 40 faulty classes included in the dataset.

Figure 5.5 reports the percentages of the faulty program versions (of the same fault class) with no fault revealing mutants. From these results we observe that 5 classes have a relatively low ratio (with less than 15%) of cases with no fault revealing mutants. 12 classes have ratios between 15%-35%, while 3 classes have a relatively large number of faults (without fault revealing mutants). Thus, we can conclude that mutation is not particularly good at detecting faults of these three classes, (*Operator-ControlFlow-OILN*, *Operand-Array-DCCA* and *HigherOrder-Expression-HEXP*). Interestingly, the OILN and HEXP classes are faults belonging to the general

category of omission faults, i.e., faults due to missing code. Omission faults form a known weakness of code-based techniques [VM97] and thus, having a strong link with more than half of them is important. The other problematic category is the DCCA class that regards the size of arrays indicating the need for mutation operators related to these faults.

5.4.5 Links between mutant types and fault classes

To investigate the link between mutant types and fault classes, we measure, for every considered class, the ratio of faulty programs with fault revealing mutants. Thus, we expect a high ratio when there is a strong link, and a low ratio when there is a weak link. Since every type of mutants involve different number of mutant instances, we also normalize our results with respect to the number of mutants involved. This way we can see whether there are significant differences between the pairs of mutant types and fault classes.

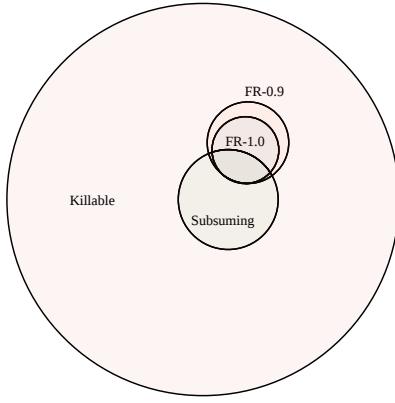
Figure 5.4 presents the ratios of faulty versions with fault revealing mutants for all pairs of mutant type and fault classes. From these plots we can see that some mutant types (*SCALAR.BINARY-SCALAR.BINARY*) are linked with specific fault classes (OAAN), while some mutant types (*POINTER.ATOM-POINTER.UNARY*) are linked with many fault classes. This suggests that for specific cases there is a strong link between mutant type and revealed class of fault. When normalizing with respect to the number of mutants the link is less clear, but strong for specific pairs, such as the mutant types *CALL-SHUFFLEARGS* and *SCALAR.BINARY-DELSTMT* with fault class OAAN.

In conclusion, our results suggests that considering fault classes is important as every class is linked with different mutant types. The differences between mutant types and fault classes provide further evidence that all types of mutants are needed and that there is no dominant type of mutants. Perhaps more importantly, our results reveal that experimental results need to be validated with a diverse class of faults, each one of which should be separately be considered.

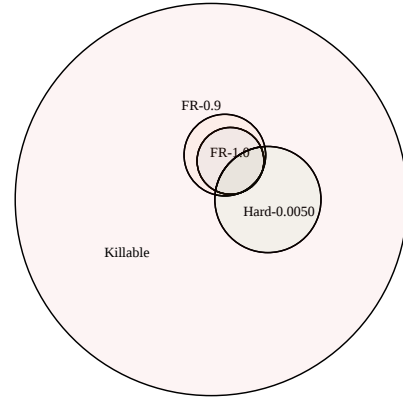
5.5 Conclusion

In this chapter we studied the relatively differences between the mutant quality indicators. We found that all indicators identify only a few (less than 10%) mutants as good and that there is no consensus on which mutants should be considered as good. We also found that all mutant types generate valuable mutants, fact indicating that all types of mutant should be used. Overall, we find that some isolated mutant types contribute more on the good mutants, the general trend is that the discriminative power of the mutant type is limited. Perhaps more importantly, we find a weak connection between the fault revelation and quality indicators, suggesting the need for specialized approaches targeting the particular class of fault revealing mutants. Finally, our results demonstrate that the fault revelation ability of mutants differs significantly across the studied classes of faults, indicating that future studies should consider the particular fault classes they target and are involved in the experimental datasets.

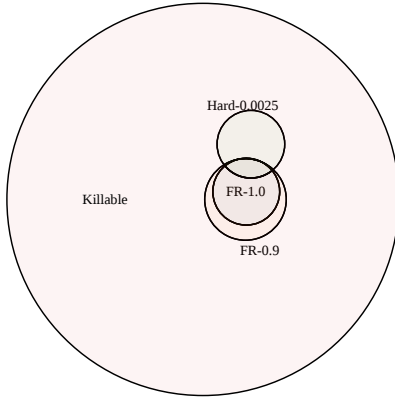
The next chapter presents a mutant reduction technique that targets the fault-revealing mutants.



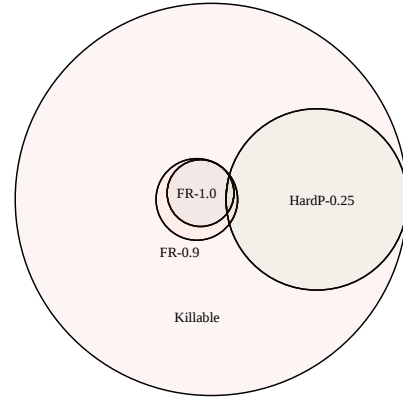
(a) *Fault Revealing VS. Subsuming*



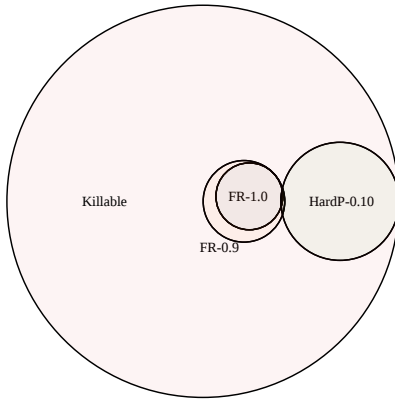
(b) *Fault Revealing VS. Hard-to-kill (5%)*



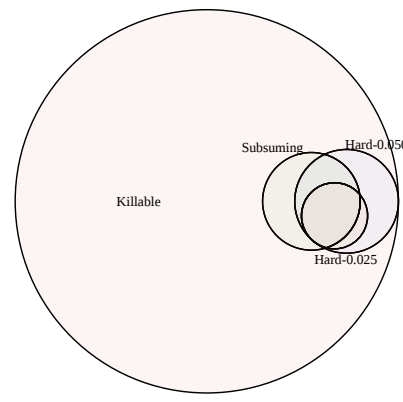
(c) *Fault Revealing VS. Hard-to-kill (2.5%)*



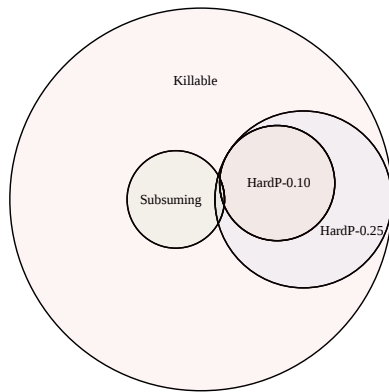
(d) *Fault Revealing VS. Hard-to-prop. (25%)*



(e) *Fault Revealing VS. Hard-to-prop. (10%)*

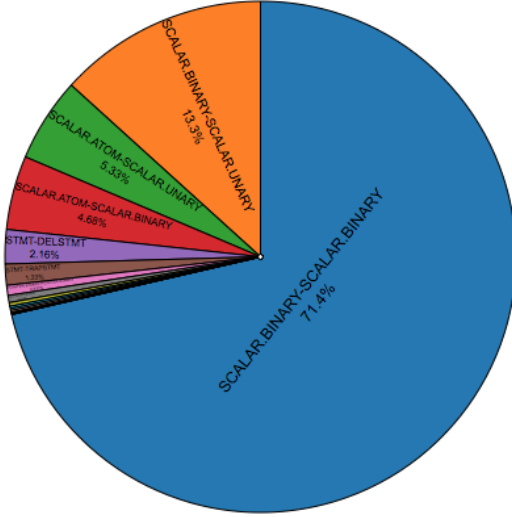
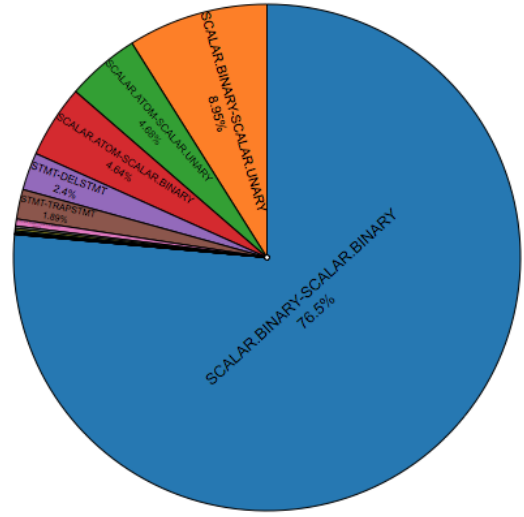
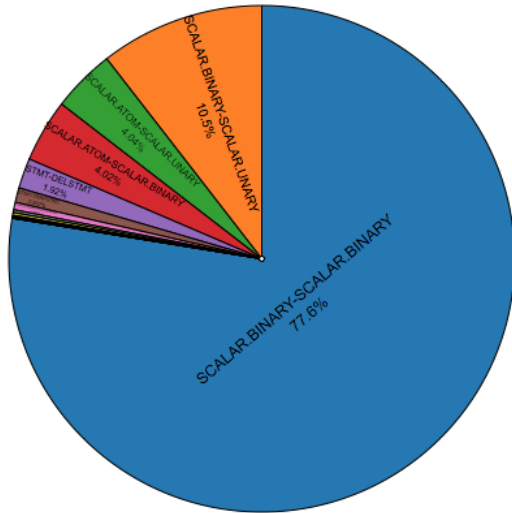
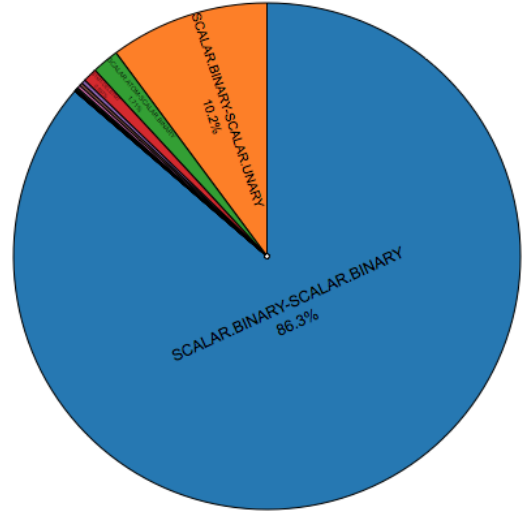


(f) *Hard-to-kill VS. Subsuming*



(g) *Hard-to-prop. VS. Subsuming*

Figure 5.1: *Relations between different mutant quality indicators.*

(a) *Fault Revealing mutants*(b) *Subsuming mutants*(c) *Hard-to-kill mutants (5%)*(d) *Hard-to-prop. mutants (10%)***Figure 5.2:** *Types of mutants involved in the mutant quality indicator categories.*

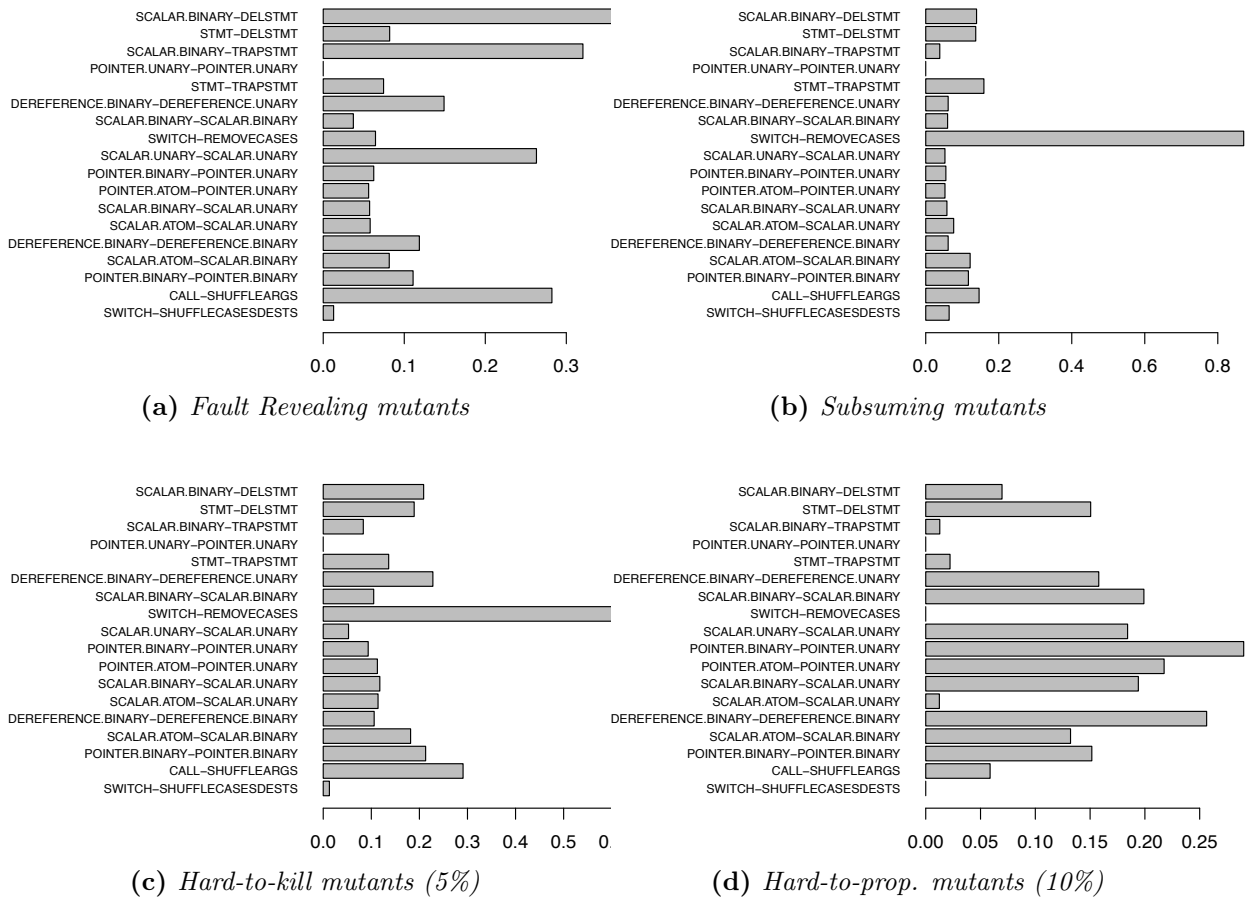
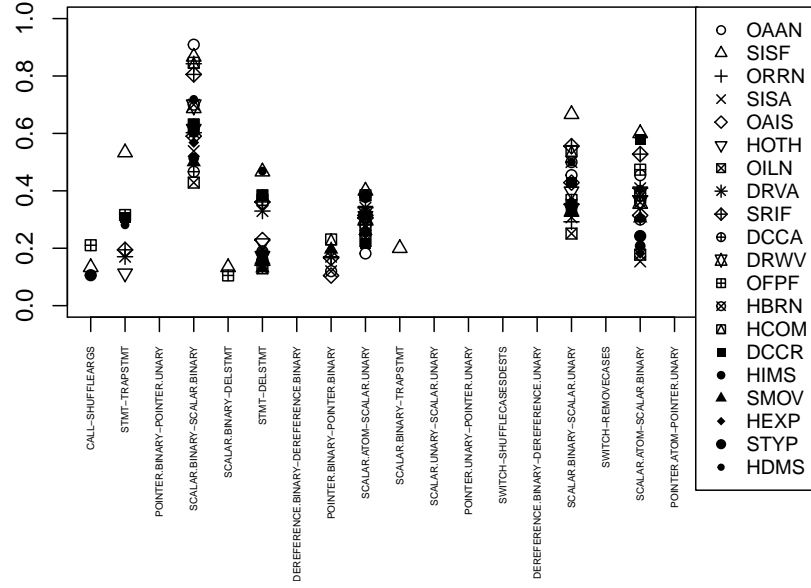
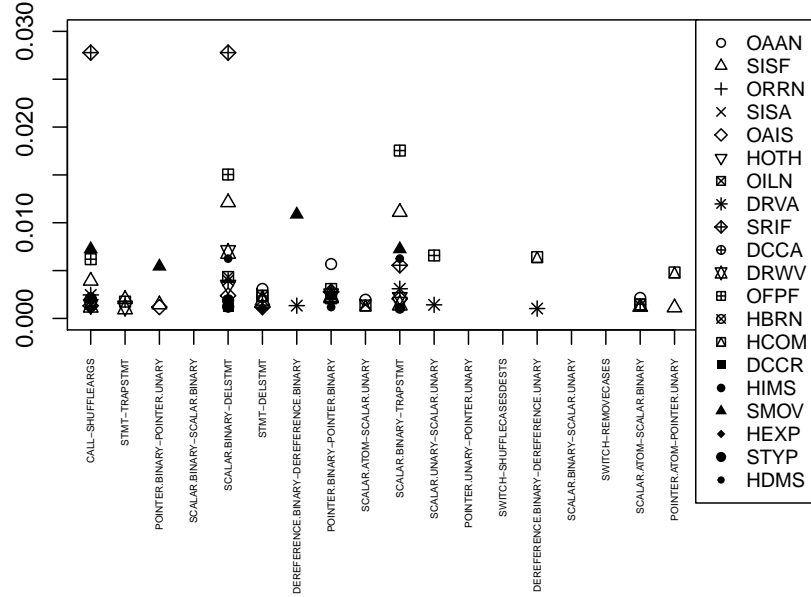


Figure 5.3: Ratio of mutants involved in quality indicator categories per mutant type.



(a) Ratio of faults for every fault class



(b) Normalized ratio of faults for every fault class

Figure 5.4: Ratio of faulty versions with fault revealing mutants (among all faults of the same type) per fault class and mutant type.

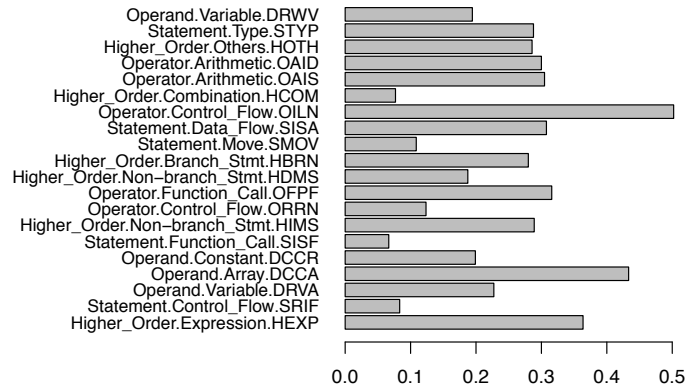


Figure 5.5: Faulty versions (see Table 5.1) without fault revealing mutants (ratios)

SELECTING FAULT REVEALING MUTANTS

Mutant selection refers to the problem of choosing, among a large number of mutants, the (few) ones that should be used by the testers. This chapter presents a machine learning approach, named FaRM, that tackles the mutant selection problem. The main focus is the fault revealing mutants, i.e., the mutants that are killable and lead to test cases that uncover unknown program faults. Experimental results show that FaRM achieves a good trade-off between application cost and effectiveness (measured in terms of faults revealed) and outperforms all the existing mutant selection methods.

This chapter is based on the work published in the following papers:

- Thierry Titchou Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon and Koushik Sen. 2019. Selecting Fault Revealing Mutants. *Empirical Software Engineering (EMSE)*. <https://doi.org/10.1007/s10664-019-09778-7>. 2019
- Thierry Titchou Chekam, Mike Papadakis, Tegawendé F. Bissyandé, and Yves Le Traon. 2018. Poster: Predicting the fault revelation utility of mutants. *In Proceedings of the 40th International Conference on Software Engineering: Companion (ICSE-Companion 2018)*, 2018

Chapter content

6.1	Introduction	67
6.2	Context	69
6.2.1	Problem Definition	69
6.2.2	Mutant Selection	71
6.2.3	Mutant Prioritization	72
6.3	Approach	72
6.3.1	Implementation	75
6.3.2	Demonstrating Example	75
6.4	Research Questions	77
6.5	Experimental Setup	79
6.5.1	Benchmarks: Programs and Fault(s)	79
6.5.2	Automated Tools Used	81
6.5.3	Experimental Procedure	82
6.5.4	Mutant Selection and Effort Metrics	83
6.6	Results	84
6.6.1	Assessment of killable mutant prediction (RQ1 and RQ2)	84
6.6.2	Assessment of fault revelation prediction	85
6.6.3	Mutant selection	87
6.6.4	Mutant prioritization	90
6.6.5	Experiments with large programs (RQ7)	96
6.7	Discussion	100
6.7.1	Working Assumptions	100
6.7.2	Threats to Validity	101
6.7.3	Representativeness of test subjects	102
6.7.4	Redundancy between the considered faults	103
6.7.5	Other Attempts	104
6.8	Conclusions	105

6.1 Introduction

We show in Chapter 4 that mutation testing is one of the most effective techniques with respect to fault revelation. Researchers typically use mutation as an assessment mechanism (measuring effectiveness) for their techniques [Pap+19], but it can be used as every other test criterion. To this end, mutation can be used to assess the effectiveness of test suites or to guide test generation [AO08; FZ12; PI18; Pap+18].

Unfortunately, mutation testing is expensive. This is due to the large number of mutants that require analysis. An important cost parameter is the so-called *equivalent mutants*, which are mutants forming equivalent program versions [Pap+15; AO08]. These need to be manually inspected by testers since their automatic identification is not always possible [BA82].

While the problem of the equivalent mutants have been partly addressed by recent methods such as the Trivial Compiler Equivalence (TCE) [Pap+15], the problem of the large number of mutants remains challenging. Yet, addressing this problem will in return contribute to addressing the equivalent mutant problem: any approach that is effective in reducing the large number of mutants, would indirectly reduce the equivalent mutant problem since less equivalent mutants will be available.

Nevertheless, producing a large number of mutants is impractical. The mutants need to be analyzed, compiled, executed and killed by test cases. Perhaps, more importantly testers need to manually analyse them in order to design effective test cases. The scalability, or lack thereof, of mutation testing, with respect to the number of mutants to be processed, is thus a key factor that hinders its wide applicability and large adoption [Pap+19]. Consequently, if we can find a lightweight and reasonably effective way to diminish the number of mutants without sacrificing the power of the method, we would then manage to significantly improve the scalability of the method. Since the early days of mutation testing, researchers attempted to find such solutions by forming many mutant reduction strategies [Pap+19], such as selective mutation [ORZ93; WM95a] and random mutant selection [TA+79].

Our goal is to form a mutant selection technique that identifies killable (non equivalent) mutants that are fault revealing, prior to any mutant execution. We consider as fault revealing, any mutant (i.e. test objective) that leads to test cases capable of revealing the faults in the program under test. We argue that such mutants are program specific and can be identified by a set of static program features. In this respect, we need features that are simultaneously generic, in order to be widely applicable, and powerful to approximate well the program and mutant semantics.

We advance in this research direction by proposing a machine learning-based approach, named *FaRM*, which learns on code and mutants' properties, such as mutant type and mutation location in program control-flow graphs, as well as code complexity and program control and data dependencies, to (statically) classify mutants as likely killable/equivalent and likely fault revealing. This approach is inspired by the prediction modelling line of research, which has recorded high performance by using machine learning to triage likely error-prone characteristics of code [MGF07; KS16].

The use case scenario of *FaRM* is a standard testing scenario where mutants are used as test objectives, guiding test generation. To achieve this, we train on a set of faulty programs that

have been tested with mutation testing, prior to any testing or test case design for the particular system under analysis. Then, we predict the killable (non equivalent) and fault revealing mutants based on which we test the particular system under analysis. The training corpus can include previously developed projects (related to the targeted application domain) or previous releases of the tested software. In a sense, we train on system(s), say x , and select mutants on the system under test, say y , where $x \neq y$.

Experimental results using 10-Fold cross validation on $1,692 + 45$ faulty program versions show a high performance of *FaRM* in yielding an adequately selected set of mutants. In particular our method achieves statistically significantly better results than the random, selective mutation and defect prediction (mutating the areas predicted by defect prediction), mutant selection baselines by revealing 23% to 34% more faults than any of the baselines. Similarly, our mutant prioritization method achieves statistically significant higher Average Percentage of Faults Detected (APFD) [Hen+16] values than the random prioritisation (4% to 9% higher in the median case). With respect to test execution, we show that our selection method requires less execution time (than random).

We also demonstrate that our method is capable of selecting killable (non-equivalent) mutants. In particular, by building an equivalent classification method, using our features, we achieve an AUC value of 0.88 and 95%, 35% precision and Recall. These results indicate drastic reductions on the efforts required by the analysis of equivalent mutants. A combined approach, named *FaRM**, achieves similar to *FaRM* fault revelation, but potentially at a lower cost (lower number of equivalent mutants), indicating the capabilities of our method.

In summary, this chapter makes the following contributions:

- It introduces the fault revealing mutant selection and fault revealing mutant prioritization problems.
- It demonstrates that the killability and fault revealing utility of mutants can be captured by simple static source code metrics.
- It presents *FaRM*, a mutant selection technique that learns to select and rank mutants using standard machine learning techniques and source code metrics.
- It provides empirical evidence suggesting that *FaRM* outperforms the current state-of-the-art mutant selection and mutant prioritization methods by revealing 23% to 34% more faults and achieving 4% to 9% higher average percentage of revealed faults, respectively.
- It provides a publicly available dataset of feature metrics, kill and fault revelation matrices that can support reproducibility, replication and future research.

This chapter is organized as follows. Section 6.2 provides background information on the mutant selection problem and defines the targeted problem(s). Section 6.3 overviews the proposed approach. Evaluation research questions are enumerated in Section 6.4, while experimental setup is described in Section 6.5 and experimental results are presented in Section 6.6. A detailed discussion on the applicability of our approach and the threats to validity are given in Section 6.7. Section 6.8 concludes this chapter.

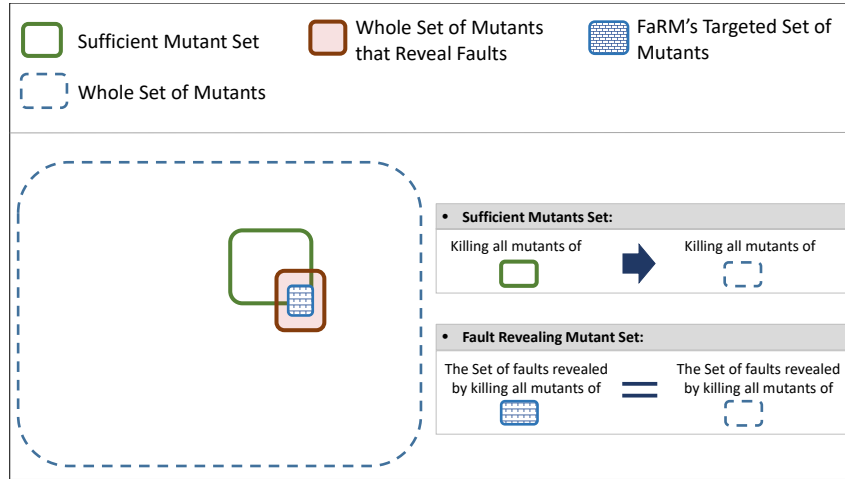


Figure 6.1: *Fault revealing mutant selection. Contrast between sufficient mutant set selection and fault revealing mutant selection. Sufficient mutant set selection aims at selecting a minimal subset of mutants that is killed by tests that also kill the whole set of mutants. Fault revealing mutant selection aims at selecting a minimal subset of mutants that is killed by tests that reveal the same underlying faults as the tests that kill the whole set of mutants.*

6.2 Context

6.2.1 Problem Definition

Our goal is to select among the many mutants the (few) ones that are fault revealing, i.e., mutants that lead to test cases that reveal existing, but unknown, faults. This is a challenging goal since only 2% (according to our data) of the killable mutants are fault revealing.

The fault revealing mutant selection goal is different from that of the “traditional” mutant reduction techniques, which is to reduce the number of mutants [Off+96a; WM95b; FPO18; Pap+19]. Mutant reduction strategies focus on selecting a small set of mutants that is representative of the larger set. This means, that every test suite that kills the mutants of the smaller set, also kills the mutants of the large set. Figure 6.1 illustrates our goal and contrasts it with the “traditional” mutant reduction problem. The blue (and smallest) rectangle on the figure represents the targeted output for the fault revealing mutant selection problem.

In line with previous research [Pap+18] we show in Chapter 5 that the majority of the mutants, even in the best case, are “irrelevant” to the sought faults. This means that testers need to analyse a large number of mutants before they can find the actually useful ones (the fault revealing ones), wasting time and effort. According to our data, 17% of the minimal mutants (ideal mutant reduction), i.e., subsuming mutants (a set of mutants with minimal overlap that are sufficient for preserving test effectiveness [JH09; KPM10; ADO14]) is fault revealing. This also indicates that the majority of the mutants, even in the best case, are “irrelevant” to the sought faults. We therefore claim that mutation testing should be performed only with the mutants that are most likely to be fault revealing. This will make possible the best effort application of the method.

Formally, we consider two aspects of this selection problem: the mutant selection one and the mutant prioritization one.

The **fault revealing mutant selection problem** is defined as:

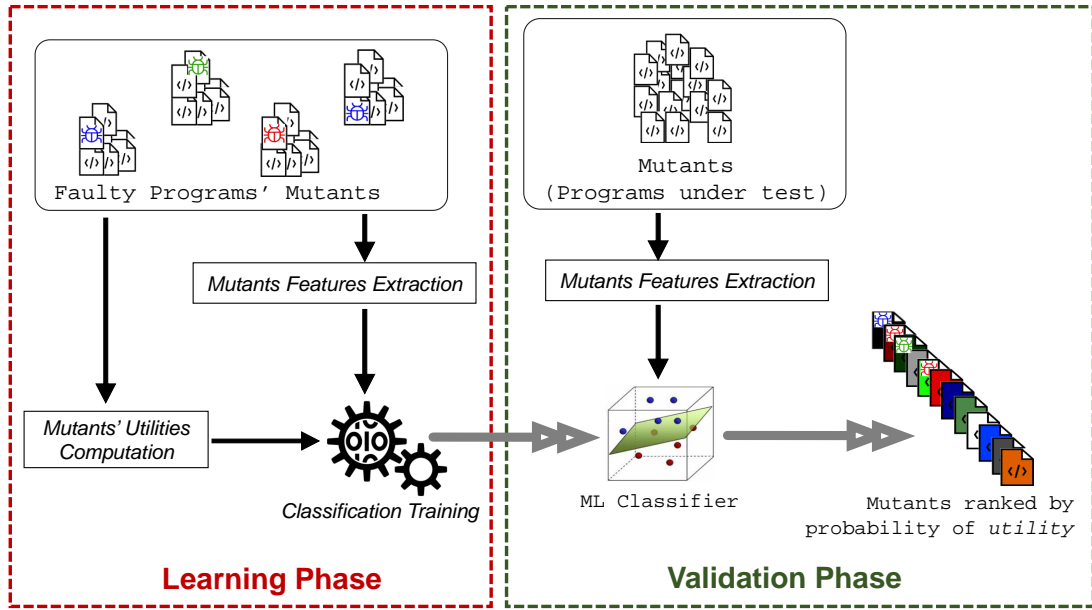


Figure 6.2: Overview of the FaRM approach. Initially, FaRM applies supervised learning on the mutants generated from a corpus of faulty program versions, and builds a prediction model that learns the fault revealing mutant characteristics. This model is then used to predict the mutants that should be used to test other program versions. This means that at the time of testing and prior to any mutant execution, testers can use and focus only on the most important mutants.

Given: A set of mutants M for program P .

Problem: Subset selection. Select a subset of mutants, $S \in M$, such that $F(S) = F(M)$ and $(\forall m \in S), (F(S - \{m\}) \neq F(M))$.

S represents a subset of M ; $F(X)$ represents the number of faults in P that are revealed by the test suites that kill all the mutants of the set X . In practice, the challenge is to approximate well S , statically and prior to any test execution, by finding a relatively good trade-off between the number of selected mutants (to minimise) and the number of faults revealed by their killing (to maximize).

Similarly, the **fault revealing mutant prioritization problem** is defined as:

Given: A set of mutants, M and the set of permutations of M , PM for program P .

Problem: Find $Pm' \in PM$ such that $(\forall Pm'')(Pm'' \in PM) (Pm'' \neq Pm') [f(Pm') \geq f(Pm'')]$

PM represents the set of all possible mutant orderings of M , and $f(X)$ represents the average percentage of faults revealed by the test cases that kill the selected mutants in the given order X (measures the area under the curve representing the faults revealed by the killing of each one of the mutants in the order). The challenge is to statically and prior to any test execution, rank the mutants so that the fault revealing potential is maximized when killing any (arbitrary) number of them. The idea is that fault revelation is maximized whenever the tester decides to stop killing mutants.

6.2.2 Mutant Selection

In the literature many mutant selection methods have been proposed [Pap+19; FPO18] by restricting the considered mutants according to their types, i.e., applying one or more mutant operators. Empirical studies [Kur+16a; DOL13], have shown that the most successful strategies are the statement deletion [DOL13] and the E-Selective mutant set [Off+96a; ORZ93]. We therefore compare our approach with these methods. We also consider the random mutant selection [T A+79] since there is evidence demonstrating that it is particularly effective [Zha+10b; PM10a].

