Examining the Impact of Propagating and Partitioning for Mutation Analysis of C Programs

by

Ali Arab
B.Sc. Sharif University of Technology, 2014

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2017

APPROVAL

Name:	Ali Arab
Degree:	Master of Science
Title of Thesis:	Examining the Impact of Propagating and Partitioning for Mutation Analysis of C Programs
Examining Committee:	Dr. Anoop Sarkar Chair
	Dr. William (Nick) Sumner, Assistant Professor, Senior Supervisor
	Dr. Arrvindh Shriraman, Associate Professor, Supervisor
	Dr. Richard Vaughan, Associate Professor, Internal Examiner
Date Approved:	May 11, 2017

Abstract

Mutation analysis is a technique for assessing the quality of test suites by seeding artificial defects into a program. The application of this method is still limited as it places a high demand on computational resources. There are techniques called infection, propagation, and partitioning that leverage the information available at run-time to reduce the execution time of mutation analysis. Although the effectiveness of these techniques has been investigated for Java, it is not known how they behave on programs written in low-level languages such as C. This thesis makes contributions toward investigating the effectiveness and efficiency of infection, propagation, and partitioning optimizations for programs written in C. It also explores the impact of statically pruning redundant mutants on these optimizations. The analysis of five real-world applications, with 402,000 lines of code in total, suggests that while infection might have the same effectiveness in C and Java, propagation and partitioning could be more effective for C. Additionally, it shows that static pruning reduces the impact of infection, propagation, and partitioning by 5.51%, 7.29%, and 11.90% respectively.

Keywords: Mutation Analysis, Program Analysis

Acknowledgments

I would like to express my sincere gratitude to my senior supervisor Prof. Nick Sumner for the continuous support of my study and research.

Besides my supervisor, I would like to thank Dr. Arrvindh Shriraman, Dr. Richard Vaughan, and Dr. Anoop Sarkar for serving as my thesis committee.

Last but not the least, I would like to thank my family for supporting me throughout my study.

Contents

Approval ii

Abstract iii

Acknowledgments iv

Contents v

List of Tables vii

List of Figures ix

1 Introduction 1

2 Background 3

2.1 Mutation Analysis . . . 3

2.2 Mutation Operators . . . 6

2.2.1 Absolute Value Insertion . . . 7

2.2.2 Relational Operator Replacement: . . . 7

2.2.3 Conditional Operator Replacement . . . 8

2.3 Optimization Techniques . . . 8

2.3.1 Reduction in the number of generated mutants . . . 9

2.3.2 Reduction in the compilation time . . . 9

2.3.3 Reduction in the mutants execution time . . . 10

2.4 Conclusion . . . 12

3 Approach 13

3.1 Notation . . . 13

3.2 Monitoring infected execution states . . . 14
3.2.1 Dynamic . . . 14
3.2.2 Static . . . 17
3.3 Propagating infected Execution States . . . 18
3.4 Partitioning Infected Execution States . . . 20
3.5 C Specific Issues . . . 22

4 Experimental Results and Analysis **25**
4.1 Effectiveness . . . 26
4.2 Static Pruning Impact . . . 27
4.3 Efficiency and Scalability . . . 28
4.4 Equivalency Likelihood of Operators . . . 30
4.5 Threats to Validity . . . 32

5 Conclusion and Future Work **34**
5.1 Future Work . . . 34

Bibliography **36**

List of Tables

2.1 Mutation Operators 7
2.2 Non-redundant mutations for relational operator [10] 9
2.3 Example to illustrate infection and propagation 11
2.4 Example to illustrate partitioning 11

3.1 Side effect examples 15
3.2 Expressions 16
3.3 Mutations 16
3.4 Constant Integer Replacement Pruning Rules 17
3.5 Absolute Value Pruning Rules 18
3.6 Comparison Mutation Pruning Rules 18
3.7 Mutations 19
3.8 Entries in ExprValues map after evaluation of $expr_3$ 20
3.9 ExprValues map after evaluation of expr_5 20
3.10 Partitioning Example 21

4.1 Benchmark 25
4.2 Survival rate for each optimization 27
4.3 Survival rate for each optimization with using static pruning 28
4.4 Percentage change in survival rate for infection, propagation, and partitioning after using static pruning (%) 28
4.5 Estimated run-time of mutation analysis for each optimization (seconds) 29
4.6 Ratio of estimated run-time compared to using only coverage 30
4.7 Processing time for each optimization (seconds) 30
4.8 Probability of generating test-equivalent mutants for each operator 32

4.9 Probability of generating test-equivalent mutants for each operator after using static pruning 32

List of Figures

2.1 Mutation Analysis Flowchart [8] 4
2.2 The original program 5
2.3 The Mutant 5

3.1 The original assignment 23
3.2 The transformed assignment 23

Chapter 1

Introduction

Software testing is an important part of software development cycle. Software testing techniques are concerned with detecting as many defect as early as possible. The National Institute of Standards and Technology estimated that inadequate software testing costs the United States $59.9 billion annually [3]. This cost emphasizes the need to have reliable and optimized testing techniques.

One major challenge in software testing is the impossibility of testing a program against all possible inputs. The reason is that for some programs the input space is infinite [1]. So, a limited number of inputs needs to be selected, which forms the test suite. A test suite is a collection of test cases. A test case is a set of test inputs, execution conditions, and expected outputs created to determine if the program satisfies requirements [7]. The program is executed for each test case and the the result of the evaluation is compared with the expected output. A difference in the output indicates the existence of a fault in the program. In addition to test cases, test criteria are used to to increase the confidence in correctness of the programs [17]. A test criterion is a constraint that needs to satisfied by the the test suite. Mutation testing is one of the effective testing criteria which determines the effectiveness of a test suite [2].

Mutation analysis measures the quality of test suite in terms of its ability to to detect faults. A set of mutation operators is used to insert small faults to the program, one a a time. Each new version is called a mutant. The test suite is evaluated against all generated mutants. For a mutant, if the test suite fails on a test, the mutant is said to be killed. Otherwise, the mutant survives. The outcome of the process is the ratio of killed mutants over the total number of mutants. The higher the ratio, the better the quality of the test suite.

A major challenge in mutation analysis is the high computational cost of executing test suite

against all mutants. There are numerous optimizations in the literature to reduce the number of executions. One method is to exploit the information available at run-time during the execution of the test suite [9]. If for a test, a mutation does not impact the state of an execution (infection), execution of the corresponding mutant is not necessary for that specific test. Furthermore, this pruning can be applied if the impacted execution state does not propagate to the enclosing expressions (propagation). Finally, among mutants for which the impacted execution state has been propagated to the enclosing expressions, if two of them have the same execution state, only of them needs to be executed (partitioning). These methods have been shown to be effective at reducing the mutation analysis time for programs written in Java [9]. However, their effectiveness for other languages has not been investigated yet.

In particular, we want to know if the results extend to C. For low level languages such as C, the optimizations might show different behaviour resulting in different effectiveness and efficiency, which needs to be investigated. In addition, the data provided by the optimizations can be leveraged to have a better understanding of the mutations. For example, as another area to explore, the optimizations' output can be used to know which operators are more likely to produce mutants that their execution is not necessary.

Additionally, some of the mutants can be pruned statically. If without running the test suite it can be inferred that the mutant is equivalent to the original program, the execution of the mutant can be avoided.

