EXPLORING DESIGN DECISIONS FOR MUTATION TESTING

BY

FARAH HARIRI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

ABSTRACT

Software testing is by far the most popular technique used in industry for quality assurance. One key challenge of software testing is how to evaluate the quality of test suites in terms of their bug-finding capability. A test suite with a large number of tests, or that achieves a high statement or branch coverage, does not necessarily have a high bug-finding capability.

Mutation testing is widely used in research to evaluate the quality of test suites, and it is often considered the most powerful approach for this purpose. Mutation testing proceeds in two steps. The first step is mutant generation. A mutant is a modified version of the original program obtained by applying a mutation operator. A mutation operator is a program transformation that introduces a small syntactic change to the original program. The second step of mutation testing is to run the test suite and determine which mutants are killed, i.e., which mutants lead to tests having a different output when run on them compared against running on the original program. Mutation testing produces a measure of quality of the test suite called mutation score. The mutation score of a given test suite is the percentage of mutants killed by that test suite out of the total number of generated mutants.

In this dissertation, we explore three design decisions related to mutation testing and provide recommendations to researchers in those regards. First, we look into mutation operators. To provide insights about how to improve the test suites, mutation testing requires both high quality and diverse mutation operators that lead to different program behaviors. We propose the use of approximate transformations as mutation operators. Approximate transformations were introduced in the emerging area of approximate computing for changing program semantics to trade the accuracy of results for improved energy efficiency or performance. We compared three approximate transformations with a set of conventional mutation operators from the literature, on nine open-source Java subjects. The results showed that approximate transformations change program behavior differently from conventional mutation operators. Our analysis uncovered code patterns in which approximate mutants survived (i.e., were not killed) and showed the practical value of approximate transformations both for understanding code amenable to approximations and for discovering bad tests. We submitted 11 pull requests to fix bad tests. Seven have already been integrated by the developers.

Second, we explore the effect of compiler optimizations on mutation testing. Multiple mutation testing tools were developed that perform mutation at different levels. More recently mutation testing has been performed at the level of compiler intermediate representation

(IR), e.g., for the LLVM IR and Java bytecode/IR. Compiler optimizations are automatic program transformations applied at the IR level with the goal of improving a measure of program performance, while preserving program semantics. Applying mutations at the IR level means that mutation testing becomes more susceptible to the effects of compiler optimizations. We investigate a new perspective on mutation testing: evaluating how standard compiler optimizations affect the cost and results of mutation testing performed at the IR level. Our study targets LLVM, a popular compiler infrastructure that supports multiple source and target languages. Our evaluation on 16 Coreutils programs discovers several interesting relations between the numbers of mutants (including the numbers on equivalent and duplicated mutants) and mutation scores on unoptimized and optimized programs.

Third, we perform an empirical study to compare mutation testing at the source (SRC) and IR levels. Applying mutation at different levels offers different advantages and disadvantages, and the relation between mutants at the different levels is not clear. In our study, we compare mutation testing at the SRC and IR levels, specifically in the C programming language and the LLVM compiler IR. To make the comparison fair, we develop two mutation tools that implement conceptually the same operators at both levels. We also employ automated techniques to account for equivalent and duplicated mutants, and to determine hard-to-kill mutants. We carry out our study on 16 programs from the Coreutils library, using a total of 948 tests. Our results show interesting characteristics that can help researchers better understand the relationship between mutation testing at both levels. Overall, we find mutation testing to be better at the SRC level than at the IR level: the SRC level produces much fewer (non-equivalent) mutants and is thus less expensive, but the SRC level still generates a similar number of hard-to-kill mutants.

To Saadeddine and Maha,
for a lifetime of sacrifices and unwavering support...

ACKNOWLEDGMENTS

I would like to start by expressing my gratitude to my advisor *Darko Marinov*. Darko was the driving force behind every research activity I have undertaken during my PhD. His zeal and passion for what he does are exciting and it rubs off on everyone who works with him. Darko is an exceptional character and builds unforgettable memories with his students; from the jokes and the time we spend reading random Wikipedia pages, to the discussions about *billion dollar* ideas no less, and of course the nights in the office during paper submission deadlines. His role as an advisor extends to even showing up at your door past midnight to comfort you and help you deal with the aftermath of a stranger trying to break in to your apartment. I don't think a student can graduate from Darko's group without carrying part of Darko in their personality. I am forever tainted by his academic curiosity, passion for programming, logical approach (filled with tautologies) towards decision making, and preference of Emacs over Vim.

The content of this thesis has been shaped by the feedback from my committee members Sarfraz Khurshid, Sasa Misailovic, and Tao Xie. Tao's constructive criticism and engaging discussions were especially helpful. I am also lucky to have collaborated with Sasa and Sarfraz on multiple occasions.

During my five years of PhD, I have been partially supported by the US National Science Foundation under Grants Nos. CCF-1012759, CCF-1319688, CCF-1409423, CCF-1421503, CCF-1434590, CCF-1438982, CCF-1439957, CCF-1566363, CCF-1566589, CCF-1629431, CCF-1652517, CCF-1703637, CCF-1704790, CNS-0958199, CNS-1239498, and the DARPA grant FA8750-12-C-0284. I was also supported by the Saburo Muroga Endowed Fellowship during my first year.

Research is only fun and interesting when it is done in group with collaborators. Thank you to Babak Behzad, Hayes Converse, Hoang-Vu Dang, Lamyaa Eloussi, Vimuth Fernando, Milos Gligoric, Alex Gyori, Milica Hadži-Tanović, Sarfraz Khurshid, Owolabi Legunsen, Yu Lin, Yafeng Lu, Qingzhou Luo, Muhammad Suleman Mahmood, Sasa Misailovic, Karl Palmskog, August Shi, Marc Snir, Lingming Zhang, and Weizhe Zhang. I have learned a lot from working with them. Thank you to my internship mentors Petr Hosek and Chuan Sun. They have made my time at Google fun and rewarding.

Alex, August, Lamyaa, and Owolabi are more than collaborators; they are also my lab mates. We joined Darko's group at the same time and hopped on the journey of grad school together. August has been a wonderful office mate, I appreciate his kindness, big heart, and clandestine sense of humor. I especially like how he consistently succeeds at hitting the absolute minimum in the number of syllables needed to answer a question. I am thankful for all the times I spent with everyone doing research, having late dinners on deadline nights, our trip to Santa Cruz, and more.

The staff at the computer science department are remarkable. The lovely smiles of Kathy Runck, Maggie Metzger, Elaine Wilson, Kara McGregor, and Viveka Perera Kudaligama used to light my day! They are the knights behind the scenes who make sure everything is smooth and flowing.

Thank you to my undergraduate advisor Fadi Zaraket for being an amazing mentor and support. He introduced me to research and opened for me opportunities to pursue graduate studies.

Being far from family during these five years of PhD has been made lighter by the exceptionally warm Muslim community of Urbana-Champaign. The list of names is way too long to include here, but I would like to send a personal thank you to every member of the community, especially the families. The beautiful memories that I have shared with them made the corn fields feel like a home away from home.

I am especially thankful to the Taha family. Thinking about Dr. Ahmed Taha and Khadiga Abdel Wahab's love and support through thick and thin brings tears to my eyes. Wherever I go, I will always carry them in my heart. Some of the nicest memories are ones I had with their children; events that we all organized together, sleep overs with Aliaa and Gehad, and traveling on road trips just to name a few. I will never forget Gehad showing up with soup on a cold night, carrying me to my apartment after having my wisdom teeth removed, and sharing many first time experiences together. A special thanks filled with love to Aliaa for our deep conversations and for her help and support on multiple occasions, especially organizing my wedding dinner in Urbana.

Thank you to *ma homie* Mariam Saadah. I met Mariam in my first year of PhD, and ever since we developed bonds that only grew stronger with time. Our late night calls, extended road trip from California to Illinois, random sleep overs, and consistent weekly routines for achieving personal growth goals are only a few of the many memories we painted throughout the years. You know someone is a good friend when she makes sure you never run out of bananas while you are working in your office. My gratitude extends to the entire Saadah family; Dr. Abdul Karim, Dr. Eman, Malaak, Raneem, and Baraa.

Thank you to Hanan Jaber for five years of incomparable friendship. You have been my teacher, my role model, and my close friend. The time we spend, even if so little, always counts a lot. My gratitude extends to the entire Jaber family; Dr. Hazem, Eman, Haneen, Mona, Aya, and Ayaat.

The spring of my fourth year of PhD was special, thanks to Bushra Hamad. We laughed and we cried, we wrestled and we comforted each other, we just connected on a unique level. I am particularly a fan of the one night we spent working on our definition of the *Weltanschauung*.

Lots of love and gratitude to Lubna Boozeyah, Ghada Hassaan, Faten Mahayri, Mona Saleh, Humna Shahid, Faria Kalim, Safa Messaoud, Huda Ibeid, and Betül Ozkaldi . You entered joy to my heart and supported me even without me asking on multiple occasions. I am thankful to Izzat El Hajj, Ihab Nahlous, Marc Ghossoub, Laurence Rustom, Hussein Sibai, and all the students in the Lebanese group of Urbana-Champaign for our times together.

Keeping in shape is important, especially with a career of sitting behind the screen. I enjoyed playing basketball with Laila Bardan. Laila brings back the child in me (sometimes embarrassingly). I love our late night masjid meetings around coding exercises. Thank you to Dr. Hany Youssef for the Taekwondo classes, they taught me discipline.

I am grateful for having my sisters Faten and Aya in my life. They are the best friends that I will never lose. I am also blessed to have met Yasser Shalabi during my time at UIUC. The progression from being my next door office neighbor at Siebel to my fiance and now husband was a colorful experience. I am grateful to Yasser for his big big love (the repetition is not a typo). He is an amazing partner in every aspect. Our discussions on various technical and philosophical topics are precious.

Saving best for last; words and sentences fail when it comes to my parents. There is no good enough tool to express my deepest gratitude to my mom Maha, and my dad Saadeddine. After the grace of God, all the thanks goes to them for what I have become today. Their love and involvement in my life are unmatchable. To mama: you are my hero with super human powers. From sacrifices that left physical scars, to prayers you send my way, you are always watching over me. Your impact on my life is something that I keep discovering and will never be able to contain or draw boudaries around. My dad is the one who implanted in me the first seed of loving knowledge. If it wasn't for him, I would have never thought of pursuing a PhD. He is the one who set me on a quest for knowledge, and I hope to continue on that path even after PhD. I hope one day I will be as good of a teacher and mentor as he has been for his students and as good of an asset to my country as he has been.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Software's role in our lives is becoming increasingly important. Automation is entering into almost every facet of everyday life, e.g. self-driving cars, transportation, trading, cryptocurrency, online shopping, personal assistants, home security, medical devices, and smart things, just to list a few. Consequently, ensuring software quality is critical to safeguard what software has power over, including our privacy, wellbeing, and even physical safety.

Software testing is by far the most popular technique used in industry for quality assurance. Tests are either written manually by developers or generated by test generation tools. Conceptually, testing is simple: the code under test is run against a test suite that consists of multiple tests; each test provides some input and checks the actual output against the expected output. However, testing has several challenges in practice. One key challenge is to evaluate the quality of test suites. A test suite that has a large number of tests, or achieves a high statement or branch coverage, does not necessarily have a high bug-finding capability. A test suite can achieve a high coverage without exercising some particular bug-finding paths in the code under test, and even if the test suite exercises the buggy paths of interest, it may still not find the bugs if the test oracle does not capture the intended behavior. Therefore, we need a better way to measure the quality of the test suite, and this is where mutation testing comes into play.

Mutation testing is widely used in research to evaluate the quality of test suites, and it is often considered the most powerful approach for this purpose [16, 56]. The intuition behind mutation testing is the following: we assess the quality of test suites by their capability to find seeded changes. Because we do not have access to the possible bugs a priori, mutation testing introduces small syntactic changes to the code under test and relies on the idea that a test suite's ability to kill these small changes is a proxy of its ability to detect actual bugs in the real world.

Mutation testing proceeds in two phases. First, a number of mutants are generated by applying *mutation operators*, which are program transformations that introduce small syntactic changes, to the original code under test. Second, the test suite is run against the mutants to determine which mutants are *killed*, i.e., which mutants lead to tests having a different output when run on them compared against running on the original program. (More precisely, we are interested in *strong mutant killing*, based on the observable output, rather than weak mutant killing, based on the intermediate state of a test's execution [16].) Finally, the *mutation score* is computed as the ratio of the number of killed mutants to the number of all generated mutants. Mutation operators aim to produce semantically non-equivalent programs. Some mutants cannot be killed by any test; these mutants are semantically *equivalent* to (albeit being syntactically different from) the original code under test. *Duplicated mutants* [88] are defined as definitely equivalent to one another but not definitely equivalent to the original code. Ideally, one would want to avoid the problem of equivalent and duplicated mutants, and perform mutation testing only on non-equivalent and non-duplicated (NEND) mutants. Determining which mutants are equivalent/duplicated is undecidable in general [16, 22].

In this dissertation, we look into three design decisions related to mutation testing and provide recommendations to researchers in those regards. First, we look into mutation operators. To provide insights about how to improve the test suites, mutation testing requires both high quality and diverse mutation operators that lead to different program behaviors. We propose the use of approximate transformations as mutation operators. Approximate transformations were introduced in the emerging area of *approximate computing* for changing program semantics to trade the accuracy of results for improved energy efficiency or performance. Researchers proposed various approximate transformations at the level of programming languages, compilers, and computer systems [32, 38, 44, 45, 76, 77, 92, 93, 95, 104, 109, 115]. For example, loop perforation [76,104] is a compiler-level approximate transformation that causes amenable loops to execute only a subset of iterations. Degrading floating-point precision is another common language-level [93] and system-level approximate transformation [95, 115]. In our research, we establish the novelty of the proposed operators based on approximate transformations and provide, based on our study, valuable findings related to the practice of mutation testing.

Second, we study the effect of compiler optimizations on performing mutation testing. Multiple mutation tools were developed that perform mutation at different levels, including traditional mutation testing at the level of source code (*SRC*), e.g., for the `C` language [17,18, 29,30,41,54,55] and `Java` [57,70] and more recently mutation testing at the level of compiler intermediate representation (*IR*), e.g., for the LLVM IR [98,99,107], and Java bytecode/IR [6, 96]. Compiler optimizations are automatic program transformations applied at the IR level with the goal of improving a measure of program performance, while preserving program semantics. Applying mutations at the IR level means that mutation testing becomes more susceptible to the effects of compiler optimizations. We investigate whether mutation testing should be applied with or without compiler optimizations, and provide insights about the interplay between mutations and compiler optimizations.

Third, we perform an empirical study to compare mutation testing at the SRC and IR levels. Applying mutation at different levels offers different advantages and disadvantages, and the relation between mutants at the different levels is not clear. We inform the practice about how mutants of the SRC and IR level relate to each other, and which of the two levels is more cost effective to perform mutation testing at.

1.1 THESIS STATEMENT AND CONTRIBUTIONS

Our thesis is that exploring new design decisions in mutation testing can lead to making more informed and efficient choices about mutation testing. This dissertation makes three contributions. First, we find approximate transformations to be effective mutation operators. Second, we recommend enabling compiler optimizations while performing mutation testing as it is more cost beneficial. Third, we recommend performing mutation testing at the SRC level as opposed to at the IR level for efficiency. In the following sections of this chapter, we elaborate on these three contributions.

1.2 APPROXIMATE TRANSFORMATIONS AS MUTATION OPERATORS

Mutation testing is a well-established approach for evaluating the quality of a test suite [56]. It produces modified versions of the code, called mutants, using a set of syntactic transformations, called mutation operators, that aim to change program semantics. It then runs the test suite on the mutants to quantify how well the suite detects the program modifications. To provide insights about how to improve the test suites, mutation testing requires both high-quality and diverse mutation operators that lead to different program behaviors.

We propose approximate transformations as a new class of mutation operators that lead to different program behaviors from those produced by conventional mutation operators. Mutation operators and approximate transformations both aim to change program semantics. Hence, approximate transformations are an attractive choice for new mutation operators that can provide novel insights about tested code and test suites. Our analysis of three approximate transformations—loop perforation, integer-to-short precision degradation, and double-to-float precision degradation—shows that they often complement conventional mutation operators.

We perform our evaluation on nine open-source Java subjects. To evaluate the effectiveness of approximate transformations as mutation operators, we use a combination of techniques established in prior work on mutation testing:

Mutation Score: We compare mutation scores [16, 56] of three approximate transformations with the mutation scores of 14 conventional mutation operators from the popular PIT mutation testing framework for Java [6]. Mutation scores are percentages of mutants detected, or *killed*, by the test suite [56].

Minimal Mutants: We check whether approximate transformations generate mutants that are in the minimal mutants set, computed across mutants from both approximate transformations and conventional mutation operators. Minimal mutants dynamically subsume all other mutants, and they are considered harder to kill than all other mutants [15, 41].

Sufficient Mutation Operators: We check whether approximate transformations are in the set of sufficient mutation operators, computed using *selective mutation analysis* [83, 86]. Tests that kill mutants generated by sufficient mutation operators also kill mutants generated by all other operators.

The results show that approximate transformations are effective mutation operators. Loop perforation has similar mutation scores as conventional mutation operators. However, we observe that mutation score alone is not sufficient for evaluating the effectiveness of approximate transformations. While precision degradation operators have significantly lower mutation scores, our analysis shows that their low mutation scores are not due to the mutants being semantically equivalent to the original code. Rather, the operators expose bad tests that do not exercise the code with boundary values. Approximate transformations also generate mutants in the minimal mutant sets.

To better understand the pattern of computations affected by approximate transformations, we manually inspected a sample of mutants from the approximate transformations. For loop perforation, we identify four code patterns, e.g., reductions and conditional computation on elements. For precision degradation, we identify three code patterns, e.g., when computed results are within a specified error bound. Further, the identified patterns allow us to draw more general comparisons with a broader set of mutation operators from recent literature [57, 69, 96]. Our analysis shows that approximate transformations complement the conventional mutation operators.

Based on our inspection, we propose a new way of reasoning about surviving (i.e., not killed) mutants generated by approximate transformations. In traditional mutation testing, a mutant can survive either because it is semantically equivalent to the original code, or because of bad (buggy, inadequate, or missing) tests. We discover that, with approximate transformations, there is a third option—*a surviving mutant can indicate the presence of approximable code.* Code is approximable if it can be transformed to produce results different from the original code, but such results still meet the specification. (Note that this third way of interpreting a surviving mutant may apply to other mutants as well.) Our inspection shows that for loop perforation, 63.83% of surviving mutants indicate bad tests, and 19.15% indicate approximable code. We find no equivalent mutants, but the remaining 17.02% are hard to inspect. For precision degradation, 53.13% of surviving mutants indicate bad tests, 14.58% indicate equivalent mutants, and 11.46% indicate approximable code. The remaining 20.83% are hard to inspect.

We identify common testing practices that can help improving bad tests: (i) achieving greater loop coverage, (ii) exercising loop conditions, (iii) exercising boundary values, and (iv) checking correctness of all output elements. We identify the instances of bad tests in all nine subjects. Even though these insights are not new to the testing community, the real value lies in the fact that the approximate transformations help detect those problems and bring them to the attention of the developer who might not have such considerations in mind. We created 11 pull requests to improve the bad tests. The developers already integrated seven of our pull requests in their code.

The contributions of our study are:

- **Concept:** We are the first to study the interplay between approximate transformations and mutation testing operators.
- **Framework:** We developed ApproxiMate, an extension to the PIT framework. It supports approximate transformations as mutation operators and integrates analyses from studies on mutation testing.
- **Evaluation:** Our results show that approximate transformations complement conventional mutation operators: they generate mutants in the minimal mutants set and are often in the sufficient mutation operators set.
- **Insights:** We present code patterns revealed by approximate transformations. We discuss how to interpret the results of mutation testing with approximate transformations and how to improve bad tests. Developers already accepted seven out of 11 pull requests that we submitted for fixing bad tests.

1.3 EFFECTS OF COMPILER OPTIMIZATIONS

As previously explained, mutation testing proceeds in two steps; the first step is mutant generation, and the second step is running the test suite for each generated mutant to determine which mutants are killed and to compute the mutation score. While the number of killed mutants depends on the test suite, the number of generated mutants depends on the mutation operators and the level at which the operators are applied. Mutation operators have been proposed for many programming languages, including Ada, C, Cobol, C#, Fortran, Java, and SQL [16]. Mutation testing was also recently applied at the level of compiler intermediate representation (IR) [98, 99, 107] using LLVM [63, 79]. One IR usually supports multiple source and target languages. For example, LLVM is a widely used compiler infrastructure that supports multiple source languages (including C/C++ via the *Clang* front-end [78]) and multiple target languages (X86-32, X86-64, and ARM). When mutation testing is implemented once at the IR level, it enables mutation testing effectively "for free" for all source languages supported by the IR, without having to implement a special tool for every one of them. However, applying mutations at the IR level means that mutation testing becomes more susceptible to the effects of compiler optimizations.