6.2.2.1 Random Mutant Selection

Random mutant sampling [T A+79] forms the simplest mutant selection technique, which can be considered as a natural baseline method. Interestingly, previous studies found it particularly effective [Zha+10b; PM10a]. Therefore, we compare with it.

We use two random selection techniques, named as SpreadRandom and DummyRandom. SpreadRandom iteratively goes through all program statements (in random order) and selects mutants (one mutant among the mutants of each statement), while DummyRandom selects them from the set of all possible mutants. The first approach is expected to select mutants residing on most of the program statements, while the second one is expected to make a uniform selection.

6.2.2.2 Statement Deletion Mutant Selection

Mutant selection based on statement deletion is a simple approach that, as the name suggests, deletes every program statement (once at a time). To avoid introducing compilation issues (mutants that do not compile) and introduce relatively strong mutants, the statement deletion is usually applied on parts of a statement (deleting parts of expressions, i.e., the expression $a + b$ becomes a or b). Empirical studies have shown that statement deletion mutant selection is powerful (achieves a very good trade-off between the number of selected mutants and test effectiveness) and has the advantage of introducing few equivalent mutants [DOL13].

6.2.2.3 E-Selective Mutant Selection

E-Selective refers to the 5 operator mutant set introduced by Offutt *et al.* [Off+96a; ORZ93]. This set is the most popular operator set [Pap+19] that is included in most of the modern mutation testing tools. This set includes the mutants related to relational, logical (including conditional), arithmetic, unary and absolute mutations. According to the study of Offutt *et al.* [Off+96a] this set has the same strengths as a much larger comprehensive set of operators. Although there is empirical evidence demonstrating that the E-Selective set has lower strengths than a more comprehensive set of operators [Kur+16a], it still provides a very good trade-off between selected mutants and strengths [Kur+16a].

6.2.3 Mutant Prioritization

Mutant prioritization has received little or even no attention in literature (refer to the Related Work Chapter 3, section 3.2 for details). Given the absence of other methods, we compare our approach with the random baselines. We also consider alternative schemes, such as Defect Prediction prioritization.

6.2.3.1 Random Mutant Prioritization

Random mutant prioritization forms a natural baseline for our approach. Comparing with random orderings is a common practice in test case prioritization studies [Rot+01; Hen+16] and shows the ability of the prioritization method to systematically order the sought elements. Similarly to mutants selection, we applied two random ordering techniques, the SpreadRandom and DummyRandom. SpreadRandom orders mutants by iteratively going through all program statements (in random order) and selects one mutant among the mutants of each statement (statement-based orders), while DummyRandom orders them from the mutant set (uniform orders).

6.2.3.2 Defect Prediction Mutant Prioritization

Naturally, one of the main attributes determining the utility of the mutants is their location. Thus, instead of selecting mutants based on other properties, one could select them based on their location. To this end, we form a prioritization method that predicts and orders the error-prone code locations, i.e., code parts that are most likely to be faulty. Then, we mutate the predicted code areas and form a baseline method. Such an approach is in sense equivalent to the application of mutation testing on the results of defect prediction. Moreover, such a comparison demonstrates that mutants depend on the attributes (features) we train on not solely on their location.

6.3 Approach

Our objective is to select mutants that lead to effective test cases. In view of this, we aim at selecting and prioritizing mutants so that we reveal most of the faults by analysing the smallest possible number of mutants.

We conjecture that mutant selection strategies should account for the properties that make them killable and fault revealing. Defect prediction studies [MGF07; KS16] investigated properties related to error-prone code locations, but not related to mutants. Mutation testing is a behaviour oriented criterion and requires mutants introducing small and useful semantic deviations. Therefore, we propose building a model, which captures the essential properties that make mutants valuable (in terms of their utility to reveal faults).

Figure 6.2 depicts the *FaRM* approach, which learns to rank mutants according to their fault revealing potential (likelihood to reveal (unknown) faults). Initially, *FaRM* applies supervised learning on the mutants generated from a corpus of faulty program versions, and builds a prediction model. This model is then used to predict the mutants that should be used to test the

particular instance of the program under test. This means that at the time of testing and prior to any mutant execution, testers can use and focus only on the most important mutants.

Regarding *FaRM* supervised learning training phase (when the prediction model is built), the faulty programs mutants's features are extracted and used as training data's features and, their utilities are computed and used as training data's expected output. The mutant's utility for fault revealing and killable mutant prediction is respectively the mutants' fault revealing and killability information. Regarding the validation phase, features of the mutants of the program under test are extracted and used as validation data's features to predict the mutants' utilities with the trained model. Mutants with high predicted utility are the useful ones.

Definition: For a given problem, we define as classifier's *performance* the prediction performance of the classifier, which is the accuracy of the predictions (precision, recall, F-measure and Area Under Curve metrics that are detailed in Section 6.5.3) of the classifier for the given problem.

ML-based measurement of mutant utility. The selection process in *FaRM* is based on training a predictor for assessing the probability of a mutant to reveal faults. To that end, we explore the capability of several features, which are designed to reflect specific code properties which may discriminate a useful mutant from another. Let us consider a mutant M associated to a code statement S_M on which the mutation was applied. This mutant can be characterized from various perspectives with respect to (1) the complexity of the relevant mutated statement, (2) the position of the mutated code in the control-flow graph, (3) dependencies with other mutants, (4) the nature of the code block where S_M is located.

ML features for characterizing mutants. Recently, the studies of Wen et al. [Wen+18], Just et al. [JKA17] and Petrovic and Ivankovic [PI18] found a strong connection between mutants' utility and the surrounding code (captured by the AST father and child nodes). Therefore, in addition to the mutant types, typically considered by selective mutation approaches [Off+96a; NAM08; Pap+19], we also considered the information encoded on the program AST. We include three such features, the Data type at the mutant location, the parent AST node (of the mutant expression) and the child AST node (of the mutant expression), in our machine learning classification scheme.

Let B_M be the control-flow graph (CFG) basic block associated to a mutated statement S_M containing the mutated expression E_M . Table 6.1 provides the list of all 28 features that we extract from each mutant. The features named *TypeAstParent*, *TypeMutant*, *TypeStmtBB*, *AstParentMutantType*, *OutDataDepMutantType*, *InDataDepMutantType*, *OutCtrlDepMutantType*, *InCtrlDepMutantType*, *DataTypesOfOperands* and *DataTypesOfValue* are categorical. We represented them using one hot encoding. Besides the categorical features listed above, all other features are numerical. The values of numerical features are normalized between 0 and 1 using *feature scaling*, more precisely *min-max normalization/scaling*.

A demonstrating example on how mutant features are computed is given in the following subsection (section 6.3.2). After extracting feature values, we feed them to a machine learning classification algorithm along with the killability and fault revealing information for each mutant for a set of faults. The training process then produces two classifiers (one for the killable (non equivalent) and one for the fault revealing mutants) which, given the feature values of a given mutant, they are capable of computing the utility probabilities for this mutant, i.e., its probability to be killable and its probability to be fault revealing.

Complexity	Complexity of statement S_M approximated by the total number of mutants on S_M
CfgDepth	Depth of B_M according to CFG
CfgPredNum	Number of predecessor basic blocks, according to CFG, of B_M
CfgSuccNum	Number of successors basic blocks, according to CFG, of B_M
AstNumParents	Number of AST parents of E_M
NumOutDataDeps	Number of mutants on expressions data-dependents on E_M
NumInDataDeps	Number of mutants on expressions on which E_M is data-dependent
NumOutCtrlDeps	Number of mutants on statements control-dependents on E_M
NumInCtrlDeps	Number of mutants on expressions on which E_M is control-dependent
NumTieDeps	Number of mutants on E_M
AstParentsNumOutDataDeps	Number of mutants on expressions data-dependent on E_M 's AST parent statement
AstParentsNumInDataDeps	Number of mutants on expressions on which E_M 's AST parent expression is data-dependent
AstParentsNumOutCtrlDeps	Number of mutants on statements control-dependent on E_M 's AST parent expression
AstParentsNumInCtrlDeps	Number of mutants on expressions on which E_M 's AST parent expression is control-dependent
AstParentsNumTieDeps	Number of mutants on E_M 's AST parent expression
TypeAstParent	Expression type of AST parent expressions of E_M
TypeMutant	Mutant type of M as matched code pattern and replacement. Ex: $a+b \rightarrow a-b$
TypeStmtBB	CFG basic block type of B_M . Ex: <i>if-then</i> , <i>if-else</i>
AstParentMutantType	Mutant types of M's AST parents
OutDataDepMutantType	Mutant types of mutants on expressions data-dependents on E_M
InDataDepMutantType	Mutant types of mutants on expressions on which E_M is data-dependent
OutCtrlDepMutantType	Mutant types of mutants on statements control-dependents on E_M
InCtrlDepMutantType	Mutant types of mutants on expressions on which E_M is control-dependent
AstChildHasIdentifier	AST child of expression E_M has an identifier
AstChildHasLiteral	AST child of expression E_M has a literal
AstChildHasOperator	AST child of expression E_M has an operator
DataTypesOfOperands	Data types of operands of E_M
DataTypeOfValue	Data type of the returned value of E_M

Table 6.1: Description of the static code features

By using these two classifiers we form three approaches, two of them using each one of the classifiers alone and one of them by combining them. The first two, named *FaRM* and *PredKillable*, respectively classify mutants according to their probability to be fault revealing and killable. The third one, named *FaRM**, divides the mutant set in two subsets, likely killable and likely equivalent (based on *PredKillable* predictions), separately ranks them according to their fault revealing probability and concatenates them by putting the likely killable subset first. Figure 6.3 show an example of mutant ranking by *FaRM**. The motivation for *FaRM** results from the hypothesis that equivalent mutants could be noise to *FaRM* and, *PredKillable* performs better at filtering equivalent mutants (or predicting killable mutants). Given that fault revealing mutants are killable, we expect them to have higher predicted utility value with both *FaRM* and *PredKillable*. Therefore, *FaRM** gives priority to the most likely fault revealing mutants that are also most likely killable.

We implement a prioritization scheme by considering the ranking of all mutants in accordance to the values of the developed probability measure. This forms our mutant prioritization approaches. Our mutant selection strategy sets a threshold probability value (e.g., 0.5) or a cut-off point according to the number of the top ranked mutants to keep only mutants with higher utility probability scores in the selected set. This forms our mutant selection approach. For the combined approach (*FaRM**) we divide the mutant set in the killable and equivalent subsets by using a cut-off point of 0.5.

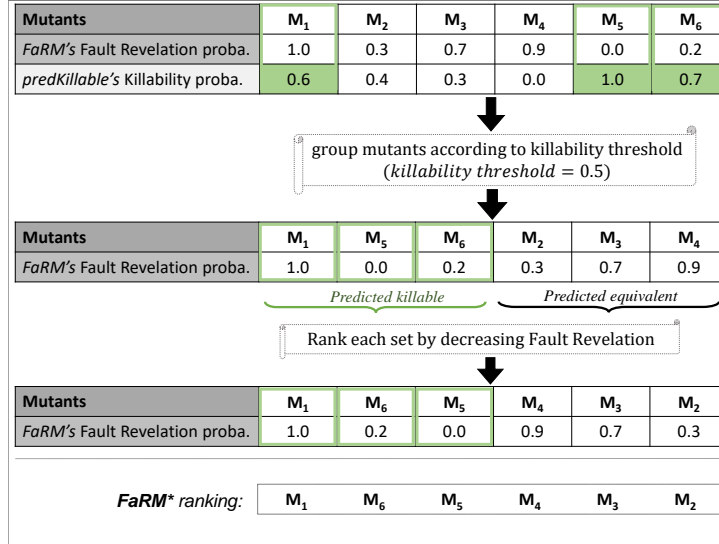


Figure 6.3: Example of mutant ranking procedure by FaRM*. the ranking is a concatenation of the ranked predicted killable mutants and the ranked predicted equivalent mutants.

6.3.1 Implementation

We implemented *FaRM* as a collection of tools in C++. We leverage stochastic gradient boosting [Fri02] (decision trees) to perform supervised learning. Gradient boosting is a powerful ensemble learning technique which combines several trained weak models to perform classification. Unlike common ensemble techniques, such as random forests [Bre01], that simply average models in the ensemble, boosting methods follow a constructive strategy of ensemble formation where models are added to the ensemble sequentially. At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learnt so far [NK13]. We use the FastBDT [Kec16] implementation by setting the number of trees to 1,000 and the trees depth to 5.

6.3.2 Demonstrating Example

Here we provide an example on how the features of Table 6.1 are computed. We consider the program in figure 6.4 (extracted from the Codeflaws benchmark, ID: 598-B-bug-17392756-17392766), on which mutation is applied. We present the feature extraction for a mutant M , which is created by replacing the right side decrement operator by the right side increment operator on line 16 ($m - -$ becomes $m ++$). We also present in figure 6.5-a the mutant, the abstract syntax tree (AST) of the mutated statement (*while* condition) in figure 6.5-b and in figure 6.5-c the control flow graph (CFG) of the function containing the mutated statement.

The features, for mutant M , are computed as following:

- The *complexity* feature value is the number of mutants generated on the statement containing the mutant M (Line 16). In this case 72 mutants. Thus, the *complexity* is 72.
- The *CfgDepth* feature value is the minimum number of basic blocks to follow, along the CFG, from *main* function's entry point to the basic block containing M (*BB2*). In this case 1 basic block as shown in Figure 6.5-c. Thus, the *CfgDepth* is 1.
- The *CfgPredNum* feature value is the number of basic blocks directly preceding the basic block containing M (*BB2*) on the control flow graph. In Figure 6.5-c there are 2 basic blocks (*BB1*

and *BB3*). Thus, the *CfgPredNum* is 2.

- The *CfgSuccNum* feature value is the number of basic blocks directly following the basic block containing *M* (*BB2*) on the control flow graph. In Figure 6.5-c there are 2 basic blocks (*BB3* and *BB4*). Thus, the *CfgSuccNum* is 2.

- The *AstNumParents* feature value is the number of AST parents of the mutated expression. In this case, the only AST parent is the relational expression, in Figure 6.5-b, whose sub-tree is rooted on the greater than sign ($>$). Thus the feature value is 1.

- The *NumOutDataDeps* feature value is the number of mutants on expressions data dependent on the mutated expression. In this case, looking at Figure 6.4, the value of variable *m* written in the mutated expression $m - -$ is only used in the same expression. Thus the feature value is the number of mutants on the mutated expression $m - -$.

- The *NumInDataDeps* feature value is the number of mutants on expressions on which the mutated expression is data dependent. In this case, looking at Figure 6.4, the value of variable *m* used on the mutated expression $m - -$ is either written on the *scanf* statement at line 15, or in the same expression. Thus the feature value is the sum of the number of mutants on the statement at line 15 and the number of mutants on the mutated expression $m - -$.

- The *NumOutCtrlDeps* feature value is the number of mutants on statements control dependent on the mutated expression. In this case, looking at Figure 6.4, no statement is control dependent on the mutated expression $m - -$. Thus the feature value is 0.

- The *NumInCtrlDeps* feature value is the number of mutants on expressions on which the mutated statement is control dependent. In this case, looking at Figure 6.4, no expression controls the mutated expression. Thus the feature value is 0.

- The *NumTieDeps* feature value is the number of mutants on the right decrement expression (mutated expression).

- The *AstParentsNumOutDataDeps* feature value is the number of mutants on expressions data dependent on the AST parent of the mutated expression. In this case, looking at Figures 6.4 and 6.5-b, the value of the relational expression (AST parent of $m - -$) is not used in other expressions. Thus the feature value is 0.

- The *AstParentsNumInDataDeps* feature value is the number of mutants on expressions on which the AST parent of the mutated expression is data dependent. In this case, looking at Figures 6.4 and 6.5-b, the value of the relational expression (AST parent of $m - -$) only depends on the value of expression $m - -$. Thus the feature value is the number of mutants on expression $m - -$.

- The *AstParentsNumOutCtrlDeps* feature value is the number of mutants on statements control dependent on the AST parent of the mutated expression. In this case, looking at Figures 6.4 and 6.5-b, all the statements in basic block *BB3* are control dependent on the relational expression (AST parent of $m - -$). Thus the feature value is the sum of the number of mutants in lines 17, 18 and 19 of the code in Figure 6.4.

- The *AstParentsNumInCtrlDeps* feature value is the number of mutants on expressions on which the AST parent of the mutated expression is control dependent. In this case, looking at Figures 6.4 and 6.5-b, no expression controls the relational expression (AST parent of the mutated expression $m - -$). Thus the feature value is 0.

- The *AstParentsNumTieDeps* feature value is the number of mutants on the relational expression, AST parent of the mutated right decrement expression. The feature value here is the number of mutants of the relational expression of operator greater than.

- The *TypeAstParents* feature value is AST type of the AST parent expression of the mutated

- expression. Here, that is the AST type of the relational expression with operator greater than.
- The *TypeMutant* feature value is the type of the mutant as a string representing the matched and replaced pattern. The feature value is " $() -- \rightarrow () ++$ ".
 - The *TypeStmtBB* feature value is the type of the basic block containing the mutated statement. The feature value here is the type of *BB2* (see Figure 6.5-c), which is "While Condition".
 - The *AstParentMutantType* feature value is the aggregation of types of the mutants on the AST parents of the mutated expression. That is the aggregation of the mutants types of the relational expression whose sub-tree is rooted on the greater than sign ($>$) as shown in Figure 6.5(b). The aggregation of a set of mutant types is performed by summing up the one encoding vectors of the mutants types, allowing each mutant type to be represented in the encoding.
 - The *OutDataDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumOutDataDeps*.
 - The *InDataDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumInDataDeps*.
 - The *OutCtrlDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumOutCtrlDeps*.
 - The *InCtrlDepMutantType* feature value is the aggregation (as computed for *AstParentMutantType*) of the mutant types of the mutants counted to compute *NumInCtrlDeps*.
 - The *AstChildHasIdentifier* feature value is the Boolean value representing whether the mutated expression has an identifier as operand. In this case, the mutated expression has the identifier *m* as operand. Thus, the value of the feature is 1 (True).
 - The *AstChildHasLiteral* feature value is the Boolean value representing whether the mutated expression has a literal as operand. In this case, the mutated expression does not have the literal as operand. Thus, the value of the feature is 0 (False).
 - The *AstChildHasOperator* feature value is the Boolean value representing whether the mutated expression has an operator. In this case, the mutated expression has the operator right decrement operator $--$. Thus, the value of the feature is 1 (True).
 - The *DataTypesOfOperands* feature value is the datatype of the operand of the right decrement operation $--$. That is the datatype of *m* which is "*int*".
 - The *DataTypeOfValue* feature value is the datatype of the value of the mutated expression, Which is "*int*" as the data type of *m*.

6.4 Research Questions

When building prediction methods, the first thing to investigate is their prediction ability. Thus, our first question can be stated as:

RQ1: *How well does our machine learning method predict the killable mutants?*

Similarly, our second question can be stated as:

RQ2: *How well does our machine learning method predict the fault revealing mutants?*