This work makes the following contributions to mutation analysis:

- Examining the effectiveness of the infection, propagation, and partitioning optimizations in C
- Investigated the impact of static pruning on effectiveness of the three optimizations

The results show that the optimizations reduce the number of executions by 50%. Furthermore, static pruning reduces the positive impact of the three optimizations. The average reduction is 5.51%, 7.29%, and 11.90% for infection, partitioning, and propagation respectively. Finally, the results suggest that absolute value insertion, bitwise mutation, and constant integer replacement are more likely to produce mutations that does not impact the program's execution state.

Chapter 2 describes the background and concepts used in this thesis. The optimizations and static pruning are explained in detail in chapter 3. Chapter 4 describes the experiments and reports the results. Finally, the conclusion and future work are presented in chapter 5.

Chapter 2

Background

In the testing phase of the software development cycle, a set of tests are created to increase the confidence in the correctness of the program. However, if the program passes all the test cases, it does not mean that the program is perfect. It could be the case that the test cases are not sufficient. For example, the test cases may not cover all parts of the code. To address this issue, some methods have been developed to measure the effectiveness of the test suites. One of the well known methods is *mutation analysis* [2].

Mutation analysis provides programmers with means to evaluate the quality of test suites. it injects small faults to the program, one at a time, and creates new instances of the program which are called *mutants*. If the mutant causes the test suite to fail, the mutant is said to be *detected* or *killed*. The greater the number of killed mutants, the better the quality of the test suite.

In the following section, the mutation analysis algorithm will be discussed. We will see that how high computational cost of mutation analysis has prohibited this method from being a practical method in software industry. In section 2.3 we introduce some of the techniques that have been developed to address this issue.

2.1 Mutation Analysis

Figure 2.1 illustrates the steps involved in mutation analysis. In the first step, by applying syntactic changes to the original program, a set of faulty programs are generated. For example, an integer literal is replaced with another constant integer such as one or zero to create a new version. As another example, an expression that performs addition can be modified to perform multiplication.

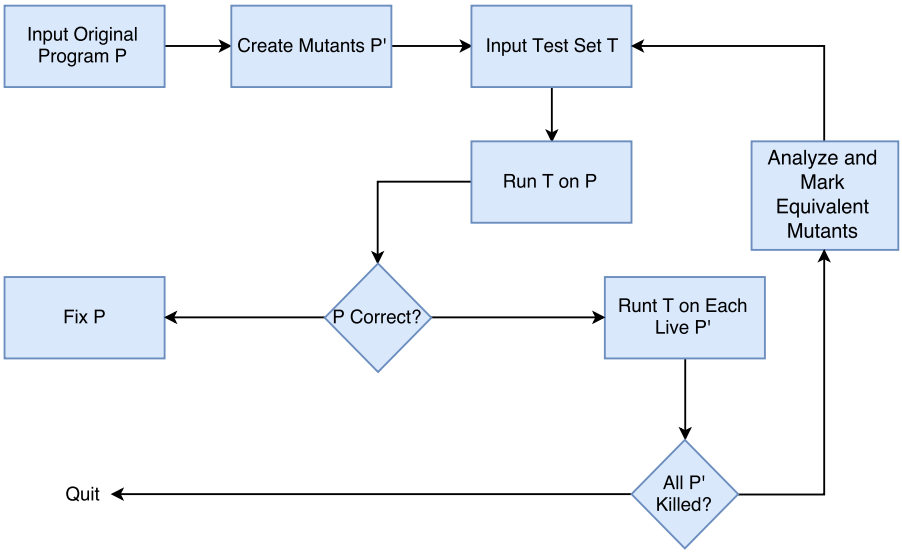

Figure 2.1: Mutation Analysis Flowchart [8]

Practically, it is not feasible to generate all possible mutants as the number of them would be infeasibly high. The reason for the infeasibility is the excessive number of potential faults that a program can have. Therefore, a subset of them that are close enough to the original program are selected. This approach is supported by two hypotheses: *the competent programmer hypothesis* (CPH) and *the coupling effect* [13, 14].

The CPH states that programs written by programmers tend to be close to the correct version of the program. Therefore, mutation analysis deals with mutants that are generated by applying simple changes which represent the programmer's faults [13].

The coupling effect claims that if test suite is sensitive enough that it can detect simple faults, it will be able to detect complex faults as well. In other words, it means that complex mutants are coupled to the simple mutants [14].

The rules that define how a mutant can be generated from the original program are called *mutation operators*. The common mutation operators can be classified into four categories : *statement mutations*, *operator mutations*, *variable mutations*, and *constant mutations*. Table 2.1 shows a subset of mutation operators.

In the next step, the the provided test suite is executed against the original program. In case

there are any failing test cases, they need to fixed before proceeding to the next step. Afterwards, the test suite needs be run against each mutant. For a mutant p, if any of the test cases fail, the mutant is said to be killed, otherwise, it is called a *surviving mutant*.

The set of survived mutants can be used by the programmer to improve the quality of the test suite. The programmer needs to add test cases that are able to kill the surviving mutants. However, it is not always possible to create test cases that are able to kill all the mutants. In some cases, the mutant is semantically identical to the original program. As an example, consider the two programs in figure 2.2 and 2.3. There is no test case that can distinguish between the two programs because they both have the same functionality. Such mutants are called *equivalent mutants*. Because program equivalence is undecidable, it is not possible to find all equivalent mutants automatically. The existence of equivalent mutants is recognized as one of the major challenges of mutation analysis [16].

```
1  ...
2  if(x == 2 && y == 2)
3   z = x + y;
4  ...
```

Figure 2.2: The original program

```
1  ...
2  if(x == 2 && y == 2)
3   z = x * y;
4  ...
```

Figure 2.3: The Mutant

In the final step, the output of the method, *the mutation score*, is calculated. The mutation score is an indication of the quality of the test suite. The mutation score can be calculated by dividing the number of killed mutants by the total number of non-equivalent mutants. The value for the mutation score is between zero and one. The value one is the best value that it can take, meaning that the test suite is able to detect all non-equivalent mutants. Test suites with this characteristic are called 100% mutation adequate.

$$\text{Mutation Score} = \frac{\text{Number of Killed Mutants}}{\text{Total Number of Non Equivalent Mutants}} \tag{2.1}$$

In spite of the effectiveness of mutation analysis in measuring the quality of test suites, it is still not recognized as a practical testing method. The first reason is that due to generating a large number of mutants, executing the test suite against each one of them results in a high computational cost. To address this issue, some optimization techniques have been suggested to reduce the cost of execution, such as *mutant sampling*, *selective mutation*, *mutant clustering*, and run-time optimizations which will be discussed in the following sections [12, 20, 6, 9].

The second reason is the amount of human effort that is needed to detect equivalent mutants. If the equivalent mutants are not detected, the mutation score can never reach 100%. Because of undecidability of detecting equivalent mutants, there is no automated method to detect equivalent mutants; so, always some human effort is needed. It has been observed that on average, the manual analysis of one mutant for equivalence is 15 minutes [5].

In the most recent work, it has been suggested to use available compiler technology to address the problem of detecting equivalent mutants. The work exploits the fact that for some of the equivalent mutants, their resulting object file, obtained by performing optimizations, are equal. In other words, if transformation of the mutant and the original program yield the same result, it can be inferred that they are equivalent. This method is able to detect 30% of equivalent mutants [16].