Compiler optimizations have to preserve the behavior of the code and produce semantically equivalent programs; in contrast, mutation operators aim to produce semantically non-equivalent programs. Some mutants cannot be killed by any test; these mutants are semantically equivalent to the original code under test. Researchers have proposed several heuristics [9, 20, 43, 48, 50, 82, 84, 85, 96, 97] that help in determining which mutants are more *likely* equivalent or non-equivalent to the original code. Most recently, Papadakis et al. [88] proposed a technique for finding mutants that are *definitely* equivalent to the original code by comparing the compiled versions of the original code and its mutants; their experiments applied the mutation operators on the source code, specifically in the C programming language. They also use the same technique to determine what they call *duplicated mutants* that are definitely equivalent to one another but not definitely equivalent to the original code.

We present an empirical study of the effects of compiler optimizations on mutation testing at the compiler IR level. Our study aims to investigate whether mutation testing should be applied with or without compiler optimizations, providing the user with insights about the interplay between mutations and compiler optimizations. We evaluate several traditional classes of mutation operators (proposed by Offutt et al. [83] for selective mutation). Specifically, we implemented for LLVM the following four classes of mutation operators:

- AOR replaces every arithmetic operator with another arithmetic operator;
- LCR replaces every logical connector with another logical connector;
- ROR replaces every relational operator with another relational operator;
- ICR replaces every integer constant c with a different value from the set $\{-1, 0, 1, c-1, c+1\}$.

A similar set of mutation operators is often used in mutation tools for the C language, e.g., by Andrews et al. [17] or Jia et al. [55]. For our evaluation, we used programs from Coreutils [36]. Coreutils are the basic command-line utilities used in Unix, e.g., `mkdir`, `mv`, or `rm`. Coreutils are frequently used as experimental objects in studies involving C programs or LLVM [24, 34, 62, 72].

We compiled Coreutils using two opposite optimization levels: `-O0` is the basic level that aims at fast compilation and only applies minimal optimizations, and `-O3` is one of the most aggressive optimization levels that applies advanced compiler optimizations. We also mutated each program using each of our mutation operators, applying the compiler optimizations *both* before and after mutation. We identified equivalent and duplicated mutants by comparing the resulting binaries as done by Papadakis et al. [88]. We determined the mutation score by running the mutants against the companion test suites for their programs. For evaluation, we compared the resulting number of mutants and the mutation scores of the test suites across the two levels of compiler optimizations.

Our findings can be summarized as follows:

- The total number of generated mutants is higher (11.7% overall) at the `-O3` level than at the `-O0` level. This is surprising because the overall number of instructions is lower at the `-O3` level than at the `-O0` level.

- The percentage of equivalent and duplicated mutants is higher (15.9pp[1] on average) at the `-O3` level than at the `-O0` level. This is expected and matches prior work [88], because `-O3` applies more optimizations after applying the mutation operators. Surprisingly, after removing equivalent and duplicated mutants, the number of remaining (non-equivalent, non-duplicated) mutants is *lower* at the `-O3` level than at the `-O0` level. This points to the importance of properly controlling for equivalent and duplicated mutants, especially at high optimization levels.

- The mutation score is persistently lower at the `-O3` level than at the `-O0` level, even when removing equivalent and duplicated mutants. This points to the importance of properly using the mutation score to interpret the results of mutation testing, especially at high optimization levels.

- The results are fairly similar across different mutation operators, indicating that the general conclusions are more likely due to the compiler optimizations than due to the specific operators.

In brief, our study shows that, if one intends to apply mutation testing at the LLVM IR level, it is advisable to use a very high optimization level, but it is necessary to properly control for equivalent and duplicated mutants and to carefully interpret the overall mutation score.

[1]The unit *pp*, from "percentage point", represents the difference of values that are already expressed in percentages.

1.4 COMPARING THE SRC AND IR LEVELS

Mutation testing has traditionally been applied at the SRC level and more recently at the IR level. Applying mutation at different levels offers different advantages and disadvantages. First, the mutation testing time is dominated by the time to run the test suite on each mutant. Depending on the number of generated mutants, a given level can be more efficient than the other. Second, the quality of the mutation score is related to the quality of the generated mutants; if a level generates many easy to kill mutants, the mutation score could be misleadingly high. Third, while mutation testing is most commonly used to evaluate test suites through mutation scores, sometimes one would reason about individual mutants when augmenting test suites for higher mutation scores. At the SRC level, the mutants are changes in the source code, making it easier to reason which (if any) tests could kill the mutant [116]. However, the mutants at the IR level are much harder to reason about; it is hard to map the mutations from one level to another, so understanding the relation between mutations at the two levels and determining which is better to use requires an empirical study. Lastly, mutating at the SRC level requires implementing a mutation tool for each programming language. In contrast, the IR level enables implementing one tool that can apply to multiple source languages that compile to the same compiler IR. For example, as already mentioned in Section 1.3, the LLVM IR [79] is a popular IR that supports multiple languages, including `C`, `C++`, `Objective-C`, `Objective-C++`, `OpenMP`, `OpenCL`, and `CUDA`.

We present an extensive comparison of mutation testing at the SRC and IR levels for the `C` language and the LLVM compiler IR. To make the comparison fair, we use two mutation tools that implement conceptually the same operators at both levels, specifically the operators listed in Section 1.3.

Although we make a best effort to use the same operators at both SRC and IR levels, mutant generation can still lead to different mutants at the two levels. This difference is inherent in the nature of the two levels, SRC and IR, and in their interplay with compiler optimizations: SRC applies mutations *before* optimizations, and IR applies mutations *after* optimizations. For example, consider the simple addition statement `z = x + y;`. At the SRC level, mutant generation using AOR would replace '`+`' with other arithmetic operators. In contrast, at the IR level, the compiler may find that `x` and `y` have constant values, and the optimizations (constant propagation and constant folding) would eliminate the add operation, so no AOR mutation would apply. On the other hand, if this statement was in a loop, then loop unrolling could make two copies of this statement, so mutating any one of the two copies at the IR level would not generate the same mutant as at the SRC level (that effectively mutates both copies at once). Our goal is *not* to generate exactly the same set of mutants at both levels; while in principle one could engineer tools to do exactly that, there would be nothing to compare among the resulting set of mutants. Our goal is to compare the two levels with conceptually the same mutation operators applied in a natural manner at each level.

We implement both tools in LLVM, a language-agnostic compiler infrastructure. We also employ automated techniques to account for equivalent and duplicated mutants, and to determine hard-to-kill mutants. We statically determine which mutants must be equivalent and duplicated using the trivial compiler equivalence (TCE) technique proposed by Papadakis et al. [88]. We dynamically determine which mutants may be equivalent using a relatively large test pool for each program. We also determine which mutants are hard-to-kill using both minimal mutant sets [15] and surface mutant sets [40].

We perform our study on 16 Coreutils programs. Coreutils is a library of command-line Unix utilities written in `C`. The source distribution includes a regression test suite for many of those programs. Most testing is done through Bash scripts, each of which runs multiple tests (that invoke the utility binary). For this study, we break those scripts into smaller scripts that have only individual tests, which allows us to study the effect of various test suites on mutation testing at the SRC and IR levels. In total, we carry out our study on 948 tests.

The summary of our findings is as follows. First, the *total* number of generated mutants and the number of non-equivalent and non-duplicated mutants is *lower* at the SRC level than at the IR level. Second, removing equivalent and duplicated mutants, using techniques such as TCE [88], is essential to ensure faster mutation testing and to avoid skewing mutation results. Third, the mutation scores at the SRC and IR levels are *highly correlated* and can be used as good proxies of each other. Fourth, the summary conclusions comparing SRC and IR for all operators combined apply to individual mutation operators. Fifth, the SRC and IR levels generate a similar number of hard-to-kill mutants.

Overall we find that mutation testing at the SRC and IR levels is more similar than dissimilar, but with the SRC level proving to be often better than the IR level. As a result, we recommend that researchers apply mutation testing at the SRC level rather than at the IR level, although the SRC level requires implementing a mutation tool for each source programming language. In retrospect, our effort on building a tool at the IR level did not produce a tool that we recommend others to use, but it did allow us to compare the two levels.

1.5 DISSERTATION ORGANIZATION

The rest of the dissertation is organized as follows:

- **Chapter 2** presents our study on using approximate transformations as mutation operators [47].
- **Chapter 3** presents the details of our empirical study of the effects of compiler optimizations on mutation testing at the compiler IR level [46].
- **Chapter 4** presents an extensive study comparing mutation testing at the SRC and IR levels for our Coreutils subjects.
- **Chapter 5** presents work in the literature on the different areas related to our research.
- **Chapter 6** summarizes the contributions of the dissertation and potential future work that can build on top of these contributions.

CHAPTER 2: APPROXIMATE TRANSFORMATIONS AS MUTATORS

In this chapter we present our study on using approximate transformations as mutation operators. Our evaluation on nine open-source Java subjects focuses on the following three research questions:

RQ2.1: How effective are approximate transformations as mutation operators, compared to conventional mutation operators?

RQ2.2: What code patterns do approximate transformations as mutation operators reveal?

RQ2.3: How can approximate transformations as mutation operators help software testing practice?

2.1 EXAMPLE

This section illustrates mutation testing and approximate transformations and shows a surviving approximate transformation mutant that resulted in an accepted pull request.

2.1.1 Code and Test

The snippet in Figure 2.1 is from `vectorz` [8] (SHA: `9c688f1`), one of the subjects in our study. The snippet shows the instance method `Matrix#swapRows`; the class `Matrix` represents $m \times n$ matrices of type `double`. `swapRows` takes integers `i` and `j`, and then it changes the `Matrix` instance by swapping rows `i` and `j`. A parametrized test that directly invokes `swapRows` is `doSwapTest`. It operates on instances of `AMatrix`, a superclass of `Matrix` (Figure 2.3). `doSwapTest` first makes a copy, `m2`, of the input `m` (when `m` is of type `Matrix`, so is `m2`), swaps the first two rows in `m2`, and asserts that `m` and `m2` are not equal. Then, it swaps the first two rows in `m2` again and asserts that `m2` is now equal to `m`.

2.1.2 Mutation Testing

Mutation testing proceeds in two steps; it first generates mutants and then runs the tests on each mutant. **Generating mutants.** Mutation testing generates *mutants*—code that differ from the original by small syntactic changes, specified by *mutation operators*, e.g., replacing multiplication with division as in Figure 2.2a (dark background).

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j * cols;
  int cc = columnCount();
  for (int k = 0; k < cc; k++) {
     int i1 = a + k;
     int i2 = b + k;
     double t = data[i1];
     data[i1] = data[i2];
     data[i2] = t;
  }
}
```

Figure 2.1: Code from `vectorz` [8]

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j / cols ;
  int cc = columnCount();
  for (int k = 0; k < cc; k++) {
     int i1 = a + k;
     int i2 = b + k;
     double t = data[i1];
     data[i1] = data[i2];
     data[i2] = t;
  }
}
```

(a) Killed mutant changes `*` to `/`.

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j * cols;
  int cc = columnCount();
  for (int k = 0; k < cc; k+=2 ) {
     int i1 = a + k;
     int i2 = b + k;
     double t = data[i1];
     data[i1] = data[i2];
     data[i2] = t;
  }
}
```

(b) Surviving LPM mutant skips iterations.

Figure 2.2: A mutation by a conventional mutation operator and a mutation by LPM of the code in Figure 2.1

Executing mutants. Mutation testing executes the test suite on each mutant. If a test exhibits different behavior when running on a mutant than when running on the original code, that mutant is considered *killed*. Typically, tests pass on the original code, so a mutant is killed when a test fails on the mutant. For instance, when `doSwapTest` is run on the mutant in Figure 2.2a, the mutant computes the index of the second row in the swap as `0` (instead of `cols`). The first row to swap is also `0`, so no swap happens. The non-equality assertion on `m2` and `m` fails when run on this mutant, suggesting that the test suite is good enough to kill this semantically different mutant.

Mutation score. Mutation testing produces a *mutation score*—the percentage of killed mutants. Higher mutation scores imply higher-quality test suites; a test suite that is strong enough to kill a larger percentage of mutants is likely strong enough to detect more faults