After demonstrating that our classification method predicts satisfactorily the fault revealing mutants, we continue by investigating its ability to practically support mutant selection with respect to the actual measure of interest, the revealed faults, and with respect to the random baseline techniques. Therefore, we investigate:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  void rotate(char *s, int n, int k) {
5      char t[10000];
6      memcpy(t, s + n - k, k);      // 49 mutants
7      memcpy(t + k, s, n - k);      // 65 mutants
8      memcpy(s, t, n);              // 10 mutants
9  }
10
11 int main(int argc, char *argv[]) {
12     char s[10000];
13     int m, l, r, k;
14     scanf("%s", s);                // 6 mutants
15     scanf("%d", &m);                // 3 mutants
16     while (m-- > 0) {              // 72 mutants
17         scanf("%d%d%d", &l, &r, &k); // 6 mutants
18         l--;                        // 55 mutants
19         rotate(s + l, r - l, k);    // 60 mutants
20     }
21     printf("%s\n", s);              // 7 mutants
22     return 0;                      // 3 mutants
23 }

```

Figure 6.4: Example program where mutation is applied. The C language comments on each line show the number of mutants generated on the line.

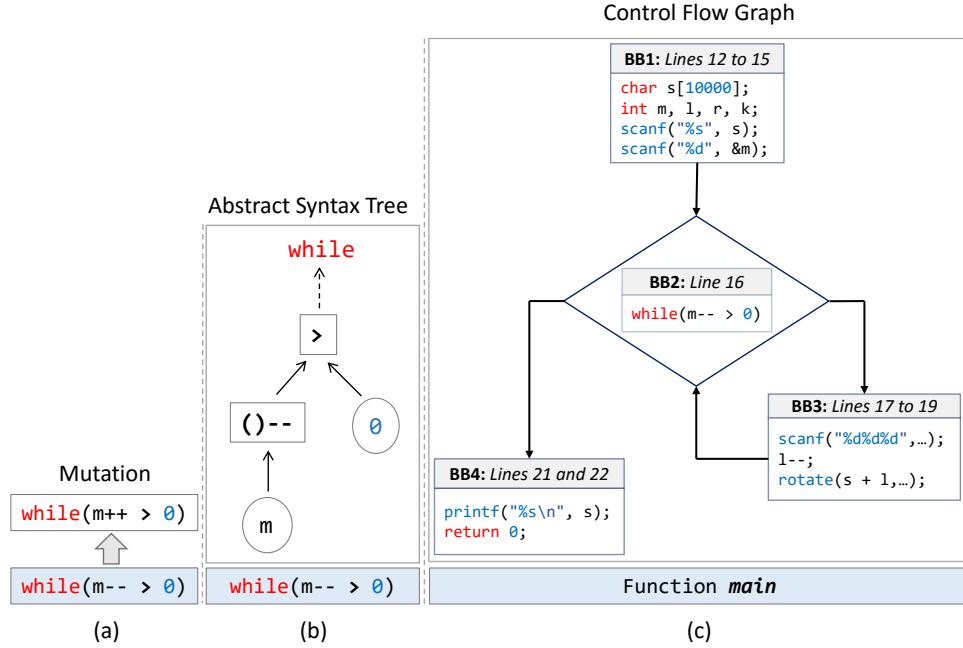


Figure 6.5: (a) An example of mutant M from the example program from Figure 6.4, (b) the abstract syntax tree of the mutated statement and (c) the control flow graph of the function containing the mutated statement.

RQ3: How do our methods compare against the random strategies with respect to the fault revealing mutant selection problem?

In addition to the random strategies, we also compare with the current state-of-the-art mutant selection methods. Thus, we ask:

RQ4: How do our methods compare against the *E-Selection* and *SDL* with respect to the fault revealing mutant selection problem?

As we already discussed an alternative mutant cost reduction technique is mutant prioritization.

Hence, we ask:

RQ5: *How do our methods compare against the random strategies with respect to the fault revealing mutant prioritization problem?*

In addition to the random strategies, we also compare with the defect prediction mutant prioritization baseline. Therefore, we ask:

RQ6: *How do our methods compare against the defect prediction mutant prioritization method?*

Finally, by demonstrating the benefits of our approach, we turn to investigate the generalization ability of our approach on larger and complex programs. Therefore we conclude by asking:

RQ7: *How well do our method generalise its findings on independently selected programs that are much larger and complex?*

6.5 Experimental Setup

6.5.1 Benchmarks: Programs and Fault(s)

For the purposes of our study we need a large number of programs that are not trivial and are accompanied with real faults. The fault set has to be large and of diverse types. Unfortunately, mutation testing is costly and its experimentation requires generating strong test suites (see Chapter 4). Therefore, there are two necessary tradeoffs, between the number of faults to be considered, the strengths of the test suites to be used and the size of the used programs.

To account for these requirements, we used the Codeflaws benchmark [Tan+17]. This benchmark consists of 7,436 programs (among which 3,902 are faulty) selected from the Codeforces¹ online database of programming contests. These contests consist of three to five problems, of varied difficulty levels. Every user submits its programs that resolve the posed problems. In total, the benchmark involves programs from 1,653 users “with diverse level of expertise” [Tan+17].

Every fault in this benchmark has two program instances: the rejected ‘*faulty*’ submission and the accepted ‘*correct*’ submission. Overall, the benchmark contains 3,902 faulty program versions of 40 different defect classes. It is noted that every faulty program instance in our dataset is unique, meaning that every program we use is different from the others (in terms of implementation). To the best of our knowledge, this is the largest number of faults used in any of the mutation testing studies. The size of the programs varies from 1 to 322 with an average of 36 lines of code. Applying mutation testing on Codeflaws yielded 3,213,543 mutants and required a total of 8,009 CPU days for all computations.

To strengthen our results and demonstrate the ability of our approach to handle faults made by actual developers, we also used the CoREBench [BR14] benchmark. CoREBench includes real-world complex faults that have been systematically isolated from the history of C open source projects. These programs are of 9-83 KLoC and are accompanied by developer test suites. It is noted that every CoREBench fault forms a single fault instance (it differs from the other faults).

We used the available test suites augmented by KLEE [CDE08]. Although these test suites greatly increased the cost of our experiment, we considered their use of vital importance as

¹<http://codeforces.com/>

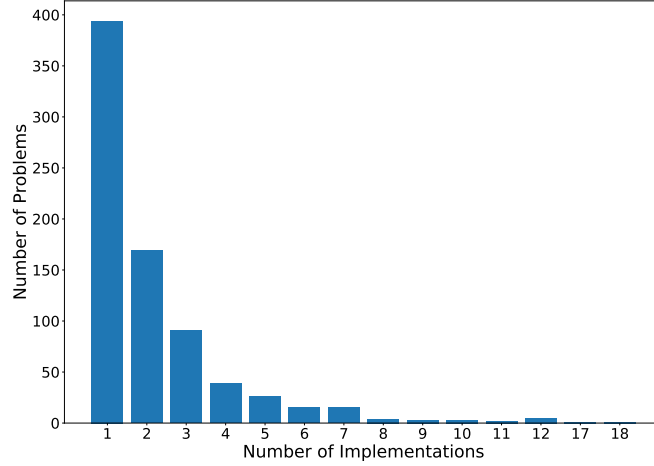


Figure 6.6: *Distribution of Codeflaws Benchmark problems by number of implementations.*

otherwise our results could be subject to “noise effects” (as presented in Chapter 4).

Due to the very high cost of the experiments and technical difficulties to reproduce some faults, we conducted our analysis on 45 faults (22 in Coreutils, 12 in Find and 11 in Grep). Applying mutation testing on these 45 versions yielded 1,564,614 mutants and required a total of 454 CPU days of computation (without considering the test generation and machine learning computations and evaluations). Test generation resulted in a test pool composed of 122,261 and 22,477 test cases related to Codeflaws, CoREBench.

The goal of our study is to evaluate the fault revealing ability of the mutants we select. However, approximately half of our faults are trivial ones (triggered by most of the test cases), and their inclusion in our analysis would artificially inflate our results. Thus, we restrict our analysis on the faults that are revealed by less than 25% of the test cases involved in our test suites. Taking such a threshold is usual in fault injection studies [SiR18], but it ensures that the targeted faults and our focus is on faults that are hard enough to find. Practically, taking a lower threshold will significantly reduce the number of faults to be considered hindering our ability to train, while taking a higher threshold will make all the approaches perform similarly, as the faults will be easy to reveal. Overall, from the Codeflaws benchmark we consider 1,692 out of the 3,902 ones (1,692 are the non-trivial faults) and from the CoreBench benchmark 45 faults.

Figure 6.6 shows the distribution of number of problems by number of implementations for the considered faulty programs from Codeflaws. We observe that 85% of the problems have at most 3 implementations.

Despite that Codeflaws benchmark faults were mined from a programming contest, the faults nevertheless are relatively small syntactical mistakes. We observe on figure 6.7 that 82% of the faults are fixed by modifying a single line of source code. This ensures that we are compatible with the competent programmer hypothesis², which is one of the basic assumptions of mutation testing [DLS78].

²The competent programmer hypothesis states that programs have a small syntactic distance from the correct version so that we need a few keystrokes to correct the program

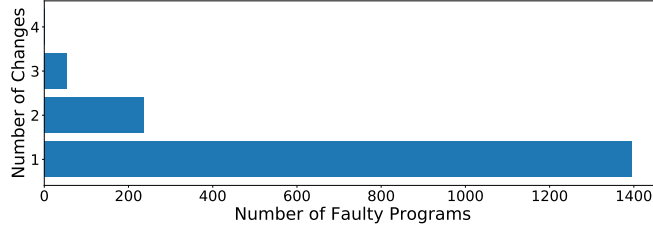


Figure 6.7: *Distribution of CodeFlaws Benchmark faulty programs by number of lines of code changed to fix the fault.*

6.5.2 Automated Tools Used

We used KLEE [CDE08] to support the test generation. We used KLEE with a relatively large timeout limit, equal to two hours per program, the Random Path search strategy, with Randomize Fork Enabled, Max Memory 2048 MB, Symbolic Array Size 4096 elements, Symbolic Standard input size 20 Bytes and Max Instruction Time of 30 seconds. This resulted in 26,229 and 1,942 test cases for CodeFlaws and CoREBench. Since the automatically generated test cases do not include any test oracle, we used the programs’ fixed version as oracle. We considered as failing, every test case that resulted in different observable program output when executed in the ‘faulty’ from that in the ‘correct’-fixed one. Similarly, we used the program output to identify the killed mutants. We deemed a mutant as killed if it resulted in a different output than in the original program.

We used our built mutation testing tool (*Mart*), presented in Chapter 8, that operates on LLVM bytecode. Actually all our metrics and analysis were performed on the LLVM bytecode. We used the default configuration of *Mart* which consists of 18 operators, composed of 816 transformation rules. These include all those that are supported by modern mutation testing tools [Off+96a; Pap+19; Col+16] and are detailed in Chapter 8.

Applying mutation testing on CodeFlaws and CoREBench yielded 3,213,543 and 1,564,614 mutants.

To reduce the influence of redundant and equivalent mutants, we applied TCE [Pap+15; Har+16; Kin+18]. Since we operate on LLVM bytecode we compared the mutated optimized LLVM codes using the *llvm-diff* utility. *llvm-diff* is a tool like the known Unix *diff* utility but for LLVM bytecode. TCE Detected 1,457,512 and 715,996 mutant equivalences on CodeFlaws and CoREBench. Note that the equivalent and redundant mutants detected by TCE are removed from the mutants set and neither executed nor considered in the experiments.

The execution of the mutants post TCE resulted in killing the 87% and 54% of the mutants for CodeFlaws and CoREBench. It is important to note that our tool applies mutant test execution optimizations by recording the coverage and program state at the mutation points avoiding the execution of mutants that do not infect the program state [PM10b]. This optimization enables huge test execution reductions and forms the current state of the art at the test execution optimizations [Pap+19]. Despite these optimization our tool required a total of 8,009 and 454 CPU days of computations for CodeFlaws and CoREBench indicating the large amount of computation resources required to perform such an experiment.

6.5.3 Experimental Procedure

To answer our research questions we performed an experiment composed of three parts. The first part regards the prediction ability of our classification method, answer RQ1 and RQ2, the second regards the fault revealing ability of the approaches, answer RQ3-RQ6, and the third regards the fault revealing ability of our approach on large independently selected programs, answer RQ7. To account for our use case scenario, in our experiments we always train and evaluate our approach on a different sets of programs (CodeFlaws) or program versions (CoREBench).

As a first step we used KLEE to generate test cases for all the programs we study and formed a pool of test cases by joining the generated and the available test cases. We then constructed a mutation-fault matrix, which records for every test case the mutants that it kills and whether it reveals the fault or not (we construct a matrix for every single fault we study). We also record the execution time needed to execute every mutant-test pair so that we can simulate the execution cost of the approaches. We make the data available³.

To measure fault revelation we mutated the faulty program versions. This is important in order to avoid making any assumption related to the interaction of mutants and faults, aka Clean Program Assumption evaluated in Chapter 4. Based on this matrix we compute the fault revealing ratio for each mutant. The *fault revealing ratio* is the ratio of tests that kill the mutant and reveal the fault to the total number of tests that kill the mutant.

First experimental part: The first task of prediction modeling is to evaluate the contribution of the used features. We computed the information gain values for each one of the used features. Higher information gain values represent more informative features for decision trees. Demonstrating the importance of our features helps us understand what is the most important factors affecting the utility of mutants. Having measured information gain, we then measure the prediction ability of our classification method by evaluating its ability to predict killable and fault revealing mutants. For this part of the experiment we considered as fault revealing the mutants that have fault revealing ratio equal to 1. We relax this constraint in the second part of the experiment.

We evaluate the trained classifiers using four typically adopted metrics such as the precision, recall, F-measure and Area Under Curve (AUC). The *precision* of a classifier is defined as the number of items that are truly relevant among the items that the classifier predicted to be relevant. The *recall* of a classifier is defined as the number of items that are predicted to be relevant by the classifier among all the truly relevant items. The F-measure of a classifier is defined as the weighted harmonic mean of the precision and recall, it is also named *F1 score*. The Area Under Curve (AUC) of a classifier is the area under the Receiver Operating Characteristic (ROC) curve (The ROC curve shows how many true positive classifications can be gained as more and more false positives are allowed) [Zhe15]. Precision represents the ratio of the identified killable and fault revealing mutants out of those classified as such. Recall represents the ratio of the identified killable and fault revealing mutants out of all existing ones. In classification usually recall and precision are competitive metrics in the sense that higher values of one imply lower values for the other. To better compare classifiers researchers use the F-measure and AUC metrics. These measure the general classification accuracy of the classifier. Higher values denote better classification.

To reduce the risk of overfitting, we applied a 10-fold cross validation by partitioning our program

³<https://mutationtesting.uni.lu/farm>

set into 10 parts and iteratively train on 9 parts and evaluation on one. We report the results for all the partitions.

This experiment part was performed on the Codeflaws programs.

Second experimental part: Our analysis requires comparing mutation-based strategies with respect to the actual value of interest, the number of faults revealed. Given that killing a mutant does not always result in revealing a fault, we train the classifier in accordance with the actual fault revealing ratios (i.e., the ratio of tests that kill a mutant and also reveal faults).

We then select and prioritise our mutants. To evaluate and compare the studied approaches with respect to fault revelation, we follow a typical procedures [Kur+16a; NAM08] (also followed in Chapter 4) by randomly selecting test cases, from the formed test pools, that kill the selected mutants. In case none of the available test cases on our test pool kills the mutant we treat it as equivalent. We repeat this process for each one of the studied approaches. As done in the first part of the experiment we report results using a 10-fold cross validation.

For the mutant selection problem we randomly pick a mutant and then randomly pick a test case that kills it. Then we remove all the killed mutants and pick another one. If the mutant is not killed by any of the test cases on our test pool we treat it as equivalent. We repeat this process 100 times and compute the probability of revealing each one of the faults.

For the mutant prioritisation case we follow the mutant order by picking test cases that kills each mutant. We do not attempt to kill a mutant twice. Again, we repeat this process 100 times and compute the Average Percentage of Faults Detected (APFD) values, which is typical metric used test case prioritization studies [Hen+16]. Again we align the compared approaches with respect to their cost (number of mutants need manual analysis) and compare their effectiveness.

To account for coincidental results and the stochastic selection of test cases and mutants we used the Wilcoxon test, which is a non-parametric test, to determine whether the Null Hypothesis (that there is no difference between the studied methods) can be rejected. In case the Null Hypothesis is rejected, then we have evidence that our approach outperforms the others. Even when the null hypothesis does not hold, the size of the differences might be small. To account for this effect we also measured the Vargha Delaney effect size \hat{A}_{12} (Section 2.5.3).

This experiment part was performed on the Codeflaws programs.

Third experimental part: To further evaluate the fault revealing ability of our approach, we applied it on the CoreBench programs. We also adopted the 10-fold cross validation as for the experiments on Codeflaws. We report results related to both fault revelation and APFD values. The CoreBench corpus is small in size and hence *FaRM* might not be particularly important. However, in case the signal of our features is strong, we will be able to experience the benefits of our method even with those few data.

6.5.4 Mutant Selection and Effort Metrics

When comparing methods, a comparison basis is required. In our case we measure fault revelation and effort. While measuring fault revelation based on the fault set we use is direct, measuring effort/cost is hard. Effort/cost depends on a large number of uncontrolled parameters, such as the followed procedure, level of automation, skills, underlying infrastructure and the learning

curve. Therefore, we have to account for different scenarios. and we adopt three frequently used metrics; the number of selected mutants, the number of test cases generated and the number of mutants requiring analysis.

The first metric (selected mutants) represents the number of mutants that one should use when applying mutation testing. This is a direct and intuitive metric as it suggest that developers should select a particular set of mutants to generate (form an actual executable codes), execute and analyse. Although such a metric conforms to our working scenario, it does not focus on the required test generation effort involved. Generating test cases is mostly a manual task (due to the test oracle problem) and so, we also consider a second metric, the number of test cases that can be generated based on a selected set of mutants.

We also adopt a third metric, the number of mutants that need to be analysed (equivalent mutants and those we pick, i.e., analysed in order to generate test cases). This metric somehow reflects the effort a tester needs to put in order to kill or identify as equivalent the selected mutants (under the assumption that equivalent mutants require the same effort as the test generation).

To fairly compare the random selection methods, we select mutants until we analyse the same number of mutants as analysed by our selection method. This establishes a fixed cost point for all the approaches and compare their effectiveness.

There are other cost factors, such as the mutant-test execution cost and the analysis of equivalent mutants (for the first two metrics) that we investigate separately. The reason for that is that we would like to see if our approaches are also faster to execute and require reasonably less equivalent mutants.

6.6 Results

6.6.1 Assessment of killable mutant prediction (RQ1 and RQ2)

To check the prediction performance of our classifier we performed a 10-Fold cross-validation for three different selected sets. These were the results of applying *PredKillable* to predict killable mutants and selecting the 5%, 10% and 20% of the top ranked mutants. The *PredKillable* classifier achieves 98.8% 5.7%, 10.7% precision, recall and F-measure when selecting the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 98.8% and 98.7% (precision), 11.4% and 22.8% (recall), 20.4% and 37.0% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular the *PredKillable* has 12.3%, 12.2% and 12.1% higher precision, and 0.7%, 1.4% and 2.8% higher recall values for the 5%, 10% and 20% sets of mutants.

When using *PredKillable* to predict non killable mutant, the classifier achieves 95.1% 35.0%, 51.2% precision, recall and F-measure when selects the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 79.1% and 49.3% (precision), 58.6% and 73.2% (recall), 67.3% and 58.9% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular the *PredKillable* has 81.6%, 65.7% and 35.8% higher precision, and 30.1%, 48.7% and 53.3% higher recall values for the 5%, 10% and 20% sets of mutants.

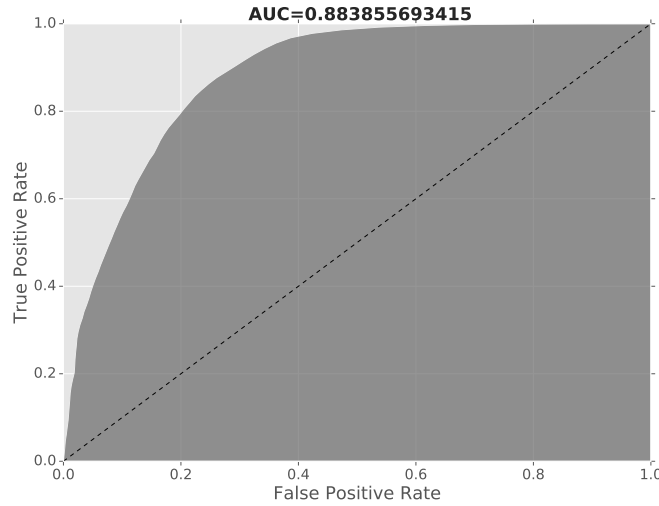


Figure 6.8: Receiver Operating Characteristic For Killable Mutants Prediction on Codeflaws

To train our models, approximately 48 CPU hours were required, while to perform the evaluation (perform mutant selection) it required less than a second. Since training should only happen once in a while, the training time is considered acceptable. Of course the cost of selecting and prioritizing mutants is practically negligible.

The Receiver operating characteristic (ROC) shown in Figure 6.8 further illustrates performance variations of the classifier in terms of true positive and false positive rates when the discrimination threshold changes: the higher the area under curve (AUC), the better the classifier. Our classifier achieves an AUC of 88%. These results establish that the code properties that were leveraged as features for characterizing mutants provide, together, a good discriminative power for assessing the fault revealing potential of mutants.

6.6.2 Assessment of fault revelation prediction

ML prediction performance Similarly to subsection 6.6.1 we performed a 10-Fold cross-validation for three different selected sets in order to check the prediction performance of our classifier. These were the results of applying *FaRM* and selecting the 5%, 10% and 20% of the top ranked mutants. The *FaRM* classifier achieves 5.7% 12.8%, 7.8% precision, recall and F-measure when selects the 5% of the mutants. With respect to 10% and 20% sets of mutants, it achieves 4.9% and 3.9% (precision), 22.0% and 35.1% (recall), 8.0% and 7.0% F-measures. These values are higher than those that one can get by randomly sampling the same number of mutants. In particular *FaRM* has 3.5%, 2.7% and 1.7% higher precision, and 7.8%, 12.1% and 15.1% higher recall values for the 5%, 10% and 20% sets of mutants.

The cost of training and evaluation are same as those reported in section 6.6.1.

The Receiver operating characteristic (ROC) shown in Figure 6.9 further illustrates performance variations of the classifier in terms of true positive and false positive rates when the discrimination threshold changes: the higher the area under curve (AUC), the better the classifier. Our classifier achieves an AUC of 62%.

We believe that such a result is encouraging due to the nature of the developer mistakes. As developers make mistakes in an non-systematic way, for the same problem, some may make

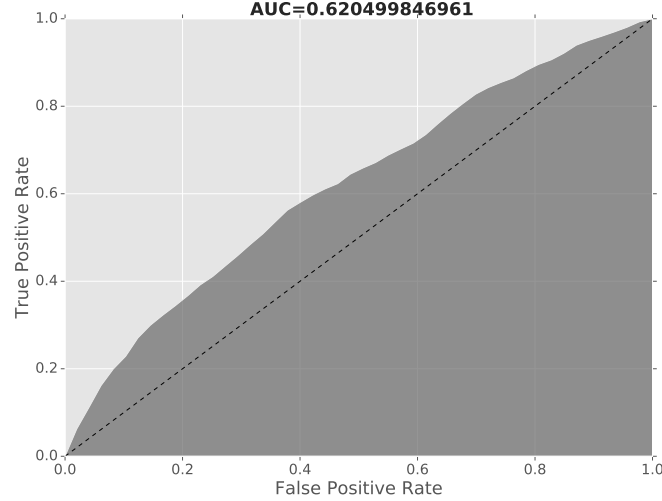


Figure 6.9: Receiver Operating Characteristic For Fault Revealing Mutants Prediction on Codeflaws

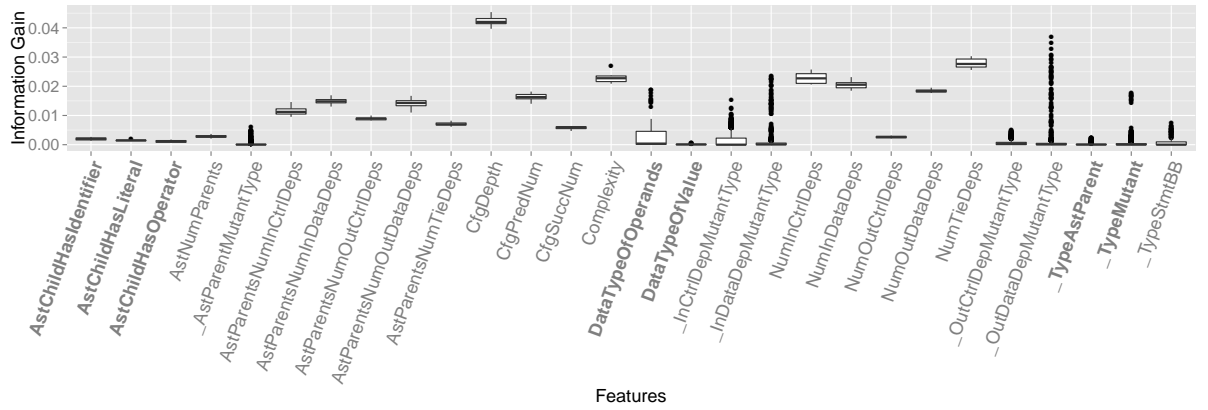


Figure 6.10: Information Gain distributions of ML features on Codeflaws

mistakes while some others may not, the only thing we can hope for is to form good heuristics, i.e., identify mutants that maximize the chances to reveal faults. Therefore, it is hard to get much higher AUC values. Nevertheless, we expect future research to built on and improve our results by forming better predictors.

Overall, the above results demonstrate that the code properties that were leveraged as features for characterizing mutants provide, together, a discriminative power to assess the fault revealing potential of mutants.

Considered features We provide in Figure 6.10 the distribution of information Gain values for the various features considered in this work. Information gain (IG) measures how much “information” a feature gives us about the class we want to predict. The IG values are computed by the supervised learning algorithm during the training process. These data enable the assessment of the potential contribution of every feature to a prediction model. Experimental training process provides evidence in Figure 6.10 that the suggested features (in bold) contribute significantly less than several other features that we have designed for *FaRM*. Interestingly, together with complexity, the features related to control and data dependencies are the most informative ones. Here we should note that IG values do not suggest which features to select and which not.

Actually our results show that we need all the features.

6.6.3 Mutant selection

6.6.3.1 Comparison with Random (RQ3)

Figure 6.11 shows the distribution of the fault revelation of the mutant selection strategies when selecting the 2%, 5% and 10% of the top ranked mutants. As can be seen from the plot, both *FaRM** and *FaRM* outperforms both DummyRandom and spreadRandom. Both DummyRandom and spreadRandom outperform *PredKillable*. When selecting 2% of the mutants the difference, for both *FaRM* and *FaRM**, of the median values is 22% and 24% for the DummyRandom and SpreadRandom respectively. This difference is increasing when selecting the 5% of the mutants and goes to 34% and 34% for *FaRM* and, 24% and 24% for *FaRM**. When selecting 10% of the mutants the difference becomes 20% and 17% for both *FaRM* and *FaRM**. Regarding *PredKillable*, the difference with DummyRandom and SpreadRandom at the 2% mutant selection threshold is 23% and 21% respectively. This difference increase for the 5% to 37% and 37%. For the 10% threshold is 43% and 46%.

To check whether the differences are statistically significant we performed a Wilcoxon rank-sum test, which is a non-parametric test that measures whether the values of one sample are higher than those of the second sample. We adopt a statistically significant level $\alpha < 0.05$ (note that the same results apply with the significance level $\alpha < 0.01$) below of which we consider the differences as statistically significant. We also computed the Vargha Delaney \hat{A}_{12} effect size value between the approaches.

The statistical test showed that *FaRM* and *FaRM** outperforms both DummyRandom and SpreadRandom with statistically significant difference. both DummyRandom and SpreadRandom outperform *PredKillable* with statistically significant difference. As expected the differences between DummyRandom and SpreadRandom are not significant. It is noted that all comparisons are aligned with respect to the number of mutants that need analysis, which as we already explained represents the manual effort involved. The Vargha Delaney \hat{A}_{12} value between the approaches show that for the 2% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 60% and 63% of the cases respectively. These values are slightly higher for *FaRM** where it is better than DummyRandom and SpreadRandom in 62% and 65% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 84% and 82% of the cases. For the 5% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 66% of the cases. *FaRM** is better than DummyRandom and SpreadRandom in 64% and 65% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 88% and 84% of the cases. For the 10% threshold, *FaRM* is better than DummyRandom and SpreadRandom in 65% and 63% of the cases respectively. *FaRM** is better than DummyRandom and SpreadRandom in 64% and 61% of the cases respectively. DummyRandom and SpreadRandom are respectively better than *PredKillable* in 87% and 85% of the cases.

Regarding the test execution time of the involved methods, our approach has an advantage but this is minor. The median difference between *FaRM* and DummyRandom and SpreadRandom was 12 and 39 seconds per program respectively. This means that *FaRM* required 12 and 29 seconds less execution time than the random baselines. While these differences are considered

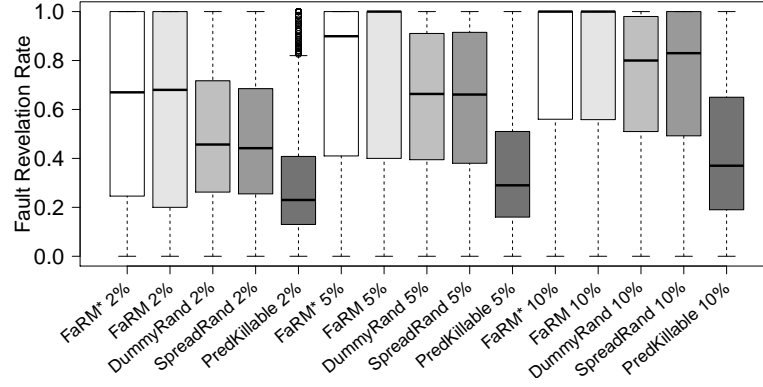


Figure 6.11: Fault revelation of the mutant selection strategies on Codeflaws. All three FaRM and FaRM* sets outperform the random baselines.

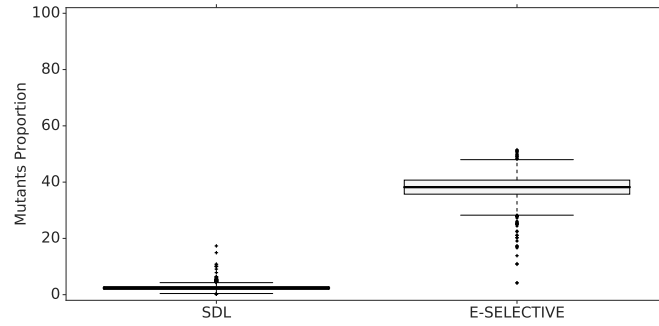


Figure 6.12: Proportion of SDL and E-SELECTIVE mutants among all mutants for Codeflaws subjects.

as minor they demonstrate that *FaRM* has significantly higher fault revelation ability than the compared baselines without introducing any major overhead.

Overall, our results suggest that *FaRM* and *FaRM** significantly outperforms the random baselines with practically significant differences, i.e., improvements on the ratios of revealed faults were between 4% to 34%. *PredKillable* is outperformed by all the approaches.

6.6.3.2 Comparison with SDL & E-Selective (RQ4)

This section aims to compare the fault revelation of our approach with that of the SDL and the E-Selective mutants sets.

In order to compare our approach with SDL selection, the selection size is set to the number of SDL mutants. In the Codeflaws subjects, SDL and E-SELECTIVE mutants represent in median respectively 2% and 38% of all mutant as seen in Figure 6.12.

Our analysis is designed as following. For each subject, the $|SDL|$ top ranked mutants of *FaRM* are selected (where $|SDL|$ is the total number of SDL mutants). We also select the $|SDL|$ top ranked mutants with the random approaches. Then, the fault revelation of each approach's selected mutants set is computed for comparison and presented in Figure 6.13. We observe that *FaRM* and *FaRM** respectively have 30% and 27% higher median fault revelation than SDL. *PredKillable* has 25% lower median fault revelation than SDL. We also observe that SDL has similar fault revelation with the random selections (respectively 3% and 2% lower than DummyRandom and SpreadRandom).

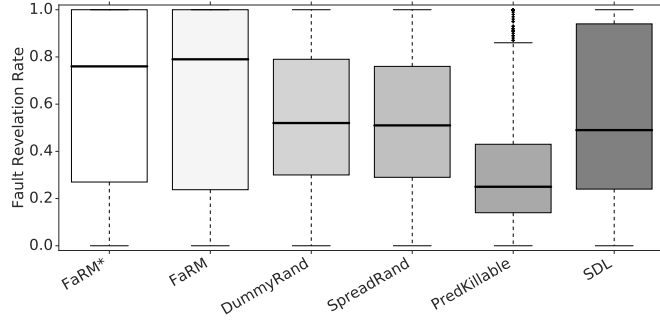


Figure 6.13: Fault revelation of FaRM compared with SDL on Codeflaws. FaRM sets outperform the SDL selection. Approximately 2% (number of SDL mutants) of all the mutants are selected.

We also performed the Wilcoxon rank-sum test as in section 6.6.3. The statistical test showed that both *FaRM* and *FaRM** outperform SDL, and SDL outperforms *PredKillable*. The difference between SDL and DummyRandom and SpreadRandom is not statistical significant. We also computed the Vargha Delaney \hat{A}_{12} value between the approaches and found that *FaRM* and *FaRM** are respectively better than SDL in 54% and 55% of the cases. SDL is better than *PredKillable* in 79% of the cases.

Similar to the experiment performed above to compare our approach with SDL, we perform another experiment to compare *FaRM* with E-Selective selection. The fault revelation results are presented in Figure 6.14. We observe that for a selection size equal to the number of E-Selective mutants, all selection approaches except *PredKillable* and DummyRandom achieve the highest median fault revelation. Given that E-Selective mutants are roughly 38% of all the mutants, which is relative large set, we make the comparison with the E-Selective set for smaller selection size namely 5% and 15% thresholds of the top ranked mutants (w.r.t all mutants). The E-Selective mutants of the given sizes are randomly selected from the whole E-Selective mutant set. The fault revelation results are presented in Figures 6.15 and 6.16. We can observe that *FaRM* and *FaRM** respectively have 31% and 22% higher median fault revelation than E-Selective for thresholds 5%. For the 15% threshold, both have 9% higher median fault revelation. *PredKillable* has 38% and 47% lower median fault revelation than E-Selective for thresholds 5% and 15% respectively. We also observe that E-Selective has similar fault revelation with the random selections (respectively 2% and 1% higher than DummyRandom and SpreadRandom for selection size 5% and respectively 3% and 0% higher than DummyRandom and SpreadRandom for selection size 15%).

The Wilcoxon rank-sum statistical test shows that both *FaRM* and *FaRM** outperform E-Selective, and E-Selective outperforms the *PredKillable*. The difference between E-Selective and the random approaches is not statistical significant. We also computed the Vargha Delaney \hat{A}_{12} effect size value between the approaches and found that for the 5% and 15% thresholds, *FaRM* is better than E-Selective in 64% and 63% of the cases respectively. *FaRM** is better in 62% and 61% of the cases respectively, and *PredKillable* is worse in 86% and 82% of the cases respectively.

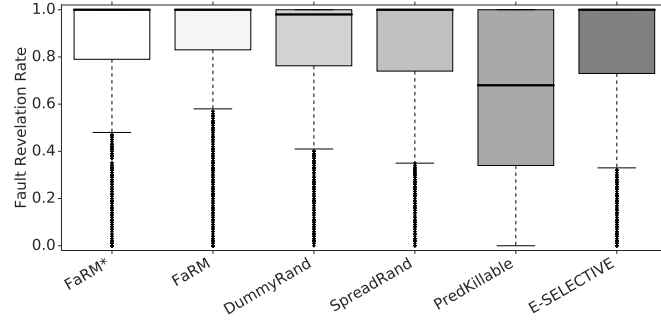


Figure 6.14: Fault revelation of FaRM compared with E-Selective on Codeflaws. Approximately 38% (number of E-Selective mutants) of all the mutants are selected.

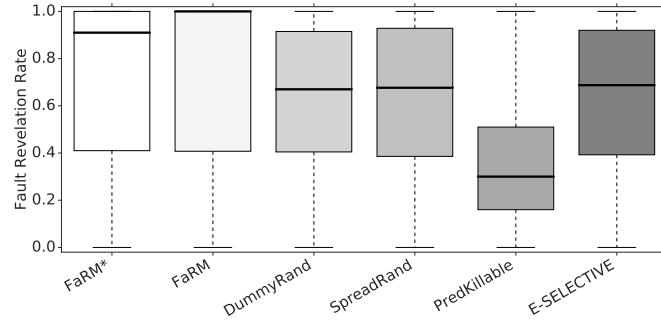


Figure 6.15: Fault revelation of FaRM compared with E-Selective for selection size 5% of all mutants. FaRM and FaRM* sets outperform E-Selective selection.

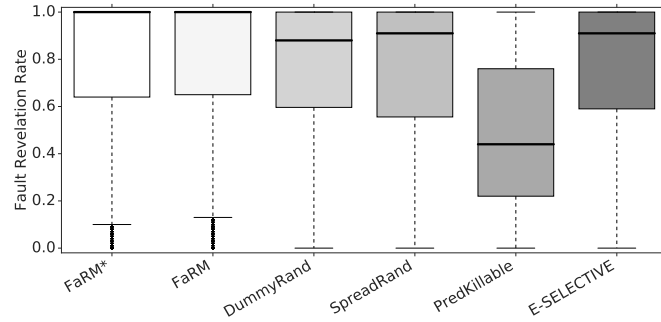


Figure 6.16: Fault revelation of FaRM compared with E-Selective for selection size 15% of all mutants. FaRM sets outperform E-Selective selection.

6.6.4 Mutant prioritization

6.6.4.1 Comparison with Random (RQ5)

Selected Mutants Cost Metric.

Figure 6.17 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While *FaRM* and *FaRM** respectively yield an APFD median of 98% and 97%, and *PredKillable* yields an APFD median of 72%, *DummyRandom* and *SpreadRandom* reach median APFD values of 93% and 94% respectively. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline, when the main cost factor (number of mutants that need analysis) is aligned, we can infer that it is generally better with practically important differences (of 4%). Note that the highest possible improvement over the random baseline is 6%

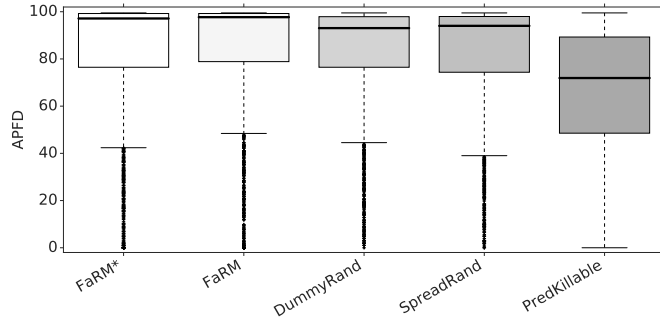


Figure 6.17: APFD measurements considering all mutants for the selected mutants cost metric for Codeflaws. The FaRM prioritization outperform the random baselines.