2.2 Mutation Operators

Table 2.1 describes some common mutation operators. To clarify, three of the operators will be explained in more details.

Operator	Description
ABS: Absolute Value Insertion	$\{(e, \text{abs}(e)), (e, -\text{abs}(e))\}$
AOR: Arithmetic Operator Replacement	$\{(x,y)\|x,y \in \{+,-,*,/,\%\} \wedge x \neq y\}$
LCR: Logical Connector Replacement	$\{(x,y)\|x,y \in \{\&\&,\|\|\} \wedge x \neq y\}$
SOR: Shift Operator Replacement	$\{(x,y)\|x,y \in \{>>,<<\} \wedge x \neq y\}$
ROR: Relational Operator Replacement	$\{(x,y)\|x,y \in \{>,>=,<,<=,==,!=\} \wedge x \neq y\}$
UOI: Unary Operator Insertion	$(v,--v),(v,v--),(v,++v),(v,v++)$
CRCR: Integer constant replacemen	$\{(c_i,x)\|x \in \{1,-1,0,c_i+1,c_i-1,-c_i\}\}$
OAAA: Arithmetic assignment mutation	$\{(x,y)\|x,y \in \{+=,-=.*=,/=\} \wedge x \neq y\}$
OBBN: Bitwise operator mutation	$\{(x,y)\|x,y \in \{\&,\|,\hat{}\} \wedge x \neq y\}$
OCNG: Logical context negation	$\{(e,!(e))\|e \in \{\text{if}(e), \text{while}(e)\}$

Table 2.1: Mutation Operators

2.2.1 Absolute Value Insertion

Absolute Value Insertion operator replaces each arithmetic expression with absolute value and negative absolute value of the expression. The function abs which returns the absolute value of an expression is used to implement this operator. The operator has been designed to force each numeric expression to have negative and positive values [1]. For instance, the statement $x = 1 + a$ is mutated to create the following statements:

$$x = 1 + \text{abs}(a) \tag{2.2}$$

$$x = 1 + -\text{abs}(a) \tag{2.3}$$

2.2.2 Relational Operator Replacement:

Relational Operator Replacement replaces each operator in the set $\{<, <=, >, >=, =, !=\}$ with other members to create new mutants [1]. For instance, mutating the statement if($a < b$) will create the following mutants:

$$\text{if}(a > b) \tag{2.4}$$

$$\text{if}(a >= b) \tag{2.5}$$

$$\text{if}(a <= b) \tag{2.6}$$

$$\text{if}(a == b) \tag{2.7}$$

$$\text{if}(a! = b) \tag{2.8}$$

2.2.3 Conditional Operator Replacement

Conditional Operator Replacement is designed to make sure that the program executes different branches. It forces the the condition to be true, false, or negates the condition [1]. For example mutating the statement if($a > b$) produces the following statements:

$$\text{if(true)} \tag{2.9}$$

$$\text{if(false)} \tag{2.10}$$

$$\text{if}(!(a > b)) \tag{2.11}$$

2.3 Optimization Techniques

Due to infeasibly high number of generated mutants, mutation analysis is notorious to be computationally expensive. However, there have been some efforts to make this technique practical. The proposed techniques can be categorized into three types: reduction in the number of generated mutants, reduction in the compilation time, and reduction in the execution cost. In the following sections, the three types of techniques will be described.

2.3.1 Reduction in the number of generated mutants

Since the test suite needs to be tested on all mutants, reduction in the number of mutants can lead to significant improvement in the execution time. The goal of these techniques is to find a minimal subset of mutants in such a way that the mutation score of the subset is almost the same as the original one. The mutation score is involved to make sure that the test effectiveness has not been affected significantly. One of the methods introduced to reduce the number of generated mutants is selective mutation [20].

The mutants are generated by applying mutation operators on the expressions, but not all the generated mutants are independent. In selective mutation, the goal is to find a minimal set of mutation operators while keeping the test effectiveness as high as possible. One approach is to use non-redundant mutations. In this method, instead of generating all mutants, a reduced, yet sufficient, set of mutants are generated [10]. For example, it has been shown that the following three replacements are sufficient when mutating relational operators [10]:

<=	↦	<, ==, true	<	↦	<, !=, false
>=	↦	>, ==, true	>	↦	>=, !=, false
==	↦	<=, >=, true	!=	↦	<, >, true

Table 2.2: Non-redundant mutations for relational operator [10]

2.3.2 Reduction in the compilation time

In order to perform mutation analysis, the mutants need to be executed on test suite. Although initially interpreter-based approaches were used, because of their inefficiency, they were replaced by compiler based methods [11]. In compiler based methods, each mutants is compiled to machine code for later execution. Because the execution of machine code is faster than interpretation, this approach is more efficient than interpretation techniques. However, since there are a huge number of mutants, the problem that needs to be dealt with is the time consuming task of mutant compilation.

To solve the above mentioned issue, *mutant schema* technique was introduced. In this method, instead of generating and compiling each mutant separately, all possible mutations are encoded into one source-level program called *metamutant*. Therefore, the only compilation cost would be one complication of the metamutant program. Because of the need to insert some extra checks

in the program, the execution of the metamutant is slightly slower than the execution of the mutants obtained by compiling mutants separately, however, the overhead is negligible considering the improvement achieved in compilation time [18].

2.3.3 Reduction in the mutants execution time

Another way to reduce the cost of mutation analysis is to optimize the process of mutant execution. The goal of these optimizations is to decide on the result of the execution, fail or pass, as soon as possible mostly using dynamic analysis.

For a test to fail on a mutant, three conditions should be held, 1) the test need to reach mutated code, 2) the test needs to achieve a different execution state, and 3) the state difference needs to *propagate* to an observable output. Hence, if one of the three conditions is not met, the mutant can be excluded from execution. If the program state of the mutant after execution is different from the program state of the original program, it is said to be *infected*. If executing a test on a mutant does not infect the execution state, running the test will not result in detection of that mutant. In this case, the mutant is called *test-equivalent* for that test [9].

As an example, consider table 2.3. The original expression $a * (b + c)$ has been mutated to $a * (b - c)$. The first column shows the values for each test. For the first test, because c is zero, the value of original expression $a + b$ and the mutated expression, $a - b$, are the same. So the the expression is not infected and causes the the mutant to be test equivalent for this test. On the other hand, for the second test, the value of $b + c$ and $b - c$ are different, however, although there is local infection, because of multiplication by zero, it can not be propagated to the enclosing expression. So, the mutant is test equivalent for this test as well. Finally, for the third test, there is local infection and it propagates to the enclosing expression.

As another optimization, the mutants that survive from the previous filtering can be *partitioned* based on their execution state. For mutants that have the same state, only one of them needs to be executed because the result of other mutants would be the same as the partition representative. Table 2.4 shows an example in which the mutants have been partitioned into two cells [9].