```
private void doSwapTest(AMatrix m) {
  if ((m.rowCount()<2)||(m.columnCount()<2)) return;
  m=m.clone();
  AMatrix m2=m.clone();
  m2.swapRows(0, 1);
  assert(!m2.equals(m));
  m2.swapRows(0, 1);
  assert(m2.equals(m));
  ...
}
```

Figure 2.3: Test of the `swapRows` method in Figure 2.1

in the code under test [17, 58].

2.1.3 Approximate Transformations

Loop perforation. Loop perforation is an approximate transformation [76,104], that transforms loops like `for (int i = 0; i < len; i++) {...}` to execute only a subset of its iterations. In general, perforation can change the value in the initialization expression, the termination condition, or the increment. We consider loop perforations that skip every other loop iteration. Figure 2.2b shows an LPM (Loop Perforation Mutator) mutant that changes the loop increment, `k++`, to `k+=2` (light background). With this perforation, `doSwapTest` executing on the mutant will only swap every other element (at even-numbered indices) in the specified rows.

Precision degradation. Precision degradation is an approximate transformation that changes the type of a numerical expression or a variable. Specifically, we downcast results of `int` or `double` arithmetic expressions.

The `int-to-short` (ITS) transformation changes result of the expression to be of type `short` (values in the range $-32,768$ to $32,767$). An example ITS mutant is replacing `a + k` on line 7 of Figure 2.1 with `(short)(a + k)`. ITS drops higher-order bits, which may result in a large error magnitude.

If '`a`' is instead a double-precision variable, the `double-to-float` (DTF) transformation changes the expression `a + k` (where the type of `k` is automatically cast to `double`) to `(double)((float)(a + k))`. The cast back to `double` here is necessary in Java to preserve the type. The resulting computation produces imprecise results, usually with a small error magnitude, because it drops lower mantissa bits. Note that our ITS and DTF transformations are finer-grained variants of the actual approximate transformations; we only cast computations as opposed to types as performed in some prior work [93].

2.1.4 Analysis of Approximate Transformation for Mutation

For the LPM mutant in Figure 2.2b, `doSwapTest` swaps only elements at even-numbered indices in the specified rows. Since the assertions only check that `m1` and `m2` are not equal after the first swap, and equal after the second swap, `doSwapTest` passes. Because `doSwapTest` is the only test that covers this mutant, the mutant *survives*, i.e., it is not killed. The survival of this LPM mutant suggests that there is some weakness in the test suite, i.e., some tests are "bad" (buggy, inadequate, or missing). Specifically, this surviving mutant indicates that the assertions are not strong enough to detect the skipping of every other element during the swap. We submitted a pull request to check whether elements in the swapped rows are as expected; our pull request was accepted by the `vectorz` developers.

For the ITS mutant on line 7, `doSwapTest` is invoked only with small integers (matrices with small dimensions), so the mutant survives. To kill the mutant, one would write a test with large matrices where the column count exceeds the range of `short`.

2.2 STUDY METHODOLOGY

ApproxiMate is our framework for evaluating approximate transformations as mutation operators. In this section, we describe ApproxiMate's implementation and analyses, the mutation operators studied, and our evaluation subjects.

2.2.1 The ApproxiMate Framework

The ApproxiMate framework extends PIT [6], implements approximate transformations as mutation operators, and can compute the full mapping from tests to killed mutants of tests to killed mutants, as has been done in previous studies [11, 100, 102]. We implement the approximate transformations as follows:

- We implement the loop perforation mutator (LPM) to skip every other iteration of loops, because other patterns of skipped iterations have similar power to identify approximable code [74]. We use SPOON [108] to find code locations of `for` loops that have increment (`i++`) or decrement (`i--`) statements. These locations are passed to our modified PIT extended with LPM, which uses the ASM library [19] to change the `iinc` bytecode instruction so that increments become `i+=2` and decrements become `i-=2`.

- We implement precision degradation, DTF and ITS, using casting. Recall (Section 2.1.3) that ITS is the `int-to-short` precision degradation operator; it casts results of `int` arith-

Table 2.1: PIT mutation operators

Type	Name	Acronym
Default	Conditionals Boundary Mutator	CBM
	Increments Mutator	IM
	Invert Negatives Mutator	INM
	Math Mutator	MM
	Negate Conditionals Mutator	NCM
	Return Values Mutator	RVM
	Void Method Calls Mutator	VMCM
Non-Default	Constructor Calls Mutator	CCM
	Inline Constant Mutator	ICM
	Member Variable Mutator	MVM
	Non Void Method Calls Mutator	NVMCM
	Remove Conditionals Mutator	RCM
	Remove Increments Mutator	RIM
	Switch Mutator	SM

metic expressions to `short`. DTF is the `double`-to-`float` precision degradation operator; it casts results of `double` arithmetic expressions to `float` and then back to `double` to preserve the type. The ITS and DTF implementations perform casting at the bytecode level.

ApproxiMate uses all other mutation operators available in PIT: seven active-by-default operators and seven non-default operators (Table 2.1), which we enabled to increase the variety of mutation operators in our experiments. ApproxiMate computes mutation scores using only mutants that are covered by the tests. The comparative analyses require the exact mapping from tests to mutants killed. Because PIT cannot capture the test that killed a mutant because of memory or other runtime errors, we exclude such mutants from the mutation score computations and the comparative analyses.

2.2.2 Comparative Analyses in ApproxiMate

In mutation testing, it is desirable to use as few mutants as possible while still resulting in the same confidence of the mutation testing results. Prior research investigated means to identify the subset of mutants that are harder to kill and representative of the other mutants [15, 41, 83, 116]. If approximate transformations generate mutants that are harder to kill than mutants generated by conventional mutation operators, it suggests that they are relatively effective as mutation operators. We use two techniques from the literature to compare the mutants from approximate transformations with those from conventional mutation operators: minimal mutants analysis [15,41] and selective mutation analysis [73,86].

Minimal mutants analysis. Minimal mutants [15, 41] are used as proxies for finding what mutants are harder to kill compared with the all other mutants [13]. We use minimal mutants, which are based on dynamic subsumption:

- ***Definition:*** A mutant m *dynamically subsumes* another mutant m' if the set of tests that kill m is a subset of the set of tests that kill m'. Intuitively, m is harder to kill than m' because only some tests that kill m' can kill m.
- ***Condition:*** If mutants generated from approximate transformations are in the set of minimal mutants, then they subsume (and are therefore harder to kill than) mutants from some conventional mutation operators.
- ***Computation:*** We apply the algorithm proposed by Gopinath et al. [41] to compute the set of minimal mutants.[1]

Selective mutation analysis. Selective mutation analysis is a heuristic technique for reducing the number of mutants to be run [73,83,86]. The general idea in selective mutation analysis is to find a set of sufficient mutation operators:

- ***Definition:*** Sufficient mutation operators are a subset of all mutation operators, such that tests which kill mutants generated by the sufficient mutation operators also kill all mutants generated by the operators that are in the complement of the sufficient set.
- ***Condition:*** If approximate transformations are in the set of sufficient mutation operators, it indicates that they are part of operators that are representative of all mutation operators.
- ***Computation:*** We analyze only the mutants killed by the existing tests, assuming that all other mutants cannot be killed [39]. Our algorithm for selecting sufficient operators is close to what was done in prior work using existing test suites [39], but there are two main differences. First, we do not restrict the number of iterations for removing mutation operators. Second, we apply test-suite reduction in each iteration to create a tailored test suite which is sufficient to kill only mutants generated by the currently-selected operators. This is close to previous studies on selective mutation testing [83, 86] where, on each

[1]Gopinath et al. refer to minimal mutants as surface mutants in their work.

Table 2.2: Subjects used in our study

Subject	SLOC	Tests	Short Description
commons-imaging	31377	169	Imaging library
commons-io	9957	1098	IO library
HikariCP	4256	96	Database connectivity pool
imglib2	31839	337	Image processing library
vectorz	44009	453	Vector and matrix library
jblas	10356	39	Matrix library
OpenTripPlanner	64202	356	Trip planner
la4j	9368	801	Linear algebra library
meka	36512	306	Machine learning library

iteration, a test suite is *generated* to kill only mutants from the selected operators, and the generated tests are checked to see that they kill all mutants.

Each iteration of the algorithm starts by finding and removing the operator that generates the most number of mutants. The second step in each iteration is to apply test-suite reduction [117] to construct a reduced test suite which kills only mutants generated from the remaining operators. If the reduced test suite kills *all* mutants (not just mutants generated from the remaining operators), the algorithm continues to the next iteration by greedily removing the operator that generates the next highest number of mutants. If a reduced test suite that kills all mutants cannot be generated, we continue the same iteration by putting the removed operator back in the set and removing the next highest mutant-generating operator. The algorithm halts when we cannot remove any more operators and still kill all mutants. The operators that remain after the algorithm halts form the set of sufficient mutation operators.

2.2.3 Evaluation Subjects

We use nine open-source Java subjects in our evaluation of the approximate transformations as mutation operators. Table 2.2 shows for each subject the source lines of code (SLOC) it has, the total number of test methods, and a description. The subjects vary widely in size and come from different domains: image processing, machine learning, linear algebra, and databases applications. The subjects are from GitHub and are a mix of (1) subjects used in previous software testing papers [64, 67], and (2) computationally-intensive subjects that may have more opportunities for applying approximate transformations because they come from domains (e.g., linear algebra, image processing, machine learning) that may benefit

Table 2.3: Number of mutants per operator

Project	Conv. Avg	LPM	ITS	DTF
commons-imaging	1577.43	275	1097	362
commons-io	653.31	37	191	0
HikariCP	192.69	6	17	1
imglib2	646.54	264	296	245
vectorz	2426.93	1009	1991	1466
jblas	323.79	155	147	29
OpenTripPlanner	2265.71	160	623	478
la4j	644.93	311	569	487
meka	593.85	266	192	153
Average	1036.13	275.89	569.22	357.89

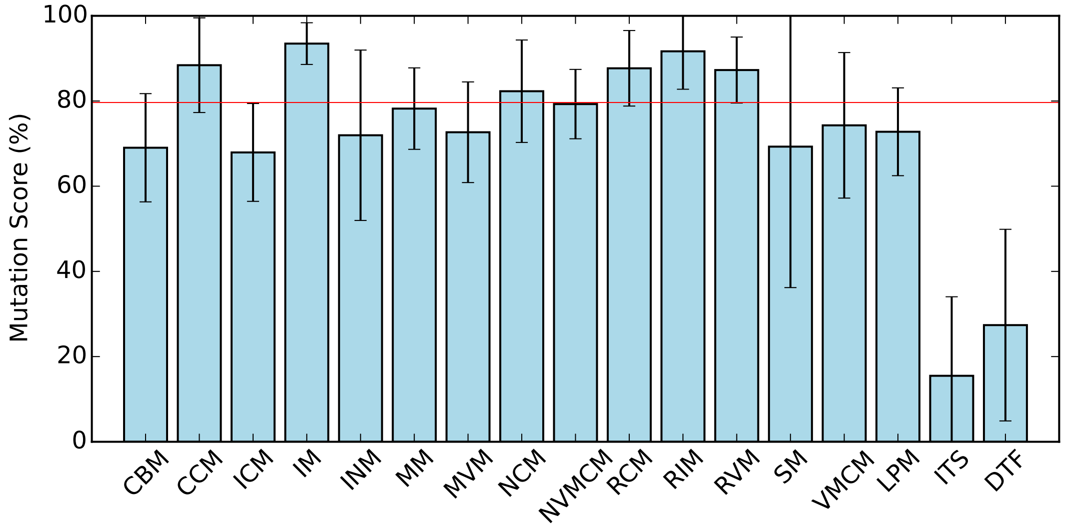


Figure 2.4: Mutation scores per operator

more from approximate computing techniques [27, 104].

2.3 QUANTITATIVE ANALYSIS RESULTS

This section contains answers to **RQ2.1**: how effective are approximate transformations as mutation operators, compared to conventional mutation operators, in terms of mutation scores, minimal mutants analysis, and selective mutation analysis.

2.3.1 Effectiveness by Mutation Scores

Table 2.3 shows the number of mutants generated and covered by tests per mutation operator for all subjects. The "Conv. Avg" column shows the average number of mutants generated by conventional mutation operators for each subject. Columns "LPM", "ITS" and "DTF" show the number of mutants generated by the approximate transformations. Figure 2.4 shows the average mutation score per operator across all subjects. Each bar represents a mutation operator; the rightmost three bars are for approximate transformations—LPM, ITS, and DTF. The y-axis shows average mutation score per operator across all subjects. The red horizontal line is the average mutation score of all conventional mutation operators across all subjects. The error margin on each bar shows the standard deviation.

Loop perforation. On average, LPM generates only 275.89 mutants, compared with 1036.13 for conventional mutation operators. This is because there are much fewer loops (the only locations that LPM can mutate) relative to the number of locations that conventional mutation operators can mutate. The average mutation score for LPM (72.78%) is *slightly lower* than that of conventional mutation operators (79.65%) but it is not a low outlier, compared to other operators.

Precision degradation. The number of mutants generated by ITS and DTF are 569.22 and 357.89, respectively. These are significantly fewer than the average number of mutants generated by conventional mutation operators (1036.13). The average mutation scores for ITS and DTF are 15.49% and 27.39%, respectively (Figure 2.4). These are significantly lower than the average score of 79.65% for conventional mutation operators. In fact, ITS and DTF scores are the lowest among all operators (including LPM).

Discussion. The LPM mutation scores are closer to the mutation scores of conventional mutation operators, suggesting that LPM mutants are as easy/hard to kill as mutants generated from conventional mutation operators. The mutation scores for ITS and DTF are very low compared to the scores for conventional mutation operators. A further analysis of survived mutants in Section 2.5 shows that this is not due to a high number of equivalent mutants, but rather to bad tests that do not exercise the code with large values crossing the precision boundaries. We perform a more detailed qualitative analysis on LPM, ITS, and DTF mutants in Section 2.4.

2.3.2 Effectiveness by Minimal Sets of Mutants

We compute minimal mutant sets, as described in Section 2.2.2, to see if mutants generated by approximate transformations are in the minimal mutant set, meaning they are not

Table 2.4: Minimal mutants per operator

Project	Conv. Avg	LPM	ITS	DTF
commons-imaging	6.79	1	0	0
commons-io	37.07	1	1	0
HikariCP	4.57	1	0	0
imglib2	13.79	4	5	3
vectorz	18.36	14	1	9
jblas	2.21	2	0	1
OpenTripPlanner	15.29	2	0	1
la4j	13.57	17	3	17
meka	7.00	2	2	2
Average	13.18	4.89	1.33	3.67

subsumed by other mutants. Table 2.4 shows, for each subject, the breakdown of the counts of the minimal mutants. The column "Conv. Avg" shows the average number of minimal mutants generated from conventional mutation operators; the remaining columns show the number of minimal mutants for each approximate transformation. Approximate transformations show up in the minimal set of mutants—at least one of the last three columns, LPM, ITS, and DTF, is not `0`—for all subjects. The average numbers of mutants contributed by LPM, ITS, and DTF to the set of minimal mutants are 4.89, 1.33, and 3.67, respectively. We conclude that, when used as mutation operators, approximate transformations can generate mutants that are not subsumed by mutants generated from conventional mutation operators.

2.3.3 Effectiveness by Selective Mutation Analysis

Table 2.5 presents the sets of sufficient mutation operators computed using the greedy selective mutation analysis algorithm presented in Section 2.2.2. For each subject, we show the number of conventional mutation operators ("# Conv. Operators") and the selected approximate transformations ("Approx. Operators") that are in the sufficient mutation operator set.

Approximate transformations appear among the sufficient mutation operators in six of the nine subjects (`commons-io`, `imglib2`, `vectorz`, `jblas`, `la4j`, and `meka`). The fact that approximate transformations end up in the sufficient mutation operator sets shows that they are important, because sufficient mutation operators are meant to be representative of all operators; tests good enough to kill these mutants are good enough to kill the mutants from all the other operators (Section 2.2.2). Furthermore, when we perform selective mutation analysis

Table 2.5: Selective mutation operator analysis

Project	**# Conv. Operators**	**Approx. Operators**
commons-imaging	7	n/a
commons-io	9	ITS
HikariCP	8	n/a
imglib2	9	LPM,DTF
vectorz	10	LPM,ITS,DTF
jblas	4	DTF
OpenTripPlanner	8	n/a
la4j	9	LPM,ITS,DTF
meka	5	LPM,DTF

with only conventional mutation operators, we find that the sufficient mutation operators for most subjects are the same as those corresponding to the number of conventional mutation operators from the "# Conv. Operators" column in Table 2.5; the only exception was `meka`. From these subjects where approximate transformations are in the set of sufficient mutation operators, it seems approximate transformations are necessary to represent themselves, as the conventional mutation operators do not subsume the approximate transformations.

2.4 CODE PATTERNS

This section provides answers to **RQ2.2**, on code patterns that approximate transformations reveal. We describe the results of our qualitative analysis to answer these subquestions:

RQ2.2.1: What code patterns do LPM mutants reveal?

RQ2.2.2: What code patterns do ITS/DTF mutants reveal?

RQ2.2.3: How are approximate transformations different from conventional mutation operators and how can they help mutation testing?

Answers to these questions help with understanding the type of computations affected by the proposed operators. Moreover, these answers guide the analysis in Section 2.5 on practical impact.

Methodology. For LPM, we randomly sampled and inspected 5% of killed mutants and 5% of surviving mutants for each subject. ITS and DTF generate significantly higher numbers of mutants than LPM in some subjects, so we sampled and inspected only 1% (121 mutants) of

Table 2.6: Code patterns for killed and survived loop perforation and precision degradation mutants

Approximate Transformation	**Code Patterns**	**#Surviving**	**#Killed**
Loop Perforation	Initialization loop	3	2
	Conditional computation on elements	14	22
	Computation on all elements	17	56
	Reduction	2	9
Precision Degradation	Result is within a precision range	95	0
	Result is outside a precision range	0	15
	Computing large values	1	8
	Indexing beyond the size of `short`	0	2
Total		132	114

their killed and surviving mutants. Table 2.6 shows code patterns we found during inspection. Sections 2.4.1 and 2.4.2 further explain these patterns.

2.4.1 Code patterns for LPM mutants

Initialization loop. When a loop is used to initialize elements in a data structure, an LPM mutant that skips loop iterations may leave some elements uninitialized. Mutants of this pattern are killed by tests that rely on all elements to be initialized. However, we also find cases where such mutants survived, e.g., in method `Index#toSet()` of `vectorz`, shown in Figure 2.5. LPM skips some iterations in the loop that initializes elements of set `ss`. The only test for this method, `testSetCreate`, passes when LPM skips an iteration that adds a duplicated value to `ss`. The mutant produces the same result as the original code and reasoning about its survival can help improve the test suite with tests that kill this mutant by not having duplicated data.

Conditional computation on elements. As a loop iterates over all elements in a data structure, the loop body checks whether a property holds before performing some computation. We find examples of this pattern in `commons-imaging`, `vectorz`, and `jblas`. Consider the example in class `DoubleMatrix` of `jblas` shown in Figure 2.6. The LPM mutant is not killed by `testArgMinMax()`, because the index with the minimum element is not skipped. The test suite can be improved by adding more tests with input data where the minimum element(s) are in a variety of different indices. In general, mutants that involve checking

```
Set<Integer> toSet() {
  TreeSet<Integer> ss=new TreeSet<Integer>();
  for (int i=0; i<data.length; i++) {
    ss.add(data[i]);
  }
  return ss;
}

@Test
void testSetCreate() {
  Index ind=Index.of(1,3,3,3,5);
  Set<Integer> s=ind.toSet();
  assertEquals(3,s.size());
  assertEquals(Index.createSorted(ind.toSet()), Index.of(1).includeSorted(s));
}
```

Figure 2.5: *Initialization Loop* LPM code pattern from `vectorz` [8] and its corresponding test

```
public int argmin() {
  if (isEmpty()) { return −1; }
  double v = Double.POSITIVE_INFINITY;
  int a = −1;
  for (int i = 0; i < length; i++) {
    if (!Double.isNaN(get(i)) && get(i) < v) {
      v = get(i); a = i;
    }
  }
  return a;
}

@Test
public void testArgMinMax() {
  A = new DoubleMatrix(4, 3, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0);
  assertEquals(0, A.argmin(), eps);
  assertEquals(11, A.argmax(), eps);
}
```

Figure 2.6: *Conditional Computation on Elements* LPM code pattern from `jblas` [3] and its corresponding test

```
public double reduce(double init, double[] data, int offset, int length) {
  double result=init;
  for (int i=0; i<length; i++) {
    result=apply(result,data[offset+i]);
  }
  return result;
}
```

Figure 2.7: *Reduction* LPM code pattern from `vectorz` [8]

a property (or searching for a value) and potentially exiting the loop early tend to survive when tests do not check for both the cases when the property holds and when it does not (e.g., they only `assertTrue` but do not have some `assertFalse` for a different input). We find such surviving mutants in `vectorz`, `jblas`, `meka`, and `OpenTripPlanner`. Killed mutants of this pattern often modify data structures where most elements satisfy the property; skipping iterations misses important computations that affect test outcomes.

Computation on all elements. As a loop iterates over the elements in a data structure, its body performs an independent computation on each element. For example, the computation may involve setting values at corresponding indices in another array, or modifying the current element in the input array. Tests tend to kill LPM mutants for this code pattern when the test assertion iterates over all elements in the resulting array to check that the value at each index is correct. We observe loops of this pattern often in image-processing applications, which process matrices of pixels (e.g., `commons-imaging` and `imglib2`). In math applications with vector and matrix operations (e.g., `jblas`, `la4j`, and `vectorz`), these LPM mutants are commonly killed because the assertions check that every element has the expected value.

Reduction. As a loop iterates over all elements in a data structure, the loop body applies a reduction operation, aggregating all values in the data structure to one representation. This pattern commonly occurs in math applications (e.g., `vectorz` and `jblas`). An example in class `Op2` of `vectorz` is shown Figure 2.7. The `reduce` method applies an operation `apply` to each element in a subarray of the `data` array. Tests typically kill such LPM mutants because the final result is a single value and the tests assert that the resulting value is equal to an expected value. It is also uncommon that the input array is such that the elements skipped are all identity elements with respect to the applied operation. Most mutants of this pattern are killed; the few that survived are such that test inputs exercise only one loop iteration; therefore applying LPM is of no effect. To kill these mutants, developers need to add tests that execute the loop with more than one iteration.

2.4.2 Code patterns for ITS and DTF mutants

Result is within/outside a precision range. When the specification of the operation is such that, for all allowed inputs, the result is always going to be within the degraded precision range, then such mutants should always survive. An example of a surviving DTF mutant is in `ColorConversions#convertHSLtoRGB` in `commons-imaging`, which converts HSL to RGB pixels by multiplying each pixel (a `double` between 0.0 and 1.0) by `255`. Degrading precision only slightly changes the accuracy of the result, within the tolerance bound of the test. On the other hand, when the test execution leads to values that go outside the precision range, such mutants are killed. We find surviving mutants of this pattern for DTF in `vectorz`, `OpenTripPlanner`, and `la4j`. We find instances where mutants of this pattern are killed in `meka`, `la4j`, `OpenTripPlanner`, `jblas`, and `imglib2`.

Computing large values. When computations involve large numbers, degrading the precision easily leads to different results, e.g., due to overflow. Hence, ITS mutants involving such computations tend to be killed by tests that expect a much larger value than the mutant returns. However, in `OpenTripPlanner`, we find an ITS mutant of this pattern that survived because the tests do not check that computed hash code values are correct, and there is no collision in the hash codes computed at lower precision.

Indexing beyond the size of `short`. ITS mutants get killed when they cause the indices of data structures to exceed their bounds when large `int` values are cast to `short` values that overflow. For instance, class `CRSMatrix` in `la4j` has a method `set`. The test that kills the ITS mutant creates a matrix with dimensions greater than `Short.MAX_VALUE`. When the ITS mutant is run on the input matrix, the `int` to `short` precision degradation causes overflow, leading to an `ArrayOutOfBoundsException (AOOBE)`. Also, in class `CellRandomAccess` of `imglib2`, an overflow occurs when an `int` value used to walk through all positions in a large matrix is cast to a `short`, causing an `AOOBE`.

2.4.3 Comparing approximate transformations with conventional mutation operators

LPM vs. conventional mutation operators. We check whether mutants generated by conventional mutation operators apply to each loop header on which an LPM mutant was generated. In total, seven conventional mutation operators (CBM, ICM, IM, NCM, NVMCM, RIM, RCM) can be applied on the same lines as LPM. Only two of these seven conventional mutation operators generate mutants that behave somewhat similarly to LPM mutants: Inline Constant Mutator (ICM) and Negate Conditionals Mutator (NCM). ICM changes the constant of the loop initialization to skip only the first iteration. NCM changes the loop condition to skip the entire loop body. LPM falls between the ICM and NCM in terms of the number of skipped iterations.

We conclude that LPM is complementary to conventional mutation operators; reasoning about their killed/surviving mutants helps developers generate new tests that exercise the code in new ways, improving their test suites (Section 2.5). Our conclusion holds for mutation operators in three other Java mutation tools: MuJava [69], Javalanche [96], and Major [57]. Replace_Constant from Javalanche and Constant Value Replacement from Major produce similar effects as PIT's ICM; Negate_Jump, and Unary_Operator from Javalanche, and Unary Operator Replacement, and Branch Condition Manipulation from Major have similar effects as PIT's NCM. Our understanding of code patterns exercised by LPM mutants enable us to perform such an analysis on mutation operators from other frameworks. If one of these tools had an operator such as that would replace the constant for every `iinc` bytecode instruction, it would produce the mutants that LPM produces (but also more mutants, as LPM applies this operator only for loop increments)

ITS/DTF vs. conventional mutation operators. We do not compare the precision degradation operators with the conventional mutation operators because the mutants they generate are not matched by any of the conventional mutation operators that modify arithmetic expressions. As our inspection in Section 2.5 shows, these mutants provide guidance towards writing better tests that exercise boundary values.

Patterns for approximate transformations and tailored mutation. The patterns we identified open up a research opportunity to achieve additional savings in mutation testing. Our findings related to code patterns can enable performing tailored mutation testing [14] or specialized selective mutation [60] to find (parts of) applications where approximate transformations can be effective as mutation operators.

2.5 IMPACT ON SOFTWARE TESTING PRACTICES

This section answers **RQ2.3**, on the practical impact on software testing of approximate transformations as mutation operators. We describe the results of our qualitative analysis to answer these subquestions:

RQ2.3.1: How often do surviving mutants from approximate transformations indicate that tests are bad, mutant is equivalent, or code is approximable?

RQ2.3.2: Do insights from inspecting surviving mutants from approximate transformations help developers?

2.5.1 Bad test, equivalent mutant, or approximable code?

Surviving mutants are traditionally regarded as either (1) signaling buggy, inadequate, or missing tests (*BadTest*) or (2) semantically equivalent to the original code, i.e., equivalent mutants. However, inspecting mutants generated from approximate transformations, we discovered a third possibility: the mutant survived because the original code is approximable (*ApproxCode*). That is, the mutant is semantically different from the original code but produces acceptable outcomes that are within a tolerable range. *This third interpretation applies to mutants from all operators, not just the ones generated by approximate transformations, changing the way mutation testing results should be interpreted in general.*

Of our inspected LPM mutants, 63.83% indicate bad tests (*BadTest*) and 19.15% indicate approximable code (*ApproxCode*); we find no equivalent mutants, and the remaining 17.02% are hard to inspect. Of our inspected ITS and DTF mutants, 53.13% indicate bad tests, 14.58% are equivalent, 11.46% indicate approximable code, and the remaining 20.83% are hard to inspect. Section 2.4 discussed the patterns that approximate transformations reveal, explaining the contexts in which those patterns signal approximable code. Section 2.5.2 describes how *BadTest*s inspired better testing and describes some pull requests we made to fix *BadTest*s.

We find mutants indicating *ApproxCode* in `vectorz`, `la4j`, `jblas`, and `meka`. An example from `la4j` is method `Matrix#shuffle()`, which makes a copy of an input matrix and uses a loop to randomly shuffle elements in the copy. Applying LPM to the shuffling loop is practically not observable, because the specification of the expected output is non-deterministic [101]. For ITS the surviving mutants for *ApproxCode* are equivalent, while for DTF the surviving *ApproxCode* mutants are within the precision range defined in the application.

Determining whether code is approximable is not an easy task. It is highly dependent on the quality of the oracles in the test suites that determine the ranges of acceptable output. Approximate computing often relies on the *usage context* (i.e., specific applications and application-level requirements) to determine if code is approximable. Such usage context is not available for the developers of general-purpose libraries (like most of our subjects) that can be used in a myriad of contexts. Therefore, the tests for these libraries are written in a conservative way, and consequently, our set of identified approximable patterns are necessarily conservative as well.

2.5.2 Do insights from surviving mutants help improve testing practice?

We find that surviving mutants of the *BadTest* category can be killed by adding tests that (1) achieve better loop coverage, (2) achieve better coverage of the loop condition, (3) exercise the code with larger inputs that cross the precision boundaries, or (4) check all output elements. Even though these insights are not new to the testing community, the real value lies in the fact that the approximate transformations are able to detect those problems, bringing them to the attention of the developer who might not have such considerations in mind. We also submitted pull requests that fix bad (buggy, inadequate, or missing) tests, to evaluate whether these insights can help developers improve their test suites. Seven of the 11 pull requests that we submitted were already integrated by developers into `vectorz`, `HikariCP`, `commons-imaging`, `imglib2`, and `commons-io`. We next discuss the categories and the pull requests.

Achieve better loop coverage. 11 out of 30 LPM *BadTest* cases have tests that do not achieve full loop coverage, i.e., they do not have tests that exercise zero, one, *and* more than one loop iterations [16]. As Table 2.7 shows, the tests frequently cover either zero or one iteration. We discover the lack of full loop coverage while inspecting surviving LPM mutants in `vectorz`, `jblas`, `OpenTripPlanner`, and `commons-io`. The causes of low loop coverage that we observed are when (1) a test exercises the code with small inputs (e.g., one dimensional matrices) and (2) a test searches for a value that always happens to be the first element in the input data, so that the loop iterates only once before exiting. For example, in `jblas`, the method `argmin()` returns the index of the minimum element in a matrix. All tests that cover `argmin()` use input that is sorted in ascending order, so `argmin()` always returns `0`.

Achieve better coverage of loop condition. While inspecting the 14 surviving LPM mutants for the code pattern "Conditional computation on elements" (Section 2.4.1), we find 12 of them are cases of *BadTest* in two categories: either the conditional check on the elements is never performed, or the conditional check is only performed on even-numbered iterations. In several cases the tests exercise the loop with only valid inputs, so the conditional check for errors that happen in the loop body is never performed. LPM helps direct the developer's attention into those critical parts of the code. An example from `commons-io` (SHA:`733dc26`) is shown in Figure 2.8. The method `resolveProxyClass()` from the class `ClassLoaderObjectInputStream` is only exercised by the test `testResolveProxyClass`. The test passes only one interface (`Comparable.class`) to the loop in `resolveProxyClass`. Thus, applying LPM to that loop will not cause `testResolveProxyClass` to fail, i.e., the resulting mutant survives, unless more than one interface is passed to `resolveProxyClass()`. Our pull request containing such a test was accepted by the `commons-io` developers.

Table 2.7: Lessons pearned dfor better testing practices from LPM *BadTest* cases

Bad Testing Pattern	**#Cases**	**Learned Testing Practice**
Zero iterations	7	Better loop coverage
One iteration	4	Better loop coverage
Loop condition (LC) Not Taken	8	Better coverage of LC
LC taken on even iterations	4	Better coverage of LC
Weak or no assertion	6	Check all output elements
Small Inputs	51	Exercise boundary values
Other[2]	1	-
Total	81	

Exercise boundary values. All *BadTest* cases for ITS and DTF are due to tests using small inputs. This means that the current tests do not use values that exceed the precision bounds of `short` for ITS and `float` for DTF, and we can write a test that can kill the mutant. A DTF example from `vectorz` is in the class `Quaternions`, which represents numbers from the quaternions number system using `double` precision. The method `mul()` computes the product of two quaternions. Mutants casting any of the arithmetic operations involved in the computation survive because the numerical values are very small.

Check all output elements. Multiple mutants are not killed because of the weakness or absence of assertions in the tests. For example, `meka` is a machine learning library. The tests cover the mutants, but most of the tests do not have assertions, and coming up with strong assertions is non-trivial. Another example is in `vectorz` (shown in Figure 2.1). The only test that covers the method `Matrix#swapRows()` does not check that all elements in the swapped rows are as expected (detailed discussion is in Section 2.1). We submitted a pull request to add assertions and it has been accepted.

2.6 THREATS TO VALIDITY

Internal. In our implementation, the tools and scripts that we use may contain bugs. To mitigate this, we use PIT [6] and Spoon [108], which are well-tested tools commonly used by researchers. We tested our implementations of approximate transformations on many small examples, and multiple collaborators reviewed the scripts that we used for running

[2]This is a case in `meka`; a setter method resets the values in a matrix, but the new values are almost equal to the old values, so the effect of skipping iterations is not observable. A better test would exercise the function such that the difference between the new and old values is observable.