(DummyRandom has a median APFD value of 94%). Nonetheless, *PredKillable* is worse than the random baseline.

To account for the stochastic nature of the compared methods and increase the confidence on our results, we further perform a statistical test. Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., samples of *FaRM* and *DummyRandom*, *FaRM* and *SpreadRandom*, *FaRM** and *DummyRandom*, *FaRM** and *SpreadRandom*, *PredKillable* and *DummyRandom*, and *PredKillable* and *SpreadRandom* respectively. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperform random mutant selection with statistical significance while random mutant selection outperform *PredKillable*. On the other hand, as expected, the Wilcoxon test revealed that there is no statistical difference between the performance of *DummyRandom* and that of *SpreadRandom*.

When examining mutant selection strategies there are two main parameters that influence the application cost. These are the killable and equivalent mutants that testers need to analyse. When analysing a killable mutant our ability to select fault revealing ones is important, while increasing the chance to get a killable mutant is also important. Therefore, it could be that our *FaRM* is better simply because it selects killable mutants and not fault revealing ones. To account for this factor we removed all non-killable mutants from our sets and recompute our results. This helped eliminating the influence of non-killable mutants, from both approaches.

Our results show that the performance improvement of *FaRM* and *FaRM** over *SpreadRandom* and *DummyRandom* is also effective when considering only killable mutants (approximated by our test suites). Figure 6.18 shows the relevant distributions of APFD, which are visibly similar to the distributions for all mutants (all values are slightly higher when considering only killable mutants). This result suggest that *FaRM* and *FaRM** are indeed capable of identifying fault revealing mutants, independent of the equivalent mutants involved.

To provide a general view of the trends, Figure 6.19 illustrates the overall (median) effectiveness of the mutant prioritization by *FaRM*, *FaRM** and *PredKillable* in comparison with random strategies. We note that for all percentages of mutants, *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 20% (maximum difference is 34%) when selecting 2% to 8% of mutants. *FaRM* reaches a plateau around 5% of mutants, as the median fault revelation is maximal. This suggests that a hint for the mutant selection size for *FaRM* could be 5% of the mutants.

Finally, we examined the differences between the approaches in terms of execution time. Al-

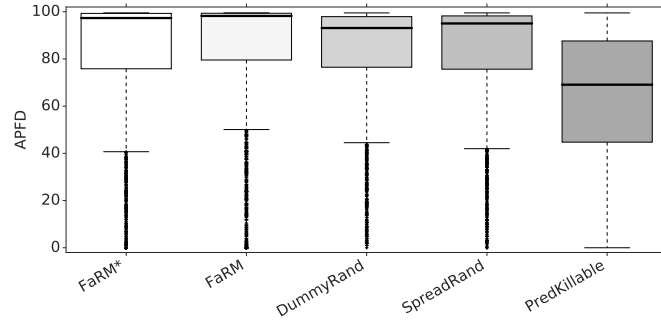


Figure 6.18: APFD measurements considering only killable mutants for the selected mutants cost metric on CodeFlaws. The FaRM prioritization outperform the random baselines, independent of non-killable mutants.

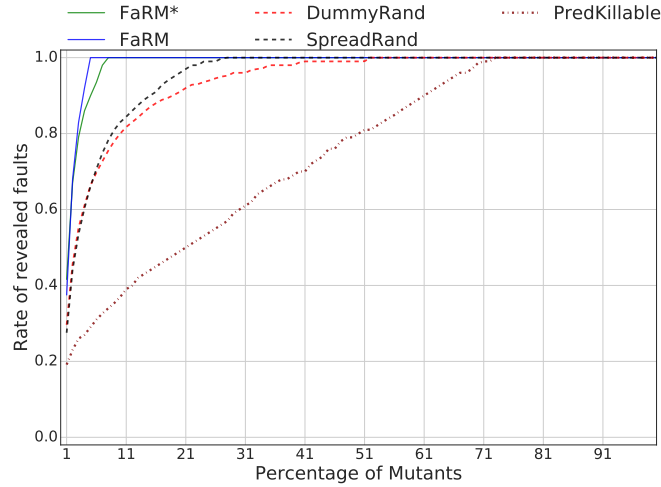


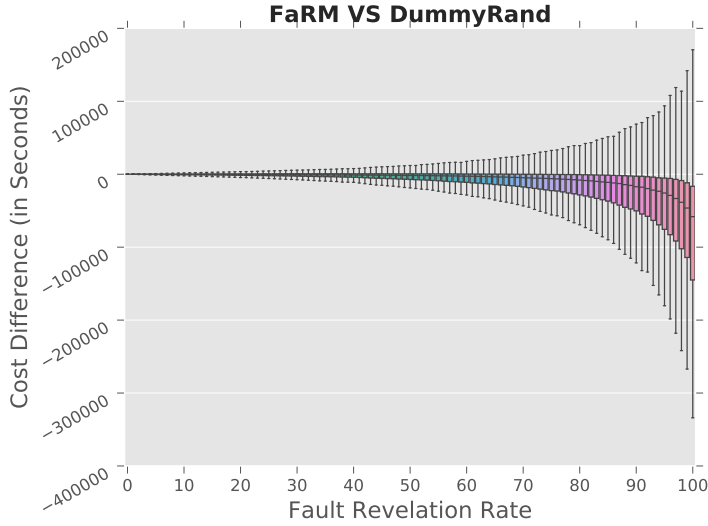
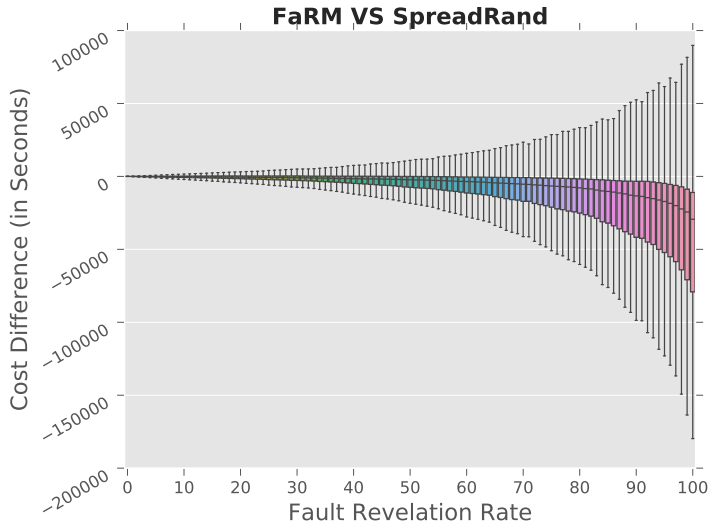
Figure 6.19: Mutant prioritization performance in terms of faults revealed (median case) for the selected mutants cost metric on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies.

though we do not explicitly aim at reducing the test execution cost, we expect some benefits due to our methods' ability to prioritise the mutants, which results in a reduced execution time [ZMK13]. Figure 6.20 illustrate, in a box-plot form, the overall execution time differences between the *FaRM* and the random baselines with respect to the attained fault revelation, measured in seconds. Although the differences can be significant in some (rare) cases, the expected (median values) ones are -58,167 and -29,373 seconds (-16 and -8 hours) for DummyRandom and SpreadRandom. This result indicates that our approach has also an advantage with respect to test execution, which sometimes becomes significant.

Conclusively, our results demonstrate that *FaRM* is indeed effective as it is statistically superior to random baselines, independent of the equivalent mutants involved. It provides 4% higher APFD values, which means that when testers analyse mutants (to strengthen their test suites) they get a 4% improvement on their fault revelation ability. Note that the highest possible improvement over the random baseline is 6% (DummyRandom has a median APFD value of 94%).

Required Tests Cost Metric.

Figure 6.21 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While both *FaRM* and *FaRM** yield an

(a) Cost of *FaRM*- Cost of DummyRand.(b) Cost of *FaRM*- Cost of SpreadRand**Figure 6.20:** Execution cost of prioritization schemes

APFD median of 81%, and *PredKillable* yields an APFD median of 76%, DummyRandom and SpreadRandom reach median APFD values of 77%. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline, when the main cost factor (number of test that need to be designed and executed) is aligned, we can infer that it is generally better with practically important differences (of 4%). The *PredKillable* performs quite similarly to the random baseline.

To account for the stochastic nature of the compared methods and increase the confidence on our results, we further perform a statistical test. Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., each of *FaRM* and *FaRM** compared with each of *PredKillable*, DummyRandom and SpreadRandom. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperform random baseline with statistical significance. On the other hand, the Wilcoxon test revealed that there is no statistical difference between the performance of *PredKillable*, DummyRandom and SpreadRandom.

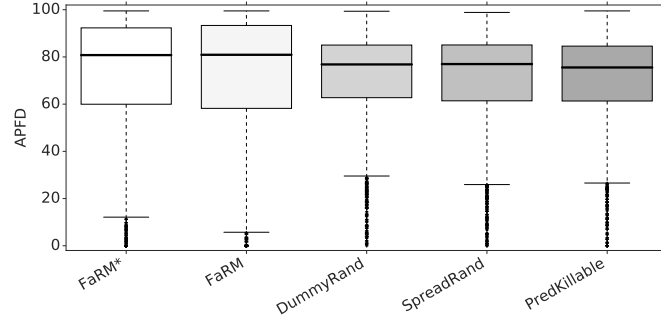


Figure 6.21: *APFD measurements for the required tests cost metric on CodeFlaws. The FaRM prioritization outperform the random baselines.*

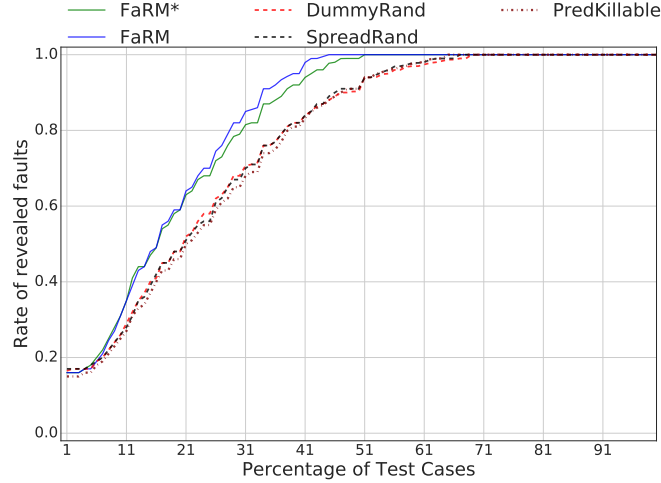


Figure 6.22: *Required tests prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies.*

The results of the Vargha Delaney effect size show that *FaRM* is better than *DummyRandom*, *SpreadRandom* and *PredKillable* in 58%, 61% and 60% of the cases respectively. *FaRM** is better than *DummyRandom*, *SpreadRandom* and *PredKillable* in 58%, 61% and 59% of the cases respectively.

To provide a general view of the trends, Figure 6.22 illustrates the overall (median) effectiveness of the required test prioritization by *FaRM*, *FaRM** and *PredKillable* in comparison with random strategies. We note that for all percentages of tests, *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 10% (maximum difference is 15%) for the 20% to 45% top ranked tests.

Analysed Mutants Cost Metric.

The analysed mutants cost metric measures the minimum number of mutants that need to be analysed, including equivalent mutants, following a mutant prioritization approach, before the fault is revealed. A good mutant prioritization approach will minimize the analysed mutants cost. Following, we compare the analysed mutants cost metric between our approaches and the random baselines. The analysed mutants cost metric is calculated for each approach and for each bug of the benchmark. We compare the approaches statistically with Wilcoxon rank-sum test and the Vargha Delaney effect size.

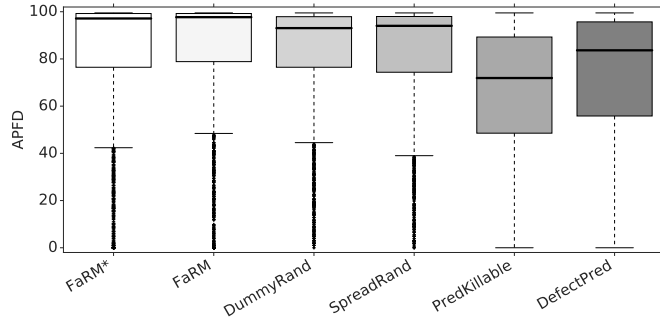


Figure 6.23: APFD measurements considering all mutants. The FaRM prioritization outperform the defect prediction.

The results show that *FaRM*, *FaRM** and *PredKillable* are better than *DummyRandom* and *SpreadRandom* with statistical significance displayed by a p-value much lower than the significance level. *FaRM* is better than *DummyRandom* and *SpreadRandom* in 57% and 61% of the cases respectively. The performance difference is higher for *FaRM** where it is better than *DummyRandom* and *SpreadRandom* in 60% and 64% of the cases respectively. *PredKillable* is better than *DummyRandom* and *SpreadRandom* in 60% and 65% of the cases respectively.

*FaRM** shows a larger improvement than *FaRM* over the random baseline, but there is no statistical significance difference between *FaRM* and *FaRM**. Furthermore, *FaRM** outperforms *PredKillable* with statistical significant difference, and is better in 53% of the cases. There is no statistical significant difference between *FaRM* and *PredKillable*.

Conclusively, our results demonstrate that *FaRM* and *FaRM** are indeed effective as they are statistically superior to random baselines.

6.6.4.2 Comparison with Defect Prediction (RQ6)

Selected Mutants Cost Metric. Figure 6.23 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the *FaRM*, *FaRM**, *PredKillable* and the random approaches. While *FaRM* yields an APFD median of 98.0%, defect prediction (*DefectPred*) reach median APFD value of 83.7%. These results reveal that the general trend is in favour to our approach. As our approach is much better than the defect prediction approach, when the main cost factor (number of mutants that need analysis) is aligned, we can infer that it is generally better with practically important differences (of 14%). Even the random approaches are better than the defect prediction approach. Nevertheless, *PredKillable* is worse than defect prediction.

The Wilcoxon test results yielded p-values much lower than our significance level for the samples of *FaRM* and *DefectPred*, and *FaRM** and *DefectPred*. Therefore, we can definitively conclude that *FaRM* and *FaRM** outperforms defect prediction with statistical significance. On the other hand, the Wilcoxon test also revealed that there is statistical significant difference between the performance of *DefectPred* and *dummyRandom*, and *DefectPred* and that of *spreadRandom* respectively. Nonetheless, *DefectPred* outperforms *PredKillable* with statistical significance. The Vargha Delaney \hat{A}_{12} effect size value shows that *FaRM* and *FaRM** are better than *DefectPred* in 76% of cases. While *DummyRandom* and *SpreadRandom* are better than *DefectPred* in 71% and 70% of the cases respectively.

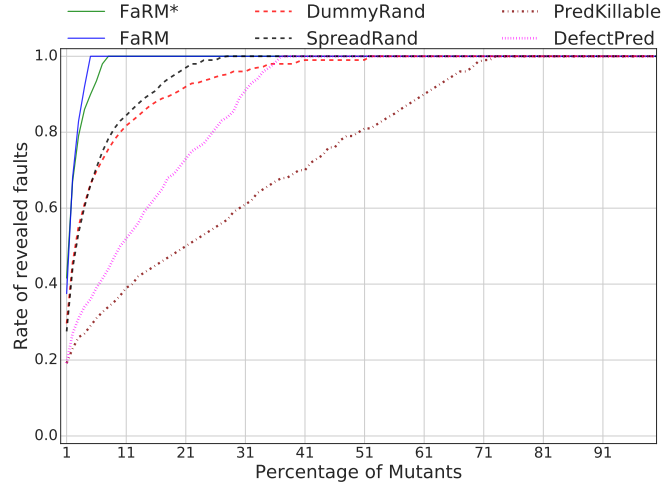


Figure 6.24: Mutant prioritization performance in terms of faults revealed (median case) on CodeFlaws. The x-axis represent the number of considered mutants. The y-axis represent the ratio of the fault revealed by the strategies.

To provide a general view of the trends, Figure 6.24 illustrates the overall (median) effectiveness of the mutant prioritization by *FaRM* in comparison with the defect prediction approach. We note that for all percentage of mutants, *FaRM* outperforms the defect prediction approach. The performance improvement goes around 40% to 66% of more faults revealed when 2% until 8% of mutants are executed.

6.6.5 Experiments with large programs (RQ7)

Selected Mutants Cost Metric.

in CoREBench, all APFDs values are much higher than in CodeFlaws, with *FaRM*, *FaRM**, DummyRandom and SpreadRandom having median APFD value of 99%, and *PredKillable* a median APFD value of 94%. The maximum possible improvement is 1% (given that the random baseline has a median of 99%). This is caused by the large number of redundant mutants involved. To demonstrate this we check the relation between mutation score and percentage of considered mutants. Figure 6.27 illustrates the overall (median) mutation score achieved (y-axis) by the tests killing the percentage of mutants recorded in x-axis. From this graph we can see that all approaches reach their maximum median mutation score value when considering more than 30% of the mutants. This implies that the benefits are reduced for every approach that consider more than 30% of the involved mutants.

Interestingly, both Figure 6.27 and Figure 6.28 demonstrate that *FaRM* guides the mutant selection towards mutants that do not maximize the mutation score nor the subsuming mutation score (random mutant selection achieves higher mutation and subsuming mutation scores than *FaRM*). Instead the selected mutant maximize fault revelation as demonstrated in Figures 6.25 and 6.26.

Given that a large proportion of the mutants are not killable (Figure 6.27), we present in Figure 6.29 the sensitivity of the approaches with regard to the equivalent mutants, to see how they are ranked. We observe that *PredKillable* does quite well at ranking the killable mutants first, and *FaRM** inherit of such characteristic from *FaRM* relatively well. We also observe that

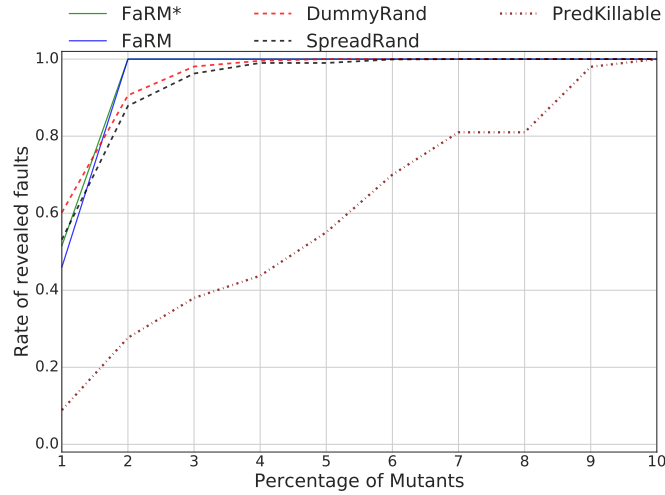


Figure 6.25: FaRM performance in terms of faults revealed (median case) on CoREBench considering all mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies.

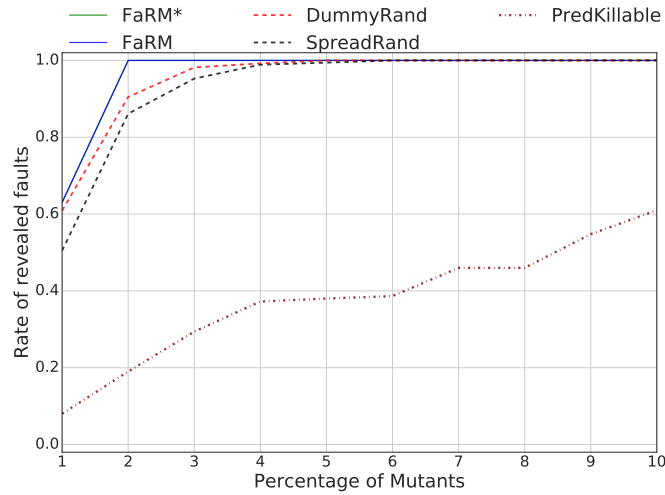


Figure 6.26: FaRM performance in terms of faults revealed (median case) on CoREBench considering only killable mutants. The x-axis represent the number of considered mutants, while the y-axis represent the ratio of the fault revealed by the strategies.

FaRM tend to keep equivalent mutants away from the top ranks.

To provide a general view of the fault revelation trend, Figures 6.25 and 6.26 illustrate the overall (median) effectiveness of the mutant prioritization by *FaRM* in comparison with random strategies for the ratios of selected mutants from 1% to 10%. We note that for all percentage of mutants, *FaRM* outperforms random-based prioritization. The performance improvement goes from 0% to 10% of more faults revealed when 5% and 2% of mutants are killed. These trends are similar with those we observe on CodeFlaws, suggesting that *FaRM* effectively learns the properties of the important mutants.

Required Tests Cost Metric.

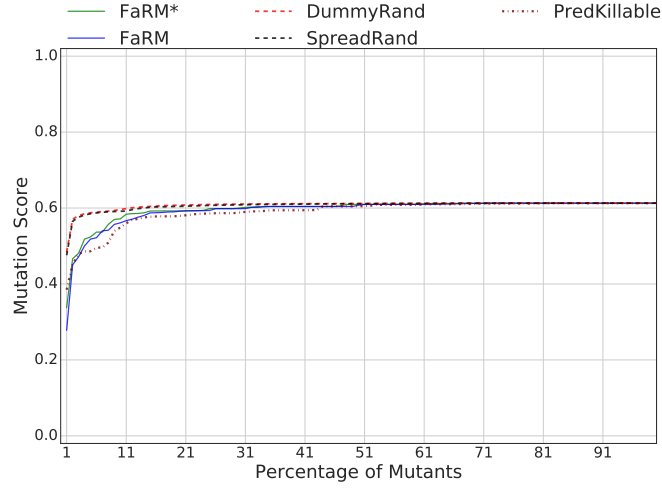


Figure 6.27: Mutation score (median case) on CoREBench. The x -axis represent the number of considered mutants, while the y -axis represent the mutation score attained by the strategies.

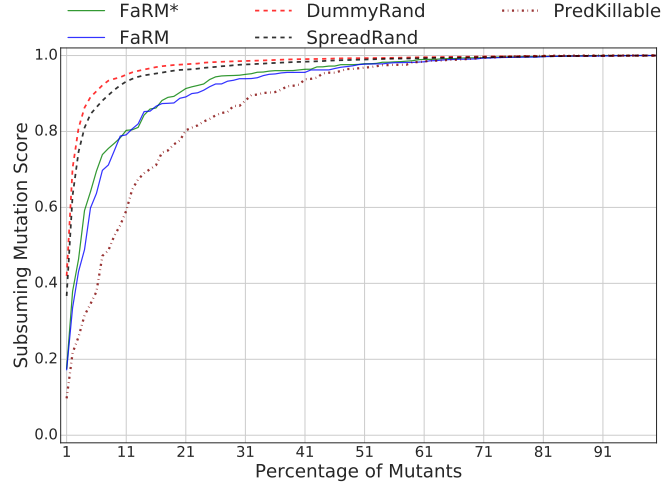


Figure 6.28: Subsuming Mutation score (median case) on CoREBench. The x -axis represent the number of considered mutants, while the y -axis represent the subsuming mutation score attained by the strategies.

Figure 6.30 shows the distributions of APFD (Average Percentage of Faults Detected) values for all faults, using the five approaches under evaluation. While both *FaRM* and *FaRM** yield an APFD median of 92%, and *PredKillable* yields an APFD median of 79%, DummyRandom and SpreadRandom reach median APFD values of 83% and 81% respectively. These results reveal that the general trend is in favour to our approach. As our approaches *FaRM* and *FaRM** are better than the random baseline, when the main cost factor (number of test that need to be designed and executed) is aligned, we can infer that it is generally better with practically important differences (of 9%). The *PredKillable* performs slightly worse than the random baseline.

The results of the Vargha Delaney \hat{A}_{12} effect size show that *FaRM* is better than DummyRandom, SpreadRandom and *PredKillable* in 74%, 77% and 86% of the cases respectively. *FaRM** is better than DummyRandom, SpreadRandom and *PredKillable* in 70%, 74% and 81% of the cases respectively. *PredKillable* is worse than the DummyRandom and SpreadRandom in 70% and 66% of the cases respectively.

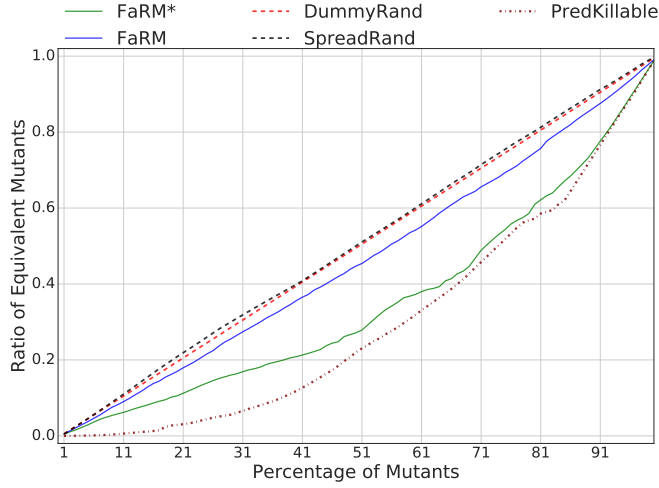


Figure 6.29: Ratio of equivalents (median case) on CoREBench. The x-axis represent the number of considered mutants, while the y-axis represent the proportion of equivalent mutants selected by the strategies.

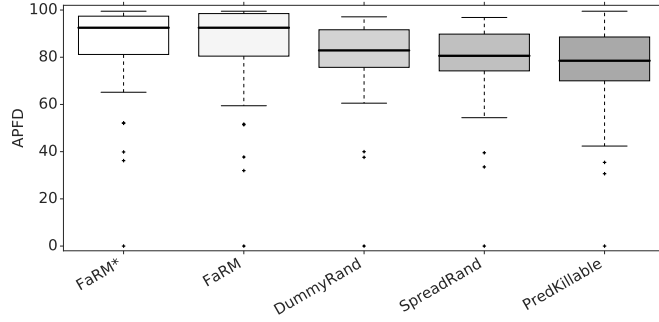


Figure 6.30: APFD measurements on CoREBench for the required tests cost metric. The FaRM prioritization outperform the random baselines.

To provide a general view of the trends, Figure 6.31 illustrates the overall (median) effectiveness of the required test prioritization by *FaRM*, *FaRM** and *PredKillable* in comparison with random strategies. We note that *FaRM* and *FaRM** outperforms random-based prioritization while *PredKillable* is outperformed by the random-based prioritization. Overall, we observe that the fault revelation benefit of *FaRM* over the random approaches is above 30% (maximum difference is 70%) for the 5% to 20% top ranked tests.

Analysed Mutants Cost Metric.

The results of the Vargha Delaney \hat{A}_{12} effect size values related to the analysed mutants cost metric show that *FaRM* and *FaRM** are better than DummyRandom and SpreadRandom. *FaRM* is better than DummyRandom and SpreadRandom in 58% and 60% of the cases respectively. The performance difference is higher for *FaRM** where it is better than DummyRandom and SpreadRandom in 61% and 63% of the cases respectively. *PredKillable* is better than DummyRandom and SpreadRandom in 56% and 58% of the cases respectively.

*FaRM** shows a larger improvement than *FaRM* over the random baseline.

Taken together our results demonstrate that *FaRM* and *FaRM** achieves significant improvements over the random baselines on both CodeFlaws and CoREBench fault sets. Therefore, the

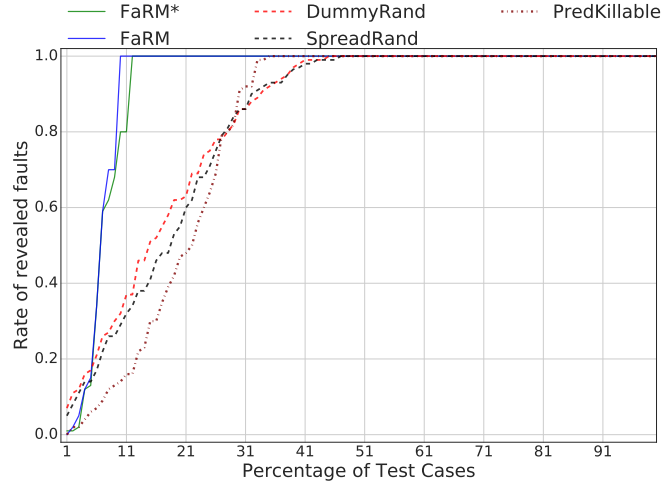


Figure 6.31: Required tests prioritization performance in terms of faults revealed (median case) on CoREBench. The x-axis represent the number of considered tests. The y-axis represent the ratio of the fault revealed by the strategies.

improvements made by *FaRM* and *FaRM** can be considered as important.

6.7 Discussion

6.7.1 Working Assumptions

Our approach uses machine learning to support mutation testing. As such it makes some assumptions that should hold in order to be applicable and effective. First, we assume that there are sufficient historical data from applications of the same context or previous software releases. This means that we need to have a diverse and comprehensive set of defects where mutation testing has been applied. Of course these defects need to belong to the targeted, by the testing procedure, class of defects. In the absence of sufficient defects, we can relax this requirement by training on hard-to-kill or subsuming mutants. This can be easily performed, the same way we train for equivalent mutants, as long as we have a large codebase that is sufficiently tested.

Second, we assume that defect causes are repeated. This is an important assumption as in its absence machine learning cannot work. We believe that it holds given the evidence provided by the *n*-version programming studies [Lev95; KL86] and the empirical observations in the context of Linux kernel [Pal+11].

Third, we assume that mutants are linked with targeted defects. This assumption comes with the use of mutation testing. We believe that it holds given the empirical evidence provided in Chapter 4 which is inline with recent studies [PI18; RWK17; Pap+18; Jus+14]. Finally, we assume that fault revelation utility can be captured by static features such as the ones used in this study. We are confident that this assumption holds given the reports of Petrovic and Ivankovic [PI18] on the utility of the AST features in mutant selection and the evidence we provide here.

6.7.2 Threats to Validity

We acknowledge the following threats that could have affected the validity of our results. One possible external validity threat lies in the nature of the test subjects we used. Individually, the majority of programs in comparison experiments are small in size, and may not be representative of real-world programs. Our mitigation strategy is discussed in the following subsection (section 6.7.3). Moreover, since the properties of the fault revealing mutants reside on the code parts that are control and data dependent to and from the faults, the cumulative size of relevant code parts (based on which we get the feature values) should be small. Therefore, for such a study, the most important characteristics should be the faulty code area and its dependencies. Since we have a large and diverse set of real faults, we feel that this threat is limited. Future work should validate our findings and analysis to larger programs.

Another potential threat relates to the mutation operators we used. Although we have considered a variety of operators, we cannot guarantee that they yield representative mutants. To diminish this threat we used a large number of operators (816 simple operators across 18 categories) covering the most frequently used C features. We also included all the operators adopted by the modern mutation testing tools [Pap+19].

Threats to internal validity lie in the use of recent machine learning algorithms to the detriment of established and widely used techniques. Nevertheless, these threats are minimized as gradient boosting is gaining a momentum in the research literature as well as the practice of machine learning.

Similarly, there might be some issues related to code redundancy, duplicated code, that may influence our results. We discuss our redundancy mitigation strategy on section 6.7.4.

Another internal validity threat may be due to the features we use. These have not been optimized with any feature selection technique. This is not a big issue in our case as we use gradient boosting that automatically performs feature selection. To verify this point we trained a Deep Learning model that also performs feature selection and checked its performance. The result showed insignificant differences from our method. Additionally, we retrained our classifiers using the features with information gain greater or equal to 0.02 and got results similar to random, suggesting that all our features are needed. Future research should shed light on this aspect by complementing and optimizing our feature set.

Other internal validity threats are due to the way we treated mutants as equivalent. To deal with this issue, we used KLEE, a state of the art test generation tool and the accompanied test suites. As the programs we are using are small KLEE should not have a problem at generating effective test suites. Together these tools kill 87% of all the mutants, demonstrating that our test suites are indeed strong. Since the 13% of the mutants we treat as equivalent is in line with the results reported by the literature [Pap+15], we believe that this threat is not important. Unfortunately, we cannot practically do much more than that, as the problem is undecidable [BA82].

Finally, our assessment metrics may involve some threats to construct validity. Our cost measurement, number of selected, analysed mutants and number of test cases essentially captures the manual effort involved. Automated tools may reduce this cost and hence influence our measurements. Regarding equivalent mutants, we used a state-of-the-art equivalent mutant detection technique, TCE [Pap+15], to remove all trivially equivalent mutants before conducting any experiment. Therefore, the remaining equivalent mutants are those that remain undetectable by

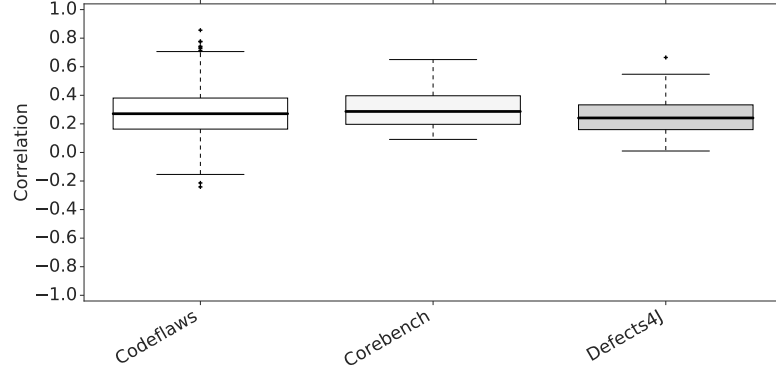


Figure 6.32: *Correlations between mutants and faults in three defect datasets. Similar correlations are observed in all three cases suggesting that Codeflaws provides good indications on the fault revealing ability of the mutants.*

the current standards. Regarding the test generation cost, we acknowledge that while automated tools manage to generate test inputs, they fail generating test oracles. Therefore, augmenting the test inputs with test oracles, remains a manual activity, which we approximate by measuring the number of tests. In our experiments we bypassed the oracle problem by using the ‘correct’ program versions as oracles. An alternative scenario involves the use of automated oracles, but these are rare in practice and we did not consider them. Overall, we believe that with the current standards, our cost measurements approximate well the human cost involved.

All in all, we aimed at minimizing any potential threats by using various comparisons scenarios, clearly evaluating the benefit of the different steps in *FaRM*, and leveraging frequently used and established metrics. Additionally, to enable replication and future research we make our data publicly available⁴.

6.7.3 Representativeness of test subjects

Most of our results are based on Codeflaws. We used this benchmark because machine learning requires lots of data and Codeflaws is, currently, the largest benchmark of real faults on C programs. Also because of its manageable size, we can automatically generate a relatively large and thorough test pool and apply mutation testing. Still this required 8,009 CPU days of computations (only for the mutant executions), indicating that we reach the experimentally achievable limits. Similarly, applying mutation testing on the 45 faults on CoREBench required 454 CPU days of computations.

The obvious differences between the size of the test subjects raise the question of whether our conclusions hold on other programs and faults. Fortunately, as already discussed our results on CoREBench have similar trends with those observed on Codeflaws. Training a classifier on CoREBench yields AUC values around 0.616, which is approximately the same (slightly lower) than the one we get from Codeflaws. This fact provides confidence that our features do capture the mutant properties we are seeking for. To further cater for this issue, we also selected the harder to reveal faults (faults revealed by less than 25% of the tests). This is a quality control practice, used in fault injection studies, ensures that our faults are not trivial.

⁴<https://mutationtesting.uni.lu/farm>

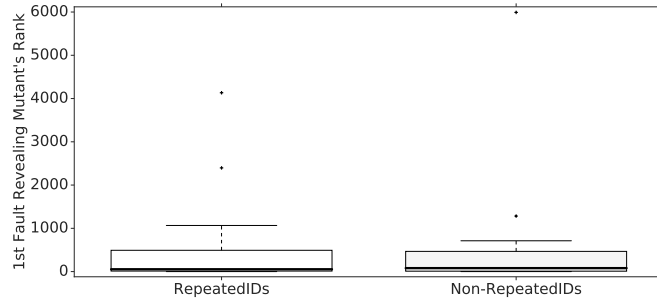


Figure 6.33: *CoREBench results on similar (repeatedIDs) and dissimilar (Non-RepeatedIDs) implementation. We observe similar trend in both cases suggesting a minor or no influence of code similarity on FaRM performance.*

Additionally, we checked the syntactic distance of the Codeflaws faults and show that it is small (please refer to Figure 6.7), similarly to the one assumed by mutation testing. This property together with the subtle faults (faults revealed by less than 25% of the tests) we select make our fault set compliant with the mutation testing assumptions, i.e., the Competent Programmer Hypothesis.

Furthermore, we computed and contrasted the correlation between mutants and faults on three defect benchmarks; CoREBench, Codeflaws and Defects4J dataset [JJE14]. Our aim is to check whether there are major difference in the relation between the faults and mutants of the three benchmarks.

Defects4J is a popular defect dataset for Java, with real faults from large open source programs. To compute the correlations on Defects4J we used the data from the study of Papadakis *et al.* [Pap+18], while for CoREBench and Codeflaws we used the data from this chapter. We computed the Kendall correlations with uncontrolled test suite size between 1% and 15% of all tests. We make 10,000 random test sets each with size randomly chosen between 1% and 15% of all the tests. Then, we compute the mutation score and the fault revelation of each test set, and compute the Kendall correlation between the mutation score and fault revelation. Figure 6.32 shows the correlations for Codeflaws, CoReBench and Defects4J. As can be seen, the correlations are similar in all three cases. Therefore, since the mutants and faults relations share similar properties on all cases, we believe that our defect set provide good indications on the fault revealing ability of our approach.

6.7.4 Redundancy between the considered faults

Code redundancy may influence our results. As depicted in Figure 6.6, in Codeflaws the number of implementations for the same problem is usually higher than one. This introduces a risk that our evaluation, test defect set, may benefit from the knowledge gained during training, in case there is another implementation for the same problem in this set. Although such a case is unlikely as all of our defects are different and form unique program versions, to remove any threat from such a factor we repeated our experiment by randomly splitting the Codeflaws subjects into training and test sets in such a way that all implementations of the same problem either appear in the training set or in the evaluation, but not both. We obtained almost identical results with the previous experiment, i.e., we get AUC values of 62% and 87% respectively for fault revelation and equivalence prediction when controlling for the implementations (having always

different problem implementations on the training and test sets). Another threat related to code redundancy may have affected our results in CoREBench. Among the 45 CoREBench faults we consider, only 20 of them are on the same components (13 in Coreutils, 3 in Find, and 4 in Grep). Note that the 13 instances of Coreutils form 3 separated set of 2, 5 and 6 bugs on the same component. We manually checked these defects and found that they all differ (they are located in different code parts and the code around the locations modified to fix the defects differs). Nevertheless, still there is a possibility that code similarities may impact positively or negatively our classifiers. Although such a case is compatible with our working scenario (we assume that we have similar historical data), so it is not a problem for our approach, it is still interesting to check the classifier performance on similar/dissimilar implementations.

To deal with this case, we divided our fault set (CoREBench) into two sets, one with the faults having similar faulty functions and one with dissimilar ones. To do so, we used the Deckard [Jia+07] tool, which computes the similarity between code instances (at the AST level). For each faulty function, the tool compares the vector representation of the sub-trees of small code snippets and reports similarity scores. Two codes are considered as similar if they have code parts with high similarity scores on the utilized abstraction, i.e., above 95% [Jia+07].

Having divided the fault sets as similar and dissimilar we then contrast the results they provide. Overall, we found insignificant differences between the two sets. Figure 6.33 compares the ranking position of the fault revealing mutants in the order provided by *FaRM*, when using the following tool parameters: similarity threshold 95, 4 strides and 50 minimum number of tokens. From these results we see that there are no significant differences between the two sets, suggesting that code redundancy does not affect our results.

In order to further reduce the threat related to code redundancy, concerning mutants that appears on same line of code in the training data and test data, we repeated the experiments by removing mutants in the test data that could cause this threat. In fact, we removed all the mutants of the test data for which there exist at least one mutant in the training data, located on the same component, that have the same features. This procedure led to removing 13% (median case) of the test data mutants. The evaluation on the remaining mutants (test data after dropping the “duplicated”) resulted in similar results as not dropping those mutants, i.e., AUC value of 62%. It is noted that 18% (median case) of the dropped mutants have different fault revelation score with their “duplicates” in training set. This has the unfortunate effect of confusing the classifier.

6.7.5 Other Attempts

Our study demonstrates how simple machine learning approaches can help improving mutation testing. Since our goal was to demonstrate the benefits of using such an approach we did not attempted to manipulate our data in any way (apart from the exclusion of the trivial faults). We achieve this goal, but still there is room for improvement that future research can exploit. For instance it is likely that classification results can be improved by pre-processing training data, e.g., exclude fault types that are problematic (see our study in Chapter 5), excluding fault types with few instances, excluding versions with low strength test suites, as well as by removing many other sources of noise.

Data manipulation strategies we attempted during our study were oversampling, the exclusive use of features with high information gain, the use of a Deep Learning classifier and targeting

irrelevant mutants (the mutants with lowest fault revealing probability). Oversampling consist of randomly duplicating the data items of the minority class in order to have a more balanced data to train the classifier. In this case, we applied oversampling of the minority class for mutants which is the fault revealing class (they represent approximately 3% of the whole data). We also attempted to replace the supervised learning algorithm used by our approach by substituting the decision tree with a deep neural network classifier. We also retrained the classifier to target irrelevant mutant (mutant not killed by fault revealing tests), the motivation being that the classifier may perform better to separate irrelevant mutants than fault revealing ones. All these attempts yielded quite similar or worse results with those we report and thus, we do not detail them.

In another attempt, we trained our classifier by only using the features that have highest information gain (those with $IG \geq 0.02$ in Figure 6.10) but achieved results similar to random mutant selection.

6.8 Conclusions

The large number of mutants involved in mutation testing has long been identified as a barrier to the practical application of the method. Unfortunately, the problem of mutant reduction remains open, despite significant efforts within the community. To tackle this issue, we introduce a new perspective of the problem: the fault revelation mutant selection. We claim that valuable mutants are the ones which are most likely to reveal real faults, and we conjecture that standard machine learning techniques can help in their selection. In view of this, we have demonstrated that some simple ‘static’ program features capture the important properties of the fault revealing mutants, resulting in uncovering significantly more faults (6%-34%) than randomly selected mutants.