The above mentioned optimizations have been implemented in Major, a mutation analysis framework in Java [9]. For Java, it's been shown that invoking these optimization would result in 40% reduction in the execution time. However, their effectiveness in other languages such as C remains unknown. There are some fundamental differences in execution model of C and Java that might affect the results. For example, Java is compiled to byte-code which then is interpreted

Test	Original	Mutated	Infected?
t1:=	$\underbrace{b+c}_{3}$	$\underbrace{b-c}_{3}$	t1-equivalent
a=0 b=3 c=0	$\underbrace{a*\underbrace{(b+c)}_{3}}_{0}$	$\underbrace{a*\underbrace{(b+c)}_{3}}_{0}$	t1-equivalent
t2:=	$\underbrace{b+c}_{4}$	$\underbrace{b-c}_{2}$	infected
a=0 b=3 c=1	$\underbrace{a*\underbrace{(b+c)}_{4}}_{0}$	$\underbrace{a*\underbrace{(b+c)}_{2}}_{0}$	t2-equivalent
t3:=	$\underbrace{b+c}_{4}$	$\underbrace{b-c}_{2}$	infected
a=2 b=3 c=1	$\underbrace{a*\underbrace{(b+c)}_{4}}_{8}$	$\underbrace{a*\underbrace{(b+c)}_{2}}_{4}$	infected

Table 2.3: Example to illustrate infection and propagation

test	Original	m1	m2	m1	m2
a=1, b=1	$\underbrace{(b+c)}_{2}$	$\underbrace{(b*c)}_{1}$	$\underbrace{(b/c)}_{1}$	$\underbrace{(b-c)}_{0}$	$\underbrace{(b\%c)}_{0}$
		$\underbrace{\qquad\qquad}_{p1}$		$\underbrace{\qquad\qquad}_{p_2}$	

Table 2.4: Example to illustrate partitioning

at run-time, but C code is compiled directly to machine code. The data available at run-time may enable Java Virtual Machine to perform some optimizations that can not be performed statically.

Major restricts its analysis to only dynamic analysis to detect test-equivalent mutants. However, for some mutants, it is possible to decide on their equivalency statically. For example, if the value of an integer is known to be positive, it can be concluded that inserting absolute value mutation for that integer is going to be equivalent to the original program, so it would be test-equivalent for all of the tests.

Another approach to reduce the execution time of running test suites is to reorder the test cases. With the assumption of presence of bug in a program, this approach tries to reach the bug as soon as possible so that the rest of the test cases can be discarded, which leads to a reduction in the execution time of test suite. One promising order is based on running time of each test case in which the test cases are sorted in descending order. This way, the long running tests will be executed last [10].

2.4 Conclusion

Although mutation analysis is known to be an effective method to evaluate the quality of test suites, its practical use has been limited because of existence of a large number of mutants which leads to extremely high analysis time. There have been some efforts to decrease the analysis time by leveraging dynamic analysis. As the most recent work, Major, a mutation analysis framework, has been developed which is able to filter some of the test-equivalent mutants using the infection, propagation, and partitioning optimizations. Since this framework solely support programs written in Java, it leaves this question unanswered that how their effectiveness is in other programming languages. In addition, none of the current works investigate the effect of static pruning on the mentioned optimizations. For some operators, it is possible to mark them as equivalent when instrumenting the program without even running the mutated statements.

To address the above mentioned questions, in this work the following questions will be answered:

- What is the effectiveness of infection, propagation, and partitioning optimizations in the C language?
- How does static pruning of mutants affect infection, propagation, and partitioning optimizations?

Chapter 3

Approach

In this chapter, three optimizations to reduce the execution time of mutations analysis are introduced. The goal of the first two optimizations is reduce the mutation analysis time by detecting as many test-equivalent mutants as possible. The third seeks the same goal by avoiding redundant execution of mutants.

3.1 Notation

The following notations are used in this chapter to illustrate the optimizations [9]:

- Test suite T
- Test $t \in T$
- Expression $expr := VARIABLE | LITERAL | \langle op, \overline{expr} \rangle$
- Expression operator op
- $n-tuple$ $(n \geq 0)$ of expressions $\overline{expr}$
- Expression value $value$
- Expression evaluation $[\![]\!] : expr \mapsto value$
- i-th evaluation of $expr$, executing $t : [\![expr]\!]_i^t = value$

An expression can be a literal, variable, or n-ary expression. As an example, the expression $(a+b)*2$ contains five expressions:

$$\underbrace{\underbrace{(\underbrace{a}_{1}+\underbrace{b}_{2})}_{3}*\underbrace{c}_{4}}_{5}$$

This expression can be represented as:

$$\begin{aligned} expr_5 &= \langle op_5, expr_3, expr_4 \rangle \\ &= \langle op_5, \langle op_3, expr_1, expr_2 \rangle, expr_4 \rangle \\ &= \langle *, \langle +, a, b \rangle, c \rangle \end{aligned}$$

3.2 Monitoring infected execution states

Performing a mutation may or may not affect the value the mutated expression. In this section, we explore two approaches, dynamic and static, to prune mutants that do not change the execution state of their target expression.

3.2.1 Dynamic

The execution state of a mutated expression is infected by a test if within a test, its value is different from the value of the original expression. During the execution of a test, an expression could be evaluated multiple times. For example, expressions that are inside a loop are evaluated several times based on the loop condition.

Definition 3.2.1. Infected execution state: Suppose expression $expr_j$ is evaluated N times within execution of test t. Additionally, the mutated version of $expr_j$ is referred as $expr_j$. Test t infects the execution state of $expr_j$ at runtime *iff*:
$\exists i \in [1..N] : [\![expr_j]\!]_i^t \neq [\![expr_j\prime]\!]_i^t$ [9]

A mutation may change the side effect. A function or expression is said to have side effect if it modifies memory, or performs an I/O operation, or calls a function that does any of those operations. If performing the mutation causes the side effect to be different in the mutated expression, Definition 3.2.1 is no longer true; because, regardless of the mutated expression's evaluated value, the program state will be changed [9]. In the presence of side effects, to be conservative, all tests that reach the the mutated expressions are treated as if they have infected the execution state of the mutants. Table 3.1 provides some examples to show which kind of mutations may change side effects.

Original	Mutated	Changes Side effect?
if(x>y)	if(x>=y)	No
if(x>y)	if(x>y++)	Yes
if(x|| foo(y))	if(false)	Maybe

Table 3.1: Side effect examples

The process in which for each test, the infected mutations are determined is referred as monitoring infection. This process is accomplished through a two step process:

1. Instrumenting the original program to insert function calls responsible for performing the analysis

2. Running the test suit against the instrumented program

In the instrumentation phase, every expression is replaced with one or more function calls, depending on the number of mutations that are applicable on them. For example, because for an integer literal, both constant integer replacement and absolute value insertion operators are applicable, the expression is replaced with two nested function calls, each corresponding to one mutation. We refer to functions in the run-time library as $eval$. For example, the transformation for the expression $(a+b)*2$ will be:

$$\underbrace{(\overbrace{a}^{expr_1}+\overbrace{b}^{expr_2})}_{expr_3}*\overbrace{c}^{expr_4}\underbrace{\qquad}_{expr_5} \quad \mapsto \quad eval(5,*,eval(3,+,a,b),c)$$

The arguments of the evaluation functions are the mutation's identifier, the operands, and the operand's identifiers. In case the expression is binary, the original operation code needs to be provided as well. This representation implicitly includes the following mapping:

- Expressions: expr id $\mapsto$ original opcode, $\overline{\text{expr id}}$

For example, for the above expression, the map has two entries as shown in table 3.2. Subscript e indicates that the ids refer to expressions.

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto *, 3_e, 4_e$

Table 3.2: Expressions

In order to identify mutations, the combination of expression id and the new opcode is used as a key. Table 3.3 shows some mutation ids and their corresponding mutated expressions for the above example.