```
protected Class<?> resolveProxyClass(final String[] ints) {
  final Class<?>[] iClasses = new Class[ints.length];
  for (int i = 0; i < ints.length; i++) {
    iClasses[i] = Class.forName(ints[i], false, loader);
  }
  try {
    return Proxy.getProxyClass(loader, iClasses);
  }
  catch (final IllegalArgumentException e) {
    return super.resolveProxyClass(ints);
  }
}

@Test
public void testResolveProxyClass() throws Exception {
  ...
  ClassLoaderObjectInputStream c =
    new ClassLoaderObjectInputStream(...);
  String[] i = new String[]{Comparable.class.getName()};
  Class<?> r = c.resolveProxyClass(i);
  assertTrue("...", Comparable.class.isAssignableFrom(r));
  c.close();
}
```

Figure 2.8: Bad test example from `commons-io` [2]

experiments. For the qualitative analysis, the classifications we assign to killed and surviving mutants may be erroneous, and there could be bias in selecting mutants to inspect. To mitigate errors in the classifying mutants, each person wrote detailed notes during inspection, based on a pre-defined format and up to two other collaborators (different from the one who did the original inspection) double-checked the notes. To reduce bias in the mutants that we selected to inspect, we randomly sampled the killed and surviving mutants in each project.
External. We analyze only Java subjects, which may not be representative of all software. To mitigate that, we choose the subjects in our study from different domains (databases, I/O, machine learning, imaging, linear algebra, etc.).
Construct. The conventional mutation operators we use in comparison may not be representative of all mutation operators. Since ours is an initial study of the effectiveness of approximate transformations as mutation operators, we have used the set of conventional mutation operators that are available in PIT, which are used in both research and practice. Furthermore, in our qualitative analysis we examine mutation operators from three other popular mutation frameworks [57, 69, 96] and find our conclusions to still hold for those.

The approximate transformations that we evaluated are a subset of all approximate transformations and they may not be representative. To mitigate that, we model popular transformations that have been widely used in the approximate computing literature. Each transformation that we implement models some key properties of the original approximate transformations, i.e., dropping parts of computation (LPM), large magnitude errors (ITS), and small magnitude errors (DTF).

CHAPTER 3: EFFECTS OF COMPILER OPTIMIZATIONS

In this chapter we present an empirical study of the effects of compiler optimizations on mutation testing at the compiler IR level. Our study aims to investigate whether mutation testing should be applied with or without compiler optimizations, providing the user with insights about the interplay between mutations and compiler optimizations. To that end, we ask the following research questions:

RQ3.1: How do compiler optimizations affect the number of generated mutants?

RQ3.2: How do compiler optimizations affect the number of equivalent and duplicated mutants?

RQ3.3: How do compiler optimizations affect the mutation score?

RQ3.4: How do these effects vary with the class of mutation operators applied?

3.1 ILLUSTRATIVE OVERVIEW

We use the program `cut` from Coreutils to illustrate our evaluation on a concrete example and to introduce some background material. The `cut` program is a standard Unix command-line utility that selects columns or fields from the input and writes them to the standard output. The Coreutils source distribution comes with the source code (`src/cut.c`) and 65 tests specifically written for this program. Each test runs `cut` for some given input (provided as the command-line arguments and the content of an input file), and checks that the actual output (in terms of both the content of `stdout` and the return exit code of the program run) matches the expected output. For example, one test gives as input the command-line argument "`-c4`" (to select the 4th character) and the input file with the content "`123`" (with no newline byte before the end of the file). The expected result is an empty string with a new line appended, conforming with `cut`'s spec. All 65 tests can be run with one shell script. We want to evaluate the effects of compiler optimizations on mutation testing for `cut`.

3.1.1 Clang and LLVM

Our mutation tool-set first uses the Clang [78] compiler front-end to translate `src/cut.c` into LLVM intermediate representation (IR), known as *bitcode*. LLVM IR also encapsulates assembly instructions and accompanying comprehensive metadata. The LLVM framework provides a rich API for alterations of the bitcode and the LLVM compiler tool-set already implements a large set of compiler optimizations. We modify the `Makefile` build configuration for Coreutils such that the compiler produces LLVM bitcode files using a specific optimization level, `-O0` or `-O3`. Note that this step applies optimizations *before* applying the mutations (and a later step will additionally apply optimizations *after* applying the mutations).

3.1.2 Compiler Optimizations

Compiler optimizations are semantics-preserving transformations applied to a program with the intention of improving the program's performance. Each optimization is generally intended to make the program smaller or faster. The optimizations are typically applied together to synergistically combine the benefits of each optimization to provide superior performance. While the `-O0` level aims for fast compilation and applies almost no optimizations, the `-O3` level aims to produce very efficient code and applies a large number of optimizations, including loop unrolling, constant propagation, and instruction combining [79]. Specifically, for the `cut` program, the `-O0` level produces LLVM bitcode with 1274 instructions, while the `-O3` level produces 1110 instructions.

For example, Figure 3.1 shows a snippet of the `cut` bitcode before and after applying optimizations. It shows the basic block in the `usage` function that returns the status code and then exits. In the unoptimized bitcode, a load instruction retrieves the status code before using it as an argument to the call to `exit`. However, in the optimized bitcode, the tail-call optimization [81] replaces the original call instruction with a tail call, allowing to remove the load instruction, thereby reducing the overall number of instructions and improving the program performance. For `cut`, the total number of call instructions at the `-O0` level is 133, of which none is a tail call. At the `-O3` level, the total number of call instructions is 120, of which 108 are tail calls. Also, the number of load instructions is reduced from 330 at `-O0` to 166 at `-O3`.

3.1.3 Mutant Generation

We implemented the mutation operators as manipulations of the LLVM IR. Specifically, we implemented two LLVM passes. The first pass takes as input an LLVM bitcode file and some class of mutation operators (e.g., AOR or ICR). It then finds all bitcode instructions that can be mutated (for AOR, all LLVM instructions that use arithmetic operators such as `add`; and for ICR, all LLVM instructions that have an integer constant), and outputs a set

Listing 3.1: Unoptimized bitcode (-O0 level)

```
if.end:
  %14 = load i32, i32* %status.addr, align 4
  call void @exit(i32 %14) #8
  unreachable
```

Listing 3.2: Optimized bitcode (-O3 level)

```
if.end:
  tail call void @exit(i32 %status) #13
  unreachable
```

Figure 3.1: Example from `cut` showing instruction count reduction

of possible mutations (e.g., replacing a specific `add` instruction with `sub`, `mul`, and `div`; or replacing an integer constant with another integer value). The second pass takes as input an LLVM bitcode file and a specific mutation to apply (as computed by the first pass) and outputs a modified LLVM bitcode file with that mutation applied. Our tool-set invokes the second pass for each and every mutation found by the first pass. Each mutated LLVM bitcode file is then compiled (and linked) into an actual executable, using the LLVM back-end at the same optimization level, either `-O0` or `-O3`, that was used initially by Clang, thus applying optimizations also *after* applying the mutations.

3.1.4 Number of Generated Mutants

For `cut`, we obtained a total of 1958 mutants at the `-O0` level and 2547 mutants at the `-O3` level. There were more mutants at the `-O3` level although it had fewer instructions overall than the `-O0` level (1110 vs. 1274), because the `-O3` level had more mutation opportunities; we define a mutation opportunity as a part of an instruction that can be mutated, e.g., the opcode or one of the operands. Across the various classes of mutation operators, we obtained the following numbers of mutants: 100 AOR, 14 LCR, 864 ROR, and 980 ICR at the `-O0` level; and 116 AOR, 102 LCR, 1215 ROR, and 1114 ICR at the `-O3` level.

3.1.5 Mutation Score

We next ran the `cut` test suite on each of the mutants, accounting for cases of “rogue” mutants that could affect the entire testing and experimental process. Such cases include mutants that encounter infinite loops (and could block all experiments) or mutants that