Our work forms a first step towards tackling the fault revelation mutant selection with the use of machine learning. As such, we expect that future research will extend and improve our results by building more sophisticated techniques, augmenting and optimizing the feature set, by using different and potentially better classifiers, and by targeting specific fault types. To support such attempts we make our subjects (programs & tests), feature, kill and fault revelation matrices publicly available.

The next chapter presents the contributions of this dissertation w.r.t automated test generation for mutation testing.

KILLING STUBBORN MUTANTS VIA SYMBOLIC EXECUTION

This chapter introduces SEMu, a Dynamic Symbolic Execution technique that generates test inputs capable of killing stubborn mutants (killable mutants that remain undetected after a reasonable amount of testing). SEMu aims at mutant propagation (triggering erroneous states to the program output) by incrementally searching for divergent program behaviours between the original and the mutant versions. We model the mutant killing problem as a symbolic execution search within a specific area in the programs' symbolic tree. In this framework, the search area is defined and controlled by parameters that allow scalable and cost-effective mutant killing. We integrate SEMu in KLEE and experimented with Coreutils (a benchmark frequently used in symbolic execution studies). Our results show that our modelling plays an important role in mutant killing. Perhaps more importantly, our results also show that SEMu kills 37% and 20% more stubborn mutants than KLEE and the mutant infection strategy (strategy suggested by previous research) within a two hour time limit, respectively.

Chapter content

7.1	Introduction	109
7.2	Context	111
7.2.1	Symbolic Encoding of Programs	111
7.2.2	Symbolic Encoding of Mutants	112
7.2.3	Example	112
7.3	Symbolic Execution	113
7.4	Killing Mutants	114
7.4.1	Exhaustive Exploration	114
7.4.2	Conservative Pruning of the Search Space	115
7.4.3	Heuristic Search	116
7.5	<i>SEMu</i> Cost-Control Heuristics	117
7.5.1	Pre Mutation Point: Controlling for Reachability	117
7.5.2	Post Mutation Point: Controlling for Propagation	118
7.5.3	Controlling the Cost of Constraint Solving	119
7.5.4	Controlling the Number of Attempts	119
7.6	Empirical Evaluation	119
7.6.1	Research Questions	119
7.6.2	Test Subjects	120
7.6.3	Employed Tools	121
7.6.4	Experimental Setup	121
7.6.5	Experimental Settings and Procedure	123
7.6.6	Threats to Validity	124
7.7	Empirical Results	124
7.7.1	Killing ability of <i>SEMu</i>	124
7.7.2	Comparing <i>SEMu</i> with KLEE	125
7.7.3	Comparing <i>SEMu</i> with infection-only	125
7.8	Conclusion	127

7.1 Introduction

Deep testing is often required in order to assess the core logic and the ‘critical’ parts of the programs under analysis. Unfortunately, performing thorough testing is hard, tedious and time consuming. As a result testing the most important program parts requires substantial efforts, skills and experience.

To deal with this issue, mutation testing aims at guiding the design of strong (likely fault revealing) test cases. The key idea of mutation is to use artificially introduced defects, called mutants, to identify the weaknesses of test suites (undetected defects form test suite deficiencies) and to guide test generation (undetected defects form test objectives). Thus, testers can improve their test suites by designing test cases that take the mutation feedback into account.

Experience with mutation testing has shown that it is relatively easy to detect a large number of mutants by simply covering them [ADO14; Pap+16; PI18]. Such trivial mutants are not useful as they fail to provide any particular guidance towards test case design [SZ13]. However, experience has also shown that there are some few mutants that are relatively hard to detect (a.k.a. stubborn mutants [YHJ14]) and can provide significant advantages when used as test objectives [PI18; YHJ14]. Interestingly, as we show in Chapter 4, these mutants form special corner cases and are linked with fault revelation. The importance of using the stubborn mutants as test objectives has also been underlined by several industrial studies [Del+18; BH13] including a large study with Google developers [PI18].

Stubborn mutants are hard to detect mainly due to a) the difficulty of infecting the program state (causing an erroneous program state when executing the mutation/defective point) and b) due to the masking effects that prohibit the propagation of erroneous states to the program output (aka failed error propagation [And+14] or coincidental correctness [Abo+19]). Either being the case, the issues linked with these mutants form corner cases which are most likely to escape testing (since stubborn mutants form small semantic deviations) [PI18].

Killing stubborn mutants (designing test cases that reveal undetected mutants) is challenging due to the variety of the code paths, constraints and data states of the program versions (original and mutant versions) that need to be differentially analysed. The key challenge here regards the handling of the failed error propagation (masking effects), which is prevalent in stubborn mutants. Effective error propagation analysis is still an open problem [Pap+19; Piz+19] as it involves state comparisons among the mutant and the original program executions that grow exponentially with the number of the involved paths (from the mutation point to the program output).

We present *SEMu*, an approach based on dynamic symbolic execution that generates test inputs capable of killing stubborn mutants. The particular focus of *SEMu* is on the effective and scalable handling of mutant propagation. Our technique executes both the original and mutant program versions with a single symbolic execution instance, where the mutant executions are “forked” when reaching the mutation points. The forked execution follows the original one and compares with it. The comparisons are performed based on the involved symbolic states and related (propagation) constraints that ensure divergences.

A key issue with both symbolic execution and mutation testing regard their scalability. To account for this problem, we develop a framework that allows defining the mutant killing problem

as a search problem within a specific area around the mutation points. This area is defined by a number of parameters that control the symbolic exploration. We thus, perform a constrained symbolic exploration, starting from a pre-mutation point (a point in the symbolic tree that is before the mutation point) and ending at a post-mutation checkpoint (a point after the mutant) where we differentially compare the symbolic states of the two executions (forked and original) and generate test inputs.

We assume the existence of program inputs that can reach the areas we are targeting. Based on these inputs, we infer preconditions (a set of consistent and simplified path conditions), which we use to constrain the symbolic exploration to only a subset of program paths that are relevant to the targeted mutants. To further restrict the exploration to a relevant area, we systematically analyse the symbolic tree up to a relatively small distance from the mutation point (performing a shallow propagation analysis).

To improve the chances for propagation we also perform a deep exploration of some subtrees. Overall, by controlling the above parameters we can define strategies with trade-offs between space (cost) and deepness (effectiveness). Such strategies allow the differential exploration of promising code areas, while keeping their execution time low.

Many techniques targeting mutation-based test generation have been proposed [Ana+13; Pap+19]. However, most of these techniques focus on generating unit-level test suites from scratch, mainly by either covering the mutants or by causing an erroneous program state at the mutation point. However, there is no work leveraging the value of existing tests to perform deep testing by targeting stubborn mutants, which are mostly hard to propagate. Moreover, none of the available symbolic execution tools aim at generating test inputs for killing mutants.

We integrated *SEMu*¹ into KLEE [CDE08]. We evaluated *SEMu* on 47 programs from Coreutils, real-world utility programs written in C, and compare it with the mutant infection strategy, denoted as infection-only, that was proposed by previous work [Zha+10a; HJL11]. Our results show that *SEMu* achieves significantly higher killing rates (approximately +37% and +20%) of stubborn mutants, for both KLEE (alone) and infection-only strategy, on the majority of the studied subjects.

In summary, this chapter makes the following contributions:

1. We introduce and implement a symbolic execution technique for generating tests that kill stubborn mutants. Our technique leverages existing tests in order to perform a deep and targeted test of specific code areas.
2. We model the mutant killing as a search problem within a specific area (around the mutation point). Such a modelling allows controlling the symbolic execution cost, while at the same time allows forming cost-effective heuristics.
3. We report empirical results demonstrating that *SEMu* has a strong mutant killing ability, which is significantly superior to KLEE and other mutation-based approaches.

¹Github link omitted due to double blind review process. In case of acceptance, the tool will be publicly available.

7.2 Context

Our work aims at the automatic test input generation for specific methods/components of the systems under test. Our working scenario assumes that testers have performed some basic testing and want to dig into some specific parts of the program. This is a frequent scenario used to increase confidence on the critical code parts (encode the core program logic) or on parts that testers feel uncertain. To do so, it is reasonable to use mutation testing by adding tests that detect the surviving mutants (mutants undetected by the existing test suite) [SZ13; YHJ14].

We consider a mutant as detected (killed) by a test when its execution leads to different observable output from that on the original program. According to our scenario, the targeted mutants are those (killable) that survive a reasonable amount of testing. This definition depends on the amount of the performed testing; strong test suites kill more mutants than weak ones, while ‘adequate’ test suites kill them all [YHJ14; AO08].

To adopt a baseline for basic or ‘reasonable amount of testing’ we augment the developer test suites with KLEE. This means that the *stubborn mutants are those that are killable and survive the developer and automatically generated test suites*. The surviving mutants form the objectives for our test generation technique.

7.2.1 Symbolic Encoding of Programs

Independently of its language, we define a program as follows.

Definition 7.2.1. *A program is a Labeled Transition System (LTS) $\mathcal{P} = (C, c_0, C_{out}, V, eval_0, T)$ where:*

- C is a finite set of control locations;
- $c_0 \in C$ is the unique entry point (start) of the program;
- $C_{out} \subset C$ is the set of terminal locations of the program;
- V is a finite set of variables;
- $eval_0$ is a predicate capturing the set of possible initial valuations of V ;
- $T : C \times GC \rightarrow C$ is a deterministic transition function where each transition is labeled with a guarded command of the form $[g]f$ where g is a guard condition and f is a function updating valuation of variables V (GC is the set of labels).

The LTS modelling a given program defines the set of control paths from c_0 to any $c_{out} \in C_{out}$. A path is a sequence of n connected transitions $\pi_P = \langle (c_0, gc_0, c_1), \dots, (c_{n-1}, gc_{n-1}, c_{n=out}) \rangle$ such that $(c_i, gc_i, c_{i+1}) \in T$ for all i . Any well-terminating execution of the program goes through one such path. Since we consider deterministic programs, this path is unique and determined by the initial valuation (i.e. the test input) v_0 of the variables V . More precisely, each path π_P defines a *path condition* $\phi(\pi_P)$ which symbolically encodes all executions going through π_P . This path condition consists of a Boolean formula such that the test with input v_0 executes through π_P iff $v_0 \models \phi(\pi_P)$. By solving $\phi(\pi_P)$ (e.g. with a constraint solver like Z3 [DB08]), one can obtain an initial valuation satisfying the path condition, thereby obtaining a test input that goes through the corresponding program path.

The execution of the resulting test input is a sequence of $n + 1$ couples of variable valuations and locations, noted $\tau_{(P,v_0)} = \langle (v_0, c_0), \dots, (v_{n-1}, c_{n-1}), (v_{n=out}, c_{n=out}) \rangle$, such that $v_0 \models eval_0$ and for all i , $v_i \models g_i$ and $v_{i+1} = f_i(v_i)$. While v_{out} is the valuation of all variables when $\tau_{(P,v_0)}$ terminates, the observable result of $\tau_{(P,v_0)}$ (its *output*), noted $Out(\tau_{(P,v_0)})$, is the subset of v_{out} restricted only to all observable variables. Since a path π encompasses a set of executions, we can also represent the set of outputs of those executions into a symbolic formula $Out(\pi)$.

7.2.2 Symbolic Encoding of Mutants

A mutation alters or deletes a statement of the original program P . Thus, a mutant is defined as a change in the transitions of P that correspond to that statement (i.e. two transitions for branching statements; one for the others).

Definition 7.2.2. Let $\mathcal{P} = (C, c_0, V, eval_0, T)$ be an original program. A mutant of \mathcal{P} is a program $\mathcal{M} = (C, c_0, V, eval_0, T')$ with $T' = (T \setminus T_m) \cup T'_m$ such that:

$$\begin{cases} T_m \subseteq T \wedge |T_m| > 0 \\ \forall (c_1, [g']f', c'_2) \in T'_m, \exists (c_1, [g]f, c_2) \in T_m : ([g']f' \neq [g]f) \vee (c'_2 \neq c_2) \end{cases}$$

It may happen that a program mutation leads to an *equivalent* mutant (i.e. semantically equivalent to the original program), that is, for any test input t , $Out(\tau_{(P,v_0)}) \equiv Out(\tau_{(M,v_0)})$. All non-equivalent mutants, however, should be discriminated (i.e. *killed*) by at least one test input. Thus, there must exist a test input t that satisfies the following three conditions (referred to as RIP [AO08; DO91; Mor90]): the execution of t on M must (i) reach a mutated transition, (ii) infect (cause a difference in) the internal program state (i.e. change the variable valuations or the reached control locations), (iii) propagate this difference up to the program outputs. One can encode those conditions as the symbolic formula: $kill(P, M) \triangleq \exists \pi_P, \pi_M : \phi(\pi_P) \wedge \phi(\pi_M) \wedge (Out(\pi_P) \not\equiv Out(\pi_M))$. Any valuation satisfying this formula forms a test input killing M . For given π_P and π_M , $kill(\pi_P, \pi_M) \triangleq \phi(\pi_P) \wedge \phi(\pi_M) \wedge (Out(\pi_P) \not\equiv Out(\pi_M))$ denotes the formula encoding the test inputs that kills M and go through π_P and π_M in P and M , respectively.

Definition 7.2.3. Let P be an original program and M_1, \dots, M_n be a set of mutants of P . Then the *mutant killing problem* is the problem of finding, for each mutant M_i :

1. two paths π_P and π_{M_i} such that $kill(\pi_P, \pi_{M_i})$ is satisfiable;
2. a test input t satisfying $kill(\pi_P, \pi_{M_i})$.

7.2.3 Example

Figure 7.1 shows a simple C program. The corresponding C code and transition system are shown in the left and middle of Figure 7.1, respectively. The transition system does not show the guarded commands for readability. The right side of Figure 7.1 shows two test inputs and their corresponding traces (as sequences of control locations of the transition system). The transition system contains 12 control locations, corresponding to the 12 numbered lines in the program. The squared nodes of the transition system represent the non-branching control locations and the circular nodes represent the branching control location. For simplicity, we assume that each line is atomic. The initial condition $eval_0$ is $x \in Int$ where Int is the set of all integers. Two

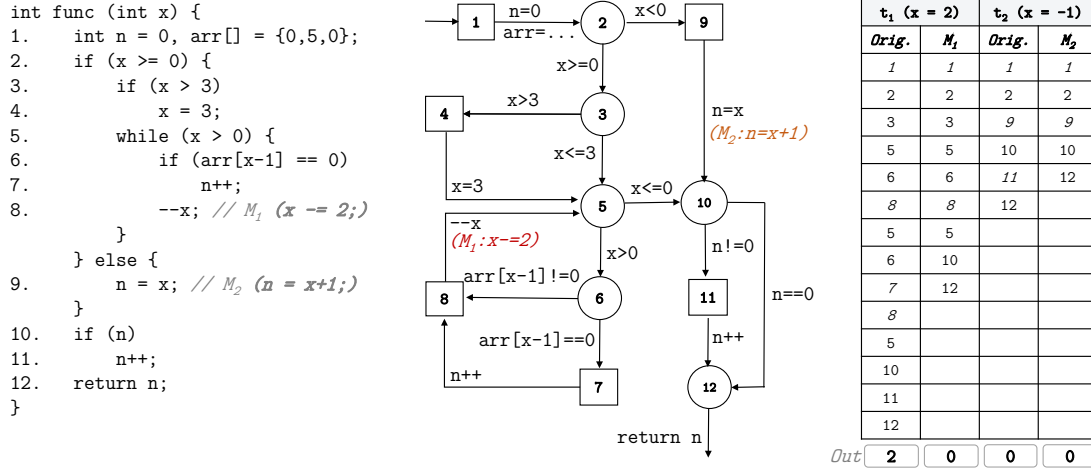


Figure 7.1: Example. The rounded control locations represent conditionals (at least 2 possible transition from them).

mutants M_1 and M_2 are generated by mutating statements 8 and 9, respectively. M_1 results from changing the statement “ $--x$ ” into “ $x = 2$ ” and M_2 results from changing the statement “ $n = x$ ” into “ $n = x + 1$ ”. The mutants M_1 and M_2 result from the mutation of the guarded command of the transitions $8 \rightarrow 5$ and $9 \rightarrow 10$, respectively.

The test execution of t_1 reaches M_1 but not M_2 , while t_2 reaches M_2 but not M_1 . Test t_1 infects M_1 and t_2 infects M_2 . The test execution of t_1 on the original program and mutant M_1 return 2 and 0, respectively. The mutant M_1 is killed by t_1 because $2 \neq 0$. Similarly, the test execution of t_2 on the original program and mutant M_2 return 0 and 0, respectively. Test t_2 does not kill mutant M_2 .

7.3 Symbolic Execution

One can apply symbolic execution to explore the different paths, using a symbolic representation of the input domain (as opposed to concrete values) and building progressively the path conditions of the explored paths. The symbolic execution starts by setting an initial path condition to $\phi = \text{True}$. At each location, it evaluates (by calling a dedicated solver) the guarded command of any outgoing transition. If the conjunction of the guard condition and ϕ is satisfiable then there exists at least one concrete execution that can go through the current path and the considered transition. In this case, the symbolic execution reaches the target location and ϕ is updated by injecting into it the guarded command of the transition. When multiple transitions are available, the symbolic execution successively chooses one and pursues the exploration, e.g. in a breadth-first manner.

As the symbolic execution progresses, it explores additional paths. The explored paths can together be concisely represented as a tree [Kin76] where each node is an execution state $\langle \phi, \sigma \rangle$ made of its path condition ϕ and symbolic program state σ (itself constituted by the current control location – program counter value – and the current symbolic valuation of variables).

Still, the tree remains too large to be explored exhaustively. Thus, one typically guides the symbolic execution to restrict the paths to explore, effectively cutting branches of the tree. Pre-conditioned symbolic execution attempts to reduce the path exploration space by setting the initial path condition (at the beginning of the symbolic execution) to a specific condition. This

precondition restricts the symbolic execution to the subset of paths that are feasible given the precondition. The idea is to derive the preconditions from pre-existing tests (aka *seeds* in the KLEE platform) that reach the particular points of interests. This allows us to provide vital guidance towards reaching the areas that should be explored symbolically, while drastically reducing the search space. In the rest of this chapter, we refer to a *preconditioned symbolic execution that explores the paths followed by some concrete executions as “seeded symbolic execution”*.

Overall, one can make the following steps to generate test inputs for a program P via symbolic execution:

1. **Precondition:** specify a logical formula over the program inputs (computed as the disjunction of the path conditions of the paths followed by the executions of the seeds) to prune out the paths that are irrelevant to the analysis.
2. **Path exploration:** explore a subset of the paths of P , effectively discarding infeasible paths.
3. **Test input generation:** for each feasible path π_P , solve $\phi(\pi_P)$ to generate a test input t whose execution $\tau_{(P,t)}$ follows π_P .

7.4 Killing Mutants

7.4.1 Exhaustive Exploration

A direct way to generate test inputs killing some given mutants (of program P) is to apply symbolic execution on both P and the mutants, thereby obtaining their respective set of (symbolic) paths. Then, we can solve $kill(\pi_P, \pi_{M_i})$ to generate a test input that kills mutant M_i and goes through π_P in P and through π_{M_i} in M_i .

Figure 7.2 illustrates the use of symbolic execution to kill mutant M_2 of Figure 7.1. We skip the symbolic execution subtree rooted at control location 3 since the corresponding paths do not reach mutant M_2 and can easily be pruned using static analysis. Also, we do not represent the symbolic variables arr and x , which are not updated in this example. The symbolic execution on the original program leads to the paths π_P^1 and π_P^2 such that $\phi(\pi_P^1) \equiv (x < 0)$, $\phi(\pi_P^2) \equiv False$, $Out(\pi_P^1) \equiv x + 1$ and $Out(\pi_P^2) \equiv x$. The symbolic execution on the mutant M_2 leads to the paths $\pi_{M_2}^1$ and $\pi_{M_2}^2$ such that $\phi(\pi_{M_2}^1) \equiv (x < -1)$ and $\phi(\pi_{M_2}^2) \equiv (x = -1)$, and $Out(\pi_{M_2}^1) \equiv (x + 2)$ and $Out(\pi_{M_2}^2) \equiv (x + 1)$.

The test generation that targets mutant M_2 solves the following formulae:

1. $kill(\pi_P^1, \pi_{M_2}^1)$. Satisfiable: example solution is $x = -2$.
2. $kill(\pi_P^1, \pi_{M_2}^2)$. Unsatisfiable: no possible output difference.
3. $kill(\pi_P^2, \pi_{M_2}^1)$. Unsatisfiable: infeasible path (π_P^2).
4. $kill(\pi_P^2, \pi_{M_2}^2)$. Unsatisfiable: infeasible path (π_P^2).

This method effectively generates tests to kill killable mutants. However, it requires a complete symbolic execution on P and on each mutant M_i . This implies that (i) all the path conditions and symbolic outputs have to be stored and analysed, and (ii) $kill(\pi_P, \pi_{M_i})$ has to be solved

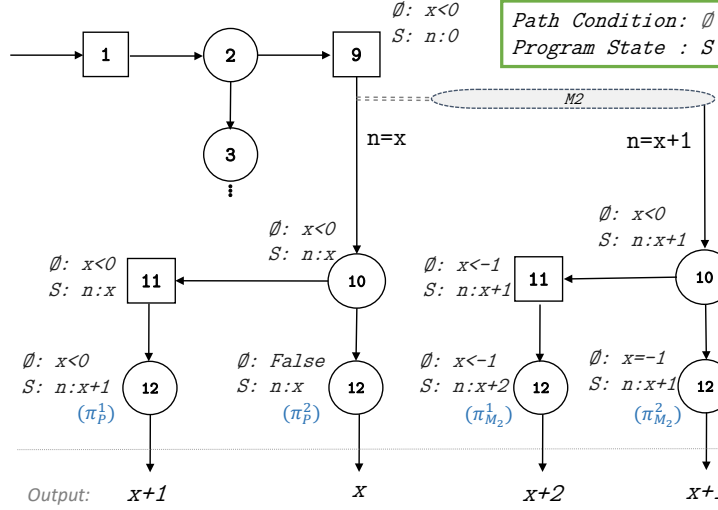


Figure 7.2: Example of Symbolic execution for mutant test generation. After control location 9, the symbolic execution on the original program contains transition $9 \rightarrow 10$ with $n = x$ while the symbolic execution of the mutant M_2 contains transition $9 \rightarrow 10$ with $n = x + 1$.

possibly for each pair of paths (π_P, π_{M_i}) . This leads to large computational cost that makes the approach impractical.

7.4.2 Conservative Pruning of the Search Space

To reduce the computational costs induced by the exhaustive exploration we apply two safe optimizations (preserve all opportunities to kill the mutants) that prune the space of program paths. We take advantage of the fact that mutants are simple syntactic alterations that share a large portion of their code with the original program.

7.4.2.1 Meta-mutation

Our first optimization stems from the observation that all paths and path prefixes of the original program P that do not include a mutated statement (i.e. location whose outgoing transitions have changed in the mutants) also belong to the mutants. Thus, the symbolic execution of P and that of the mutants may explore a significant number of identical path prefixes. As seen in Figure 7.2, the symbolic execution is identical for the original and mutant M_2 up to control location 9. Instead of making two separate symbolic executions, *SEMu* performs a shared symbolic execution based on a meta-mutant program. A meta-mutant [UOH93; PM10b; PM11] represents all mutants in a single code. A branching statement (named *mutant choice statement*) is inserted at each mutation point and controls, based on the value of a special global variable (the mutant ID), the execution of the original and mutant programs.

The symbolic execution on the meta-mutant program initialises the mutant ID to an unknown value and explores a path normally until it encounters a mutant choice statement. Then, the path is duplicated once for the original program and once for each mutant, with the mutant ID set to the corresponding value, and each duplicated path is further explored normally. While the effect of this optimization is limited to the prefixes common to the program and all mutants, it reduces

the overall cost of exploration at insignificant computation costs and without compromising the results.

7.4.2.2 Discarding non-infected mutant paths

In practice, many execution paths reach (cover) a mutant but fail to infect the program state (introducing an erroneous program state). Extending the execution along such paths is a waste of effort as the mutant will not be killed along those paths. Thus, *SEMu* terminates anticipatively the exploration of any path that reaches the mutant but fails to infect the program state.

7.4.3 Heuristic Search

Even with the aforementioned optimizations, the exhaustive exploration procedure remains too costly due to two factors: the size of the tree to explore and the number of couples of paths π_P and π_{M_i} to consider. To speed up the analysis, one can further prune the search space, at the risk of generating useless test inputs (that kill no mutant) or missing opportunities to kill mutants (by ignoring relevant paths).

A first family of heuristics reduce the number of paths to explore by selecting and prioritizing them, at the risk of discarding paths that would lead to killing mutants. A second family stop exploring a path after k transitions and solve, instead of $kill(\pi_P, \pi_{M_i})$, the formula

$$partialKill(\pi_P[..k], \pi_{M_i}[..k]) \triangleq \phi(\pi_P[..k]) \wedge \phi(\pi_{M_i}[..k]) \wedge (\sigma(\pi_P[..k]) \neq \sigma(\pi_{M_i}[..k]))$$

where, for any path π , $\pi[..k]$ denotes the prefix of π of length k and where $\sigma(\pi[..k])$ is the symbolic state reached after executing $\pi[..k]$. It holds that $kill(\pi_P, \pi_{M_i}) \Rightarrow \exists k : partialKill(\pi_P[..k], \pi_{M_i}[..k])$, since a mutation cannot propagate to the output of the program if it does not infect the program in the first place. The converse does not hold, though: statements after the mutation can cancel the effects of an infection, rendering the output unchanged at the end of the execution. The problem then boils down to selecting an appropriate length k where to stop the exploration, so as to maximize the chances of finding an infection that propagates up to the observable outputs.

As illustrated in Figure 7.2, generating a test at $k = 3$ (control location 10), requires to solve the constraint $partialKill(\pi_P[..3], \pi_M[..3]) \equiv (x < 0 \wedge x \neq x + 1)$. The constraint solver may return $x = -1$ which does not propagate the infection to the output. However, generating a test at $k = 4$ (control location 11), using the prefixes of the original path π_P^1 and mutant path $\pi_{M_2}^1$, requires to solve the constraint $x < 0 \wedge x < -1 \wedge (x \neq x + 1)$. Any value returned by the constraint solver kills the mutant.

An ideal method to kill a mutant M would explore only one path π_P and one path π_M , and up to the smallest prefix length k where the constraint solver can generate a test that kills M . However, identifying the right π_M and the optimal k is hard, as it requires precisely capturing the program semantics. To overcome this difficulty, *SEMu* defines heuristics to prune non-promising paths on the fly and to control at what point (what prefix length k) to call the constraint solver. Once candidate path prefixes are identified, *SEMu* invokes the solver to solve $partialKill(\pi_P[..k], \pi_M[..k])$.

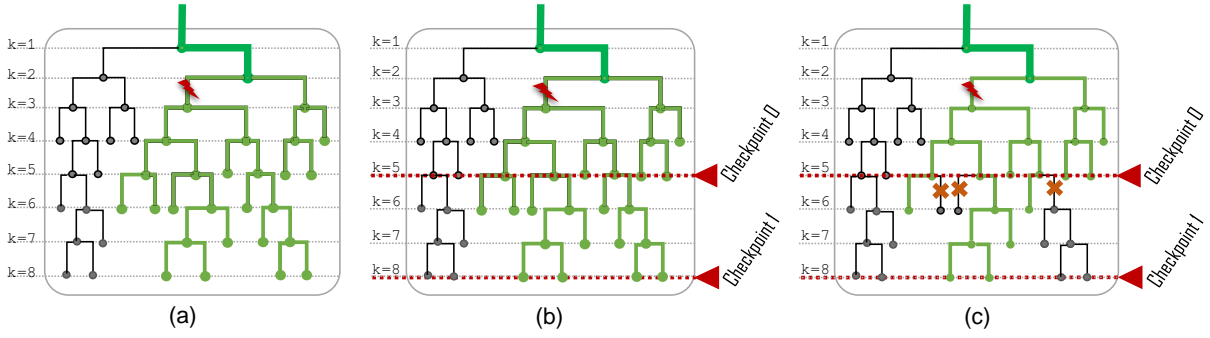


Figure 7.3: Illustration of SEMu cost-control parameters. Subfigure (a) illustrates the Precondition Length where the green subtree represents the candidate paths constrained by the precondition (the thick green path prefix is explored using seeded symbolic execution). Subfigure (b) illustrates the Checkpoint Window (here CW is 2). Subfigure (c) illustrates the Propagation Proportion (here PP is 0.5) and the Minimum Propagation Depth (here if MPD is 1 the first test is generated, for unterminated paths, from Checkpoint 1).

7.5 SEMu Cost-Control Heuristics

SEMu consists of parametric heuristics to control the symbolic exploration of promising code regions. Any configuration of *SEMu* sets the parameters of the heuristics, which together define which paths to explore and the test generation process. *SEMu* also takes as inputs the original program, the mutants to kill and a set of pre-existing test inputs to drive the seeded symbolic execution. During the symbolic exploration, *SEMu* selects which paths to explore and when to stop the exploration to generate test inputs based on the obtained path prefix.

7.5.1 Pre Mutation Point: Controlling for Reachability

To improve the efficiency of the path exploration, it is important to quickly prune paths that are infeasible (cannot be executed) or irrelevant (cannot reach the mutants). To achieve this, we leverage seeded symbolic execution (as implemented in KLEE) where the seeds are pre-existing tests. We proceed in two steps. First, we explore the paths in seeded mode up to a given path prefix length (as the number of transitions). Then, we stop following the seeds' executions and proceed with a non-seeded symbolic execution. The location of the switching point, on the explored paths, thus determines where the exploration stops using the precondition. In particular, if it is set to the entry point of the program then the execution is equivalent to a full non-seeded symbolic execution. If it is set beyond the output then it is equivalent to a fully seeded symbolic execution. Formally, let Π denote the complete set of paths of a program P , $\{t_1, \dots, t_n\}$ be the set of seeds, and l be the chosen seeded symbolic execution length. Then the set of explored paths resulting from the seeded symbolic execution of length l and with seeds $\{t_1, \dots, t_n\}$ is the largest set $\Pi' \subseteq \Pi$ satisfying $\pi \in \Pi' \Rightarrow \exists t_i : t_i \models \phi(\pi[..l])$. We define as *precondition length* the number of branching control locations on the path prefix $\pi[..b_{(\pi, PL)}]$, where $l = b_{(\pi, PL)}$, and $b_{(\pi, i)}$ is the number of transitions to be traversed, along π , in order to reach the i^{th} branching control location, for all i .

This heuristics is illustrated in Figure 7.3a where the thick (green) segments represent the portion of the tree explored by seeded symbolic execution and the subtree below (light green) represents

the portion explored by non-seeded symbolic execution. The precondition leads to pruning the leftmost subtree.

Accordingly, the first parameter of *SEMu* controls the *precondition length* (PL) at which to stop the seeded symbolic execution. Instead of demanding a specific length l , the parameter can take two values reflecting two strategies to define l dynamically: *GMD2MS* (Global Minimum Distance to Mutant Statement) and *SMD2MS* (Specific Minimum Distance to Mutant Statement). When set to *GMD2MS*, the precondition length is defined, *for all explored paths*, as the length of the smallest path prefix that reaches a mutated statement. When set to *SMD2MS*, the precondition length PL is defined, *individually for each path* π , such that $b_{(\pi, PL)}$ is the length of the smallest prefix $\pi[..b_{(\pi, PL)}]$ of this path that reaches a mutated statement.

7.5.2 Post Mutation Point: Controlling for Propagation

From the mutation point, all paths of the original program are explored. When it comes to a mutant, however, it happens that path prefixes that cover and infect the program state fail to propagate the infection to the outputs. These prefixes should be discarded to reduce the search space. Accordingly, our next set of parameters controls where to check that the propagation goes on, the number of paths to continue exploring from those checkpoints, and when to stop the exploration and generate test inputs. Overall, those parameters contribute to reducing the number of paths explored by the symbolic execution as well as the length k of the path prefixes from which tests are generated.

7.5.2.1 Checkpoint Location

The first parameter is an integer named the *Checkpoint Window* (CW) which determines the location of the checkpoints. Any checkpoint is a program location with branching statements (i.e. transitions with guarded command $[g]f$ such that $g \neq \text{True}$) that is found after the mutation point. Then, the checkpoint window defines the number of branching statements (that are not checkpoints) between the mutation point and the first checkpoint, and between any two consecutive checkpoints. The effect of this parameter is illustrated in Figure 7.3b. The marked horizontal lines represent the checkpoints. In this case, the checkpoint window is set to 2, meaning that there are two branching statements between two checkpoints. At each checkpoint, *SEMu* can perform two actions: (1) discard some branches (path suffixes) of the current path prefix (by ignoring some of the branches) and (2) generate tests based on the current prefix. Whether and how those two actions are performed is determined according to the following parameters.

7.5.2.2 Path Selection

The parameter *Propagating Proportion* (PP) specifies the percentage of the branches that are kept to pursue the exploration, whereas the parameter *Propagation Selection Strategy* (PSS) determines the strategy used to select these branches. We implemented two strategies: random (RND) and Minimum Distance to Output (MDO). The first one simply selects the branches randomly with a uniform probability. The second one assigns a higher priority to the branches that can lead to the program output more rapidly (i.e. by executing fewer statements). This distance is estimated statically based on the control flow and call graphs of the program. The two

parameters are illustrated in Figure 7.3c, where the crossed subtrees represent branches pruned at Checkpoint 0.

7.5.2.3 Early Test Generation