Mutation ID	Mutated Expression
$(5, -)$	$(a + b) - c$
$(3, /)$	$(a / b) * c$

Table 3.3: Mutations

The instrumented program, while functioning the same as the original one, performs some additional analysis in order to monitor the infections. Each $eval$ function applies a set of mutations on the current expression; mutations that infect the execution state of the expression are marked as infected for the current test.

To handle expressions with side effects, the instrumentation keeps the order and the number of evaluations the same as the original program. To maintain this characteristic, instrumentation of some of the statements needs special care. Conditional operators are of one of such statements.

The conditional operators have a short circuiting characteristic which needs to be preserved. In short circuiting, the second argument will be evaluated only if the evaluation of the first argument is not sufficient to determine the value of the whole expression. For example, in expression $a\&\&b$, b is evaluated only if a is true. Therefore, it is not valid to transform this expression to $eval(id, a, b)$ because in this case, b will always be evaluated, regardless of the value for a. To handle this case, each expression $a\ op\ b$, where op is a logical operator, will be replaced by two function calls:

a op b $\mapsto$ lhs(a) op rhs(b)

The lhs function evaluates a and return its value. When evaluation of rhs is needed, the rhs function is executed. Based on the operator type, rhs can infer the value of a and return the right value.

3.2.2 Static

Some mutations can be pruned at instrumentation time. These mutations are either the ones that will not infect the execution state of the mutated expressions, or are redundant mutations that are duplication of other mutants [10]. The first case happens when the mutated expression is semantically the same as the original expression. For instance, inserting absolute value insertion for a positive integer will create an equivalent mutant. The second case occurs when the mutated expression is equivalent to the result of other mutations. Tables 3.4, 3.5 and 3.6 illustrate the cases that mutations are pruned for constant integer replacement, absolute value insertion, and comparison mutation operator respectively.

Value	Pruned Mutations
1	Replace with one
	Minus one
	Negative
0	Replace with zero
	Minus_One
	Plus one
	Negative
-1	Replace with minus one
-2	Plus one

Table 3.4: Constant Integer Replacement Pruning Rules

Value	Pruned Mutations
Postive	Absolute Value
Negative	Negative Absolute Value
zero	Absolute Value
	Negative Absolute Value

Table 3.5: Absolute Value Pruning Rules

Condition	Pruned Mutations
LHS is unsigned and evaluates to zero	Greather Than or Equal
	Greather Than
RHS is unsigned and evaluates to zero	Less Than or Equal
	Less Than

Table 3.6: Comparison Mutation Pruning Rules

3.3 Propagating infected Execution States

The local infection of an expression might or might not change the execution state of the enclosing expressions. If the local infection affect the execution state of an enclosing expression, the infection is said to be propagated to the enclosing expression. The mutants that have local infections which do not propagate to the enclosing expressions will be test-equivalent mutants [9].

Definition 3.3.1. Propagation of Infected Execution States: Let $expr_j$ be a mutated expression that its execution state has been infected withing execution of test t. The infected execution state of $expr_j$ propagates to an enclosing expression $expr_e$ iff:
$\exists i \in 1...N$
$[\![\langle op_e, ...\langle ..., expr_j, ...\rangle ...\rangle]\!]_i^t \neq [\![\langle op_e, ...\langle ..., expr_j, ...\rangle ...\rangle]\!]_i^t$ [9]

In the compound expression, the only mutated expression is $expr_j$; so, if the infected execution state does not propagate to any enclosing expression $expr_e$, the propagation of infected execution state of $expr_j$ does not need to be monitored anymore because it will not lead to any infected execution state.

The run-time library explained in the previous section can be extended to implement this optimization. While evaluating compound expressions, the run-time library maintains a mapping which contains information about mutation values. We refer to this map as ExprValues. In this map, the keys are mutation ids and the mapped values are the value of the currently-executing expressions. The map includes the mutations that have infected the execution state of the current expression, and also mutations from sub-expressions that have propagated to the current expression. Once a mutation does not propagate to any of enclosing expressions, it will be removed from the mapping.

To keep track of surviving mutations, the top level expressions need to be instrumented as well. If a mutation reaches one of these locations, it will be marked as surviving. Assignments, return statements, argument, and conditions are examples of such points. We refer to these locations as commit points. In commit points, a handler function will be added which is responsible for marking the surviving mutations and clearing the expression value map.

The following example illustrates the propagation process [9].

Consider the following expression:

$$\underbrace{\underbrace{(\overbrace{a}^{expr_1} + \overbrace{b}^{expr_2})}_{expr_3} > \overbrace{c}^{expr_4}}_{expr_5}$$

Table 3.7 shows the mutations of the expression and also the input values. Ids with subscript m are used to denote the mutations.

Test	Mutations
t:=	$1_m \mapsto 3_e, -$
a=2	$2_m \mapsto 3_e, *$
b=2	$3_m \mapsto 5_e, <$
c=2	$4_m \mapsto 5_e, >=$

Table 3.7: Mutations

The evaluation starts from the innermost expression which in this case is $expr_3$, $a+b$. Within execution of $expr_3$, the run-time library applies the mutations whose target expression is $expr_3$

i.e 1_m and 2_m. Table 3.8 shows the values of applied mutations. Because mutation 2_m does not infect the execution state of the $expr_3$, it will be removed form the ExprValues map.

Expression Values
$1_m \mapsto 0$
~~$2_m \mapsto 4$~~

Table 3.8: Entries in ExprValues map after evaluation of $expr_3$

In the next step, the run-time library executes the evaluator function for $expr_5$. First, it propagates the mutations from sub-expression $expr_3$. So, mutation 1_m will be propagated to the $expr_5$. Since it infects the execution state of $expr_5$, it will remain in the ExprValues map. Next, mutations 3_m and 4_m will be applied on expression $expr_5$. Among these new mutations, only mutation 3_m infects the execution state of $expr_5$. So, the final state of ExprValues map would be:

Expression Values
$1_m \mapsto$ false
$3_m \mapsto false$
~~$4_m \mapsto$ true~~

Table 3.9: ExprValues map after evaluation of expr_5

3.4 Partitioning Infected Execution States

There are cases that two mutations show the same behaviour. In other words, the expression value they produce for the mutated expression is the same for all executions withing a test. So, if one of them is detected, it can be implied that the other one will be detected as well. Similarly, test-equivalency of one of them implies test-equivalency of the other one. Based on this observation, the set of mutations M_{expr_j} that infect the top-level expression $expr_j$ can be partitioned based the value they have produced. For test t, if two mutations have the same value for all executions, they will be put in the same partition [9].

Definition 3.4.1. Partitioning of Infected Execution States: Let M_{expr_j} represents the set of mutated expressions which are infected during execution of test t. Partition P of M_{expr_j} satisfies the following properties:

$$\bigcup_{p \in P} p = M_{expr_j} \wedge \forall p_1, p_2 \in P : p_1 \neq p_2 \Rightarrow p_1 \cap p_2 = \emptyset$$
$$\forall expr_j\prime, expr_j\prime\prime \in M_{expr_j} :$$
$$expr_j\prime, expr_j\prime\prime \in p \Leftrightarrow \forall i : [\![expr_j\prime]\!]_i^t = [\![expr_j\prime\prime]\!]_i^t \text{ [9]}$$

In mutation analysis, to reduce the number of mutants to be executed, one representative mutant can be selected from each partition $p \in P$. If a test t can can detect the representative mutant, all other mutant in the partition can be marked as detected. If the test t can not detect the mutant, all of the mutations inside the partition will be t-equivalent [9].