```
%12 = and i8 %dash_found.0.ph430.i, 1
%tobool134.i = icmp eq i8 %12, 0
br i1 %tobool134.i, label %L1, label %L2
```

Figure 3.2: Example from `cut` for duplicated mutants (`-O3` level)

excessively write to disk. Our tool-set includes a sophisticated runner to handle these cases. If a mutant causes any test in the suite to fail, the mutant is *killed*. Higher-quality test suites kill more mutants, and the percentage of mutants killed is called the *mutation score*. The actual value depends on the generated mutants, which in turn depend on the compiler optimization level. Specifically, for `cut`, we find that the test suite kills a total of 1091 and 1402 of the mutants generated at the `-O0` and `-O3` levels, respectively. The corresponding mutation scores are 55.7% and 55.0%, respectively. The same test suite thus appears seemingly better when evaluated with the mutants generated at the `-O0` level than at the `-O3` level.

3.1.6 Equivalent and Duplicated Mutants

Some of the mutants that are generated, while syntactically different in the mutated LLVM, may end up being semantically equivalent to the original `cut` program. No test can kill any equivalent mutant, so ideally all equivalent mutants should be removed from the set of generated mutants. However, determining mutant equivalence is undecidable in general [16, 22]. Our tool-set uses the recently proposed trivial compiler equivalence [88] to perform a bit-by-bit equality comparison between the compiled binaries for the original code and the mutants. If a mutant binary is exactly the same as the original binary, then the mutant is definitely equivalent; if the binaries differ, then we cannot be sure. In the case of `cut`, this technique finds 66 and 111 equivalent mutants at the `-O0` and `-O3` levels, respectively.

Moreover, even if we cannot establish that some mutants are definitely equivalent to the original code, we can find that these mutants are equivalent to one another—following Papadakis et al. [88], we call such mutants *duplicated*. We use the same technique that compares compiled binaries of mutants to find the mutants that are definitely duplicated but may not be equivalent to the original code. For `cut`, this technique finds 11 duplicated mutants (0.5% of all generated mutants) at the `-O0` level and 360 (14.1% of all generated mutants) at the `-O3` level.

Figure 3.2 shows a snippet from `cut` that leads to duplicated mutants. The second instruction compares whether the boolean from the register `%12` is equal to zero and saves the result in the register `%tobool134.i` which is checked in the next branch instruction. The comparison instruction presents two mutation opportunities: one for ROR (replacing the relational operator `eq` with one of {`ne,ugt,uge,ult,ule,sgt,sge,slt,sle`}) and the other for ICR (replacing `0` with one of {`1,-1`}). Four duplicated mutants are generated from this instruction: replacing `eq` with one of {`ne,ugt,sgt`} or replacing `0` with `1`. (Note that these four mutants are *not* equivalent to the original code but are equivalent to one another.) Basically, checking whether a boolean is not equal or greater than `0` (i.e., the boolean is not false) is semantically the same as checking whether the boolean is equal to `1` (i.e., the boolean is true).

Identifying equivalent and duplicated mutants allows us to remove some mutants from mutation testing, which makes mutation testing faster (because we need not run tests on those removed mutants) and provides a more accurate mutation score (because we can use a more precise number of generated and killed mutants). More specifically, we should remove all equivalent mutants, and from each equivalence class of duplicated mutants, we should remove all mutants but one to act as a representative of the equivalence class. We call the set of mutants resulting from this removal the *non-equivalent, non-duplicated (NEND) mutants*. Applying this to `cut`, we end up with 1881 and 2076 NEND mutants at the `-O0` and `-O3` levels, respectively.

3.1.7 Revisiting Mutation Score

Revisiting mutation score when considering only NEND mutants, it turns out that the `cut` test suite kills 1085 and 1148 NEND mutants at the `-O0` and `-O3` levels, respectively. The absolute numbers of mutants killed among NEND mutants are lower than the absolute numbers of mutants killed among all generated mutants. The reason is that some equivalence classes of duplicated mutants were killed, and thus removing those duplicated mutants also reduces the number of killed mutants. (Note that removing equivalent mutants never reduces the number of killed mutants, because equivalent mutants cannot be killed.) As a result, we find that the mutation scores are 57.6% and 55.3% for the NEND mutants at the `-O0` and `-O3` levels, respectively. Both of these values are higher than the corresponding values for all generated mutants. We conclude that using only NEND mutants gives a more accurate evaluation of the test suite; users should prefer the `-O3` level, but they should carefully interpret the mutation score obtained at the `-O3` level. We will see that most relationships we have mentioned in this section are not specific to `cut` but actually hold for (almost) all 16 Coreutils programs that we evaluate.

3.2 EXPERIMENTAL SETUP

We describe the programs we use in our evaluation, the mutation tool-set we built for the evaluation, and the comparison strategy we used to identify equivalent and duplicated mutants.

3.2.1 Object Programs

Our evaluation uses programs from Coreutils, a well-studied set of programs frequently used as benchmarks for research in testing [24,34,62,72]. Specifically, we use Coreutils version 6.11; while not the most recent, this version is often used in research, including studies on compiler optimizations [34]. We selected 16 programs for our evaluation. We focused on the programs with test directories that explicitly label these programs as test targets (to avoid accidentally killing mutants by tests that do not target the specific program). Out of 27 such programs, our infrastructure had problems with 9, e.g., they had tests with non-deterministic results (known as *flaky tests* [68]). In the case of a flaky test, the output is dependent on the testing environment in addition to the test inputs, so both the original code and its equivalent mutants can pass or fail regardless of the chosen test. We made certain that for all 16 selected programs, (1) all tests pass on the original code, (2) all equivalent mutants produce the same output as the original code, and (3) all duplicated mutants from the same equivalence class return the same result. Each of these 16 programs comes with a number of tests, typically one or more shell scripts that invoke the program multiple times.

3.2.2 Compiler Optimizations

In these experiments, we use LLVM 3.8.1. We selected two opposite optimization levels for our experiments. The `-O0` level provides fast compilation and serves as the baseline for comparison. The `-O3` level is one of highest optimization levels in LLVM 3.8.1, enabling some of the most time-intensive optimizations.

3.2.3 Mutation Tool-Set

We implemented both mutant generation and mutant execution. For mutant generation, we wrote LLVM passes that first identify the points where mutation operators could be applied and then systematically apply these operators to modify the LLVM bitcode files (as described in Section 3.1). We implemented four classes of mutation operators: AOR,

Table 3.1: Total number of LLVM instructions and the number of mutation opportunities (per operator class and total), at both `-O0` and `-O3` levels

Program	`-O0`						`-O3`					
	Total	Mutation Opportunities					Total	Mutation Opportunities				
	Inst	AOR	LCR	ROR	ICR	Sum	Inst	AOR	LCR	ROR	ICR	Sum
chmod	675	7	15	38	256	316	403	8	15	36	227	286
chown	292	3	1	16	126	146	233	3	3	16	110	132
cut	1274	25	7	96	357	485	1110	29	51	135	417	632
dd	2436	93	74	196	684	1047	2080	91	100	234	705	1130
du	1199	11	6	62	470	549	835	22	15	76	389	502
head	1744	59	7	100	604	770	994	55	17	106	477	655
join	2088	50	6	117	791	964	1958	71	33	208	749	1061
mkdir	251	1	7	12	101	121	159	1	7	9	53	70
mv	604	5	4	31	257	297	372	3	9	28	216	256
readlink	140	1	0	6	44	51	95	1	0	5	35	41
rm	322	2	1	16	162	181	200	2	4	17	113	136
rmdir	349	2	1	21	100	124	199	2	6	23	71	102
tac	871	31	1	54	205	291	581	26	8	60	180	274
tail	3030	74	27	192	1126	1419	2175	87	47	262	997	1393
test	1895	64	24	144	529	761	1711	102	43	237	642	1024
touch	606	5	9	45	242	301	413	5	16	41	219	281
tr	3116	107	25	144	1108	1384	2219	99	104	205	854	1262
wc	1172	43	20	74	346	483	842	47	28	73	319	467
Overall	22064	583	235	1364	7508	9690	16579	654	506	1771	6773	9704

LCR, ROR, and ICR (described briefly in Section 1.3). Note that some of the mutation operators for the source language do not apply at the LLVM level. For example, "replace an arithmetic-assignment operator by another operator" replaces C-level assignment operators "`+=`", "`-=`", "`*=`", and "`/=`" with one another, but such assignment operators are de-sugared at the LLVM level. Each of our mutation operators is applied to generate syntactically distinct mutants. We integrated our LLVM passes into the `Makefile` build configuration such that it can generate all the mutants at the specified optimization level. Each of the 16 programs was compiled using each of the two selected optimization levels, producing two original LLVM bitcode files of each program. Each LLVM bitcode file was then subjected to all the mutation operators, generating our set of mutants.

For mutant execution, we created a framework that sandboxes and parallelizes runs of each program's companion test suite against its set of mutants. The framework automatically determines which mutants are killed. When a test terminates, it is easy to determine if a mutant is killed (test failed) or not (test passed). However, in some cases, mutants can exhibit unexpected behavior. First, mutations can massively increase the number of iterations of a loop by altering the guard condition, to the point where mutants can have infinite loops. Our framework handles such cases by time-limiting each test to 30 seconds; mutants that ran out of time are considered killed. Second, a mutant could write arbitrary data to the file system (and Coreutils programs and their tests already perform many file-system operations, which makes this case harder to detect). Our framework handles these cases by limiting the size of the files that a process could write. Finally, the entire mutation testing process was very time intensive, as dozens or hundreds of tests needed to be run on hundreds or thousands of mutants per program, so our framework parallelized these runs. We ran our experiments on three 24-core machines with Scientific Linux. When all the tests completed, we were able to assign each test suite a mutation score for each relevant set of mutants.

3.2.4 Mutant Comparison

It is important to remove equivalent and duplicated mutants, because they can artificially inflate or deflate the mutation score. We compared the mutants by computing checksums, specifically using `md5sum`, of the final binaries. We then compared the checksums for each mutant to those of the progenitor programs to identify which files have the same content; collisions are highly unlikely using `md5sum`.

To establish a firm baseline for comparison, the original programs had to be compiled using the same process as the mutants but without the application of mutation operators. Initially, we compiled the original programs without creating an intermediate LLVM file, going directly from the source code to the object file that was linked into the final executable. However, LLVM adds debug information during its compilation process, which was present in the mutants but not in the original executables. This provided enough differentiation to make it seem as if no mutant was equivalent to the original program. Even using `strip` to remove some symbols from the object files did not make their content match exactly. We changed the `Makefile` to add the original program to the compilation pipeline, making sure not to mutate it.

3.3 EXPERIMENTAL RESULTS

We next discuss the results obtained in our experiments. Table 3.1 shows some statistics for the 16 Coreutils programs, specifically the total number of the LLVM bitcode instructions and the number of mutation opportunities for various mutation operators at different compiler optimization levels. The number of mutation opportunities is equal to the number of instructions mutated by the corresponding operator for all operators except for ICR, where it reflects the number of integer constant occurrences (and one instruction may have more than one integer constant operand). The last row shows the sum of the values for each column. Overall, there are fewer LLVM instructions at the `-O3` level than at the `-O0` level

Table 3.2: The number of generated mutants (#M), the number (#E) and percentage (E%) of equivalent mutants, the number (#D) and percentage (D%) of duplicated mutants, and the number of NEND mutants, at both `-O0` and `-O3` levels

Program	`-O0`						`-O3`					
	#M	#E	E%	#D	D%	#NEND	#M	#E	E%	#D	D%	#NEND
chmod	1069	41	3.8	9	0.8	1019	952	90	9.4	131	13.7	731
chown	467	15	3.2	0	0.0	452	453	67	14.7	41	9.0	345
cut	1958	66	3.3	11	0.5	1881	2547	111	4.3	360	14.1	2076
dd	4208	131	3.1	22	0.5	4055	4721	297	6.2	797	16.8	3627
du	1723	74	4.2	11	0.6	1638	1682	146	8.6	178	10.5	1358
head	2699	110	4.0	27	1.0	2562	2513	250	9.9	306	12.1	1957
join	2902	112	3.8	24	0.8	2766	3980	340	8.5	496	12.4	3144
mkdir	368	23	6.2	3	0.8	342	253	15	5.9	15	5.9	223
mv	907	32	3.5	5	0.5	870	792	82	10.3	98	12.3	612
readlink	192	7	3.6	0	0.0	185	140	5	3.5	12	8.5	123
rm	543	12	2.2	0	0.0	531	458	28	6.1	42	9.1	388
rmdir	479	25	5.2	11	2.3	443	417	16	3.8	48	11.5	353
tac	1151	58	5.0	12	1.0	1081	1120	67	5.9	111	9.9	942
tail	4673	158	3.3	35	0.7	4480	5409	501	9.2	677	12.5	4231
test	3077	75	2.4	66	2.1	2936	4717	248	5.2	710	15.0	3759
touch	1083	52	4.8	17	1.5	1014	983	125	12.7	101	10.2	757
tr	4280	161	3.7	44	1.0	4075	4624	212	4.5	610	13.1	3802
wc	1780	68	3.8	18	1.0	1694	1729	108	6.2	237	13.7	1384
Overall	33559	1220	3.6	315	0.9	32024	37490	2708	7.2	4970	13.2	29812

(16579 vs. 22064). This reduction is expected, as the unoptimized code often simply moves data using instructions like `alloca`, `load`, and `store`, while the optimized code removes such instructions (e.g., Figure 3.1 discussed in Section 3.1.2). However, there is overall a *similar number* of mutation opportunities at the `-O3` and `-O0` levels (9704 vs. 9690). The optimized and unoptimized versions of the code have a similar number of instructions that perform the actual computations (or operate on constant values) and to which our operators thus apply.

3.3.1 Number of Mutants

Table 3.2 shows the total number of mutants generated for each program, the number and percentage of equivalent and duplicated mutants, and the number of NEND mutants at both the `-O0` and `-O3` levels. The last row shows the overall values, which are (1) the sums of the numbers of respective mutants in a given column and (2) the overall percentages of equivalent and duplicated mutants (computed as the weighted average across all programs).

From Table 3.2, we see that the overall number of all generated mutants is 11.7% higher at the `-O3` level (37490 mutants) than at the `-O0` level (33559 mutants). However, this relationship between the number of mutants does *not* follow for most of the individual programs. In fact, the relationship is the opposite for all programs except for six (`cut`, `dd`, `join`, `tail`, `test`, and `tr`); these six programs generate a far larger number of mutants at the `-O3` level, thus raising the average and leading to the overall conclusion. The Wilcoxon paired rank test for the numbers of all generated mutants has a p-value of 0.76, indicating that the difference between `-O0` and `-O3` levels is *not* statistically significant.

In brief, we obtain the following answer for RQ3.1: *The overall number of generated mutants is lower at the `-O0` level than at the `-O3` level, but the opposite holds for most programs and difference is not statistically significant.*

3.3.2 Equivalent and Duplicated Mutants

We next analyze the number of equivalent and duplicated mutants in more detail. Table 3.2 shows a detailed breakdown for such mutants at both optimization levels. We can see that the overall percentages of both equivalent and duplicated mutants are higher at the `-O3` level (7.2% and 13.2%, respectively) than at the `-O0` level (3.6% and 0.9%, respectively). The comparison between these percentages is similar overall as for almost every individual program.

The overall number of NEND mutants is 6.9% *lower* at the `-O3` level (29812 mutants) than at the `-O0` level (32024 mutants). Moreover, the number of NEND mutants is lower at the `-O3` level than at the `-O0` level for almost every program. Only three programs (`cut`, `join`, and `test`) have more NEND mutants at the `-O3` level than at the `-O0` level. The Wilcoxon paired rank test shows statistically significant difference between the numbers of NEND mutants ($p < 0.05$). This suggests that mutation testing can be faster at the `-O3` level than at the `-O0` level because there are fewer NEND mutants to run at the `-O3` level, and the more optimized programs likely run faster as well.

In brief, we obtain the following answer for RQ3.2: *The relative number of both equivalent and duplicated mutants is higher at the `-O3` level than at the `-O0` level; as a result, the overall absolute number of NEND mutants is lower at the `-O3` level than at the `-O0` level (despite the overall absolute number of all generated mutants being higher at the `-O3` level than at the `-O0` level).*

3.3.3 Mutation Score

We next consider the mutation score, arguably the most important metric in mutation testing. The number of mutants is an important internal metric because it determines the time needed to perform mutation testing, but the mutation score is an external metric used

Table 3.3: The mutation score for all generated mutants and for only NEND mutants, at both `-O0` and `-O3` levels

Program	`-O0`		`-O3`	
	All	NEND	All	NEND
chmod	35.7	36.9	33.7	35.7
chown	32.7	33.8	29.8	33.3
cut	55.7	57.6	55.0	55.3
dd	32.8	33.9	31.4	31.5
du	41.6	43.4	36.2	38.0
head	17.8	18.7	15.7	17.6
join	54.5	56.6	40.0	42.1
mkdir	52.9	56.4	55.3	57.8
mv	53.8	55.5	50.6	51.4
readlink	39.5	41.0	40.7	39.8
rm	42.7	43.6	35.3	37.1
rmdir	29.4	31.3	28.3	29.7
tac	41.3	43.5	37.0	38.4
tail	31.3	32.4	24.5	26.5
test	37.6	38.7	37.3	38.8
touch	39.4	41.6	38.2	41.0
tr	59.2	61.5	59.2	59.8
wc	32.4	33.8	26.7	26.0
Overall	40.4	41.9	37.0	38.5

to compare the quality of test suites. Table 3.3 shows the mutation score values. We note two interesting comparisons.

First, *between the optimization levels*, the corresponding overall mutation score values are lower at the `-O3` level than at the `-O0` level, both for all mutants and for only NEND mutants. Moreover, this holds not only for the overall values but for *most* individual programs, again not only for all mutants but also for only NEND mutants. The Wilcoxon paired rank test shows statistically significant differences between the mutation score values at `-O0` and `-O3` levels ($p < 0.005$ for all mutants, and $p < 0.001$ for only NEND mutants).

In brief, we obtain the following answer for RQ3.3: *The mutation score values are lower at the `-O3` level than at the `-O0` level both for all mutants and for only NEND mutants.*

Second, *between all mutants and only NEND mutants*, the mutation score value for almost every program is lower for all mutants than the corresponding mutation score value for only NEND mutants (Table 3.3). For example, consider `dd`: at `-O0`, the mutation score value for all mutants (32.8%) is lower than the value for only NEND mutants (33.9%); and similarly, at `-O3`, the value for all mutants (31.4%) is lower than the value for only NEND mutants

Table 3.4: The number of generated mutants (#M), the percentages of equivalent (E%) and duplicated (D%) mutants, and the number of NEND mutants, split across mutation operators classes at the `-O0` level

Program	`-O0`															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	28	0.0	0.0	28	30	0.0	0.0	30	342	0.0	0.0	342	669	6.1	1.3	619
chown	12	0.0	0.0	12	2	0.0	0.0	2	144	0.0	0.0	144	309	4.8	0.0	294
cut	100	0.0	0.0	100	14	0.0	0.0	14	864	1.0	0.0	855	980	5.8	1.1	912
dd	370	0.0	0.0	370	148	0.0	0.0	148	1764	0.0	0.0	1764	1926	6.8	1.1	1773
du	44	0.0	0.0	44	12	0.0	0.0	12	558	0.0	0.0	558	1109	6.6	0.9	1024
head	236	0.0	0.0	236	14	0.0	0.0	14	900	0.0	0.0	900	1549	7.1	1.7	1412
join	200	0.0	0.0	200	12	0.0	0.0	12	1053	0.0	0.0	1051	1637	6.7	1.4	1503
mkdir	4	0.0	0.0	4	14	0.0	0.0	14	108	0.0	0.0	108	242	9.5	1.2	216
mv	20	0.0	0.0	20	8	0.0	0.0	8	279	0.0	0.0	279	600	5.3	0.8	563
readlink	4	0.0	0.0	4	0	N/A	N/A	0	54	0.0	0.0	54	134	5.2	0.0	127
rm	8	0.0	0.0	8	2	0.0	0.0	2	144	0.0	0.0	144	389	3.0	0.0	377
rmdir	8	0.0	0.0	8	2	0.0	0.0	2	189	0.0	0.0	189	280	8.9	3.9	244
tac	124	0.0	0.0	124	2	0.0	0.0	2	486	0.0	0.0	486	539	10.7	2.2	469
tail	296	0.0	0.0	296	54	0.0	0.0	54	1728	0.0	0.0	1728	2595	6.0	1.3	2403
test	256	0.0	0.0	256	48	0.0	0.0	48	1296	0.0	0.0	1296	1477	5.0	4.4	1336
touch	20	0.0	0.0	20	18	0.0	0.0	18	405	0.0	0.0	405	640	8.1	2.6	571
tr	428	0.0	0.4	426	50	0.0	0.0	50	1296	0.0	0.0	1296	2506	6.4	1.6	2305
wc	172	0.0	1.1	170	40	0.0	0.0	40	666	0.0	0.0	666	902	7.5	1.5	820
Overall	2330	0.0	0.1	2326	470	0.0	0.0	470	12276	0.0	0.0	12265	18483	6.5	1.6	16968

(31.5%). We do not compare here the value at `-O0` for all mutants and at `-O3` for only NEND mutants because those are not corresponding values.

There is no general relationship between the mutation score values for all mutants and only NEND mutants. Consider some mutation score value $\frac{k}{m}$, where m is the number of all generated mutants and k is the number of mutants killed among those m. If we remove (only) $e > 0$ equivalent mutants, we must get a higher $\frac{k}{m-e}$. If we remove (only) $d > 0$ duplicated mutants, and none of them are killed, we must get a higher $\frac{k}{m-d}$. If all the duplicated mutants are killed, we must get a lower $\frac{k-d}{m-d}$. In general, we remove e equivalent and d duplicated mutants, and the number of killed duplicated mutants d' is between 0 and d, so the resulting $\frac{k-d'}{m-e-d}$ must be between $\frac{k}{m-e-d}$ and $\frac{k-d}{m-e-d}$; the resulting mutation score can be higher or lower than the original mutation score.

3.3.4 Analysis Across Mutation Operators

We have discovered several relationships between the numbers of all mutants, equivalent and duplicated mutants, NEND mutants, and mutation score values, compared across different compiler optimization levels. However, the analysis so far has been across the mutants generated by *all* mutation operators. Do these relationships vary when considering mutants of each mutation operator individually? We revisit the initial questions while breaking down the numbers for each operator classes.

Table 3.5: The number of generated mutants (#M), the percentages of equivalent (E%) and duplicated (D%) mutants, and the number of NEND mutants, split across mutation operators classes at the `-O3` level

Program	-O3															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	32	0.0	25.0	24	30	0.0	0.0	30	324	8.9	17.2	239	566	10.7	6.3	469
chown	12	0.0	25.0	9	6	0.0	0.0	6	144	6.9	16.6	110	291	19.5	1.3	230
cut	116	0.0	26.7	85	102	0.0	1.9	100	1215	7.5	13.4	960	1114	1.7	3.1	1060
dd	362	0.0	23.2	278	200	7.5	3.0	179	2106	7.3	17.2	1588	2053	6.2	4.8	1825
du	88	0.0	17.0	73	30	0.0	0.0	30	684	5.7	11.7	565	880	12.1	3.7	740
head	220	0.0	12.7	192	34	0.0	0.0	34	954	5.8	12.2	781	1305	14.8	5.5	1038
join	284	2.4	30.9	189	66	0.0	0.0	66	1872	7.2	10.1	1546	1758	11.2	3.4	1500
mkdir	4	0.0	0.0	4	14	0.0	0.0	14	81	9.8	8.6	66	154	4.5	1.9	144
mv	12	0.0	25.0	9	18	0.0	5.5	17	252	5.1	19.0	191	510	13.5	4.3	419
readlink	4	0.0	50.0	2	0	N/A	N/A	0	45	6.6	11.1	37	91	2.2	3.3	86