Generating test inputs before the end of the symbolic execution (on the path prefixes) allows us to reduce its computation cost. Being placed after the mutation point, all checkpoints are potential places where to trigger the test generation. However, generating sooner reduces the chances of seeing the infection propagate to the program output. To alleviate this risk, we introduce the parameter *Minimum Propagation Depth* (MDP), which specifies the number of checkpoints that the execution must pass through before starting to generate tests. In Figure 7.3c, if MDP is set to 1 then tests are generated from Checkpoint 1 (for the two remaining paths prefixes). Note that in case MDP is set to 0, tests are generated for the crossed (pruned) path prefixes at Checkpoint 0.

7.5.3 Controlling the Cost of Constraint Solving

Remember that *partialKill* requires the state of the original program and the mutant to be different. The subformulae representing the symbolic program states can be large and/or complex, which may hinder the performance of the invoked constraint solver. To reduce this cost, we devise a parameter *No State Difference* (NSD) that determines whether to consider the program state differences when generating tests. When set to *True*, $\text{partialKill}(\pi_P[..k], \pi_M[..k])$ is reduced to $\phi(\pi_P[..k]) \wedge \phi(\pi_M[..k])$; however, its solution has lower chances of killing mutant M .

7.5.4 Controlling the Number of Attempts

It is usually sufficient to generate a single test that covers the mutant to kill it. However, the stubborn mutants that we target may not be killed by the early attempts (applied closer to the mutation point) and require deeper analysis. Furthermore, a test generated to kill a mutant may collaterally kill another mutant. For those reasons, generating more than one test for a given mutant can be beneficial. Doing this, however, comes at higher test generation and test execution costs. To control this, we devise a parameter *Number of Tests Per Mutant* (NTPM) that specifies the number of tests generated for each mutant (i.e. the number of *partialKill* formulas solved for each mutant).

7.6 Empirical Evaluation

7.6.1 Research Questions

We first empirically evaluate the ability of *SEMu* to kill stubborn mutants. This is an essential question, since there is no point in evaluating *SEMu* if it cannot kill some of the targeted mutants.

RQ1 What is the ability of *SEMu* to kill stubborn mutants?

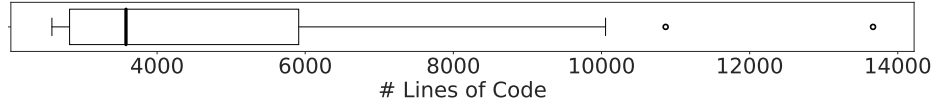


Figure 7.4: *Size of the test subjects.*

Since the results of RQ1 indicate a strong killing ability of *SEMu*, we turn our attention to the question of whether the killing ability is due to the extended symbolic exploration that is anyway performed by KLEE. We thus, compare *SEMu* with KLEE by running KLEE in the seed mode (using the initial test suite as a seed for KLEE test generation) to generate additional tests. Such a comparison is also a first natural baseline to compare with. These motivate RQ2:

RQ2 How does *SEMu* compare with KLEE in terms of killed stubborn mutants?

Perhaps not surprisingly, we found that *SEMu* outperforms KLEE. This provides evidence that our dedicated mutation-based approach is indeed suitable for mutation-based test generation. At the same time though, our results raises further questions on whether the superior killing ability of *SEMu* is due to mutant infection (suggested by previous research) or due to mutant propagation (specific target of *SEMu*). In case we find that mutant infection is sufficient for killing stubborn mutants then mutant propagation should be skipped in order to save effort and resources. To investigate this, we ask:

RQ3 How does *SEMu* compare with the infection-only strategy in terms of killed stubborn mutants?

7.6.2 Test Subjects

To answer our research questions, we experimented with the C programs of GNU Coreutils² (version 8.22). GNU Coreutils is a collection of text, file, and shell utility programs widely used in unix systems. The whole codebase of Coreutils is made of more than 60,000 lines of C code³.

The repository of Coreutils contains developer tests for the utilities programs which are system tests written in shell or perl scripts that involve more than 20,000 lines of code³.

Applying mutation analysis on all Coreutils programs requires excessive amount of effort. Therefore, we randomly sampled 60 programs, based on which we performed our analysis. Unfortunately, in 13 of them mutation analysis took excessive computational time (due to costly test execution), for which we terminated the analysis. Therefore, we analysed 47 programs. These are: `base64`, `basename`, `chcon`, `chgrp`, `chmod`, `chown`, `chroot`, `cksum`, `date`, `df`, `dirname`, `echo`, `expr`, `factor`, `false`, `groups`, `join`, `link`, `logname`, `ls`, `md5sum`, `mkdir`, `mkfifo`, `mknod`, `mktemp`, `nproc`, `numfmt`, `pathchk`, `printf`, `pwd`, `realpath`, `rmdir`, `sha256sum`, `sha512sum`, `sleep`, `stdbuf`, `sum`, `sync`, `tee`, `touch`, `truncate`, `tty`, `uname`, `uptime`, `users`, `wc`, `whoami`. Figure 7.4 presents the size of these subjects.

For each subject we selected the 3 functions that were covered by the largest number of developer tests (from the initial test suite).

²<https://www.gnu.org/software/coreutils/>

³Measured with `cloc` (<http://cloc.sourceforge.net/>)

7.6.3 Employed Tools

We implemented our approach on top of LLVM⁴ using the symbolic virtual machine KLEE [CDE08]. The version of our tool is based on the KLEE revision 74c6155, LLVM 3.4.2. Our implementation modified (or added) more than 8,000 lines of code on KLEE, which we will make publicly available. To convert system tests into the format of seeds required by KLEE for the seeded symbolic execution, we use SHADOW [PKC16].

Our tool requires the targeted mutants to be represented in a meta-mutant program (presented in Section 7.4.2.1), which were produced using the *Mart* mutant generation tool (Presented in Chapter 8). *Mart* mutates a program by applying a set of mutation operators (code transformations) to the original LLVM bytecode program.

7.6.4 Experimental Setup

7.6.4.1 Selected Mutants

To perform our experiment we need to form our target mutant set. To do so, we employed *Mart* by using its default configuration and generated 172,919 mutants. This configuration generates a comprehensive set of mutants based on a large set of mutation operators, consisting of 816 code transformations. It is noted that the operator set includes the classical 5 operators [Off+96a] that are used by most of the today's studies and mutation testing tools. Unfortunately, space constraints prohibit us from detailing the operator set. See Chapter 8, Section 8.1 for further details.

To identify the stubborn mutant set we started by eliminating trivial equivalent and duplicated mutants, and form our initial mutant set M_1 . To do so, we applied Trivial Compiler Equivalence (TCE) [Pap+15], a technique that statically removes a large number of mutant equivalences. In our experiment, TCE removed a total number of 102,612 mutants as being equivalent or duplicated. This gave us 70,307 mutants to be used for our initial mutant set, i.e., $M_1=70,307$.

Then, we constructed our initial test suites TS (composed of the developer test suite augmented with a simple test generation run of KLEE). To generate these tests with KLEE, we set a test generation timeout of 24 hours, while using the same configurations presented by the authors of KLEE [CDE08] (except for larger memory limit and `max-instruction-time`, set to 9GB and 30s respectively). This run resulted in 5,161 tests (2,693 developer tests and 2,468 tests generated by the initial run of KLEE).

We then executed the initial test suites (TS) with the initial mutant set (M_1) and identified the live and killed mutants. The killed mutants were discarded, while the live ones formed our target mutant set (denoted it as M_2), i.e., M_2 is the target of *SEMu*. In our experiment we found that M_2 included 26,278 mutants, which is approximately 37% of M_1 . It is noted that M_2 is a superset of the stubborn mutants as it includes both stubborn and equivalent mutants. Unfortunately, judging mutant equivalence is undecidable and thus, we cannot remove such mutants before test generation. Therefore, to preserve realistic settings we are forced to run *SEMu* on all M_2 mutants.

⁴<https://llvm.org/>

To evaluate *SEMu* effectiveness we need to measure the extent to which it can kill stubborn mutants. Unfortunately, M_2 contains a large proportion of equivalent mutants [SZ13], which may result in significant underestimations of test effectiveness [Kur+16b]. Additionally, M_2 may contain a large portion of subsumed mutants (mutants killed collaterally by tests designed to kill other mutants), which may inflate (overestimate) test effectiveness [Pap+16]. Although we discarded easy-to-kill mutants, it is still likely that a significant amount of ‘noise’ still remains.

To reduce such biases (both under and over estimations) [Kur+16b], there is a need to filter out the subsumed mutants by forming the subsuming mutant set [Pap+19]. The subsuming mutants are mainly distinct (in the sense that killing one of them does not alter, increase or decrease, the chances of killing the others) providing objective estimations of test effectiveness. Unfortunately, identifying subsuming mutants is undecidable and thus, several mutation testers, e.g., Ammann et al. [ADO14], Papadakis et al. [Pap+16], Kurtz et al. [Kur+16b] suggested approximating them through strong test suites. Therefore, to approximate them, we used the combined test suite that merges all tests generated by KLEE and *SEMu* across the execution of its 128 different configurations, $\bigcup_{\forall i} TS_{x_i}$, where x_0 is KLEE and x_i ($i > 0$) are the *SEMu* configurations (refer to Section 7.6.4.2 for details). This process was applied on M_2 and resulted in a set of 529 mutants. In the rest of this chapter we call the mutants belonging to M_3 as reference mutants. We use M_3 for our effectiveness evaluation.

Overall, through our experiments we used two distinct mutant sets, M_2 and M_3 . To preserve realistic settings, the former is used for test generation, while the later is used for test evaluation (to reduce bias).

7.6.4.2 *SEMu* Configuration

To specify relevant values for our modelling parameters we performed ad-hoc exploratory analysis on some small program functions. Based on this analysis we specify 2 relevant values for each of the 7 parameters (defined in Section 7.5). These values provided us the basis for constructing a set of configurations (parameter combinations) to experiment with. In particular the values we used are the following: Precondition Length: *GMD2MS* and *SMD2MS*, Checkpoint Window: *0* and *3*, Propagating Proportion: *0* and *0.25*, Propagating Selection Strategy: *RND* and *MDO*, Minimum Propagation Depth: *0* and *2*, No State Difference: *True* and *False*, Number of Tests Per Mutant: *1* and *5*.

We then experiment with them in order to select the *SEMu* configuration and form our approach. It is noted that different values and combinations form different strategies. Examining them is a non-trivial task since the number of configurations is exponentially increased, i.e., $2^7 = 128$ and mutant execution takes considerable amount of time. In our study, the total test generation of the various configurations and KLEE took roughly 276 CPU days, while the execution of the mutants took approximately 1,400 CPU days.

To identify and select the most prominent configuration, we executed our framework on all test subjects under all configurations x_i where $i \in [1, 128]$. We restrict the symbolic execution time to 2 hours. We then randomly split the set of test subjects into 5 buckets of equal size (each one containing 20% of the test subjects). Then, we pick 4 buckets (80% of the test subjects) and select the best configuration by computing the ratio of killed reference mutants. We assess the generalization of this configuration on the left out bucket (5th bucket that includes 20% of the test

subjects). To reduce the influence of random effects, we repeated this process 5 times by leaving every bucket out for evaluation. At the end we selected the median performing configuration (performance on the bucket that had been left out). It is noted that such a cross validation process is commonly used in order to select stable and potentially generalizable configurations.

Based on the above procedure we selected the *SEMu* configuration: $PL = \text{GMD2MS}$, $CW = 0$, $PP = 0.25$, $PSS = \text{RND}$, $MPD = 2$, $NSS = \text{False}$, $NTPM = 5$.

7.6.5 Experimental Settings and Procedure

To perform our experiment we set, on KLEE, the following (main) settings (which are similar to the default parameters of KLEE): a) we set a memory usage threshold of 8 GB, (a threshold never reached by any of the studied methods), b) we set the search strategy on Breadth-First Search (BFS), which is commonly used in patch testing studies [PKC16] and c) we set a 2 hours time limit for each subject.

It is noted that our current implementation supports only BFS. We believe that such a strategy fits well with our purpose as it is important that the mutants and original program paths are explored in a lock step in order to enable state comparison at the same depth. The time budget of 2 hours was adopted because it is frequently used in test generation studies, e.g., [PKC16], and forms a time budget that is neither too big nor too small. It is noted that since *SEMu* performs a deeper analysis than the other methods, adopting a higher time limit would probably lead to an improved performance, compared to the other methods. Of course reducing this limit could lead to reduced performance.

We then evaluated the generated test suites by computing the ratio of reference mutants that they kill. Unfortunately, in 11 among the 47 test subjects we considered, none of the evaluated techniques managed to kill any mutant. This means that for these 11 subjects we approximate having 0 stubborn mutants and thus, we discarded those programs. Therefore, the following results regard the 36 programs for which we could kill at least one stubborn mutant.

To answer RQ1 we compute and report the ratio of the reference mutants killed, i.e., M_3 set, by *SEMu* when it targets the 26,278 surviving mutants, i.e., M_2 set.

To answer RQs 2 and 3 we compute and contrast the ratio of the reference mutants killed by KLEE (executed in "seeding" mode), the infection-only strategy (a strategy suggested by previous research [Zha+10a; HJL11]) and *SEMu* (for fair comparison, we used the initial test suite as seeds for the three approaches). We also report and contrast the number of mutant-killing tests that were generated. Since the generated tests may include large numbers of redundant tests, i.e., a test is redundant with respect to a set of tests when it does not kill any unique mutant compared to the mutants killed by the other tests in the set [Pap+19], we compare the sizes of non-redundant test sets, which we call mutant-killing test sets. The size of these sets represents the raw number of end objectives that were successfully met by the techniques [AO08; Pap+19].

To compute the mutant-killing test sets we used a greedy heuristic. This heuristic incrementally selects the tests that kill the maximum number of mutants that were not killed by the previously selected tests.

7.6.6 Threats to Validity

All in all we targeted 133 functions from 47 programs from Coreutils. This level of evidence sufficiently demonstrates the potential of our approach, but should not be considered as a general assertion of its test effectiveness.

We generated tests at the system level, relying on the developers' tests suites. We believe that this is the major advantage of our approach because this way we focus on stubborn mutants that encode system level corner cases that are hard to reveal. Another benefit of doing so is that at this level we can reduce false alarms, experienced at unit level (feasible behaviors at unit but infeasible at system level), [GFZ12]. Unfortunately though, this could mean that our results do not necessarily extend to unit level.

Another issue may be due to the tools and frameworks we used. Potential defects and limitations of these tools could influence our observations. To reduce this threat we used established tools, i.e., KLEE and Mart, that have been used by many empirical studies. To reduce this threat further we also performed manual checks and intend to make our tool publicly available.

In our evaluation we used the subsuming stubborn mutants in order to cater for any bias caused by trivial mutants [Pap+16]. While this practice follows the recommendations made by the mutation testing literature [Pap+19], the subsuming set of mutants is a subject to the combined reference test suite, which might not be representative to the input domain. Nevertheless, any issue caused by the above approximations could only reduce the mutant killed ratios and not the superiority of our method. Additional (future) experimentations will increase the generalizability of our conclusions.

The comparison between the studied methods (infection-only) was based on a time limit that did not include any actual mutant test execution time. This means that when reaching the time limit, we cannot know how successful (at mutant killing) the generated tests were. Additionally, we cannot perform test selection (eliminate ineffective tests) as this would require expensive mutant executions. While, it is likely that a tester would like to execute the mutants in order to perform test selection, leaving mutant execution out allows a fair comparison basis between the studied methods since mutant execution varies between the methods and heavily depends on test execution optimizations used [Pap+19]. Nevertheless, it is unlikely that including the mutant execution would change our results since *SEMu* generates less tests than the baselines (because it makes a deeper analysis than the baselines).

7.7 Empirical Results

7.7.1 Killing ability of *SEMu*

To evaluate the effectiveness of *SEMu* we run it for 2 hours per subject program and collect the generated test inputs. We then execute these inputs with the reference mutants and determine the killed ones. Interestingly *SEMu* kills a large portion of the reference mutants. The median percentage of killed mutants is 37.3%, indicating a strong killing ability. To kill these mutants *SEMu* generated 153 mutant-killing test inputs (each test kills at least one mutant that is not killed by any other test).



Figure 7.5: Comparing the stubborn mutant killing ability of SEMu, KLEE and the infection-only.

7.7.2 Comparing SEMu with KLEE

Figure 7.5 records the proportion of the killed reference mutants by *SEM**u*, seeded mode of KLEE and infection-only (investigated in RQ3). It is noted that the boxes include the proportions of killed mutants among the different test subjects we use. From these results we can observe that *SEM**u* has a median value of 37.3% while KLEE has a median of 0.0%.

To further validate the difference we use the Wilcoxon statistical test (paired version) to check whether the differences are significant. The statistical test gives a p-value of 0.006 suggesting that the two samples' values are indeed significantly different. As statistical significance does not provide any information related to the volume of the difference, we also compute the Vargha Delaney effect size (\hat{A}_{12} value) that quantifies the frequency the observed difference. The results give a \hat{A}_{12} of 0.736, which indicates that *SEM**u* is superior to KLEE in 73.6% of the cases.

Figure 7.6 depicts the differences and overlap between the reference mutants killed by *SEM**u* and KLEE, per studied subject. From this figure, we can observe that the number of programs with overlapping killed mutant is very small indicating that the two methods differ significantly. We also observe that *SEM**u* performs best in the majority of the cases. Interestingly, a non negligible number of mutants are killed by KLEE only. These cases fall within a small number of test subjects. We investigated these cases and found that the differences were big either because there was only one reference mutant, which was killed by KLEE alone, or because of the large number of surviving mutants that force *SEM**u* perform a shallow search. Unfortunately, *SEM**u* spends much time trying to kill every targeted mutant and thus, when a large number of them is involved, the 2 hours time limit we set is not sufficient to effectively kill them.

To better demonstrate the effectiveness differences of the methods we also record the number of the mutant killing test inputs (each test kills at least one mutant that is not killed by any other test). We found that *SEM**u* generated 153 mutant-killing test inputs, while KLEE generated only 62.

7.7.3 Comparing SEMu with infection-only

A first comparison between *SEM**u* and *infection-only* can be made based on the data from Figure 7.5. According to these data *SEM**u* has a median value of 37.3% while *infection-only* has a median of 17.2%. Interestingly, this shows a big difference in favour of our approach. To

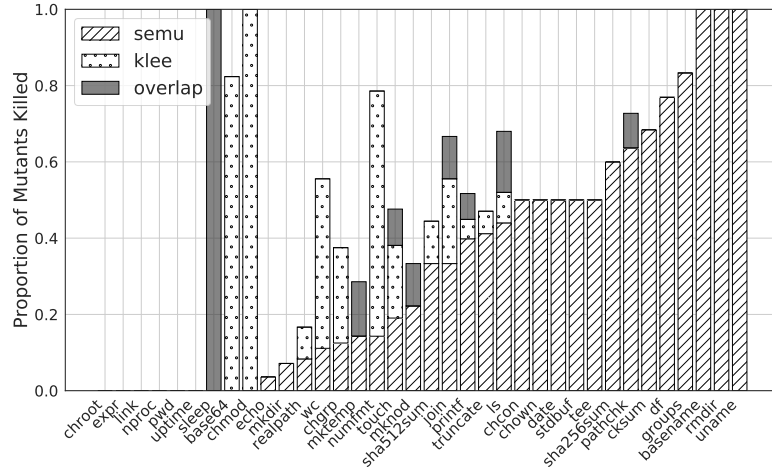


Figure 7.6: Comparing the mutant killing ability of SEMu and KLEE in per program basis.

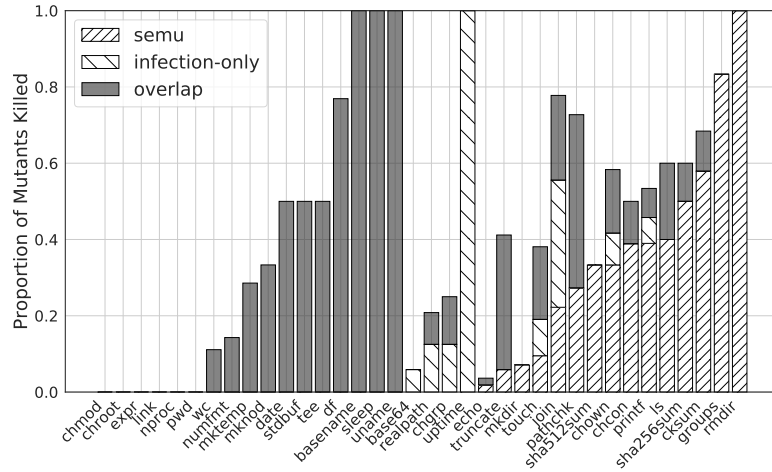


Figure 7.7: Comparing the mutant killing ability of SEMu and infection-only in per program basis.

further validate this finding, we performed a Wilcoxon statistical test and got a p-value of 0.04 suggesting that the two samples' values are statistically significant (at the commonly adopted 5% confidence level). Like in RQ2 we also computed the Vargha Delaney effect size \hat{A}_{12} and found that *SEM*_u yields higher killing rates than *infection-only* in 61% of the cases.

To demonstrate the differences we also present our results in a per test subject basis. Figure 7.7 shows the differences and overlap between the killed reference mutants. From these results we observe a large overlap between the mutants killed by both approaches, with *SEM*_u being able to kill more mutants for most of the cases. We also observe that in 5 of the cases *infection-only* performed better than *SEM*_u, while *SEM*_u performed better in 13.

Similarly, to the previous RQs we compare the strategies by counting the number of the mutant killing test inputs that were generated by the strategies. Interestingly, we found that *SEM*_u generated 87% more mutant killing test inputs than the "infection-only" one (153 vs. 82 inputs), indicating the usefulness of our framework.

7.8 Conclusion

This chapter introduced *SEMu*, a method that generates test inputs for killing stubborn mutants. *SEMu* relies on a form of shared differential symbolic execution that incrementally searches a small but ‘promising’ code region around the mutation point in order to reveal divergent behaviours. This allows the fast and effective generation of test inputs that thoroughly exercise the targeted program corner cases. We have empirically evaluated *SEMu* on Coreutils and demonstrated that it can kill approximately 37% of the involved stubborn mutants within a two hour time budget.

In the next chapter we present the mutant generation tool built to support the work presented in this dissertation. We also present an automated testing framework that integrates the mutation cost reduction techniques (*FaRM* and *SEMu*), presented in this dissertation, into the software testing process.

BUILT TOOLS AND FRAMEWORKS

This Chapter presents the tools and frameworks built as support to the techniques and studies presented in this dissertation. These tools provide a boost to mutation testing by providing researchers with material to extend the techniques presented in this dissertation and build new techniques, and to practitioners ready made tools that can be used.

This chapter is partly based on the work published in the following paper:

- Thierry Titchou Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.

Chapter content

8.1	Mart: A Mutant Generation tool for LLVM Bitcode	130
8.1.1	Overview	130
8.1.2	Mart Mutants Generation	130
8.1.3	Implementation and Usage	135
8.2	Mutaria: An Extensible and Flexible Multi-Criteria Software Analysis Framework	136
8.2.1	Overview	136
8.2.2	Background and Motivation	137
8.2.3	Mutaria Framework Overview	137
8.2.4	Case Study	141
8.2.5	Related Tools	142
8.3	Summary	142

8.1 *Mart*: A Mutant Generation tool for LLVM Bitcode

This section presents *Mart*, a mutant generation tool that implements many mutation operators and many features in order to support mutation testing techniques. *Mart* also implements the mutant reduction technique (*FaRM*) presented in Chapter 6.

8.1.1 Overview

Mart generates mutants for LLVM bitcode [LA04] (high-level languages such as C and C++ are compiled to LLVM intermediate representation for optimization and analysis). Generating mutants at the bitcode level may lead to inconsistency with source code mutation (due to loss of structural information during compilation). Nevertheless, the major advantage of generating mutants at the LLVM bitcode level is the ability to generate mutants for multiple high-level programming languages with the same tool. Two LLVM bitcode mutation tools, namely MuLL [DP18] and SRCIROR [HS18], have been developed recently but they currently support only few mutation operators and provides limited flexibility of mutation operators configuration during mutant generation. MuLL implements Arithmetic Operator Replacement, Condition Negation, Function Call Deletion and Replacement with Constant, Scalar Value Replacement. SRCIROR implements Arithmetic Operator Replacement, Logical Connector Replacement, Relational Operator Replacement and Integer Constant Replacement operators. None of those tools mutate pointers.

Mart mutant generation tool provides:

- A rich set of mutation operators (fine-grained operators [JKA17]), including operators that simulate high-level programming language’s complex expressions (such as left increment).
- An in-memory implementation of Trivial Compiler equivalence (TCE) [Pap+15] to eliminate equivalent and duplicate mutants.
- A simple description language for mutation operators configuration. The language enables users to apply a mutation operator based on the class of the operands of the mutated code’s operation.
- Generation of separated mutant bitcode files, meta-mutants bitcode file (useful for some mutant execution techniques [Wan+17]), weak mutation instrumented bitcode file and, mutant coverage instrumented bitcode file.

Mart has been used to generate mutants for studies in Chapters 5, 6 and 7. For the study in Chapter 6, *Mart* generated 4,778,157 mutant and detected 2,173,508 equivalent and duplicate mutants.

8.1.2 *Mart* Mutants Generation

Mart generates mutants for LLVM bitcode programs. *Mart* takes as input an LLVM bitcode file and optionally mutation configuration files to automatically generate mutated LLVM bitcode files.

An overview of the process implemented by *Mart*, to generate mutants, is represented in Figure 8.1. Initially, the input LLVM bitcode file is pre-processed (re-formatted to ease mutant instrumentation) and then instrumented by transforming the code using mutation operators

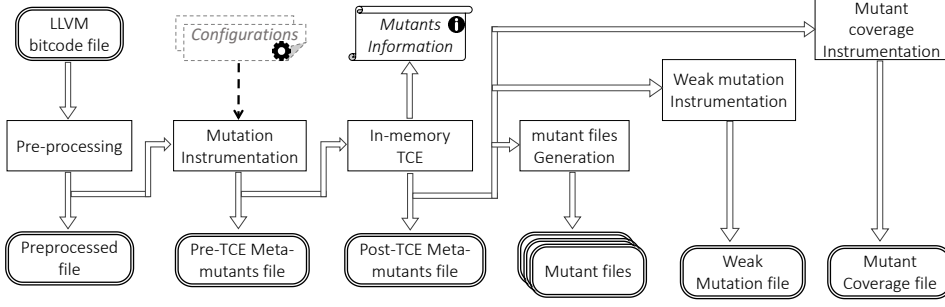


Figure 8.1: LLVM bitcode mutation process of Mart. The rounded edges rectangles with double border lines represent LLVM bitcode files. The square edge rectangles represent the steps of the mutation process. Each step is implemented by a component of Mart.

(the instrumentation can be constrained using mutation configurations). The instrumentation results in a meta-mutants program that encodes all mutants in a single module. The Meta-mutants module is then further processed by eliminating equivalent and duplicate mutants using an in-memory implementation of the Trivial Compiler Equivalence (TCE) [Pap+15]. Equivalent mutants are mutants that are semantically equivalent to the original program while duplicate mutants are mutants that are semantically equivalent to other mutants. The TCE elimination results in another meta-mutants module where equivalent and duplicate mutants, detectable by TCE, are removed. Information about mutants, such as mutant type, etc, are also exported. Finally, the post-TCE meta-mutants module is used to generate separated mutants files for different mutants, weak mutation instrumented module (to measure mutants infection) and mutant coverage module (to measure mutants reachability). The generated mutant files and instrumented files can be input to third-parties testing frameworks to be executed with test suites or improve the test suites. We recommend to use *Muteria*, presented in Section 8.2 testing framework for test execution.

In the following sub-sections, we present details about the implementation of components of *Mart*.

8.1.2.1 Preprocessing

The input LLVM bitcode file loaded as LLVM Module is transformed to enable the instrumentation with mutation operators. In this phase, the *phi nodes* of LLVM intermediate language are removed by applying a customized *reg2mem* function (which replaces registers by local variables). *Phi nodes* enable LLVM registers to be assigned and used in different basic blocks. This feature hinders *Mart* instrumentation as the instrumentation changes a single basic block at the time. The pre-processing step replaces registers with local variables for *phi nodes* by declaring, for each *phi node* register, a local variable which is assigned the register’s value at the register writing basic block and, the variable is loaded and the value used in the register reading basic block instead of the register.

8.1.2.2 Mutation Instrumentation

The mutation instrumentation of *Mart* consists of applying the defined mutation operators on compatible code locations. In this step, a configuration of the set of mutation operators to apply

as well as a configuration of the code locations to apply those mutation operators can be used to constrain the mutation.

A) Mutation Operators Representation

In order to support the mutation of complex operators of source code (e.g. recognizing C language arithmetic left increment ($++i$) or pointer de-reference followed by right decrement ($*p--$) on the LLVM bitcode level), we define abstractions of mutant operators.

Definition 8.1.1. We define as code fragment any piece of code that can be expressed as a function. Regarding a mutation, a fragment is the minimal piece of LLVM code that is syntactically changed by the mutation. This code may input LLVM registers or constant values and return some value into another register.

Definition 8.1.2. A fragments f' is compatible with another fragment f if and only if f can be replaced by f' without breaking the code's syntax.

Mart represents each mutation operator as a pair of fragments (f, f') , where f' is compatible with f . To apply the operator (f, f') at a location l of a program P , the fragment f is matched then, if found, it is replaced by the fragment f' . the resulting program M after replacing f by f' at l on P is a mutant of P . Figure 8.2-(b) illustrate an example of a mutation as executed by *Mart*.

Implementing New Mutation Operators. Given that fragments need to be matched and/or replaced, *Mart* provides an interface to implement fragments where matching and replacing functions need to be implemented. Implementing a new operator requires to implement the fragments' interfaces. The matching function inputs a list of LLVM bitcode instructions, checks whether the fragment is matched or not and, returns the fragment input addresses and output register address. The replacing function input the list of matched fragment's inputs and the list of bitcode instructions to mutate then, replace the code instructions to mutate.

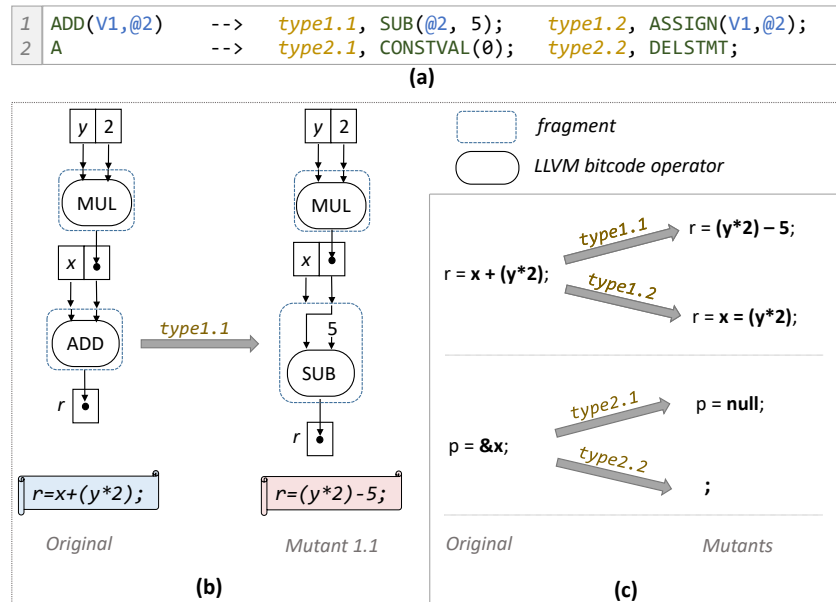


Figure 8.2: Example of bitcode mutation by *Mart*. Sub-figure (a) is an example of a mutation operators configuration description in a simple description language. Sub-figure (b) illustrates an example of code mutation; the second fragment in the original code is replaced by a mutant fragment. Sub-figure (c) presents an example of the mutation using the configuration of (a).

Table 8.1: *Mutant Types*

Mutated Code	Original <i>Fragment</i> Group	Mutant <i>Fragment</i> Group
STATEMENT	ANY STMT	TRAPSTMT
	ANY STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	SCALAR.UNARY
	SCALAR.ATOM	SCALAR.BINARY
	SCALAR.UNARY	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.BINARY
	SCALAR.BINARY	TRAPSTMT
	SCALAR.BINARY	DELSTMT
	POINTER.ATOM	POINTER.UNARY
	POINTER.UNARY	POINTER.UNARY
	POINTER.BINARY	POINTER.UNARY
	POINTER.BINARY	POINTER.BINARY
	DEREFERENCE.BINARY	DEREFERENCE.UNARY
	DEREFERENCE.BINARY	DEREFERENCE.BINARY

B) Currently Supported Mutation Operators

Currently, *Mart* implements 18 operator groups (pairs of "compatible" fragment groups). The 18 operator groups are designed to match a large number of elements of program syntax (additional operator groups can be implemented). There are 68 *fragments* implemented and the default mutation configuration is made of 816 operators (pair of *fragments*), including variations due to operand classes. These include all those that are supported by modern mutation testing tools [Off+96a]. The 18 operator groups are recorded in Table 8.1. "Original *fragment* group" refers to the matched *fragment* and "mutant *fragment* group" refers to the replacing *fragment*.

The *fragment* groups are defined as following (*p* refers to pointer values and *s* refers to scalar values):

- **ANY STMT** refers to matching any type of statement (only original *fragment*).
- **TRAPSTMT** refers to a *trap*, which cause the program to abort its execution (only mutant *fragment*).
- **DELSTMT** refers to the empty statement, thus, replacing by this is equivalent to deleting the original statement (only mutant *fragment*).
- **CALL STATEMENT** refers to a function call.