The partitioning can be done at two levels: per test, and per test suite. Partitioning per test suite is easier to implement, but the the partition cells will be smaller, which reduces the effectiveness of this optimization. In this work, the partitioning is done per test. The other benefit of partitioning per cell is that it eliminates the need to run representative of a cell if all mutations in that cell are already detected in previous tests. Table 3.10 provides an example:

Test	Partition	Representative	Executed?	Detected?
t1	1 2 3 4	1	Yes	Yes
	5 6	5	Yes	No
t2	1	1	No	
	2 3	2	No	
	4 5	4	Yes	
	6	6	Yes	

Table 3.10: Partitioning Example

In this example, in the mutation analysis phase, the test t1 is executed on a representative of its two cells, 1 and 5. Suppose that 1 is detected and 5 is t1-equivalent. In the next step, when t1 is going to be executed, the first two cells can be skipped because all mutants in those cells are already detected by test t1.

In order to implement this optimization, at any top level expression in the source code, the entries in the ExprValues map are written to a file. So, after execution of a test, there is a log file that contains the values of mutations that have propagated to the top-level expressions. This file can be used in an offline process to perform the partitioning.

3.5 C Specific Issues

In this section we explain the approach we have taken to implement the above mentioned optimizations in C. Our system consists of two major components, the source code transformer and the run-time library, which will be discussed in the following paragraphs.

The first component, the source code transformer, recognizes the expressions on which mutations are applicable and replaces them with function calls. To apply the transformations, a tool called Clang has been used. Clang is a compiler front end for C like languages such as C, C++, Objective-C, Objective-C++, and CUDA. Clang supports plugins which enables programmers to perform user-defined actions during compilation.

The transformation is done through a two step process. First, once the abstract syntax tree (AST) is generated by Clang based on the source code, we iterate over the AST to find mutations for each expression in the program. Next, in the second iteration, the AST is modified to contain the calls to the run-time library functions.

The second component is the run-time library which is loaded dynamically at execution time. Two run-time libraries have been implemented with the same interface, but with different functionalities. One of them implements the schemata semantic and the second one is responsible for performing the optimizations. Having the same interface for both of the libraries lets us to have one instrumented version of the target program and load the library at run-time based on our need.

This execution model has both advantages and disadvantages. One of the advantages is that the *eval* functions are dynamically bound when the application is loaded, which permits the application inherits the changes to the run-time library without the need to recompile. Another benefit is that since several applications use the same shared library, it reduces the main memory use and also the amount of disk space needed to store the applications. On the other hand, the major drawback of this method is that separate compilation of the application and the run-time library may eliminate some opportunities to do optimizations by the compiler.

During transformation of the source code, there are some cases that need special care. For most of the mutations, the arguments for their corresponding function in the run-time library are the mutation id, the operands and their ids, and operation code for the original expression, if applicable. However, for some mutations, instead of the value of the operands, their address needs to be passed to the run-time library functions. The reason is that such mutations need to change the value of their operands; as an example, unary insertion falls into this category. The

issue is that for some variables, even though such mutations are applicable, it is not valid to take their address. For instance, this happens when the variable is defined using *Register* keyword. This keyword is used to ask the compiler to keep the variable in a processor register, which means it will not have a memory address. To solve this issue, in AST transformation, the register keyword needs to be removed from all variable declarations.

The other problematic issue in the transformation process is existence of anonymous *enums* and unions. To add the handler function for commit points, a temporary variable is defined to hold the value of the top-level expression. Next, a handler function from the run-time library is called to mark the surviving mutants. Then, the temporary variable is returned as the value of the expression. To implement this transformation, we need to use compound statements. A compound statement is a sequence of statements surrounded by braces. The last statement determines the value of the whole statement. For example, the code in Figure 3.2 is the result of such transformation for code in Figure 3.1.

```
1  int x;
2  x= 4;
```

Figure 3.1: The original assignment

```
1  ({
2  int x;
3  int temp = eval(2);
4  assignmentHandler();
5  x = temp;
6  })
```

Figure 3.2: The transformed assignment

The problem appears when the expression is of type anonymous, because the type for temporary variable can not be specified. The solution is to deanonymize all unions and enums while transforming the AST. They can be uniquely named based on their order in tree traversal.

The other issue that needs to be dealt with is existence of *undefined behaviour* in C. Undefined behaviour is defined as a behaviour for which International Standard imposes no requirements. For example, integer division with zero as the divisor is an undefined behaviour. This

operation causes a hardware exception in x86, while PowerPC silently ignores that. This flexibility permits the compiler to choose an efficient implementation for the target platform [19]. During mutation analysis, it is possible that applying a mutation operator causes an undefined behaviour in the mutant. Since such mutants are not meaningful, they can not be considered as killed or surviving. So, they should be excluded in mutation analysis. Handling undefined behaviour is out of scope of this work, however, it has been shown that static analysis methods can be used to detect and prune such mutants [4].

Chapter 4

Experimental Results and Analysis

We evaluated the discussed methods on the five open source projects listed in Table 4.1. Each project includes a regression test suite. We consider only test cases that pass in the execution of the original program. The first column in the table shows the number of source lines of code (SLOC), ranging from 6,690 to 153,663. The number of mutations given in the third column indicates the number of mutants generated by applying mutations listed in Table 2.1. Git is a distributed revision control system. Diffutils is a package of several tools used to find differences between files. Grep is a command-line utility to search in files to find strings that match to a specific pattern. Inetutils is collection of common network programs. Finally, CPPI is used for indenting C preprocessor directives.

Project	SLOC	Number of Mutants	Version	Source
Git	153,663	700285	2.1.1	git://github.com/git/git.git
Diffutils	70,642	80272	3.5	git://git.sv.gnu.org/diffutils.git
Grep	67,874	65455	2.16	git://git.sv.gnu.org/grep.git
Inetutils	103,726	203961	1.9.4	git://git.sv.gnu.org/inetutils.git
cppi	6,690	13997	1.18	git://git.sv.gnu.org/cppi

Table 4.1: Benchmark

This section answers the following questions:

- RQ1: How common is it for a test to cover a mutant without infecting its execution state?

- RQ2: How common is it for a test to infect the execution state locally without infection propagating to the enclosing expressions?
- RQ3: How much redundancy exists due to the tests that lead to the same execution state on the same mutants?
- RQ4: What is the impact of static pruning on the infection, propagation, and partitioning optimizations?
- RQ5: How efficient and scalable are the optimizations?
- RQ6: How might these optimizations affect the efficiency of the mutation analysis?
- RQ7: Which mutation operators are more likely to generate test-equivalent mutants?

4.1 Effectiveness

In this section, we evaluate the effectiveness of each optimization, answering RQ1, RQ2, and RQ3. To measure their effectiveness, we calculate the number of mutants that survive for each optimization. To be exact, for infection, the ratio of number of mutants achieving state infection to the number of covered mutants is calculated. For propagation, this number is the ratio of the number of mutants propagating a state infection to the number of mutants achieving state infection. Finally, for partitioning, the ratio of number of partitions to the number of mutants propagating a state infection is computed. We will refer to these ratios as survival rates.