rm	8	0.0	25.0	6	8	0.0	0.0	8	153	5.2	15.6	121	289	6.9	1.7	264
rmdir	8	0.0	37.5	5	12	0.0	0.0	12	207	5.8	8.7	177	190	2.1	8.9	169
tac	104	0.0	5.7	98	16	12.5	0.0	14	540	5.5	10.5	453	460	7.6	4.5	404
tail	348	0.0	12.6	304	94	0.0	1.0	93	2358	5.8	13.0	1912	2609	13.9	5.2	2109
test	408	0.0	19.1	330	86	0.0	0.0	86	2133	5.7	10.2	1792	2090	5.9	9.9	1758
touch	20	0.0	30.0	14	32	0.0	0.0	32	369	7.5	11.6	298	562	17.2	3.9	443
tr	396	0.0	24.4	299	208	0.4	0.0	207	1845	5.5	15.4	1457	2175	4.9	2.4	2014
wc	188	0.0	17.5	155	56	0.0	1.7	55	657	6.0	14.3	523	828	8.2	4.5	722
Overall	2614	0.2	20.3	2076	1012	1.7	1.0	983	15939	6.4	13.1	12816	17925	9.2	4.8	15394

Table 3.6: The mutation score split across mutation operator classes for all generated mutants and only NEND mutants

Program	-O0								-O3							
	AOR		LCR		ROR		ICR		AOR		LCR		ROR		ICR	
	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND	All	NEND
chmod	39.2	39.2	66.6	66.6	36.8	36.8	33.6	35.3	53.1	54.1	73.3	73.3	36.1	37.6	29.1	32.6
chown	50.0	50.0	50.0	50.0	31.9	31.9	32.3	34.0	50.0	44.4	50.0	50.0	35.4	35.4	25.7	32.1
cut	70.0	70.0	57.1	57.1	59.4	60.1	50.9	54.0	68.1	68.2	72.5	72.0	54.3	55.6	52.8	54.0
dd	35.4	35.4	48.6	48.6	35.5	35.5	28.7	30.7	33.9	29.8	38.0	39.1	36.3	37.4	25.2	27.2
du	72.7	72.7	58.3	58.3	46.5	46.5	37.6	40.2	61.3	56.1	73.3	73.3	34.0	34.8	34.0	38.7
head	14.4	14.4	7.1	7.1	21.2	21.2	16.5	17.9	13.1	11.4	14.7	14.7	16.3	17.8	15.7	18.7
join	65.5	65.5	100.0	100.0	54.4	54.5	53.0	56.6	44.7	45.5	46.9	46.9	40.3	41.9	38.7	42.6
mkdir	100.0	100.0	71.4	71.4	51.8	51.8	51.6	56.9	100.0	100.0	71.4	71.4	54.3	57.5	53.2	55.5
mv	100.0	100.0	87.5	87.5	59.5	59.5	49.1	51.5	100.0	100.0	83.3	82.3	63.1	62.3	42.1	46.3
readlink	100.0	100.0	N/A	N/A	57.4	57.4	30.6	32.2	100.0	100.0	N/A	N/A	53.3	54.0	31.8	33.7
rm	50.0	50.0	0.0	0.0	37.5	37.5	44.7	46.1	50.0	66.6	50.0	50.0	33.3	34.7	35.6	37.5
rmdir	50.0	50.0	50.0	50.0	32.8	32.8	26.4	29.5	50.0	40.0	50.0	50.0	24.6	27.1	30.0	30.7
tac	46.7	46.7	50.0	50.0	46.5	46.5	35.4	39.6	66.3	66.3	31.2	35.7	38.5	38.8	28.9	31.1
tail	53.7	53.7	11.1	11.1	30.7	30.7	29.6	31.5	45.6	46.3	23.4	22.5	24.2	24.0	22.0	25.7
test	38.2	38.2	45.8	45.8	41.2	41.2	34.0	36.1	40.6	40.3	41.8	41.8	33.4	34.7	40.4	43.3
touch	60.0	60.0	11.1	11.1	46.1	46.1	35.3	38.7	60.0	57.1	31.2	31.2	44.1	44.9	33.9	39.5
tr	80.1	80.0	52.0	52.0	60.1	60.1	55.3	59.0	85.3	83.6	37.9	38.1	59.0	60.1	56.7	59.3
wc	44.7	45.2	40.0	40.0	28.3	28.3	32.8	35.6	42.0	40.0	41.0	40.0	21.9	21.4	26.0	26.8
Overall	51.4	51.4	45.1	45.1	41.9	42.0	37.8	40.5	49.2	47.5	43.7	43.9	37.3	38.1	34.6	38.1

Tables 3.4 and 3.5 show a detailed breakdown, per operator class, of the left side of Table 3.2 and the right side of Table 3.2, respectively. Table 3.6 shows the detailed breakdown per operator class of Table 3.3. When considering the breakdown into individual operator classes, it is important to note how to treat duplicated mutants, because mutants can be duplicated across different operator classes (e.g., a mutant generated using the AOR operator can be a duplicate of a mutant generated using the ICR operator). In these tables, we compute duplicated mutants for each operator as if the only mutants that exist are the mutants generated by that operator. For example, if two mutants M_a and M_i are duplicates

Table 3.7: Relationships for mutants generated by all operators together vs. mutants generated by each operator individually

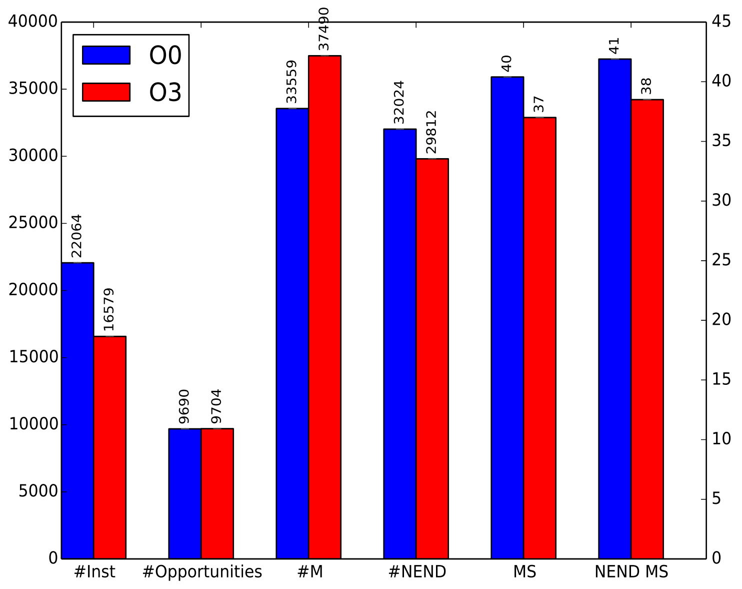
Relationship	Operators				
	All	AOR	LCR	ROR	ICR
for all generated mutants, overall #M at `-O3` > #M at `-O0`	yes	yes	yes	yes	no
for only NEND mutants, overall #NEND at `-O3` < #NEND at `-O0`	yes	yes	no	no	yes
overall E% at `-O3` ≥ E% at `-O0`	yes	yes	yes	yes	yes
overall D% at `-O3` ≥ D% at `-O0`	yes	yes	yes	yes	yes
for all generated mutants, overall mutation score at `-O3` < mutation score at `-O0`	yes	yes	yes	yes	yes
for only NEND mutants, overall mutation score at `-O3` < mutation score at `-O0`	yes	yes	yes	yes	yes

of each other, but M_a is generated by the AOR operator and M_i by the ICR operator, when considering mutants from AOR, M_a is counted as NEND and not counted as a duplicated mutant; likewise, when considering mutants from ICR, M_i is counted as NEND. The sum of all the duplicated mutants we report for each operator is lower than the number we report overall in Table 3.2, because many duplicated mutants when considering all operators together may not be duplicated when considering only one operator.

Table 3.7 summarizes the relationships across all operators and per each operator. Most relationships that hold for the mutants generated by all operators together also hold when considering only the mutants generated by each operator individually. One exception concerns the overall number of all generated mutants for the ICR operator: the overall number of all generated mutants at the `-O3` level is *higher* than the overall number of all generated mutants at the `-O0` level for all operators except ICR. Though there are a few programs where the number of generated mutants at `-O3` for ICR is higher than at `-O0` (`cut`, `dd`, `join`, `tail`, and `test`), for the majority of programs this is not the case, and these exceptions do not have many more generated mutants at the `-O3` level. The other exceptions concern the number of NEND mutants for the LCR operator and the ROR operator. Concerning the LCR operator, it does not generate many equivalent nor duplicated mutants at either optimization level. As such, because the LCR operator generates more mutants overall at the `-O3` level, it continues to have more NEND mutants at the `-O3` level as well. Concerning the ROR operator, the difference between the number of NEND mutants at the different optimization levels is relatively small (12265 at `-O0` vs. 12816 at `-O3`), and we see that there are actually only six programs that have more NEND mutants at the `-O3` level than at the `-O0` level (`cut`, `du`, `join`, `tail`, `test`, and `tr`).

In brief, we obtain the following answer for RQ3.4: *The effects of `-O0` and `-O3` levels on mutation testing are most likely due to compiler optimizations and not due to specific mutation operators.*

Figure 3.3: The overall number of instructions, mutation opportunities, generated mutants (all and NEND), and mutation score (all and NEND)



3.3.5 Visual Summary

As an overall visual summary of all our high-level results, Figure 3.3 compares the number of instructions, mutation opportunities, generated mutants (both all and NEND), and mutation score (both all and NEND) between optimization levels `-O0`, shown in (dark) blue, and `-O3`, shown in (light) red. To summarize, although there are fewer instructions in the programs compiled at the `-O3` level than at the `-O0` level, there are actually slightly more mutation opportunities at the `-O3` level, which in turn leads to more mutants generated. However, when we keep only NEND mutants, there are fewer NEND mutants at the `-O3` level. For both all mutants and only NEND mutants, the mutation score values are lower at the `-O3` level.

3.4 THREATS TO VALIDITY

Internal. Our implementation of mutation testing and the scripts for running the experiments may contain bugs. To reduce the risks, we used the well-known framework LLVM and reviewed our code and scripts to check basic functionality. We built our own scripts to run the experiments, collect results, and analyze them. We performed sanity checks on the numbers that the scripts generated, e.g., we checked that equivalent and duplicated mutants give the same results, as they should (Section 3.2.1). We also manually inspected some outliers to confirm that the results were correct.

External. The programs and tests that we used for our empirical study are a subset of all available software and may not be representative. Thus, our findings may not generalize to all software. To address this threat, we selected a total of 16 open-source programs from the widely used Coreutils distribution. We used all of the regression tests that come with each Coreutils program and specifically target it.

Construct. We use the technique proposed by Papadakis et al. to identify equivalent and duplicated mutants [88]. The technique is a heuristic and provides only a lower bound on the number of such mutants; we cannot claim we identified *all* equivalent and duplicated mutants. We also chose the basic `-O0` optimization level in LLVM as our baseline and compared it with one of the most advanced levels, `-O3`. The results could differ for other combinations of compiler optimizations, but we hypothesize that the general result holds: using higher optimization levels is beneficial, as long as equivalent and duplicated mutants are removed, and the mutation score is properly interpreted. In particular, we do not know which of the two mutation scores may better correspond to finding real faults, but the mutation scores at the `-O0` and `-O3` levels are similar, so we expect that both equally well (or equally poorly!) correspond to real faults.

CHAPTER 4: COMPARING THE SRC AND IR LEVELS

In this chapter we present an extensive comparison of mutation testing at the SRC and IR levels in the C programming language and the LLVM compiler IR. To make the comparison fair, we develop two mutation tools that implement conceptually the same operators at both levels. We also employ automated techniques to account for equivalent and duplicated mutants, and to determine hard-to-kill mutants. We carry out our study on 16 programs from the Coreutils library, using a total of 948 tests. We also perform a case study on the widely studied *Space* program. *Space* is developed by the European Space Agency and is publicly available in the Software Infrastructure Repository (*SIR*) [33].

In our study, we address the following research questions:

RQ4.1: How does the number of generated mutants differ between the SRC and IR levels?

RQ4.2: How does the number of equivalent and duplicated mutants differ between the SRC and IR levels?

RQ4.3: How does the mutation score differ between the SRC and IR levels?

RQ4.4: How do the mutation score and the number of equivalent and duplicated mutants of different mutation operator classes differ between the SRC and IR levels?

RQ4.5: Which of the two levels (SRC or IR) generates more hard-to-kill mutants?

RQ4.6: How do mutation scores at the SRC and IR levels compare with the actual bug-finding capability of test suites?

4.1 MUTATION TOOLS IMPLEMENTATION

We use two mutation tools: one tool generates mutants for `C` source code, based on Clang, and the other tool generates mutants for the code's intermediate representation (IR), based on LLVM. The following two subsections discuss the details of the implementation of each of the tools and how they get invoked by our runner to execute tests on the generated mutants.

4.1.1 Source-level Mutant Generation Tool

We implement our source (SRC)-level mutant generation tool as a source-to-source transformation tool based on Clang (version 3.8.1). Clang parses the input files and builds an abstract syntax tree (AST). We use the *LibTooling* [4] and *LibASTMatchers* [5] libraries that together enable modifying the Clang AST to perform source-to-source transformations. The mutant generation tool operates in three steps. First, we use the *llvm-cov* tool [80] to collect the code coverage achieved by the test suite; we build the code under test using Clang with the flags `-O0 -g --coverage` enabled, run the tests, and collect the coverage data. Second, we invoke our mutation tool (instead of the regular compiler) to search for candidate mutation locations in the AST that correspond to the covered lines. Then, for each of these candidates, we apply all the mutation operators that are applicable, generating one mutated source file for each mutation. Lastly, for each of the mutated source files, we compile it as usual to produce mutated executable(s) and libraries. We perform on-the-fly trivial compiler equivalence (TCE) [88] (details in Section 4.2.3) to determine if the mutant is equivalent or duplicated. If it is not, we run the tests and collect the results.

Our tool-set supports mutating multiple files. This is an essential characteristic of a mutation tool, as code is generally organized in multiple files and directories according to its functionality. For example, a significant part of the functionality used by Coreutils tools is defined in a utility directory that gets into a shared library `libcoreutils.a` that is linked to the executable. Missing on mutating code coming from `libcoreutils.a` decreases the confidence in the value of the mutation testing results we get.

4.1.2 IR-level Mutant Generation Tool

For IR-level mutant generation, we build on top of the LLVM tool we used in our study in Chapter 3. The original tool uses transformation passes in the LLVM compiler infrastructure (LLVM version 3.8.1) to generate mutants. It consists of two LLVM passes. The first pass takes as input a file containing the LLVM IR (also known as *bitcode*) and generates as output the locations that can be mutated and the mutations to apply to each location. The second pass takes as input a file with an IR and the mutation to apply, and then actually applies it. We extend the implementation of the tool with an LLVM pass that instruments the code and collects coverage at the IR instruction level, as done by some another tool for LLVM [98].

Our extended IR mutation tool mirrors the source-level mutation tool in its three steps of operation. First, we instrument the code and collect coverage per LLVM instruction. Second, we use the LLVM pass described above to generate a list of possible mutants and intersect it with the coverage info. Lastly, for each of the mutants in the list, we generate the mutated executable(s) and libraries, and perform on-the-fly TCE. If the generated mutant is not equivalent or duplicated, we run the tests and collect the results.

4.2 EXPERIMENTAL SETUP

In this section, we describe our experimental setup for comparing mutation testing at the SRC level and at the IR level. We first describe the programs and their tests we used for the evaluation, along with how we sampled the tests for smaller test suites. We then describe how we determine equivalent and duplicated mutants. We finally describe how we run the two different mutation testing tools (one for each level).

4.2.1 Evaluation Programs

For our evaluation, we use the programs from Coreutils (described in detail in Section 3.2.1). The selected subset of tools from Coreutils in this chapter is, however, different as we make more effort towards separating individual test cases. We selected 16 programs for our evaluation. We eliminated the other programs because they had too few tests, or because they had flaky tests [68].

The tests for most programs in Coreutils are manually written scripts that invoke the program multiple times, where each invocation conceptually represents a different test. Such scripts are not ideal for evaluating mutation testing. For example, the program `cut` has a test script file that contains 186 tests. If we were to execute such a test script directly on the original and mutated versions of the program, it would execute *all* 186 tests and report a failure if *any* of the 186 tests fails. Therefore, we would just know if a mutant is killed or not, but we would not get the full test-mutant matrix, i.e., we would not know for each test-mutant pair whether that test kills that mutant. If one were to use a mutation testing tool to evaluate the quality of a test suite, it is enough to know what mutants are killed by any test in the test suite. However, we want to obtain the full test-mutant matrix because it can facilitate a further analysis of mutants, e.g., computation of minimal mutant sets [15]. To get the full matrix for the programs, we manually analyzed all the test script files for the Coreutils programs used in our evaluation, and we split each long script into several shorter scripts that each runs an individual test.

We split long test scripts into shorter test scripts through a combination of automated transformations (whenever it was possible) and manual changes. To ensure that our process for splitting the test scripts does not affect the validity of the results, we executed all shorter test scripts on their respective programs to verify that each of them gives the excepted result on the original code. More precisely, executing a test on a program in Coreutils can give one of the three possible results: PASS, FAIL, or SKIP. The tests are skipped during execution when their precondition state is not established, which can happen for a number of reasons.

One reason that we commonly found for skipped tests was that they required to be run with the root privilege level. Another reason was that a few tests required the presence of more than one disk partition mounted on the file system. These tests report the SKIP result for the original program as well as for any mutant generated for the program. We did not attempt to execute tests with elevated privileges because failing mutants with a higher privilege could substantially affect the system (e.g., consider a mutant for `rm` that deletes the entire disk). Further, we inspected all tests that were getting skipped after our splitting of long test scripts into shorter test scripts. For most cases, the test was also originally skipped in the longer script due to unavailable privileges or resources, which is the correct behavior. For a few cases, the test started being skipped after our splitting. We carefully inspected the latter cases and found out that some tests were getting skipped because their setup was getting skipped—this setup usually sets some test environment variables and is performed when all tests are run by invoking `make check` from the top-most test directory; our shorter scripts do not invoke tests that way. However, the most important aspect is that the same tests are skipped consistently, and thus they do *not* affect our comparison of SRC- and IR-level mutation testing.

The number of tests found for each program also affects our selection of programs for evaluation. About half the programs have literally no tests or very few tests once skipped tests are ignored. We ignored those programs from our evaluation because the design of our experiments requires sampling from the entire test pool to form smaller test suites for each program (as described later in this section). We cannot reasonably sample from a test pool if it is already small.

4.2.2 Sampling Test Suites

When evaluating mutation testing at the SRC level and the IR level, we obtain just one mutation score for the entire test pool of a single program. However, it is difficult to compare the two different levels based solely on just one mutation score. As such, we also sample test suites from the overall test pool for each program, creating a number of smaller test suites. Specifically, from each program's test pool, we sample four different sizes of test suites: 1/2, 1/4, 1/8, and 1/16 of the test pool size. As our smallest size is 1/16 of the test pool size, our criterion then for the number of tests in the test pool for any of the programs we evaluate on is a minimum of 16, to ensure at least one test in a sampled test suite. For each size, we randomly sample the appropriate number of tests from the entire test pool to create a test suite, and we sample ten such test suites per size (with replacement across test suites). We also ensure that no two test suites are equal to each other (although they can have overlaps in tests). We use the same smaller test suites for both SRC level and IR level mutation testing. With the multiple test suites, we can draw correlations between the mutation scores at the SRC and IR levels.

4.2.3 Mutant Comparison

After generating all the mutants, we determine which mutants are equivalent and duplicated. As shown in Chapter 3, it is important to properly handle equivalent and duplicated mutants when computing mutation scores.

Once the NEND mutants are determined, one could run the entire mutation analysis only on those mutants *if* all the tests are deterministic. However, in our evaluation, we run all tests on all generated mutants as a means to find *flaky tests*. Flaky test results are unreliable (unless the same test is explored via expensive, multiple runs on the same code [37]), so we cannot easily determine if a mutant is killed or not when the test does not give deterministic results. However, we can determine that we have flaky tests by examining the results of running tests on equivalent and duplicated mutants. If a test kills an equivalent mutant, then the test must be a flaky test. Similarly, if a test kills a mutant from an equivalence class of duplicated mutants but does not kill another mutant from the same equivalence class, then the test is also flaky. By comparing the test results on equivalent and duplicated mutants, we found that tests from some programs (e.g., `join` and `uniq`) have tests that can seemingly kill equivalent mutants, while tests from some other programs (e.g., `ln`) have tests that seemingly do not equally kill all the mutants in the same class of duplicated mutants. As such, we removed several programs from our evaluation.

4.2.4 Sampling Mutants and Parallelizing Runs

We apply the two mutant-generation tools on the 16 programs we use in our evaluation. In Chapter 3, we studied mutant generation at the IR level using different optimization levels, and we found that the `-O3` optimization level is preferred. In this study, we configure both mutant-generation tools to generate mutants using the same optimization level, `-O3`.

The number of generated mutants for the Coreutils subjects is very large, exceeding 15,000 mutants in some cases, which is prohibitive to run and necessitates sampling. These numbers are much larger than those in Chapter 3 because the enhances tools mutates multiple files. Previous research [119] has shown that selective mutation based on operator selection gives similar results to random sampling. Therefore, we perform *random mutant selection* sampling 10% of the generated mutants. We then run the tests on the sampled mutants.

Table 4.1: Number of tests, generation time, number of generated, equivalent, and duplicated mutants at both levels for each program

Program	Tests	SRC							IR						
		Gen. Time (s)	#M	#E	E%	#D	D%	#NEND	Gen. Time (s)	#M	#E	E%	#D	D%	#NEND
chmod	51	15132	562	8	1.4	24	4.3	530	20173	1598	33	2.1	96	6.0	1499
dd	16	1180	276	6	2.2	20	7.2	250	21332	1250	45	3.6	23	1.8	1182
du	36	28184	718	13	1.8	26	3.6	679	8034	1693	84	5.0	47	2.8	1564
expr	86	82	53	2	3.8	1	1.9	50	13701	1357	22	1.6	71	5.2	1264
factor	31	85	36	1	2.8	1	2.8	34	251	60	6	10.0	2	3.3	52
head	85	10159	94	1	1.1	2	2.1	91	899	364	25	6.9	4	1.1	335
mkdir	36	2119	260	9	3.5	7	2.7	244	7771	889	27	3.0	23	2.6	839
readlink	159	612	294	3	1.0	16	5.4	275	6432	1082	34	3.1	36	3.3	1012
rm	60	6179	511	4	0.8	25	4.9	482	24463	1057	25	2.4	38	3.6	995
seq	37	205	98	1	1.0	6	6.1	91	3025	456	27	5.9	17	3.7	412
stat	68	445	183	5	2.7	11	6.0	167	3752	715	30	4.2	18	2.5	667
tac	52	1364	209	5	2.4	8	3.8	196	26290	535	24	4.5	2	0.4	509
tail	125	11972	303	6	2.0	10	3.3	287	5163	649	39	6.0	0	0.0	610
touch	28	2477	490	5	1.0	25	5.1	460	23435	1197	28	2.3	39	3.3	1132
unexpand	38	42	34	1	2.9	1	2.9	32	206	118	2	1.7	0	0.0	116
wc	40	1986	277	4	1.4	16	5.8	257	59190	1161	33	2.8	23	2.0	1105
Overall	948	82223	4398	74	1.7	199	4.5	4125	224117	14181	484	3.4	439	3.1	13293

The entire mutation testing process is rather time intensive, because there can be hundreds of tests running on thousands of mutants per program. Therefore, we parallelize our experiments across 16 Ubuntu 16.04.4 LTS virtual machines. For each of our two tools, we run the entire mutation testing process for a single Coreutils program on one virtual machine at a time.

4.3 RESULTS AND ANALYSIS

Table 4.1 tabulates the programs we use in our evaluation and the number of tests they have.

4.3.1 Number of Generated Mutants