- **SWITCH STATEMENT** refers to a C language like *switch* statement.
- **SHUFFLEARGS** refers to the same function call as the original, with arguments of same type swapped (e.g. $g(a, b) \rightarrow g(b, a)$). This can only be a mutant *fragment* and, requires the original *fragment* to be a function call.
- **SHUFFLECASESDEST** refers to the same *switch* statement as the original, with the basic blocks of the *cases* swapped (e.g. $\{case\ a : B_1; case\ b : B_2; default : B_3;\} \rightarrow \{case\ a : B_2; case\ b : B_1; default : B_3;\}$). This can only be used as mutant *fragment* and,

requires the original *fragment* to be a *switch* statement.

- **REMOVECASES** refers to the same *switch* statement as the original, with some *cases* deleted (the corresponding values will lead to execute the *default* basic block) (e.g. $\{case\ a : B_1; case\ b : B_2; default : B_3;\} \rightarrow \{case\ a : B_2; default : B_3;\}$). This can only be used as mutant *fragment* and, requires the original *fragment* to be a *switch* statement.
- **SCALAR.ATOM** refers to any non pointer type variable or constant (only original *fragment*).
- **POINTER.ATOM** refers to any pointer type variable or constant (only original *fragment*).
- **SCALAR.UNARY** refers to any non pointer unary arithmetic or logical operation (e.g. $abs(s)$, $-s$, $!s$, $s++$...).
- **POINTER.UNARY** refers to any pointer unary arithmetic operation (e.g. $p++$, $--p$...).
- **SCALAR.BINARY** refers to any non pointer binary arithmetic, relational or logical operation (e.g. $s_1 + s_2$, $s_1 \& s_2$, $s_1 > s_2$, $s_1 <= s_2$...).
- **POINTER.BINARY** refers to any pointer binary arithmetic or relational operation (e.g. $p + s$, $p_1 > p_2$...).
- **DEREFERENCE.UNARY** refers to any combination of pointer dereference and scalar unary arithmetic operation, or combination of pointer unary operation and pointer dereference (e.g. $(*p) -$, $*(p -)$...).
- **DEREFERENCE.BINARY** refers to any combination of pointer dereference and scalar binary arithmetic operation, or combination of pointer binary operation and pointer dereference (e.g. $(*p) + s$, $*(p + s)$...).

C) Instrumentation process

Mart mutation instrumentation visits the Control Flow Graph (CFG) of the module under mutation and for each statement (represented by a group of instructions that are data-dependent w.r.t. registers) l , create mutated versions l'_1, \dots, l'_k . A branching instruction is then inserted, to select, based on the value of a special global variable called "Mutant ID selector", the statement to execute between l, l'_1, \dots, l'_k . The resulting module is a meta-mutants module where, the module can represent a specific mutant by just setting the value of the "Mutant ID selector" variable to its ID.

Constrained Mutation. The instrumentation process is subject to possible configuration. Users can restrict the corresponding source code's source files and functions to mutate by specifying the values in a JSON file that is used during mutation instrumentation. The operators to apply can also be specified in a file where each line is a key-value with the key the matching pattern and value the list or replacing patterns (This makes a simple mutation operator description language). Each pattern is made of the *fragment* name and the list of its indexed arguments' classes. The arguments classes are *constant* (C), *scalar variable* (V), *address* (A), *pointer variable* (P) and *any expression* ($@$). The set of argument classes in the replacing pattern is a subset of those from the matching pattern (except for constants). The mutation operator description language diagram is depicted in Figure 8.3. The mutation operation configuration file can easily be created automatically with a script available with the tool. Figure 8.2 shows an example of a mutant operator description configuration where 4 mutation operators are defined (2 operators for matching the sum of a variable and any expression and, 2 for matching an address).

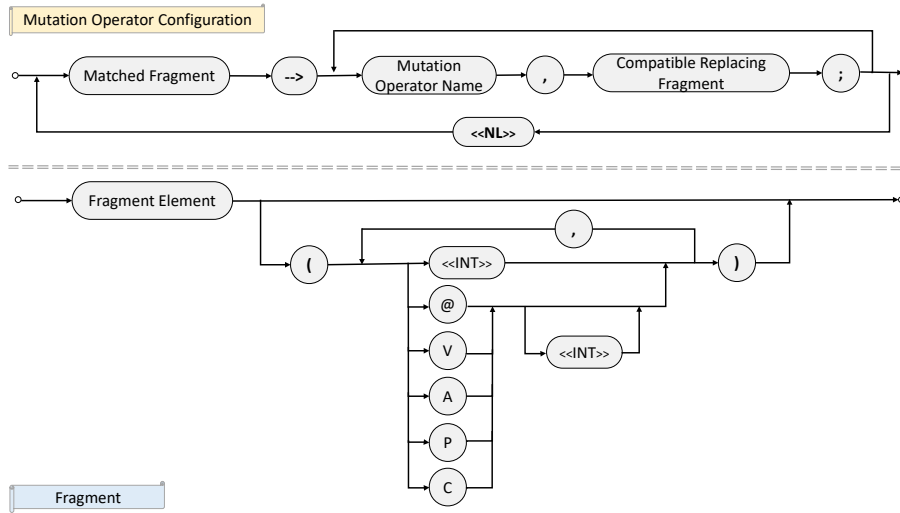


Figure 8.3: *Mutant operators description language syntax diagram*

8.1.2.3 In-Memory Equivalent and Duplicate Mutants Elimination

Mart eliminates equivalent and duplicate mutants by applying Trivial Compiler Equivalence (TCE) [Pap+15], on the mutated functions, in-memory. Our implementation of TCE applies LLVM optimizations, for each mutant, to the mutated function and uses a customized version of `llvm-diff`¹ tool to check for the difference between the optimized functions of the mutants and the original program's.

8.1.2.4 Final Mutated Files Generation

After TCE equivalent and duplicate mutants have been eliminated from the meta-mutants module, separated mutants files are generated by dumping the mutants functions used during TCE (which are also linked with the un-mutated function to make complete mutant bitcode). weak mutation and mutants coverage instrumented bitcode modules are generated by replacing, in the meta-mutants module, each mutant's code by label (function call that writes, into a file, the mutant ID of the mutants whose label is covered during test execution). A mutant coverage label is covered by any test that reaches it (the mutant location). A weak mutation label is covered by any tests that infect the mutant.

8.1.3 Implementation and Usage

Mart is implemented as a static analysis tool for LLVM bitcode (*Mart* loads the input LLVM bitcode file as an LLVM module and manipulates the module using the LLVM API). Currently, *Mart* has been tested for LLVM versions 3.4, 3.7, 3.8 and 3.9, and on Ubuntu (Linux) operating system.

¹<https://llvm.org/docs/CommandGuide/llvm-diff.html>

Mart can be used to mutate programs written in any language compilable into LLVM bitcode (compile complex C/C++ projects with `wllvm`²). The users are required to compile the code with debug information enabled in order to keep the information about source code location for mutants information. If no debug information is found in the program to mutate, the mutants information will not contain the source code locations information of the mutants.

Mart can be used through the command lines interface (CLI) or through its application programming interface API. Users of *Mart* can provide mutation configuration files (mutants operators and mutation scope), decide whether to apply in-memory TCE, decide whether to output weak mutation bitcode file, mutant coverage bitcode file and separated mutant files. See the tool weblink to get started.

8.2 *Muteria*: An Extensible and Flexible Multi-Criteria Software Analysis Framework

This section presents *Muteria*, a framework that aims to support experimentation in research to improve the practicality of mutation testing.

8.2.1 Overview

A Test Adequacy Criteria (TAC) based software analysis process, as depicted in figure 8.4 (more details in section 8.2.2) involves using TACs to evaluate and improve the test suites. Several phases (steps) of the software analysis process optimize the execution (represented with dashed lines in the figure 8.4). Many tools and techniques have been developed to help software developers analyse and test their software [Off11; SWF10; CDE08]. These tools and techniques are used to increase the fault detection, give guarantee of the software correctness and reduce the cost of software analysis through automation of the process [Off11]. Nevertheless, with the proliferation of programming languages, TACs and software analysis tools, developers need to exert supplementary effort to learn to use newly-developed tools, and integrate them into their test environment. Furthermore, researchers exert much effort to implement and evaluate their developed techniques and often, a great deal of engineering effort is required in order to integrate their implementation with other tools.

Muteria framework is built in response to those challenges. *Muteria* provides a collection of simplified drivers interfaces for integration of software analysis tools (implementing different aspects of the TAC-based software analysis process). Tools are integrated into *Muteria* through drivers that implement interface functions to enable *Muteria* to call the tools (these drivers can be made publicly available with the corresponding tools).

The *Muteria* framework provides:

- The Flexibility to add support for new TACs and programming languages.
- An Interface to implement drivers to integrate new tools.
- A controller that handles the integration of the tools.
- A Reporter that computes metrics and display results.

²<https://github.com/travitch/whole-program-llvm>

8.2.2 Background and Motivation

8.2.2.1 Software Analysis Process

Figure 8.4 presents an overview of a test adequacy criteria (TAC) based software analysis process, adapted from the “Two mutation processes” presented by Offut [Off11] (This is a detailed version of the Figure 1.1 presented in Chapter 1). During the process, tests that are either manually or automatically generated, are executed (after possible selection/prioritization) on the program under Analysis (PUT) to check for failures due to potential faults (in presence of faults, the process is interrupted, the user repairs the program and restarts the process). The test suites are evaluated using TACs’ coverage and improved to maximize TACs’ coverage. The TACs’ test objectives are generated by instrumenting the PUT. For faster execution, the TACs’ test objectives of interest may be selected (for instance, a random number of mutants are selected based on mutation operators in the case of mutation testing [PI18] or, most likely to be faulty statements are selected in case of statement coverage). The tests are executed (with possible optimization such as optimizing strong mutation using weak mutation [KMK13]) on the TACs’ instrumented programs and the coverage values are computed. The computed TACs’ coverage values are reported to the user who, based on the values, may generate more tests to increase the coverages.

There are two variants of the process: in *process 1*, the targeted TACs coverage is reached before the PUT is checked for correctness while in *process 2*, the PUT is checked for correctness before the TACs’ coverage is measured.

Each phase of the preceding process has been subject to research leading to development of new techniques and tools. Nonetheless, researchers exert a great deal of engineering effort to build prototypes of their techniques which, often, are not easy to use due to the engineering effort needed in order to integrate them into the software analysis process. Moreover, the experimental evaluations of the developed techniques require that scripts are implemented to integrate the prototypes with other existing tools.

8.2.2.2 Why A New Framework?

The reasons behind *Muteria* are to provide the following.

A laboratory framework for TAC-based software analysis research that allows researchers to implement and evaluate their techniques with little effort.

Simplify the development of TAC-based software analysis tools by providing out-of-the-box integration with other existing tools.

Ease the use of TAC-based software analysis techniques through rich user interfaces and configuration.

8.2.3 *Muteria* Framework Overview

We believe that a well designed software analysis framework should be easy to use, provide good user interfaces and be easy to modify for different uses. *Muteria* framework implements

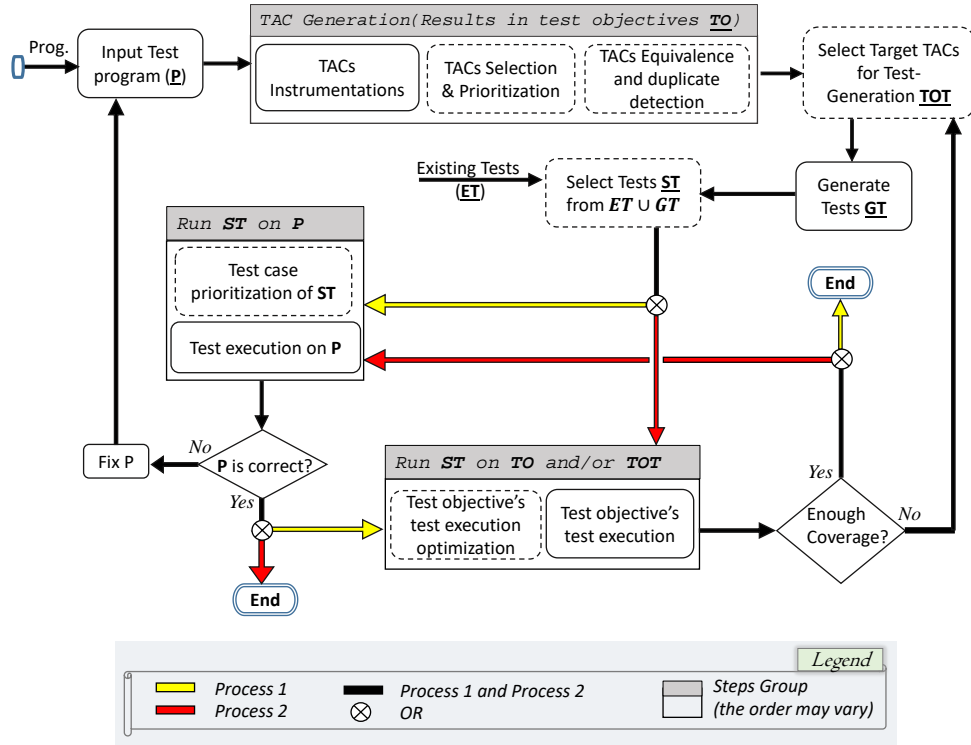


Figure 8.4: Test Adequacy Criteria (TAC) based software analysis process (adapted from Offut’s “Two mutation processes” [Off11]). The Process 1 is adapted from the “Traditional process” and Process 2 from the “Post-Mothra Process”

the different phases of the TAC-based software analysis process, depicted in figure 8.4, with extensible interfaces. *Muteria* uses a modular approach [Par72] for the implementation of its functionalities.

8.2.3.1 Design Goals

In this section, we present the main features of *Muteria* that support its design.

1. *Extensible.* The modular design of *Muteria* framework separates the phases of the TAC-based software analysis process into different components. Within each component, multiple tools that implement the corresponding phases can be integrated into the framework. Integrating a new tool into *Muteria* simply requires to extend the corresponding component’s *tool driver* interface. Such a design enables the development of drivers for new tools on a specific component, independently to the tools used in other components.
2. *Configurable.* *Muteria* provides a wide space of configurations that allow the users to have deep control over the execution of the framework. The framework allows the users to configure the execution of the software analysis process by specifying the test adequacy criteria to use during the analysis, whether to reuse preceding execution data or not (useful for example for regression testing), the level of concurrency and, which metrics to report and how to report them. The underlying test generation tools, test adequacy criteria tools and test execution optimization techniques can also be configured collectively or individually (tool specific configuration).

3. *Multi Programming Language Support.* *Muteria* framework separately support multiple programming language by integrating the testing tools of the same programming language. For instance, using the framework on a C language programs allows only the use of tools supporting C programs. Therefore, the framework's extension tools are grouped by programming languages.

8.2.3.2 User Interaction

Muteria framework provide 2 main forms of user interaction.

4. *Application Programming Interface.* Users can integrate *Muteria* into other frameworks through its application programming interface (API). Moreover, *Muteria*'s components can be used as libraries to build different frameworks.
5. *Command Lines.* As most frameworks, *Muteria* provide a rich command lines interface (CLI), allowing users to execute the framework from terminals.

8.2.3.3 Architecture

Figure 8.5 presents an overview of the architecture of *Muteria* framework. The core of the framework is made of the following components:

- *Controller.* This component organize the tasks to be executed, based on the configuration, and calls the relevant components for each of the executions. It implements the integration between the tools implementing different phases of the software analysis process.
- *Code Manager.* This component manages the code repository of the PUT. It also provides functions to build code (convert from one code representation to another).
- *Test Cases Manager.* This component provides an abstraction of test generation and test execution to the framework. Multiple test generation and test execution tools can be integrated through drivers on this component. A test execution tool handles specific test formats (e.g. JUnit tests) whose test can be generated by multiple test generation tools. Manually written tests are also managed by this component and are executed by the tool whose test format they follow. This component provides a high level functions to generate and to execute tests. These functions are mapped to the underlying tools through the tools drivers.
- *TAC Manager.* Similar to the Test Cases Manager, this component provides an abstraction of each implemented TAC's instrumentation tool. Multiple TACs tools can be integrated through drivers on this component. Each tool may implement support for multiple TACs. This component provides functions to instrument the PUT for the given TACs and to execute a test set against the instrumented programs (by calling the Test Case Manager).
- *Test Execution Optimizer.* This component provides functions to select and prioritize tests cases (e.g. for regression testing). Test execution optimizing techniques implementations can be integrated into this component. This component's object is passed to the Test Case Manager during test execution to enable implementing dynamic optimizations based on previous test executions.

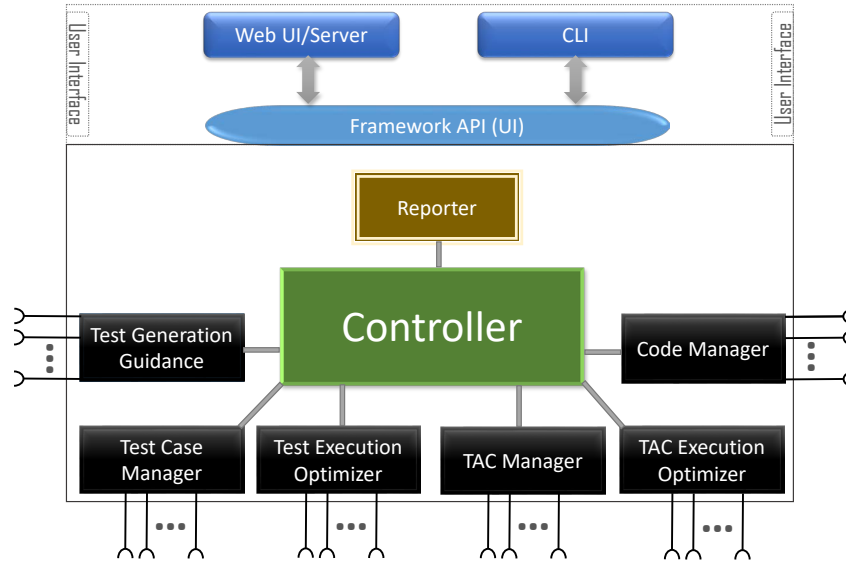


Figure 8.5: Architecture of Muteria framework. The components with black rectangle provide interfaces for corresponding tools to connect to the framework. The controller enable the integration. All components are accessible by the users through the framework API.

- *Test Generation Guidance.* This component implements functions to select, during the test generation process, "important" TACs' test objectives to focus on (e.g. select likely fault revealing statements or mutants (Chapter 6 and use them to guide automated test case generation to reveal potential faults). Test generation guidance techniques implementations can be integrated into this component.
- *TAC Execution Optimizer.* This component provides functions to select and/or prioritize TACs' test objectives for execution (e.g. using weak mutation to improve execution time of strong mutation [KMK13]). This is useful, for instance, for strong mutation in regression testing, where the optimizer could statically select the mutants likely to be relevant to the area of interest in the program under test. TACs execution optimizer implementations can be integrated into this component.
- *Reporters.* This component provides functions to compute useful metrics (such as code coverage, mutants subsumption and execution time) and present to the user. It computes the metrics and store into data files and/or render into HTML files or terminal. When using the API, the reports data are returned through the API.

8.2.3.4 Implementation

The *Muteria* framework is implemented in Python programming language. The extension tools' drivers are also implemented in Python programming language. The integrity of the code repository of the PUT is ensured using git³ (some TACs, e.g. mutation, may modify source files). There have been many challenges in the development of the framework, and the greatest were the modularization of the framework and design of tool driver interfaces. The *Muteria* framework is publicly available⁴ open source. Installation is done by running *pip install muteria*.

³<https://gitpython.readthedocs.io/en/stable/>

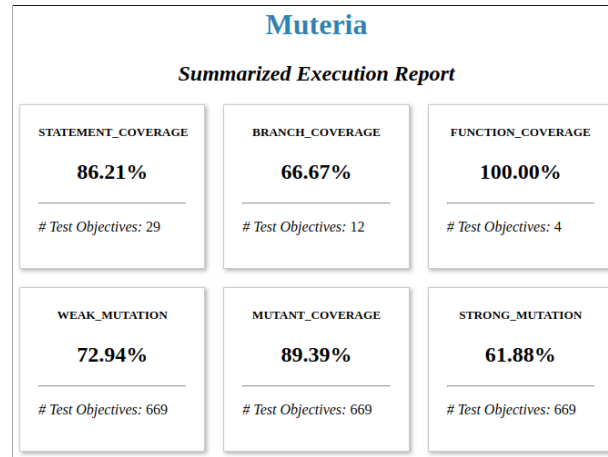
⁴<https://github.com/muteria/muteria>

Table 8.2: *Muteria in Practice: Some drivers sizes (Python LOC)*

Language	Tool Type	Tool Name	Driver Size (LOC)
Python	Statement/branch Coverage	Coverage.py	333
C/C++	Statement, Branch, Function Coverage	GNU Gcov	327
C/C++	Mutant Coverage and Strong/Weak Mutation	<i>Mart</i>	330
C/C++	Test generation	KLEE	198

Table 8.3: *Muteria in Practice: Some User Configurations*

Language	Analysis	# Config. Vars.
Python	Measure unit tests Statement/branch Coverage	8
C/C++	Measure system tests and generated tests Statement, Branch, Function Coverage and weak, strong mutation scores	16

**Figure 8.6:** *report of software analysis with Muteria.*

8.2.4 Case Study

We implemented a set of drivers for several C programming language software analysis tools, namely: *Mart* mutant generation tool (Section 8.1), KLEE [CDE08] test generation tool, GNU Gcov code coverage measurement tool. We also implemented drivers for Python code coverage measurement tool *Coverage.py*⁵. Table 8.2 summarizes the implementation sizes of the drivers.

We also present in Table 8.3 the number of configuration that the user needs to provide to make an analysis on a software using the selected tools.

The sample reported coverage information for the execution of *Muteria* on a sample C program is shown in Figure 8.6.

⁵<https://github.com/nedbat/coveragepy>

8.2.5 Related Tools

Many frameworks have been designed and developed to support TAC-based software analysis. Most of those frameworks either focus on specific programming languages, specific TAC or support specific test runners. Moreover, very often, there is no straightforward approach to integrate those with other tools. Stryker⁶ is an open-source mutation analysis framework that supports several programming languages (currently three) and enable integration with multiple test runners. Nevertheless, currently, Stryker neither provide support for adding test adequacy criteria nor support integration with various mutation tools or test generation tools. Similarly, LittleDarwin [PMD17] provides extensibility and many feature related to mutation analysis, however, it is restricted to Java programs. Open Code Coverage Framework (OCCF) [SWF10] is a framework that aim to simplify the development of code coverage measurement in multiple programming languages. OCCF does not provide mechanisms to integrate such coverage measurement tools with other types of tools such as test generation tools. OCCF is orthogonal with *Muteria* and can be used alongside *Muteria* by developing *Muteria* drivers for the tools developed with OCCF. The Mothra mutation framework [Off11] was built with the goal to be expandable and adaptable. In fact, Mothra tool-set was designed to be like a *laboratory* for future research [Off11], which is also an important philosophy for *Muteria*. Nevertheless, Mothra was designed for Fortran programs and for mutation TAC. *Muteria* learned from Mothra and generalized to support different TACs and programming languages.

8.3 Summary

This chapter presented the tools and frameworks built in the work presented in this dissertation. the details about the motivation and the implementation of the tools is presented. These tool are publicly available open source.

⁶<https://stryker-mutator.io/>

CONCLUSION

This Chapter concludes the dissertation and presents future research directions.

Chapter content

9.1	Summary	144
9.2	Future Research Directions	145

9.1 Summary

In this dissertation, we presented studies, techniques, and tools that contribute to enabling the practical use of mutation testing and support mutation testing research. We focus on mutation testing because mutation has been shown in previous research to be a good test adequacy criterion for software testing. The contributions of this dissertation can be grouped into 3 parts: 1) An empirical study that evaluates and compares mutation test adequacy criterion with other widely used test adequacy criteria on real-world software and real-world setting. 2) A mutant reduction technique that improves the scalability of mutation testing. 3) An automated test generation technique for mutation testing. The tools and datasets resulting from the work presented in this dissertation are made publicly available.

Regarding the first part, our objectives were to provide empirical evidence about the superiority of mutation over other widely used code-based test adequacy criteria, provide guidance on how to use coverage metrics to maximize fault revelation in practice, and evaluate a critical assumption made by researchers when conducting experiments in software testing. We thus conducted an empirical study, based on real-world software and real faults, where we studied the validity of the "Clean Program Assumption", which is an assumption that the coverage of a software testing criterion on a "clean" program (not having a certain fault) can generalize to its faulty (having the fault) counterpart and vice-versa. We found that the assumption does not always hold. We also studied the relationship between four popular software testing criteria (statement coverage, branch coverage, weak mutation, and strong mutation) and their fault revelation. We found that fault revelation is observed only beyond a certain threshold of coverage for the four studied criteria, and there is a relation between fault revelation and coverage value only for strong mutation. Therefore, a software tester needs to target very high coverage, beyond a certain threshold, before having the guarantee of finding potential faults.

Regarding the second part, our objectives were to reduce the number of mutants and focus on valuable mutants. We thus analysed different types of mutants, i.e., hard to kill, subsuming, hard to propagate and fault revealing, and demonstrated that the class of fault revealing mutants is unique and differs from the other mutant sets. Motivated by such findings (that it is possible to target a specific (small) set of mutants that maximize testing effectiveness), we designed *FaRM*, a supervised machine learning approach that select and prioritize mutants in order to reduce the number of analyzed mutants and therefore, the cost of mutation testing. *FaRM* significantly improves the efficiency and accuracy of mutation testing and relies on machine learning in order to identify and select fault revealing mutants, i.e., mutants that can only be killed by test cases that also reveal faults. The idea is that testers can focus only on the good mutants (fault revealing ones), which are few and reveal most of the faults in the systems under analysis. *FaRM* has been evaluated with empirical data and has been found to be significantly more effective than the random mutant selection.

As a final part, we presented *SEMu*, a scalable technique that employs symbolic execution to generate test cases for mutation testing in an efficient and automated manner. The technique targets mutants that escape traditional testing (killable mutants that are not killed by existing test suites). Killing these mutants is important for thorough testing of software components. We conducted an evaluation of our technique on real-world software and the results show that *SEMu* can successfully generate test to kill mutants and significantly improve the subsuming mutation score over KLEE and the state of the art techniques.


9.2 Future Research Directions

Following are potential future research directions that are in line with this dissertation.

- **Test adequacy criteria evaluation.** The empirical study that is presented in Chapter 4 evaluates the fault revelation of mutation and the most widely-used code-based test adequacy criteria. Nevertheless, data flow-based test adequacy criteria such as *All-Uses coverage* (see Table 3.1) were found to be useful in testing, thus, future work will consider comparing mutation with data-flow based test adequacy criteria on real-faults. Regarding the clean program assumption, this dissertation pointed out several studies that make the assumption, and it is worthwhile to evaluate the effect of the clean program assumption on those studies. Furthermore, the study in Chapter 4 shows that this assumption does not hold in the case of real faults and thus, leaves the case of artificial faults open for future research.
- **Mutation testing mutants reduction.** The mutant reduction approach (*FaRM*) proposed in this dissertation is an initial step toward using machine learning to tackle the mutant reduction problem. However, *FaRM* does not involve features engineering thus, future work will apply feature selection and design additional mutant features that could improve the effectiveness of *FaRM*. Additionally, based on the nature of the mutant reduction problem, our approach can benefit from the use of active (machine) learning where, the trained binary classifier model used to predict useful mutants can be improved based on users feedback on previous prediction. Another future work consists in specializing *FaRM* to specific fault classes (e.g. incorrect condition or null pointer dereference). This would reduce the noise caused by the proliferation of faults classes, thus, obtain a higher accuracy of prediction (as presented in Chapter 5).

Given the evolutionary nature of software, it is intuitive to learn to rank fault revealing mutants on previous version of a software to predict the fault revealing mutants on new versions. The study presented in Chapter 6 considers a scenario where the binary classifier is trained on software that are different than the test software. future work will evaluate *FaRM* in a regression testing scenario (where the model is trained only on previous versions of a system to predict fault revealing mutants on the new version). Another similar future work in regression testing setting is to build, similarly to *FaRM*, a machine learning based mutant reduction to predict the minimal set of mutants that is relevant to the version commits' change (such mutants are useful to generate regression test).

- **Automated test generation for mutation testing.** The test generation approach (*SEMu*) presented in Chapter 7 defines a set of parameters whose values are not tailored to the program under analysis. Therefore, the user needs to manually decide about the right values for each specific program or use the statistical best configuration presented in Chapter 7 (however, there are corner cases where the statistical best configuration would not perform well). Future work will extend our technique to leverage static analysis and adapt the cost control heuristic parameters to the program under analysis. Another future work will improve the strategies that are used to approximate the required propagation depth for test generation for each path, as well as the strategies used to quickly prune out the paths that are irrelevant to killing a mutant.



LIST OF PAPERS, TOOLS & SERVICES

Papers included in the dissertation:

- 2019
 - Thierry Titchou Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, Koushik Sen: Selecting Fault Revealing Mutants. Empirical Software Engineering (2019). <https://doi.org/10.1007/s10664-019-09778-7>.
 - Thierry Titchou Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.
- 2018
 - Mike Papadakis, Thierry Titchou Chekam, and Yves Le Traon. “Mutant Quality Indicators”. In: the 13th International Workshop on Mutation Analysis. Mutation 2018. 2018
 - Thierry Titchou Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon: Poster: Predicting the fault revelation utility of mutants. ICSE-Companion 2018: 408-409
- 2017
 - Thierry Titchou Chekam, Mike Papadakis, Yves Le Traon, Mark Harman: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. ICSE 2017: 597-608

Papers not included in the dissertation:

- Mingkang Ruan, Thierry Titchou, Ennan Zhai, Zhenhua Li, Yao Liu, Jinlong E, Yong Cui, Hong Xu: On the Synchronization Bottleneck of OpenStack Swift-Like Cloud Storage Systems. IEEE Trans. Parallel Distrib. Syst. 29(9): 2059-2074 (2018)
- Matthieu Jimenez, Thierry Titchou Checkam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. 2018. Are mutants really natural?: a study on how "naturalness" helps mutant selection. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18). ACM, New York, NY, USA, Article 3, 10 pages. DOI: <https://doi.org/10.1145/3239235.3240500>.

- Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon. Assessing and Comparing Mutation-based Fault Localization Techniques. CoRR abs/1607.05512 (2016)

Papers published prior PhD:

- Thierry Titcheu Chekam, Ennan Zhai, Zhenhua Li, Yong Cui, Kui Ren: On the synchronization bottleneck of OpenStack Swift-like cloud storage systems. INFOCOM 2016: 1-9

Software developed during PhD:

- *Mart*: A mutant generation tool for LLVM bitcode (<https://github.com/thierry-tct/mart>).
- *Muteria*: An Extensible and Flexible Multi-Criteria Software Analysis Framework (<https://github.com/muteria/muteria>).
- *SEMu*: A test input generation tool to kill mutants (will be made publicly available).

Services:

- Journal reviewer: JSS (in 2017), Transaction on Computers (in 2019), STVR (in 2019); STTT (in 2019).
- Conference co-reviewer: QRS 2019, ICST 2019, ICPC 2019.
- Workshop co-reviewer: Mutation 2018.



BIBLIOGRAPHY

- [AB11] Andrea Arcuri and Lionel Briand. “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering”. In: *ICSE*. 2011, pp. 1–10. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985795 *Cited on pages 21, 41.*
- [Abo+19] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. “Coincidental correctness in the Defects4J benchmark”. In: *Software Testing, Verification and Reliability* 29.3 (2019). e1696 STVR-18-0045.R2, e1696. DOI: 10.1002/stvr.1696 *Cited on page 109.*
- [Acr80] Allen Troy Acree. “On Mutation”. PhD thesis. Atlanta, Georgia: Georgia Institute of Technology, 1980 *Cited on page 26.*
- [ADO14] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. “Establishing Theoretical Minimal Sets of Mutants”. In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014–April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 21–30. DOI: 10.1109/ICST.2014.13 *Cited on pages 26, 53, 69, 109, 122.*
- [Ana+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061 *Cited on pages 6, 29, 34, 35, 110.*
- [And+06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”. In: *IEEE Trans. Software Eng.* 32.8 (2006), pp. 608–624. DOI: 10.1109/TSE.2006.83 *Cited on pages 2, 24, 25, 34–36, 41, 48.*
- [And+14] Kelly Androutsopoulos, David Clark, Haitao Dan, Mark Harman, and Robert Hierons. “An Analysis of the Relationship between Conditional Entropy and Failed Error Propagation in Software Testing”. In: *36th International Conference on Software Engineering (ICSE 2014)*. Hyderabad, India, June 2014, pp. 573–583 *Cited on pages 36, 109.*
- [AO08] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008. ISBN: 978-0-521-88038-1 *Cited on pages 12, 13, 36, 48, 53, 67, 111, 112, 123.*
- [AO16] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. ISBN: 9781107172012 *Cited on page 3.*

-
- [BA82] Timothy A. Budd and Dana Angluin. “Two Notions of Correctness and Their Relation to Testing”. In: *Acta Inf.* 18.1 (Mar. 1982), pp. 31–45. ISSN: 0001-5903. DOI: 10.1007/BF00625279 *Cited on pages 67, 101.*
- [Bar+15a] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. “Sound and Quasi-Complete Detection of Infeasible Test Requirements”. In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102607 *Cited on page 40.*
- [Bar+15b] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525 *Cited on page 40.*
- [Ber07] Antonia Bertolino. “Software Testing Research: Achievements, Challenges, Dreams”. In: *Future of Software Engineering 2007*. Ed. by Lionel Briand and Alexander Wolf. This volume. Los Alamitos, California, USA, 2007 *Cited on page 35.*