Table 4.2 reports the survival rates per program for infection, propagation and partitioning optimizations. As indicated in the first column, the infection optimization is able to identify 22-26% of mutants as test equivalent. These mutants can be excluded from execution in mutation analysis. The average survival rate is 76%, while for Java this number is reported as 79%.

Additionally, the results state that among the mutations that have infected execution state, 31-36% of them are test-equivalent. In comparison, the average survival rate for C is twenty percent lower.

Lastly, the partitioning optimization is able to prune 46-54% of the mutants that have propagated execution state. The average survival rate for C is 50% whereas the survival rate for Java is 86%. This implies the existence of more redundancy due to the surviving mutations that have the same execution state.

Based on the results, the observable trend is that all three of the optimizations are able to reduce the number of mutants that need to be executed in mutation analysis. Thus, these optimizations developed and previously evaluated for Java are applicable to C programs as well. In addition the results suggest that, in comparison to Java, while infection shows the same behavior, propagation and partitioning could be more effective in C.

Program	Infection	Propagation	Partitioning
cppi	0.74	0.64	0.50
diff	0.77	0.63	0.46
inet	0.77	0.69	0.54
grep	0.78	0.67	0.53
git	0.75	0.66	0.49
overall	0.76	0.65	0.50

Table 4.2: Survival rate for each optimization

4.2 Static Pruning Impact

In this section, we explore the impact of static pruning to answer RQ4. To evaluate this impact, the survival rates for each optimization are calculated in the presence of static pruning. To compare with the previous case, without static pruning, the percentage changes are calculated for each optimization per project.

Table 4.3 summarizes the survival rates per optimization after applying static pruning. Table 4.4 shows the corresponding percentage change for each value. It can be seen that the survival rates for all optimizations have been increased. The average increase is 5.51, 7.29, and 11.90 for infection, propagation, and partitioning respectively.

Static pruning pursues two goals. First, it tries to avoid applying mutations that generate equivalent mutants. Second, it tries to avoid generating mutants that are duplication of other mutants. Since partitioning have been impacted most, it could mean that static pruning is more successful at avoiding generating redundant mutants.

To sum up, static pruning reduces the impact of infection, propagation, and partitioning optimizations. While the survival rate for all of them have been increased, the partitioning has been impacted most and infection has been impacted least.

Program	Infection	Propagation	Partitioning
cppi	0.78	0.69	0.57
diff	0.81	0.68	0.52
inet	0.81	0.73	0.59
grep	0.84	0.73	0.60
git	0.78	0.70	0.54
overall	0.80	0.70	0.56

Table 4.3: Survival rate for each optimization with using static pruning

Program	Infection Rate	Propagation Rate	Partitioning Rate
cppi	5.41%	7.81%	14.00%
diff	5.19%	7.94%	13.04%
inet	5.19%	5.80%	9.26%
grep	7.69%	8.96%	13.21%
git	4.00%	6.06%	10.20%
overall	5.51%	7.29%	11.90%

Table 4.4: Percentage change in survival rate for infection, propagation, and partitioning after using static pruning (%)

4.3 Efficiency and Scalability

To investigate the efficiency and scalability of the optimizations, answering RQ5 and RQ6, we report the processing time to prune the mutants and also the estimated time reduction that we achieve through applying the optimizations. The latter minus the former is the net benefit of the system.

In order to analyze cost reduction in mutation analysis, we define a projection function which estimates the the execution time for mutation analysis. Let M_i be a set of mutants that are to be executed for test t_i in the mutation analysis after performing the optimizations. The execution time of running tests $t_1, t_2, ...t_N$ against their corresponding mutants can be estimated as:

$$\text{Estimated Time} = \sum_{n=1}^{N} T(t_n) * |M_n| \tag{4.1}$$

where $T(t_i)$ indicates the run-time of test t_i on schemata of the program and $|M_i|$ represents the number of mutants in the set M_i. This number sets an upper bound for the number of test executions because it assumes that all mutations are going to be executed, but as we discussed in chapter 2, some of the mutations can be skipped once they are detected in previous tests. So, the analysis could be done faster. The other limitation with this approach is that $T(t_i)$ is just an estimation for running time of test t_i in mutation analysis and it could be different from the actual execution time. The reason is that applying the mutation might change the behaviour of the program resulting in a different execution time for the tests.

Table 4.5 summarizes the estimated execution time for mutation analysis per optimization in seconds. The first columns indicates the estimated execution time after applying coverage optimization which is considered as our baseline. To make the comparison easier, table 4.6 has been created which shows the ratio of estimated mutation analysis time after applying each optimization over coverage optimization. The data reveals that infection optimization reduces the mutation analysis time by 23.6% on average. The reduction gained by applying propagation optimization is 10.5%. Finally, the partitioning is able to reduce the mutation analysis time by 15% on average. In total, the three optimizations reduce the mutation analysis time by 50% over using the coverage optimization alone.

Program	Coverage	Infection	Propagation	Partitioning
cppi	32,350.45	23,967.58	20,562.66	16,317.69
diff	3,386,521.94	2,604,932.10	2,118,677.33	1,564,561.91
inet	265,434.29	205,247.25	181,817.12	138,388.86
grep	1,474,521.44	1,161,521.28	991,683.70	784,519.14
git	12,979,270.04	9,790,214.11	8,648,067.54	6,372,579.83

Table 4.5: Estimated run-time of mutation analysis for each optimization (seconds)

Program	Infection	Propagation	Partitioning
cppi	0.74	0.64	0.50
diff	0.77	0.63	0.46
inet	0.77	0.68	0.52
grep	0.79	0.67	0.53
git	0.75	0.67	0.49
overall	0.76	0.65	0.50

Table 4.6: Ratio of estimated run-time compared to using only coverage

To investigate the optimizations' overhead, the processing time for each optimization is measured and reported in Table 4.7. The data indicate that the infection and partitioning has the least and most overhead respectively. The reason that partitioning requires more processing time is that the analysis is done using a Python script which needs to process a long text stream. The length of the stream and also low performance of Python, as an interpreted language, are the reasons that lead to high overhead for partitioning.

Program	Uninstrumented	Coverage	Infection	Propagation	Partitioning
cppi	7.35	8.12	8.16	8.93	12.03
diff	1.81	41.43	57.02	201.23	892.93
inet	12.26	20.73	21.76	27.85	79.57
grep	3.36	25.35	31.90	69.61	323.86
git	8.60	64.70	91.96	155.64	659.10

Table 4.7: Processing time for each optimization (seconds)

In conclusion, by comparing the the amount in cost reduction and optimizations' overhead, it is clear that the benefit of using infection, propagation, and partitioning is always positive.

4.4 Equivalency Likelihood of Operators

To determine which mutation operators are more likely to produce test equivalent mutants, answering RQ7, for each operator, the ratio of the number of generated test-equivalent mutants to

the total number number of generated mutations is measured. This ratio indicates the probability of generating a test-equivalent mutant for each operator.

Table 4.8 reports the probabilities of generating equivalent mutants per program and per operator. The results reveal that absolute value insertion is very likely to produce a test-equivalent mutant with probability of 85%. The next operators with highest probability are integer constant replacement, bitwise operator mutation, and logical connector replacement. The results indicate that the compound assignment mutation operator is least likely to produce a test-equivalent mutant.