Table 4.1 also shows the total number of mutants generated, the time to generate those mutants, and the number of equivalent and duplicated mutants, at each level. We can see that the *total* number of generated mutants at the IR level is about three times higher than at the SRC level. The number is also higher for the individual programs, as well as for the NEND mutants (after removing the equivalent and duplicated mutants). Looking at tables 4.2 and 4.3, we see that the overall numbers of generated mutants for AOR and LCR are similar between SRC and IR. However, the major difference comes from ROR (2335 vs. 5343 overall for NEND) and especially ICR (943 vs. 6858 overall for NEND). We perform a detailed analysis of those differences in Section 4.3.4.

We next discuss several example mutants from the function `main` of `mkdir` to illustrate cases with a one-to-one mapping between SRC and IR mutants, and other cases where one SRC mutant corresponds to multiple IR mutants, and vice versa.

One-to-one mapping between SRC and IR mutants: A mutation applied at the SRC level can in some cases be mapped directly to a mutation at the IR level. For example, in the conditional '`if (optind == argc)`', ROR at the SRC level replaces the operator '`==`' with '`>`'. Looking at the IR bitcode, we find the same mutant obtained by replacing the instruction '`icmp eq`' with '`icmp sgt`' also using ROR.

IR presents more mutation opportunities: It is intuitive to expect that IR can present more mutation opportunities than SRC knowing that one SRC statement translates into multiple IR instructions. However, that is not the only reason, and we show here an example not due to the increase in the number of instructions, but due to the more freedom an IR instruction can present in manipulating the functionality. Applying ROR on the line '`if (!change)`' negates the condition of the if statement, namely replaces it with '`if (!(!change))`'. At the IR level, the if statement translates into multiple instructions, including the following integer comparison instruction that checks for equality '`%tobool25 = icmp eq %r* %call24, null`'. The application of ROR replaces '`icmp eq`' with '`icmp neq`' and '`icmp ugt`', generating two mutants at the IR level that are semantically similar to the mutant generated at the SRC level negating the if condition. Therefore, the type of the instruction at the IR level enabled more mutants to be generated.

SRC presents more mutation opportunities: In some cases, a SRC statement can present more mutation opportunities than the corresponding IR code. For example, a line in the source of `mkdir` includes the following: '`(0400|0200|0100) | ((0400|0200|0100) >> 3) | (((0400|0200|0100) >> 3) >> 3)`'. Applying ICR at the SRC level generates multiple mutants, e.g., replacing the first occurrence of the constant '3' with '-3'. Note that there are multiple constants that ICR can mutate. Going to the corresponding IR bitcode, the constant folding compiler optimization leads to replacing the entire expression containing multiple constants with just the value '`511`'. Now there is only one constant that ICR can mutate at the IR level, so there are fewer mutants generated.

Looking at the time to generate mutants in Table 4.1, it is approximately 23 hours for SRC which is about 2.7 times lower than that of IR (62.35 hours). This is expected as the number of SRC mutants is about three times lower than the number of IR mutants.

Answering **RQ4.1**, *the number of generated mutants is much higher at the IR level than at the SRC level both before and after taking into consideration equivalent and duplicated mutants.* This difference suggests that mutation testing at the SRC level is much faster to

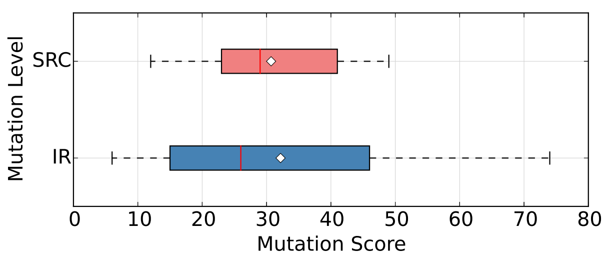


Figure 4.1: Distribution of NEND mutation score for the entire test pool at SRC and IR levels

run than mutation testing at the IR level.

4.3.2 Equivalent and Duplicated Mutants

Our experiments show that, on average, the percentage of equivalent mutants is 1.7% at the SRC level, and 3.4% at the IR level. For duplicated mutants, the percentage is 4.5% at the SRC level and 3.1% at the IR level. Note that due to sampling, those percentages can diverge from the actual numbers of equivalent and duplicated mutants. For example, if the sampling mostly chooses mutants from different equivalent classes, we end up with a low percentage of duplicated mutants. However, because our sampling is random, the variance of those numbers is not large.

Answering **RQ4.2**, *the ratios of equivalent and duplicated mutants are similar at both the SRC and IR levels*.

4.3.3 Mutation Score

The mutation score is the main metric in mutation testing, so we compare a variety of mutation scores at the SRC and IR levels. We first compare the mutation scores for only NEND mutants for the entire test pool. Second, we sample test suites from the test pool, as detailed in Section 4.2.2. For these sampled test suites, we compare the mutation score for both the set of all NEND mutants and the *refined* mutant set [40], also sometimes called removing dynamically equivalent mutants. A refined mutant set is a subset of all NEND mutants that are killed by at least one test. Similarly to previous studies [40,103,119], we use refined mutant sets to remove the mutants that *may* be equivalent to the original program.

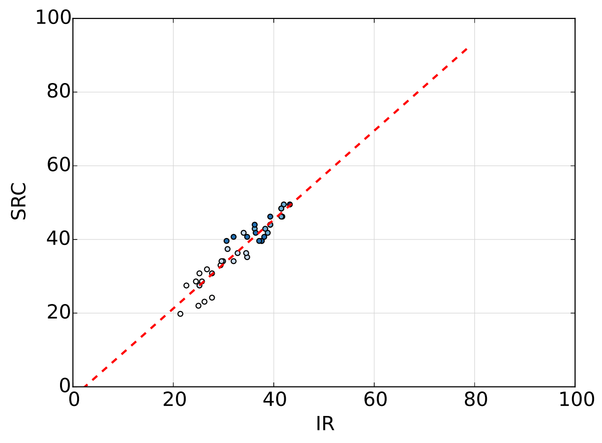


Figure 4.2: NEND mutation score for sampled test suites from `seq` at both levels

Entire Test Pool

Focusing on comparing SRC and IR levels, Figure 4.1 shows in boxplot form the distribution of the NEND mutation scores of all programs for both SRC and IR levels. The median (red line) and (unweighted) mean (white dot) mutation scores for the SRC level are similar (with SRC having slightly higher median and slightly lower mean than IR) to their respective values for the IR level. The weighted mean mutation score at the SRC level is similar to the one at the IR level, 28.6% vs. 28.4%. Examining individual values and from Figure 4.1, we can see that the mutation score for both the SRC level and the IR level are fairly similar. Indeed, the Wilcoxon paired rank test for the mutation score has a high p-value (0.74), indicating the difference of mutation scores at both levels is not statistically significant.

Sampled Test Suites

We compute the mutation score of sampled test suites of different sizes, considering both all NEND mutants and the refined NEND mutants. Our key goal here is to compare the mutation scores that the sampled test suites obtain at the SRC and IR levels: because most uses of mutation testing in research are to compare test suites, we want to know whether the SRC and IR levels agree on the quality of test suites. If the two levels largely agree, we can simply use the level that is better by other metrics (e.g., runs faster or is easier to implement). If the two levels greatly disagree, then we need to establish which one is closer to the real quality of test suites.

We use two measures of correlation to compare the SRC and IR levels: Kendall's τ_b and

Table 4.2: Number of generated, equivalent, and duplicated mutants across operator classes at the SRC level

Program	SRC															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	77	0.0	1.3	76	48	2.1	8.3	43	299	1.0	3.3	286	138	2.9	2.2	131
dd	46	0.0	0.0	46	41	9.8	0.0	37	132	1.5	11.4	115	57	0.0	3.5	55
du	89	0.0	0.0	89	58	1.7	1.7	56	425	2.4	4.2	397	146	1.4	1.4	142
expr	4	0.0	0.0	4	1	0.0	0.0	1	36	2.8	2.8	34	12	8.3	0.0	11
factor	3	0.0	0.0	3	1	0.0	0.0	1	22	4.5	0.0	21	10	0.0	0.0	10
head	11	0.0	0.0	11	5	20.0	0.0	4	54	0.0	1.9	53	24	0.0	0.0	24
mkdir	18	0.0	0.0	18	25	4.0	0.0	24	162	2.5	2.5	154	55	7.3	1.8	50
readlink	40	0.0	2.5	39	19	0.0	0.0	19	182	0.5	5.5	171	53	3.8	3.8	49
rm	45	0.0	0.0	45	57	0.0	1.8	56	287	1.0	4.9	270	122	0.8	2.5	118
seq	18	0.0	5.6	17	9	0.0	0.0	9	57	0.0	7.0	53	14	7.1	0.0	13
stat	16	0.0	0.0	16	15	0.0	0.0	15	111	3.6	7.2	99	41	2.4	0.0	40
tac	34	0.0	0.0	34	14	14.3	0.0	12	112	1.8	6.2	103	49	2.0	0.0	48
tail	33	0.0	0.0	33	27	3.7	0.0	26	177	1.1	4.5	167	66	4.5	0.0	63
touch	54	0.0	0.0	54	50	6.0	6.0	44	257	0.8	5.8	240	129	0.0	2.3	126
unexpand	2	0.0	0.0	2	1	0.0	0.0	1	20	5.0	0.0	19	11	0.0	0.0	11
wc	34	0.0	2.9	33	26	0.0	3.8	25	164	2.4	4.3	153	53	0.0	1.9	52
Overall	524	0.0	0.8	520	397	3.5	2.5	373	2497	1.6	4.9	2335	980	2.0	1.7	943

Pearson's R^2. Intuitively, τ_b checks for monotonicity: a high value would show that SRC and IR's mutation scores change in the same direction, i.e., if one increases, the other increases as well, and vice versa. Intuitively, R^2 checks for a linear relationship: a high value would show that the mutation scores at the SRC and IR levels change by a linearly proportional amount (irrespective of the direction). Using both measures together can tell us if SRC and IR mutation scores are correlated.

To visualize the correlation we want to compute between SRC and IR level mutation scores, we show Figure 4.2, a scatter plot of mutation scores computed on all NEND mutants for one sample program, `seq`. We show this scatter plot for only `seq` as most of the other programs demonstrate a similar trend. Each point in the scatter plot corresponds to a sampled test suite from `seq`. The darker the point color, the bigger the test suite is (the darkest corresponding to the entire test pool and the lightest corresponding to the size 1/16th of the test pool). The x-coordinate is the IR level mutation score, and the y-coordinate is the SRC level mutation score. The linear correlation can be seen from the figure. From this figure, we can see that for `seq`, the SRC and IR level mutation scores are very strongly correlated with one another, meaning `seq` has a high τ_b value. Furthermore, there is a strong linear correlation as well, so `seq` has a high R^2 value as well.

We compute the τ_b and R^2 values for each program for the set of all NEND mutants and the set of refined NEND mutants. The average values of τ_b (excluding `readlink` whose τ_b is undefined) and R^2 across all programs are 0.75 and 0.73, respectively, for all NEND

Table 4.3: Number of generated, equivalent, and duplicated mutants across operator classes at the IR level

Program	IR															
	AOR				LCR				ROR				ICR			
	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND	#M	E%	D%	NEND
chmod	71	0.0	0.0	71	43	0.0	0.0	43	599	3.3	6.7	554	885	1.5	6.1	834
dd	54	0.0	0.0	54	57	0.0	0.0	57	522	3.3	1.9	495	617	4.5	1.9	577
du	70	0.0	0.0	70	62	0.0	0.0	62	691	3.9	2.6	648	870	6.6	3.1	786
expr	57	0.0	0.0	57	41	0.0	0.0	41	578	1.7	6.4	531	681	1.8	4.7	637
factor	5	0.0	0.0	5	2	0.0	0.0	2	26	11.5	7.7	21	27	11.1	0.0	24
head	21	0.0	0.0	21	7	0.0	0.0	7	157	4.5	1.3	148	179	10.1	0.6	160
mkdir	47	0.0	0.0	47	20	0.0	0.0	20	346	3.2	1.7	329	476	3.4	3.2	445
readlink	46	0.0	0.0	46	32	0.0	0.0	32	450	2.9	3.6	421	554	3.8	3.4	514
rm	40	0.0	0.0	40	43	0.0	0.0	43	415	1.4	4.1	393	559	3.4	3.4	521
seq	22	0.0	0.0	22	5	0.0	0.0	5	167	1.8	1.8	161	262	9.2	5.0	225
stat	29	0.0	0.0	29	22	0.0	0.0	22	301	5.6	3.0	275	363	3.6	2.2	342
tac	31	0.0	0.0	31	23	0.0	0.0	23	221	5.9	0.0	208	260	4.2	0.0	249
tail	41	0.0	0.0	41	15	0.0	0.0	15	284	5.3	0.0	269	309	7.8	0.0	285
touch	52	0.0	0.0	52	41	0.0	0.0	41	467	2.8	4.7	433	637	2.4	2.4	608
unexpand	7	0.0	0.0	7	6	0.0	0.0	6	59	3.4	0.0	57	46	0.0	0.0	46
wc	66	0.0	0.0	66	35	0.0	0.0	35	425	2.8	3.1	400	635	3.3	1.4	605
Overall	659	0.0	0.0	659	454	0.0	0.0	454	5708	3.3	3.4	5343	7360	4.0	3.0	6858

mutants, and 0.75 and 0.73, respectively, for refined NEND mutants. We also inspected the individual values per program and noticed that there is a strong positive correlation between the mutation scores at the SRC and IR levels for most programs, with almost all of them having both a τ_b and R^2 value greater than 0.65 for both types of mutants. The exception is `factor`, which has a lower τ_b and R^2 value than all the other programs (0.54 and 0.53, respectively). We inspected `factor` and found out that `factor` has individual tests that kill a very high percentage of mutants at the SRC level, much higher than at the IR level.

Answering **RQ4.3**, *mutation scores at the SRC and IR levels are correlated and can be used as a good proxy of each other.*

4.3.4 Analysis Across Mutation Operators

So far we have analyzed mutation testing by examining the number of generated mutants, the number of NEND mutants, and the mutation score for all four classes of the mutation operators combined. To better understand how mutation testing at the SRC and IR levels compare to each other, we perform our analysis for each operator class separately, which is a finer granularity that sheds the light on whether the results are generalizable or due to a specific mutation operator.

Number of Mutants

Tables 4.2 and 4.3 show the number of mutants generated across different operator classes at the SRC and IR level, respectively. Note that the sum of NEND mutants for a given program in these tables do not add up to the total NEND shown in table 4.1 because duplicates can be across operators. Therefore, when we compute the number of NEND mutants per operator, we can end up with a larger number.

The numbers are different for IR from Chapter 3 because of the different generation process; in this chapter, we account for coverage, and we generate mutants for all covered code (not just the `tool.c` file for a given tool). In addition, we carry the analysis in this Chapter and show the results for a randomly sampled 10% subset of the total generated mutants. The total number of mutants at the SRC level is lower than at the IR level for every single operator class (524 vs. 659 for AOR, 2497 vs. 5708 for ROR, 397 vs. 454 for LCR, and 980 vs. 7360 for ICR). Similarly, the number of NEND mutants is lower at the SRC level than at the IR level (520 vs. 659 for AOR, 2335 vs. 5343 for ROR, 373 vs. 454 for LCR, and 943 vs. 6858 for ICR). For ROR, the SRC level would generate five (likely NEND) mutants replacing each relational operator by a different one from the set {`>`, `>=`, `<`, `<=`, `==`, `!=`}, whereas IR would generate nine (likely NEND) mutants replacing each operator by a different one from the set {`eq,ne,ugt,uge,ult,ule,sgt,sge,slt,sle`}. The difference of five vs. nine leads to having about twice as many ROR mutants at the IR level than at the SRC level. Inspecting the ICR mutants, IR has almost 7 times more mutants due to the large number of *getelemptr* instructions (which contributes the majority of the ICR mutants) in LLVM used to compute offsets (such as replacing array indexing `a[i]` with `a+4*i`).

Equivalent and Duplicated Mutants

The percentages of equivalent and duplicated mutants at the SRC level are similar to the corresponding ones at the IR level for each and every operator class. They are generally low, and the numbers show that both levels are similar in terms of generating mutants that are not trivially equivalent to the original program, or to one another. Note that the ratios of equivalent and duplicated mutants are affected by sampling only 10% of the mutants; in particular, the ratio of duplicated mutants is expected to be higher for all 100% of the mutants.

Table 4.4: NEND mutation score across mutation operator classes

Program	SRC				IR			
	AOR	LCR	ROR	ICR	AOR	LCR	ROR	ICR
chmod	23.7	44.2	27.3	27.5	25.4	25.6	21.1	20.1
dd	26.1	21.6	14.8	21.8	25.9	15.8	17.6	12.3
du	33.7	19.6	32.2	25.4	27.1	16.1	25.8	23.5
expr	75.0	0.0	32.4	72.7	78.9	34.1	44.4	52.7
factor	66.7	0.0	38.1	60.0	80.0	100.0	47.6	62.5
head	36.4	25.0	45.3	37.5	42.9	0.0	26.4	25.6
mkdir	55.6	12.5	34.4	32.0	48.9	40.0	27.7	24.3
readlink	28.2	10.5	25.1	16.3	21.7	18.8	14.3	9.5
rm	42.2	19.6	43.0	22.9	72.5	67.4	67.9	80.6
seq	70.6	33.3	45.3	46.2	86.4	60.0	49.1	34.2
stat	25.0	6.7	8.1	20.0	10.3	4.5	8.7	4.4
tac	38.2	8.3	19.4	20.8	19.4	4.3	19.7	7.6
tail	15.2	19.2	16.8	27.0	22.0	6.7	17.5	12.6
touch	35.2	18.2	29.6	25.4	26.9	17.1	29.6	45.4
unexpand	50.0	100.0	26.3	63.6	71.4	83.3	45.6	39.1
wc	30.3	20.0	26.1	38.5	30.3	8.6	19.8	15.0
Overall	33.3	21.2	28.9	27.4	37.5	24.2	28.0	28.1

Mutation Score

Table 4.4 shows the mutation scores of the NEND mutants across mutation operators. The overall mutation scores for all operators are similar at both the SRC and IR levels.

Answering **RQ4.4**, *the summary conclusions comparing SRC and IR for all operators combined apply to every single mutation operator and therefore are generalizable.*

4.3.5 Hard-to-Kill Mutants

So far the SRC level mutation looks better than the IR level mutation, because SRC level generates fewer NEND mutants but has similar mutation scores. However, these scores could be affected by redundant mutants [15] that do not add any value to the evaluation of test suites. These mutants can misleadingly inflate the mutation score, e.g., some mutants may be easy to kill by any test. To address this issue, we compare hard-to-kill mutants at the SRC and IR levels. Following Gopinath et al. [40], we compute two sets of mutants that can approximate hard-to-kill mutants: the *minimal set of mutants* and the *surface set of mutants*[1]. Both sets are defined using dynamically subsuming mutants: a mutant m

[1]There is some terminology mismatch: what Gopinath et al. [41] define as "surface" mutants is what Ammann et al. [15] define as "minimal" mutants, so what Gopinath et al. [41] call "minimal" should have

Table 4.5: Number of minimal mutants and surface mutants

Program	#Minimal		#Surface	
	SRC	IR	SRC	IR
chmod	9	8	10	11
dd	4	6	4	7
du	13	14	14	15
expr	7	29	8	33
factor	2	2	2	2
head	6	9	9	13
mkdir	5	7	6	11
readlink	2	5	6	9
rm	12	6	13	7
seq	6	14	7	16
stat	4	3	4	4
tac	3	5	5	9
tail	9	8	12	9
touch	12	9	12	10
unexpand	3	5	5	6
wc	7	9	8	9
Overall	104	139	125	171

subsumes another mutant m' for a test pool T if every test from T that kills m also kills m'; the subsuming mutant m is a higher quality mutant that is harder to kill. Given a set of mutants, a surface mutant set is a maximal subset that has no subsuming mutants, with subsumption computed over the entire test pool. Further, a minimal mutant set is computed the same as the surface mutant set, except subsumption is computed over a minimal test suite (i.e., a test suite that is a subset of the test pool, has the same mutation score as the entire test pool, and if one test is removed, the mutation score drops).

Table 4.5 shows the number of minimal and surface mutants computed for each program. As expected, the number of minimal mutants is never greater than the number of surface mutants. However, the difference is very small; the number of minimal mutants is similar to the number of surface mutants.

Comparing SRC and IR, we see that the numbers of minimal mutants are similar. The same applies for surface mutants. We can compute τ_b and R^2 values for minimal and surface mutants between SRC and IR similarly to how we computed for mutation scores between the two levels, but with minimal/surface mutant counts instead. The values of τ_b are 0.32 and 0.36 and those of R^2 are 0.01 and 0.00 for the minimal and surface sets of NEND mutants,

been called differently, e.g., "doubly minimal". We follow the terminology from Gopinath et al. [40] as it is a more recent paper.

respectively. The values of τ_b are moderate, showing a correlation between SRC and IR. However the values of R^2 are low.

Answering **RQ4.5**, *the SRC and IR levels generate a similar number of hard-to-kill mutants.*

4.4 CASE STUDY WITH REAL FAULTS

So far we have found that SRC and IR are good proxies for each other, with the mutation score of SRC being lower than that of IR on average. Naturally, the next question that arises is this: which of the two levels has a mutation score that is closer to the actual bug-finding capability of the test suite?

To answer this question, we conduct a case study on *Space*, an interpreter of an array definition language (ADL) developed by the European Space Agency [1]. We use the version `2.0`, most recent available version in SIR [7] at the time. It comes with a test suite of 13496 test cases and with 33 documented real faults. (The total number of versions documenting real faults that come with *Space* is 38, but only 33 of them are not equivalent to the original program for the given test suite i.e., they are not detected by the given test suite.)

4.4.1 Setup

We generate SRC and IR mutants for *Space*, and we run the entire test suite for the SRC mutants, the IR mutants, and the faulty versions. Then, we compare the output (standard output and standard error) of each test case to that we obtain when executing the test case on the original program (we compute the checksum of the outputs to compare them). If the output is the same, we consider the test to be passing for the corresponding mutant/fault; otherwise we count it as failing. A test failing for a given mutant/fault means that the test was able to detect it.

Following Andrews et al. [17], we generate 5000 test suites of randomly selected test suites of size 100 each. We then calculate for each test suite S the mutation detection ratio Am(S) defined as the number of mutants killed by S divided by the total number of mutants generated, and Af(S) as the number of faults detected by S divided by the total number of faults we have (33 faults). Then for each test suite S, we compare Am(S) of SRC and Am(S) of IR to Af(S).

4.4.2 Results

We generate a total of 9319 SRC NEND mutants and 23935 IR NEND mutants. The average of Af(S) - Am(S) for SRC is 0.013497, which is smaller, in absolute value, than the average Af(S) - Am(S) for IR (-0.057248). The average of Af(S) - Am(S) for SRC we obtain is similar to the one reported in [17], which increases our confidence in our results; despite the difference in our tool implementation and the operators used, we still obtain similar results on *Space* as those reported in the literature.

Answering **RQ4.6**, *mutation scores at the SRC level are somewhat closer to the actual bug-finding capability of the test suites than the mutation scores at the IR level.*

4.5 THREATS TO VALIDITY

Our results may not generalize to all software because the programs we chose for our evaluation may not be representative. Our programs are a subset of all available software. To address this threat, we used Coreutils, which is commonly used in previous research. We chose a total of 16 programs from Coreutils, i.e., all programs that had a non-trivial number of tests and had tests that were not flaky. For each program, we used all its tests.

In our evaluation, we use four classes of mutation operators at each level. The general results obtained could be specific to those operators and may not generalize to other classes of operators. In fact, repeating our analysis by each class of the mutation operators already shows that the general conclusions can be influenced by some class and need not be similar for each and every class. To determine equivalent and duplicated mutants, we use the trivial compiler equivalence (TCE) [88] and comparison of results for refined mutants. While TCE finds mutants that are definitely equivalent and duplicated, it only gives a lower bound on the actual number of equivalent mutants; in contrast, refined mutants give only an upper bound on the actual number of equivalent mutants. The true number of equivalent mutants is in between these bounds and could affect our findings of the mutation score. However, detecting all equivalent mutants is an undecidable problem [22], so we use both NEND and refined mutants to compare SRC and IR level mutation.

The results of our case study may not be generalizable as we use only one subject. To mitigate that, we conduct our experiments on the widely used and well documented *Space* program. Previous research [17] has built their key conclusion solely on *Space*, and we follow the same methodology.

CHAPTER 5: RELATED WORK

The work presented in this dissertation is related to several topics in approximate computing and mutation testing.

5.1 APPROXIMATE COMPUTING