- [BH13] Richard Baker and Ibrahim Habli. “An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software”. In: *IEEE Trans. Software Eng.* 39.6 (2013), pp. 787–805. DOI: 10.1109/TSE.2012.56 *Cited on page 109.*
- [BMV01] Ellen Francine Barbosa, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. “Toward the determination of sufficient mutant operators for C”. In: *Software Testing, Verification and Reliability* 11.2 (May 2001), pp. 113–136 *Cited on page 26.*
- [BP00] Lionel C. Briand and Dietmar Pfahl. “Using simulation for assessing the real impact of test-coverage on defect-coverage”. In: *IEEE Trans. Reliability* 49.1 (2000), pp. 60–70. DOI: 10.1109/24.855537 *Cited on pages 24, 35.*
- [BR14] Marcel Böhme and Abhik Roychoudhury. “CoREBench: studying complexity of regression errors”. In: *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 2014, pp. 105–115. DOI: 10.1145/2610384.2628058 *Cited on pages 34, 37–40, 47, 48, 79.*
- [Bre01] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32 *Cited on page 75.*
- [BSR13] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. “Partition-based regression verification”. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 302–311. DOI: 10.1109/ICSE.2013.6606576 *Cited on page 30.*
- [Bud80] Timothy Alan Budd. “Mutation Analysis of Program Test Data”. PhD thesis. New Haven, Connecticut: Yale University, 1980 *Cited on page 26.*
- [CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008, pp. 209–224 *Cited on pages 34, 37, 40, 54, 79, 81, 110, 121, 136, 141.*

- [Ciu+11] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. “On the number and nature of faults found by random testing”. In: *Softw. Test., Verif. Reliab.* 21.1 (2011), pp. 3–28. DOI: 10.1002/stvr.415 Cited on pages 24, 25.
- [CN06] Rich Caruana and Alexandru Niculescu-Mizil. “An Empirical Comparison of Supervised Learning Algorithms”. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML ’06. Pittsburgh, Pennsylvania, USA: ACM, 2006, pp. 161–168. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143865 Cited on page 17.
- [Col+16] 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, Saarbrücken, Germany, July 18-20, 2016*. 2016, pp. 449–452. DOI: 10.1145/2931037.2948707 Cited on page 81.
- [CS13] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782 Cited on pages 15, 34.
- [DB08] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0 Cited on pages 15, 111.
- [Del+18] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John A. Clark, and Inmaculada Medina-Bulo. “Evaluation of Mutation Testing in a Nuclear Industry Case Study”. In: *IEEE Trans. Reliability* 67.4 (2018), pp. 1406–1419. DOI: 10.1109/TR.2018.2864678 Cited on pages 3, 109.
- [DeM+88] Richard A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. Jefferson Offutt. “An Extended Overview of the Mothra Software Testing Environment”. In: *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA ’88)*. Banff Alberta, Canada, July 1988, pp. 142–151 Cited on page 26.
- [DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. “Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact”. In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435. DOI: 10.1007/s10664-005-3861-2 Cited on page 38.
- [DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *IEEE Computer* 11.4 (Apr. 1978), pp. 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136 Cited on pages 12, 13, 80.
- [DM18] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. “Search-based mutant selection for efficient test suite improvement: Evaluation and results”. In: *Information and Software Technology* 104 (2018), pp. 130–143. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2018.07.011> Cited on page 28.

- [DMV01] Márcio Eduardo Delamaro, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. “Proteum/IM 2.0: An Integrated Mutation Testing Environment”. In: Boston, MA: Springer US, 2001. Chap. Mutation Testing for the New Century, pp. 91–101. ISBN: 978-1-4757-5939-6. DOI: 10.1007/978-1-4757-5939-6_17 Cited on page 26.
- [DO91] Richard A. DeMillo and A. Jefferson Offutt. “Constraint-Based Automatic Test Data Generation”. In: *IEEE Trans. Software Eng.* 17.9 (1991), pp. 900–910. DOI: 10.1109/32.92910 Cited on pages 13, 112.
- [DOA14] Márcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. “Designing Deletion Mutation Operators”. In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 11–20. DOI: 10.1109/ICST.2014.12 Cited on page 27.
- [DOL13] Lin Deng, Jeff Offutt, and Nan Li. “Empirical Evaluation of the Statement Deletion Mutation Operator”. In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 84–93. DOI: 10.1109/ICST.2013.20 Cited on pages 27, 71.
- [DP18] A. Denisov and S. Pankevich. “Mull It Over: Mutation Testing Based on LLVM”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 25–31. DOI: 10.1109/ICSTW.2018.00024 Cited on page 130.
- [ES07] Robert B. Evans and Alberto Savoia. “Differential testing: a new approach to change detection”. In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 2007, pp. 549–552. DOI: 10.1145/1287624.1287707 Cited on page 39.
- [FA15] Gordon Fraser and Andrea Arcuri. “Achieving scalable mutation-based generation of whole test suites”. In: *Empirical Software Engineering* 20.3 (2015), pp. 783–812. DOI: 10.1007/s10664-013-9299-z Cited on page 30.
- [FI98a] Phyllis G. Frankl and Oleg Iakounenko. “Further Empirical Studies of Test Effectiveness”. In: *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*. 1998, pp. 153–162. DOI: 10.1145/288195.288298 Cited on pages 2, 24, 25, 34, 48.
- [FI98b] Phyllis G. Frankl and Oleg Iakounenko. “Further Empirical Studies of Test Effectiveness”. In: *SIGSOFT Softw. Eng. Notes* 23.6 (Nov. 1998), pp. 153–162. ISSN: 0163-5948. DOI: 10.1145/291252.288298 Cited on page 13.
- [FPO18] F. Cutigi Ferrari, A. Viola Pizzoleto, and J. Offutt. “A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 1–10. DOI: 10.1109/ICSTW.2018.00021 Cited on pages 69, 71.
- [Fri02] Jerome H Friedman. “Stochastic gradient boosting”. In: *Computational Statistics & Data Analysis* 38.4 (2002), pp. 367–378 Cited on page 75.

- [FW91] Phyllis G. Frankl and Stewart N. Weiss. “An Experimental Comparison of the Effectiveness of the All-Uses and All-Edges Adequacy Criteria”. In: *Symposium on Testing, Analysis, and Verification*. 1991, pp. 154–164. DOI: 10.1145/120807.120821 Cited on pages 24, 25.
- [FW93] Phyllis G. Frankl and Stewart N. Weiss. “An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing”. In: *IEEE Trans. Software Eng.* 19.8 (1993), pp. 774–787. DOI: 10.1109/32.238581 Cited on pages 24, 25.
- [FWH97] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. “All-uses vs mutation testing: An experimental comparison of effectiveness”. In: *Journal of Systems and Software* 38.3 (1997), pp. 235–253. DOI: 10.1016/S0164-1212(96)00154-9 Cited on pages 12, 13, 24, 25.
- [FZ12] Gordon Fraser and Andreas Zeller. “Mutation-Driven Generation of Unit Tests and Oracles”. In: *IEEE Trans. Software Eng.* 38.2 (2012), pp. 278–292. DOI: 10.1109/TSE.2011.93 Cited on pages 29, 34, 67.
- [GFZ12] Florian Gross, Gordon Fraser, and Andreas Zeller. “Search-based system testing: high coverage, no false alarms”. In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 2012, pp. 67–77. DOI: 10.1145/2338965.2336762 Cited on pages 40, 48, 124.
- [GG75] John B. Goodenough and Susan L. Gerhart. “Toward a Theory of Test Data Selection”. In: *IEEE Trans. Software Eng.* 1.2 (1975), pp. 156–173. DOI: 10.1109/TSE.1975.6312836 Cited on page 12.
- [GJG14] Rahul Gopinath, Carlos Jensen, and Alex Groce. “Code coverage for suite evaluation by developers”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 72–82. DOI: 10.1145/2568225.2568278 Cited on pages 2, 24, 34.
- [GJG17] Rahul Gopinath, Carlos Jensen, and Alex Groce. “The Theory of Composite Faults”. In: *10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-18, 2017*. 2017 Cited on page 25.
- [Gli+13] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. “Comparing non-adequate test suites using coverage criteria”. In: *International Symposium on Software Testing and Analysis, ISSTA ’13, Lugano, Switzerland, July 15-20, 2013*. 2013, pp. 302–313. DOI: 10.1145/2483760.2483769 Cited on pages 2, 24, 34, 35, 48.
- [Gli+15] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. “Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites”. In: *ACM Trans. Softw. Eng. Methodol.* 24.4 (2015), p. 22. DOI: 10.1145/2660767 Cited on pages 24, 35.
- [Gon+17] Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. “Mutant reduction based on dominance relation for weak mutation testing”. In: *Information & Software Technology* 81 (2017), pp. 82–96. DOI: 10.1016/j.infsof.2016.05.001 Cited on page 27.

- [Gop+16] Rahul Gopinath, Mohammad Amin Alipour, Iftekhhar Ahmed, Carlos Jensen, and Alex Groce. “On the limits of mutation reduction strategies”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 511–522. DOI: 10.1145/2884781.2884787 Cited on page 54.
- [Gro+16] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. “Cause reduction: delta debugging, even without bugs”. In: *Softw. Test., Verif. Reliab.* 26.1 (2016), pp. 40–68. DOI: 10.1002/stvr.1574 Cited on page 40.
- [HA13] Mohammad Mahdi Hassan and James H. Andrews. “Comparing multi-point stride coverage and dataflow coverage”. In: *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 172–181. DOI: 10.1109/ICSE.2013.6606563 Cited on page 24.
- [Har+16] Farah Hariri, August Shi, Hayes Converse, Sarfraz Khurshid, and Darko Marinov. “Evaluating the Effects of Compiler Optimizations on Mutation Testing at the Compiler IR Level”. In: *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. 2016, pp. 105–115. DOI: 10.1109/ISSRE.2016.51 Cited on page 81.
- [Hen+16] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. “Comparing white-box and black-box test prioritization”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 523–534. DOI: 10.1145/2884781.2884791 Cited on pages 20, 68, 72, 83.
- [HHD99] Robert M. Hierons, Mark Harman, and Sebastian Danicic. “Using Program Slicing to Assist in the Detection of Equivalent Mutants”. In: *Softw. Test., Verif. Reliab.* 9.4 (1999), pp. 233–262 Cited on page 52.
- [HJL11] Mark Harman, Yue Jia, and William B. Langdon. “Strong higher order mutation-based test data generation”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 2011, pp. 212–222. DOI: 10.1145/2025113.2025144 Cited on pages 29, 110, 123.
- [HJZ15] Mark Harman, Yue Jia, and Yuanyuan Zhang. “Achievements, open problems and challenges for search based software testing (keynote)”. In: *8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*. Graz, Austria, Apr. 2015 Cited on page 34.
- [HS18] Farah Hariri and August Shi. “SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France: ACM, 2018*, pp. 860–863. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3240482 Cited on page 130.
- [Hut+94] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. “Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria”. In: *Proceedings of the 16th International Conference on Software Engi-*

- neering, Sorrento, Italy, May 16-21, 1994*. 1994, pp. 191–200 *Cited on page 38*.
- [IH14] Laura Inozemtseva and Reid Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 435–445. DOI: 10.1145/2568225.2568271 *Cited on pages 2, 24, 34*.
- [JH09] Yue Jia and Mark Harman. “Higher Order Mutation Testing”. In: *Information & Software Technology* 51.10 (2009), pp. 1379–1393. DOI: 10.1016/j.infsof.2009.04.016 *Cited on page 69*.
- [JH11] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62 *Cited on page 3*.
- [Jia+07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. In: *29th International Conference on Software Engineering (ICSE’07)*. May 2007, pp. 96–105. DOI: 10.1109/ICSE.2007.30 *Cited on page 104*.
- [JJE14] René Just, Darioush Jalali, and Michael D. Ernst. “Defects4J: a database of existing faults to enable controlled testing studies for Java programs”. In: *International Symposium on Software Testing and Analysis, ISSTA ’14, San Jose, CA, USA - July 21 - 26, 2014*. 2014, pp. 437–440. DOI: 10.1145/2610384.2628055 *Cited on page 103*.
- [JKA17] René Just, Bob Kurtz, and Paul Ammann. “Inferring mutant utility from program context”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 2017, pp. 284–294. DOI: 10.1145/3092703.3092732 *Cited on pages 73, 130*.
- [Jus+14] 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-22), Hong Kong, China, November 16 - 22, 2014*. 2014, pp. 654–665. DOI: 10.1145/2635868.2635929 *Cited on pages 12, 24, 25, 36, 41, 42, 48, 100*.
- [Kec16] Thomas Keck. “FastBDT: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification”. In: *arXiv preprint arXiv:1609.06119* (2016) *Cited on page 75*.
- [Kin+18] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. “Detecting Trivial Mutant Equivalences via Compiler Optimisations”. In: *IEEE Trans. Software Eng.* 44.4 (2018), pp. 308–333. DOI: 10.1109/TSE.2017.2684805 *Cited on page 81*.
- [Kin76] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252 *Cited on pages 15, 113*.
- [Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective”. In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7 *Cited on page 40*.

-
- [KL86] John C. Knight and Nancy G. Leveson. “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming”. In: *IEEE Trans. Software Eng.* 12.1 (1986), pp. 96–109. DOI: 10.1109/TSE.1986.6312924 Cited on page 100.
- [KMK13] Sang-Woon Kim, Yu-Seung Ma, and Yong-Rae Kwon. “Combining weak and strong mutation for a noninterpretive Java mutation system”. In: *Software Testing, Verification and Reliability* 23.8 (2013), pp. 647–668. DOI: 10.1002/stvr.1480. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1480> Cited on pages 6, 137, 140.
- [KPM10] Marinos Kintis, Mike Papadakis, and Nicos Malevris. “Evaluating Mutation Testing Alternatives: A Collateral Experiment”. In: *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. 2010, pp. 300–309. DOI: 10.1109/APSEC.2010.42 Cited on pages 26, 52, 54, 69.
- [KS16] Yasutaka Kamei and Emad Shihab. “Defect Prediction: Accomplishments and Future Challenges”. In: *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*. 2016, pp. 33–45. DOI: 10.1109/SANER.2016.56 Cited on pages 67, 72.
- [Kur+14] Bob Kurtz, Paul Ammann, Márcio Eduardo Delamaro, Jeff Offutt, and Lin Deng. “Mutant Subsumption Graphs”. In: *Mutation*. 2014, pp. 176–185 Cited on pages 53, 54.
- [Kur+16a] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. “Analyzing the validity of selective mutation with dominator mutants”. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 571–582. DOI: 10.1145/2950290.2950322 Cited on pages 27, 71, 83.
- [Kur+16b] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. “Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness”. In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. 2016, pp. 142–151. DOI: 10.1109/ICSTW.2016.41 Cited on page 122.
- [LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9 Cited on pages 8, 130.
- [Lau+17] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. “Assessing and Improving the Mutation Testing Practice of PIT”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 2017, pp. 430–435. DOI: 10.1109/ICST.2017.47 Cited on page 52.
- [Lev95] Nancy G. Leveson. *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995. ISBN: 978-0-201-11972-5 Cited on page 100.

- [LMH09] Kiran Lakhotia, Phil McMinn, and Mark Harman. “Automated Test Data Generation for Coverage: Haven’t We Solved This Problem Yet?” In: *4th Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART’09)*. Windsor, UK, Apr. 2009, pp. 95–104 *Cited on page 34.*
- [LPO09] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. “An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage”. In: *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*. 2009, pp. 220–229. DOI: 10.1109/ICSTW.2009.30 *Cited on pages 12, 24, 25.*
- [Mat91] Aditya P. Mathur. “Performance, Effectiveness, and Reliability Issues in Software Testing”. In: *Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC’79)*. Tokyo, Japan, Nov. 1991, pp. 604–605 *Cited on page 26.*
- [MB99] Elfurjani S. Mresa and Leonardo Bottaci. “Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study”. In: *Software Testing, Verification and Reliability 9.4* (Dec. 1999), pp. 205–232 *Cited on page 26.*
- [MC12] Paul Dan Marinescu and Cristian Cadar. “make test-zesti: A symbolic execution solution for improving regression testing”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 2012, pp. 716–726. DOI: 10.1109/ICSE.2012.6227146 *Cited on page 30.*
- [MC13] Paul Dan Marinescu and Cristian Cadar. “KATCH: high-coverage testing of software patches”. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 2013, pp. 235–245. DOI: 10.1145/2491411.2491438 *Cited on page 30.*
- [MCZ19] Dongyu Mao, Lingchao Chen, and Lingming Zhang. “An Extensive Study on Cross-project Predictive Mutation Testing”. In: *12th IEEE International Conference on Software Testing, Verification and Validation, ICST 2019, Xian, China, April 22-27, 2019*. 2019, pp. 160–171 *Cited on page 28.*
- [Mec+18] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. “Semantic Program Repair Using a Reference Implementation”. In: *Proceedings of the 40th International Conference on Software Engineering. ICSE ’18*. Gothenburg, Sweden: ACM, 2018, pp. 129–139. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180247 *Cited on page 15.*
- [MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. “Data Mining Static Code Attributes to Learn Defect Predictors”. In: *IEEE Trans. Software Eng.* 33.1 (2007), pp. 2–13. DOI: 10.1109/TSE.2007.256941 *Cited on pages 67, 72.*
- [MMP15] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. “Guided Mutation Testing for JavaScript Web Applications”. In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 429–444. DOI: 10.1109/TSE.2014.2371458 *Cited on page 27.*

- [MOK06] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. “MuJava: a mutation system for java”. In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. 2006, pp. 827–830. DOI: 10.1145/1134425 Cited on page 27.
- [Mor90] L. J. Morell. “A Theory of Fault-Based Testing”. In: *IEEE Trans. Softw. Eng.* 16.8 (Aug. 1990), pp. 844–857. ISSN: 0098-5589. DOI: 10.1109/32.57623 Cited on pages 13, 112.
- [MR16] Jakub Mořucha and Bruno Rossi. “Is Mutation Testing Ready to Be Adopted Industry-Wide?” In: Nov. 2016, pp. 217–232. ISBN: 978-3-319-49093-9. DOI: 10.1007/978-3-319-49094-6_14 Cited on page 4.
- [NA09] Akbar Siامي Namin and James H. Andrews. “The influence of size and coverage on test suite effectiveness”. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSA 2009, Chicago, IL, USA, July 19-23, 2009*. 2009, pp. 57–68. DOI: 10.1145/1572272.1572280 Cited on pages 24, 35.
- [Nam+15] Akbar Siامي Namin, Xiaozhen Xue, Omar Rosas, and Pankaj Sharma. “MuRanker: a mutant ranking tool”. In: *Softw. Test., Verif. Reliab.* 25.5-7 (2015), pp. 572–604. DOI: 10.1002/stvr.1542 Cited on page 28.
- [NAM08] Akbar Siامي 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)*. Leipzig, Germany, Oct. 2008, pp. 351–360 Cited on pages 26, 73, 83.
- [Nat+13] Roberto Natella, Domenico Cotroneo, João Durães, and Henrique Madeira. “On Fault Representativeness of Software Fault Injection”. In: *IEEE Trans. Software Eng.* 39.1 (2013), pp. 80–96. DOI: 10.1109/TSE.2011.124 Cited on pages 26, 28.
- [NK13] Alexey Natekin and Alois Knoll. “Gradient boosting machines, a tutorial”. In: *Frontiers in neurorobotics* 7 (2013) Cited on page 75.
- [OC94] A. Jefferson Offutt and W. Michael Craft. “Using compiler optimization techniques to detect equivalent mutants”. In: *Software Testing, Verification and Reliability* 4.3 (1994), pp. 131–154. DOI: 10.1002/stvr.4370040303. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.4370040303> Cited on page 13.
- [Off+96a] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. “An Experimental Determination of Sufficient Mutant Operators”. In: *ACM Transactions on Software Engineering and Methodology* 5.2 (Apr. 1996), pp. 99–118 Cited on pages 26, 52, 69, 71, 73, 81, 121, 133.
- [Off+96b] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. “An Experimental Evaluation of Data Flow and Mutation Testing”. In: *Softw., Pract. Exper.* 26.2 (1996), pp. 165–176. DOI: 10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K Cited on pages 12, 24, 25.
- [Off11] Jeff Offutt. “A mutation carol: Past, present and future”. In: *Information and Software Technology* 53.10 (2011). Special Section on Mutation Testing, pp. 1098–1107. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2011.03.007> Cited on pages 136–138, 142.

- [Off92] A. Jefferson Offutt. “Investigations of the Software Testing Coupling Effect”. In: *ACM Trans. Softw. Eng. Methodol.* 1.1 (1992), pp. 5–20. DOI: 10.1145/125489.125473 Cited on pages 25, 42.
- [ORZ93] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. “An Experimental Evaluation of Selective Mutation”. In: *Proceedings of the 15th International Conference on Software Engineering. ICSE '93*. Baltimore, Maryland, USA: IEEE Computer Society Press, 1993, pp. 100–107. ISBN: 0-89791-588-7 Cited on pages 5, 26, 67, 71.
- [OU01] A. Jefferson Offutt and Ronald H. Untch. “Mutation Testing for the New Century”. In: ed. by W. Eric Wong. Norwell, MA, USA: Kluwer Academic Publishers, 2001. Chap. Mutation 2000: Uniting the Orthogonal, pp. 34–44. ISBN: 0-7923-7323-5 Cited on page 2.
- [Pal+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. “Faults in linux: ten years later”. In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. 2011, pp. 305–318. DOI: 10.1145/1950365.1950401 Cited on page 100.
- [Pap+15] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. “Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015, pp. 936–946. DOI: 10.1109/ICSE.2015.103 Cited on pages 5, 13, 41, 48, 56, 67, 81, 101, 121, 130, 131, 135.
- [Pap+16] 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, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 2016, pp. 354–365. DOI: 10.1145/2931037.2931040 Cited on pages 26, 52, 54, 56, 109, 122, 124.
- [Pap+18] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. “Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical study on the Relationship Between Mutants and Real Faults”. In: *Proceedings of the 40th International Conference on Software Engineering. ICSE 2018*. May 2018 Cited on pages 26, 67, 69, 100, 103.
- [Pap+19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. “Mutation Testing Advances: An Analysis and Survey”. In: vol. 112. *Advances in Computers*. Elsevier, 2019, pp. 275–378. DOI: <https://doi.org/10.1016/bs.adcom.2018.03.015> Cited on pages 3, 6, 12, 26, 29, 53, 67, 69, 71, 73, 81, 101, 109, 110, 122–124.
- [Par72] David Lorge Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623 Cited on page 138.
- [Per+08] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. “Differential symbolic execution”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Geor-*

- gia, USA, November 9-14, 2008*. 2008, pp. 226–237. DOI: 10.1145/1453101.1453131
Cited on page 30.
- [Per+11] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. “Directed incremental symbolic execution”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, pp. 504–515. DOI: 10.1145/1993498.1993558 Cited on page 30.
- [Pet+18] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. “An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 47–53. DOI: 10.1109/ICSTW.2018.00027 Cited on pages 2, 3.
- [PI18] Goran Petrovic and Marko Ivankovic. “State of Mutation Testing at Google”. In: *40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2018, May 27 - 3 June 2018, Gothenburg, Sweden*. 2018 Cited on pages 2, 3, 27, 67, 73, 100, 109, 137.
- [Piz+19] 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”. In: *Journal of Systems and Software* (2019). ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2019.07.100> Cited on pages 6, 109.
- [PKC16] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. “Shadow of a doubt: testing for divergences between software versions”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 1181–1192. DOI: 10.1145/2884781.2884845 Cited on pages 30, 37, 39, 40, 48, 121, 123.
- [PKT18a] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation”. In: *Search-Based Software Engineering*. Ed. by Thelma Elita Colanzi and Phil McMinn. Cham: Springer International Publishing, 2018, pp. 309–324. ISBN: 978-3-319-99241-9 Cited on page 29.
- [PKT18b] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets”. In: *IEEE Transactions on Software Engineering* 44.2 (Feb. 2018), pp. 122–158. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2663435 Cited on page 29.
- [PM10a] Mike Papadakis and Nicos Malevris. “An Empirical Evaluation of the First and Second Order Mutation Testing Strategies”. In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*. 2010, pp. 90–99. DOI: 10.1109/ICSTW.2010.50 Cited on pages 24–26, 34, 36, 41, 71.

- [PM10b] Mike Papadakis and Nicos Malevris. “Automatic Mutation Test Case Generation via Dynamic Symbolic Execution”. In: *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. 2010, pp. 121–130. DOI: 10.1109/ISSRE.2010.38 Cited on pages 29, 81, 115.
- [PM11] Mike Papadakis and Nicos Malevris. “Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing”. In: *Software Quality Journal* 19.4 (2011), pp. 691–723. DOI: 10.1007/s11219-011-9142-y Cited on pages 29, 40, 115.
- [PM12] Mike Papadakis and Nicos Malevris. “Mutation based test case generation via a path selection strategy”. In: *Information & Software Technology* 54.9 (2012), pp. 915–932. DOI: 10.1016/j.infsof.2012.02.004 Cited on page 40.
- [PMD17] Ali Parsai, Alessandro Murgia, and Serge Demeyer. “LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems”. In: *Fundamentals of Software Engineering*. Ed. by Mehdi Dastani and Marjan Sirjani. Cham: Springer International Publishing, 2017, pp. 148–163. ISBN: 978-3-319-68972-2 Cited on page 142.
- [PT15] Mike Papadakis and Yves Le Traon. “Metallaxis-FL: mutation-based fault localization”. In: *Softw. Test., Verif. Reliab.* 25.5-7 (2015), pp. 605–628. DOI: 10.1002/stvr.1509 Cited on page 42.
- [Rad92] Radio Technical Commission for Aeronautics. *RTCA DO178-B Software Considerations in Airborne Systems and Equipment Certification*. 1992 Cited on pages 34, 36.
- [Rei95] Stuart C. Reid. “The Software Testing Standard — How you can use it”. In: *3rd European Conference on Software Testing, Analysis and Review (EuroSTAR ’95)*. London, Nov. 1995 Cited on pages 34, 36.
- [Rot+01] Gregg Roethermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. “Prioritizing Test Cases For Regression Testing”. In: *IEEE Trans. Software Eng.* 27.10 (2001), pp. 929–948. DOI: 10.1109/32.962562 Cited on page 72.
- [RWK17] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. “An empirical study on the application of mutation testing for a safety-critical industrial software system”. In: *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*. 2017, pp. 1401–1408. DOI: 10.1145/3019612.3019830 Cited on page 100.
- [SiR18] SiR. *Software-artifact Infrastructure Repository*. <https://sir.unl.edu/portal/bios/tcas.php>. Accessed: 2018-10-20. 2018 Cited on page 80.
- [Sun+17] Chang-ai Sun, Feifei Xue, Huai Liu, and Xiangyu Zhang. “A path-aware approach to mutant reduction in mutation testing”. In: *Information & Software Technology* 81 (2017), pp. 65–81. DOI: 10.1016/j.infsof.2016.02.006 Cited on page 27.
- [SW09] Ben H. Smith and Laurie Williams. “Should software testers use mutation analysis to augment a test set”. In: *Journal of Systems and Software* 82.11 (2009), pp. 1819–1832. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2009.06.031> Cited on pages 2, 4, 5.

- [SWF10] Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. “Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages”. In: *2010 10th International Conference on Quality Software*. July 2010, pp. 262–269. DOI: 10.1109/QSIC.2010.42
Cited on pages 136, 142.
- [SZ13] David Schuler and Andreas Zeller. “Covering and Uncovering Equivalent Mutants”. In: *Softw. Test., Verif. Reliab.* 23.5 (2013), pp. 353–374. DOI: 10.1002/stvr.1473
Cited on pages 28, 109, 111, 122.
- [T A+79] Allen T. Acree, Timothy A. Budd, Richard Demillo, Richard J. Lipton, and Frederick G. Sayward. “Mutation Analysis”. In: *Technical Report GIT-ICS-79/08* (Sept. 1979), p. 92
Cited on pages 5, 67, 71.
- [Tan+11] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. “eXpress: guided path exploration for efficient regression test generation”. In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 2011, pp. 1–11. DOI: 10.1145/2001420.2001422
Cited on page 30.
- [Tan+17] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. “Codeflaws: a programming competition benchmark for evaluating automated program repair tools”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 2017, pp. 180–182. DOI: 10.1109/ICSE-C.2017.76
Cited on pages 54–56, 79.
- [TR15] Shin Hwei Tan and Abhik Roychoudhury. “relifix: Automated Repair of Software Regressions”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015, pp. 471–482. DOI: 10.1109/ICSE.2015.65
Cited on page 48.
- [Unt09] Roland H. Untch. “On reduced neighborhood mutation analysis using a single mutagenic operator”. In: *Proceedings of the 47th Annual Southeast Regional Conference, 2009, Clemson, South Carolina, USA, March 19-21, 2009*. 2009. DOI: 10.1145/1566445.1566540
Cited on page 27.
- [UOH93] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. “Mutation Analysis Using Mutant Schemata”. In: *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '93. Cambridge, Massachusetts, USA: ACM, 1993, pp. 139–148. ISBN: 0-89791-608-5. DOI: 10.1145/154183.154265
Cited on pages 6, 115.
- [VD00] A. Vargha and H. D. Delaney. “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong”. In: *Jrnl. Educ. Behav. Stat.* 25.2 (2000), pp. 101–132
Cited on page 21.
- [Ver+18] Sten Vercammen, Mohammad Ghafari, Serge Demeyer, and Markus Borg. “Goal-oriented Mutation Testing with Focal Methods”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. Lake Buena Vista, FL, USA: ACM, 2018, pp. 23–30. ISBN: 978-1-4503-6053-1. DOI: 10.1145/3278186.3278190
Cited on page 6.

- [VM97] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997. Cited on pages 36, 43, 60.
- [Wan+17] 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, Santa Barbara, CA, USA, July 10 - 14, 2017*. 2017, pp. 295–306. DOI: 10.1145/3092703.3092714 Cited on pages 6, 130.
- [Wen+18] 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 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 2018, pp. 1–11. DOI: 10.1145/3180155.3180233 Cited on page 73.
- [WM95a] W. Eric Wong and Aditya P. Mathur. “Reducing the cost of mutation testing: An empirical study”. In: *Journal of Systems and Software* 31.3 (1995), pp. 185–196. DOI: 10.1016/0164-1212(94)00098-0 Cited on pages 5, 26, 67.
- [WM95b] W. Eric Wong and Aditya P. Mathur. “Reducing the cost of mutation testing: An empirical study”. In: *Journal of Systems and Software* 31.3 (1995), pp. 185–196. DOI: 10.1016/0164-1212(94)00098-0 Cited on page 69.
- [WMO12] Yi Wei, Bertrand Meyer, and Manuel Oriol. “Is Branch Coverage a Good Measure of Testing Effectiveness?” In: *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 194–212. ISBN: 978-3-642-25231-0. DOI: 10.1007/978-3-642-25231-0_5 Cited on pages 24, 25.
- [Woh+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. 2000. ISBN: 0-7923-8682-5 Cited on pages 21, 41.
- [Won93] W. Eric Wong. “On Mutation and Data Flow”. PhD thesis. West Lafayette, Indiana: Purdue University, 1993. Cited on page 26.
- [Xua+16] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. “B-Refactoring: Automatic test code refactoring to improve dynamic analysis”. In: *Information & Software Technology* 76 (2016), pp. 65–80. DOI: 10.1016/j.infsof.2016.04.016 Cited on page 40.
- [YHJ14] Xiangjuan Yao, Mark Harman, and Yue Jia. “A study of equivalent and stubborn mutation operators using human analysis of equivalence”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 919–930. DOI: 10.1145/2568225.2568265 Cited on pages 52, 109, 111.
- [ZH02] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Software Eng.* 28.2 (2002), pp. 183–200. DOI: 10.1109/32.988498 Cited on page 39.
- [Zha+10a] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. “Test generation via Dynamic Symbolic Execution for mutation testing”. In: Oct. 2010, pp. 1–10. DOI: 10.1109/ICSM.2010.5609672 Cited on pages 29, 110, 123.

- [Zha+10b] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. “Is operator-based mutant selection superior to random mutant selection?” In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 2010, pp. 435–444. DOI: 10.1145/1806799.1806863 *Cited on pages 27, 71.*
- [Zha+13] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. “Operator-based and random mutant selection: Better together”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 2013, pp. 92–102. DOI: 10.1109/ASE.2013.6693070 *Cited on page 48.*
- [Zha+16] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. “Predictive mutation testing”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 2016, pp. 342–353. DOI: 10.1145/2931037.2931038 *Cited on page 28.*
- [Zha+18] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang. “Predictive Mutation Testing”. In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2809496 *Cited on page 28.*
- [Zhe15] Alice Zheng. *Evaluating Machine Learning Models A Beginner’s Guide to Key Concepts and Pitfalls*. O’Reilly Media, Inc, 2015 *Cited on page 82.*
- [ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. “Software Unit Test Coverage and Adequacy”. In: *ACM Comput. Surv.* 29.4 (1997), pp. 366–427. DOI: 10.1145/267580.267590 *Cited on pages 12, 36.*
- [ZMK13] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. “Faster mutation testing inspired by test prioritization and reduction”. In: *International Symposium on Software Testing and Analysis, ISSTA ’13, Lugano, Switzerland, July 15-20, 2013*. 2013, pp. 235–245. DOI: 10.1145/2483760.2483782 *Cited on page 92.*

]