The high probability of generating test-equivalent mutants for absolute value insertion is due to the fact that a large number of of expressions hold the same sign during the test executions. Therefore, among absolute value insertion and negative absolute value insertion, one of them would be equivalent to the original expression resulting in a test-equivalent mutant. The high probability of generating test-equivalent mutants for integer constant replacement operator can be explained using a similar argument. In this case, the replacement results in the same original value.

As mentioned in the previous chapter, static pruning can be used to filter some of these mutations. Table 4.9 reports the probabilities after static pruning of mutations. Although a considerable reduction in the probability of producing equivalent mutants can be seen for integer constant replacement, the reduction for absolute value insertion and comparison mutation is trivial. The reason for low reduction in absolute value is that the target expressions, unlike constant integer replacement, can not always be evaluated statically which means the mutations can not be pruned. Regarding comparison operator, the low frequency of cases that this pruning is applicable could be the reason for the unobservable impact.

In conclusion, absolute value insertion, bitwise mutation, and constant integer replacement are the operators that are more likely to produce test-equivalent mutants. On the other hand, compound assignment mutation operator seems to produce very few test-equivalent mutants.

Operator	Git	Diffutil	Grep	Inetuils	CPPI	Overall
Comparison	0.44	0.35	0.34	0.36	0.36	0.37
Assignment	0.03	0.03	0.03	0.01	0.08	0.03
Conditional	0.27	0.22	0.23	0.23	0.27	0.24
Bitwise	NA	0.71	0.57	0.65	0.42	0.58
AbsVal	0.89	0.84	0.85	0.84	0.86	0.85
Logical	0.62	0.54	0.56	0.55	0.57	0.56
Shift	NA	0.45	0.18	0.58	0.68	0.47
Integer	0.55	0.68	0.53	0.63	0.57	0.59
Arithmetic	0.56	0.32	0.19	0.40	0.33	0.36

Table 4.8: Probability of generating test-equivalent mutants for each operator

Operator	Git	Diffutil	Grep	Inetuils	CPPI	Overall
Comparison	0.44	0.35	0.35	0.36	0.36	0.37
Assignment	0.03	0.03	0.03	0.01	0.08	0.03
Conditional	0.27	0.22	0.23	0.23	0.27	0.24
Bitwise	NA	0.71	0.57	0.65	0.42	0.58
AbsVal	0.88	0.82	0.83	0.79	0.84	0.83
Logical	0.62	0.55	0.56	0.55	0.58	0.57
Shift	NA	0.45	0.18	0.58	0.68	0.47
Integer	0.41	0.56	0.38	0.54	0.45	0.46
Arithmetic	0.56	0.32	0.19	0.40	0.33	0.36

Table 4.9: Probability of generating test-equivalent mutants for each operator after using static pruning

4.5 Threats to Validity

As in any empirical study, the reported results depend on the choice of the subject projects. To cope with this issue, several open source projects with different size and functionality have been

selected. However, although the results are consistent, they are not generalizable for all programs in C. To increase generalizability, further replication studies with a wide variety of subject programs are required.

Another threat comes from the specific implementation of the schemata. Calculation of estimated time for mutation analysis is based on running time of test suites on schemeta. Since the schemata could possibly be implemented in more efficient ways, the results may be affected by using other implementations of the schemata. Further work is need to investigate how different implementations of schemata affect the results.

Lastly, the difference in implementation method of partitioning algorithm could be also a threat to the validity. In our implementation, executing the the tests on the instrumented version of the program creates log files. These files are later used by a Python script to do the partitioning. Due to possible inefficiencies of this script, such an implementation may not perform best in terms of performance. Implementing the the partitioning algorithms with other languages is required to analyze its effect on the processing times.

Chapter 5

Conclusion and Future Work

It is believed that mutation analysis is too expensive in terms of computational cost [8]. This thesis studies the effectiveness of infection, propagation, and partitioning optimizations on mutation analysis of programs written in C. Furthermore, it examines the impact of static pruning on infection, propagation, and partitioning.

The results show that infection, propagation and optimization may be effective methods to reduce the number of executions needed for mutation analysis. The average reduction in the number of executions is 76%, 65%, and 50% for infection, propagation, and partitioning respectively. Furthermore, in comparison to Java, the results suggest that although infections showed the same effectiveness, propagation and partitioning may be more effective at pruning the mutants. We also showed that based on our estimation of time needed for mutation analyses, the benefit of using the optimizations is positive. Regarding static pruning, we observed that the impact of infection, propagation, and partitioning are reduced by around 5%, 7%, and 11% after using static pruning.

5.1 Future Work

To investigate the effect of the optimizations on mutation analysis time, we relied on an estimated running time for mutation analysis. Future work can include other methods that lead to more precise estimation or direct measurement of the time reduction.

Moreover, the implementation of the schemata could be changed to run the tests more efficiently. For instance, currently, in our implementation, all source files are instrumented, which

causes a large overhead in run-time. However, the overhead can be reduced by transforming only the file that contains the mutated expression.

In addition, the results show that propagation and partitioning lead to pruning more mutants in C. Future work can consider investigating the reasons for the difference between the effectiveness of the optimizations in Java and C. The difference could be application specific or could lie in generalizable programming differences that exist across programs due to programming styles.

Furthermore, the results can be improved by enhancing the scope of propagation optimizations. In our implementation, if a mutation reaches a top level expression, it is marked as surviving; however, even these mutations may not reach to an observable output. So, more advanced propagation algorithms can be used to identify these cases.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[2] James H Andrews, Lionel C Briand, and Yvan Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411. ACM, 2005.

[3] Robert N Charette. Why Software Fails [Software Failure]. *IEEE Spectrum*, 42(9):42–49, 2005.

[4] Mehrnoosh Ebrahimipour. Undefined Behaviour in Mutation Testing. Master's thesis, Simon Fraser University, 2016.

[5] Bernhard JM Grün, David Schuler, and Andreas Zeller. The Impact of Equivalent Mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE, 2009.

[6] Shamaila Hussain. Mutation Clustering. *Ms. Th., King's College London, Strand, London*, 2008.

[7] IEC ISO. IEEE, Systems and Software Engineering–Vocabulary. *ISO/IEC/IEEE 24765: 2010 (E)) Piscataway, NJ: IEEE Computer Society, Tech. Rep.*, 2010.

[8] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[9] René Just, Michael D Ernst, and Gordon Fraser. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.

[10] René Just, Gregory M Kapfhammer, and Franz Schweiggert. Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20. IEEE, 2012.

[11] Kim N King and A Jefferson Offutt. A Fortran Language System for Mutation-based Software Testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[12] Aditya P Mathur and W Eric Wong. An Empirical Comparison of Data Flow and Mutation-based Test Adequacy Criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[13] A Offutt. The Coupling Effect: Fact or Fiction. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 131–140. ACM, 1989.

[14] A Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[15] Jie Pan. Using Constraints to Detect Equivalent Mutants. Master's thesis, George Mason University Master's thesis, 1994.

[16] 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 *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 936–946. IEEE, 2015.

[17] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, (4):367–375, 1985.

[18] Roland H Untch. Mutation-based Software Testing Using Program Schemata. In *Proceedings of the 30th Annual Southeast Regional Conference*, pages 285–291. ACM, 1992.

[19] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[20] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An Empirical Study on the Scalability of Selective Mutation Testing. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 277–287. IEEE, 2014.