Approximate computing is an emerging area of research focusing on trading off inaccurate results for performance gains (e.g., for time reduction or less energy usage). Some approximate computing techniques involve approximate hardware [65, 75], data types [94], sampling [10, 66], or code perforation [76, 104], all of which obtained significant performance with tolerable errors in specific domains. However, most of the existing work in approximate computing does not make explicit connections to software testing research. While a recent position paper argues in favor of using approximate computing to improve various software testing tasks [38], the work presented in this dissertation shows that approximate transformations are indeed useful in mutation testing.

Researchers also proposed sensitivity profilers [25,75,76,92,111,114], which transform code, run it using representative input/output pairs, compare any differences, and suggest which parts of computations are approximable. Like sensitivity profiling, our approach transforms code and runs them on a set of tests, but our goal is different in several ways: (1) we study approximate transformations for mutation testing and compare with conventional mutation operators, (2) we execute programs on finer-grained unit tests, not coarse-grained integration tests, and (3) our results provide hints for improving tests, not just code.

5.2 MUTATION TESTING

Mutation testing has been widely-studied for decades [31,120]; Jia and Harman [56] provide a thorough survey. Multiple techniques were developed for mutation testing at different levels and for different languages [6, 17, 41, 46, 55, 96, 98]. Many tools were also introduced for multiple programming languages, e.g., including `C` [30, 55], `C++` [61], Java [59, 70, 71, 96], and others [23, 26, 51].

5.2.1 Approximate transformations

Many optimizations have been developed to speed up mutation testing, including mutant schemata [112], weak mutation [89], and higher-order mutation [55]. Researchers have also proposed new mutation operators for different domains and use cases, such as for GUI-based applications [12,87], embedded systems [110], class diagrams [42], Android applications [113], or fault-localization tasks [52]. We are the first to study approximate transformations in the context of mutation testing [47].

Researchers have studied how to improve the efficiency of mutation testing by techniques to only use the mutants that are hard-to-kill and representative of all mutants. Some heuristics for finding hard-to-kill mutants include minimal mutant analysis [15,41], static analysis [90], or use of historical data [53]. Offutt et al. [83, 86] empirically found the set of sufficient operators, operators whose generated mutants are representative of mutants generated by the other operators, and others have extended this idea to various languages and paradigms, like concurrent code [39].

While these projects have the goal to improve the efficiency of mutation testing, that is not the goal of our research on introducing approximate transformations as mutation operators. We are focused on improving the *quality* of mutation testing by utilizing new mutation operators that give different insights into improving the test suite. We do, however, use the established existing techniques to evaluate how effective approximate transformations are compared against conventional mutation operators. We show, using minimal mutant analysis, that approximate transformations generate mutants that end up in the minimal mutants set, suggesting that they are harder to kill than other mutants. Through selective mutation analysis, we find that approximate transformations are not subsumed by conventional mutation operators.

5.2.2 Compiler optimizations

Substantial progress has been made toward turning mutation testing into a broadly applicable and fully automated approach; several tools have been created explicitly for this purpose. The most related to our work are two tools that operate on the LLVM IR, namely Sen and Sousa's testing framework for automated mutant generation for transaction level modeling [107] and Schulte's `llvm-mutate` tool (which uses a different set of mutation operators than what we use) [98, 99]. However, their previous work did not study the effect of the LLVM's compiler optimizations on mutation testing (including the number of generated mutants, duplicated and equivalent mutants, and the mutation score).

Rajan et al. studied the effect of program transformations on code coverage, specifically MC/DC [49, 91]. In their study, Rajan et al. used mutation testing as an enabling method but did so without regard for compiler optimization levels.

To the best of our knowledge, no prior work to ours [46] examined the impact of the compiler optimization level on mutation testing (including the impact on equivalent and duplicated mutants and especially the impact on mutation score) as our study does. However, ours is not the first study to examine the effects of compiler optimizations and other transformations on advanced testing and verification tools. Most recently, Dong et al. investigated the interactions between compiler optimizations and symbolic execution [34]. They found that the same transformations that speed up concrete execution can negatively impact symbolic execution, especially in combination. In contrast, we find that compiler optimizations at the highest level not only can speed up program execution but also can substantially help in mutation testing.

5.2.3 The problem of equivalent mutants

Despite the progress made on increasing the applicability of mutation testing, the approach still suffers from a number of issues. One key issue is the *mutant equivalence problem*, i.e., determining which mutants are semantically equivalent to the original program. (There are variations in the definition based on the scope of testing, e.g., Ellims et al. suggested a “resource-aware” view of mutants that encompasses memory and time usage as well as functional output [35].) Budd and Angluin first noted that determining mutant equivalence automatically is generally undecidable [22]. However, a number of heuristics have been developed for identifying equivalent and duplicated mutants in some cases.

The foundational work was done by Baldwin and Sayward in their study on the use of compiler optimizations to determine mutant equivalences [20]. Using established compiler optimization techniques, they attempted to identify equivalent mutants by either “optimizing” or “de-optimizing” the produced programs and comparing them to the original. Offutt and Pan created a new approach to the problem by formulating the question of equivalence as a constraint satisfaction problem [84, 85]. In their approach, constraints are generated through analysis of the mutant’s path conditions, and empirical evaluations showed that this approach tended to be more powerful than the compiler optimization technique [82].

Papadakis et al. presented *trivial compiler equivalence*, a technique that compares a mutant’s machine code to that of its progenitor program to determine whether or not it is equivalent [88]. This technique was already broadly used in the field of compiler optimizations to determine where optimizations had yielded no improvement or other change.

In contrast to the technique proposed by Papadakis et al. that aims to find *definitely* equivalent mutants, several studies have proposed heuristics to help identify mutants that are *likely* (non-)equivalent. Schuler et al. proposed two such techniques for ranking mutants based on code coverage and dynamically inferred invariants [96, 97]. Grün et al. defined an impact function characterizing the difference between a mutant's execution and the original program's execution [43]; they reported that mutants with lower impact were more likely to be equivalent.

In a similar spirit to these heuristics, Harman et al. developed a technique for equivalence detection using program slicing [48, 50]. Their method does not automatically detect equivalent mutants after generation but does reduce their number during generation, and it also assists in the manual process of equivalence analysis by simplifying the program to a minimal state representing at least a partial answer to the equivalence question. Adamopoulos et al. proposed a different approach that uses genetic algorithms wherein mutants are evaluated using a fitness function that has a much lower value for equivalent mutants [9]. Their approach allows mutants to evolve alongside the test suites that support the program while reducing the number of equivalent mutants generated.

5.2.4 Comparing SRC and IR

The focus of Chapter 4 is on the comparison of two mutation testing tools for programs that start from a single programming language. Multiple studies [28, 40, 105] compared different mutation testing tools for `Java` programs. Our work is most similar to the work by Gopinath et al. [40]. In their work, they compared three mutation testing tools that work on `Java` applications. Two of the tools used in the study generated mutants on the `Java` byte-code (effectively the IR for `Java`) while the remaining tool generated mutants on the `Java` source code. As such, the authors were able to evaluate the effects of generating mutants at different levels, SRC vs. IR, and they found that the level at which mutants are generated does not significantly affect the mutation score. However, they used the tools out-of-the-box and did not control for mutation operators. In our study, we build mutant generation tools that use the same operators at both levels, allowing for a fairer comparison of the effects of mutants generated at different levels. Furthermore, our evaluation is on the `C` programming language. Our findings are similar to theirs in terms of mutation score, but we also study the number of equivalent and duplicate mutants, the breakdown per operator, and multiple types of mutation scores at more depth.

Selective mutation, where only a subset of mutation operators are used for evaluation, is another area of research for mutation testing. Most selection methods work by comparing the mutants generated by different mutation operators to identify a subset of the operators that produce high quality mutants [21, 39, 83, 106]. Offutt et al. [83] introduced the concept of sufficient mutation operators where they showed that using only a small number of mutation operators is sufficient to access the quality of a test suite. They performed their study for Fortran programs. Barbosa et al. [21] conducted a similar study for C programs. In both cases, the identified sufficient set of operators was similar. Gligoric et al. [39] addressed the same issue for concurrent programs where a test may have to be executed on multiple thread schedules for each mutant. For concurrent mutation operators, operator-based selection produced slightly better results compared to random mutant selection. Smith and Williams [106] evaluated the mutation operators in MuJava and concluded that the best choice of mutation operators may depend on the type of application being tested. Zhang et al. [119] compared selective mutation based on operators with random mutant selection and concluded that random mutant selection can also give similar results. A more previous study [118] combined operator-based mutant selection with random selection by sampling mutants from selected operators and reported that the combination is also effective. We compare mutation testing at the two different levels while breaking down the results based on mutation operator to see if any specific operator has a bigger effect on one level or the other. We find that for the comparison of SRC vs. IR, the results for different operators somewhat differ but are largely similar.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

In Chapter 2, we propose approximate transformations as mutation operators, and we compare them with conventional mutation operators. Specifically, we integrated loop perforation and precision degradation—common approximate transformations by the approximate computing community—into an existing mutation testing framework, and we compared and analyzed the quality of those transformations when used as mutation operators. Our results show that approximate transformations generate mutants that are not subsumed by mutants generated by conventional mutation operators. Our qualitative analysis of a number of killed and surviving approximate transformations uncovered several code patterns that developers could use to enhance their test suites. The surviving mutants inspired proposing better testing practices and helped us submit 11 pull requests to fix bad tests.

In Chapter 3, we present a study of the effects of compiler optimizations, which are widely used semantics-preserving transformations aimed at improving program performance, on mutation testing. While mutation testing and compiler optimizations are two well-studied approaches, they are seldom used together. Our study aims to find new opportunities that enhance the effectiveness and application of mutation testing by leveraging modern compiler infrastructures. Specifically, we target LLVM, a popular compiler infrastructure that supports multiple programming languages. Our evaluation uses 16 Coreutils programs. Some of the findings about the number of mutants and the mutation scores on optimized and unoptimized programs surprised us. The overall conclusion is that mutation testing can use very high optimization levels, but one should remove equivalent and duplicated mutants, and one should carefully interpret the overall mutation score. Note that our conclusion about the mutation score views mutation testing only as a means to evaluate test suites and does not necessarily aim to provide guidance for how to generate new tests to kill more mutants; indeed, it would be hard for a human to reason about the changes made to the optimized program and successfully construct test inputs that could kill the mutant.

In Chapter 4, we present an extensive study that compares mutation testing at the SRC and IR levels. While mutation testing has been performed before at the IR level, no study has been performed to compare the performance and quality of mutation testing at both levels. We perform our study on 16 applications from Coreutils programs with 948 tests. To ensure a fair comparison between the two levels, we implement our own mutation tools for SRC and IR using the same mutation operators. Yet, we find that the number of mutants generated and the mutation score can differ between the two levels. This difference highlights the fact that the mutations cannot be simply mapped between the two levels. Our results show that it is more economical to perform mutation at the SRC level. It generates fewer NEND mutants, allowing mutation testing at the SRC level to run faster as there are fewer mutants to run on. We also find that mutants generated at both levels are of similar quality. They also produce similar mutation scores that are highly correlated. Moreover, the mutation score of a given test suite at the SRC level is closer to the bug-finding capability of the test suite than the mutation score at the IR level. We believe that the better performance, the better reflection of real bug-finding capability, and the ability to reason about the generated mutants make mutation testing at the SRC level more attractive, especially considering that the results of mutation testing seem to be quite similar between the two levels.

6.2 FUTURE WORK

In this section, we outline a number of potential lines for future work that can be built as a follow up on our findings in this dissertation.

In Chapter 2, we have proposed and evaluated the use of approximate transformations as mutation operators. An idea that can be explored in the future is the dual direction. Namely, how can mutation operators benefit approximate computing? Can we use mutation operators to identify approximable code sections? Specifically, one could use the surviving mutants from performing mutation testing to target the application of approximate transformations to specific portions of the code that we find potentially approximable. One of our main findings in Chapter 2 is discovering a new way of interpreting mutation testing results; the reason for getting surviving mutants may be *approximable code.* So far in our research, we have relied on manual efforts to identify approximable code cases. Future work could automate this task. Such a technique would be very valuable for filtering out the surviving mutants due to approximable code and adjusting the mutation score accordingly.

In Chapter 3, we studied the effects of compiler optimizations on mutation testing at the LLVM IR level. It would be interesting in the future to look into the Java world, and study how the different optimizations applied by the (JIT) compiler affect mutation testing. We hypothesize in Chapter 3 that many mutants at the `-O3` level are harder to kill, but evaluating this hypothesis would require a very large number of tests, likely automatically generated, and is out of scope for this dissertation. We leave it as future work to inspect why the mutation score values are lower at the `-O3` level.

In Chapter 4, we compared mutation testing at the SRC and IR level, and we find that SRC is more efficient. This result is based on our current tool for performing IR mutations.

We think that a more sophisticated mutation tool for IR can have the potential to match or outperform the SRC tool and hence open the door for having only one mutation tool that supports multiple source languages. For example, implementing a custom mutation for the *getelemptr* LLVM bitcode instruction that ignores certain types of mutations can be one of the ways to decrease the large number of mutants generated at IR.

In sum, mutation testing has been studied for several decades and is widely used in software testing research. This dissertation explores several design decisions about mutation testing and shows that the community can still learn new lessons about mutation testing and how to improve it. We hope that the work in this dissertation is a step closer to bringing mutation testing into practice.

REFERENCES

[1] C object biographies. `http://sir.unl.edu/portal/bios/space.php`.

[2] Commons-io. `https://github.com/apache/commons-io`.

[3] Jblas. `https://github.com/mikiobraun/jblas`.

[4] Libtooling. `http://clang.llvm.org/docs/LibTooling.html`.

[5] Matching the Clang AST. `http://clang.llvm.org/docs/LibASTMatchers.html`.

[6] Real world mutation testing. `http://pitest.org`.

[7] Software-artifact infrastructure repository. `http://sir.unl.edu/portal/index.php`.

[8] Vectorz. `https://github.com/mikera/vectorz`.

[9] Adamopoulos, K., Harman, M., and Hierons, R. M. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO* (2004), pp. 1338–1349.

[10] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., and Stoica, I. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys* (2013), pp. 29–42.

[11] Ahmed, I., Gopinath, R., Brindescu, C., Groce, A., and Jensen, C. Can testedness be effectively measured. In *FSE* (2016), pp. 547–558.

[12] Alegroth, E., Gao, Z., Oliveira, R., and Memon, A. Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study. In *ICST* (2015), pp. 1–10.

[13] Alipour, M. A., Shi, A., Gopinath, R., Marinov, D., and Groce, A. Evaluating non-adequate test-case reduction. In *ASE* (2016), pp. 16–26.

[14] Allamanis, M., Barr, E. T., Just, R., and Sutton, C. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516* (2016).

[15] Ammann, P., Delamaro, M. E., and Offutt, J. Establishing theoretical minimal sets of mutants. In *ICST* (2014), pp. 21–30.

[16] Ammann, P., and Offutt, J. *Introduction to Software Testing.* Cambridge University Press, 2008.

[17] Andrews, J., Briand, L., and Labiche, Y. Is mutation an appropriate tool for testing experiments? In *ICSE* (2005), pp. 402–411.

[18] Andrews, J. H., and Alipour, A. MutGen tool. `https://github.com/alipourm/cmutate`.

[19] ASM. `http://asm.ow2.org/`.

[20] Baldwin, D., and Sayward, F. Heuristics for determining equivalence of program mutations. Tech. rep., DTIC Document, 1979.

[21] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. Toward the determination of sufficient mutant operators for C. *STVR 11*, 2 (2001), 113–136.

[22] Budd, T. A., and Angluin, D. Two notions of correctness and their relation to testing. *Acta Informatica 18*, 1 (1982), 31–45.

[23] Budd, T. A., Lipton, R. J., DeMillo, R., and Sayward, F. The design of a prototype mutation system for program testing. In *AFIPS* (1899), p. 623.

[24] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), pp. 209–224.

[25] Carbin, M., and Rinard, M. C. Automatically identifying critical input regions and code in applications. In *ISSTA* (2010), pp. 37–48.

[26] Chan, W., Cheung, S. C., and Tse, T. Fault-based testing of database application programs with conceptual data model. In *QSIC* (2005), pp. 187–196.

[27] Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Analysis and characterization of inherent application resilience for approximate computing. In *DAC* (2013), p. 113.

[28] Delahaye, M., and Du Bousquet, L. A comparison of mutation analysis tools for Java. In *QSIC* (2013), pp. 187–195.

[29] Delamaro, M. E., and Maldonado, J. C. Proteum tool for mutation testing of C programs. `https://github.com/magsilva/proteum`.

[30] Delamaro, M. E., and Maldonado, J. C. Proteum—A tool for the assessment of test adequacy for C programs. In *PCS* (1996), pp. 79–95.

[31] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[32] Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., OReilly, U.-M., and Amarasinghe, S. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices* (2015), vol. 50, pp. 379–390.

[33] Do, H., Elbaum, S., and Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10*, 4 (2005), 405–435.

[34] Dong, S., Olivo, O., Zhang, L., and Khurshid, S. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE* (2015), pp. 205–215.

[35] Ellims, M., Ince, D., and Petre, M. The Csaw C mutation tool: Initial results. In *TAIC PART* (2007), pp. 185–192.

[36] Foundation, F. S. Coreutils – GNU core utilities. `http://www.gnu.org/software/coreutils/coreutils.html`.

[37] Gligoric, M., Jagannath, V., and Marinov, D. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *ICST* (2010), pp. 55–64.

[38] Gligoric, M., Khurshid, S., Misailovic, S., and Shi, A. Mutation testing meets approximate computing. In *ICSE NIER* (2017), pp. 3–6.

[39] Gligoric, M., Zhang, L., Pereira, C., and Pokam, G. Selective mutation testing for concurrent code. In *ISSTA* (2013), pp. 224–234.

[40] Gopinath, R., Ahmed, I., Alipour, M. A., Jensen, C., and Groce, A. Does choice of mutation tool matter? *SQJ* (2016), 1–50.

[41] Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., and Groce, A. Measuring effectiveness of mutant sets. In *Mutation* (2016), pp. 132–141.

[42] Granda, M. F., Condori-Fernández, N., Vos, T. E. J., and Pastor, O. Mutation operators for UML class diagrams. In *CAiSE* (2016), pp. 325–341.

[43] Grün, B. J., Schuler, D., and Zeller, A. The impact of equivalent mutants. In *Mutation* (2009), pp. 192–199.

[44] Gupta, V., Mohapatra, D., Park, S. P., Raghunathan, A., and Roy, K. Impact: imprecise adders for low-power approximate computing. In *ISLPED* (2011), pp. 409–414.

[45] Han, J., and Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS* (2013), pp. 1–6.

[46] Hariri, F., Shi, A., Converse, H., Khurshid, S., and Marinov, D. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE* (2016), pp. 105–115.

[47] Hariri, F., Shi, A., Legunsen, O., Gligoric, M., Khurshid, S., and Misailovic, S. Approximate transformations as mutation operators. In *ICST* (2018).

[48] Harman, M., Hierons, R., and Danicic, S. *Mutation Testing for the New Century.* Kluwer Academic Publishers, 2001.

[49] Heimdahl, M. P., Whalen, M. W., Rajan, A., and Staats, M. On MC/DC and implementation structure: An empirical study. In *DASC* (2008), pp. 5.B.3–1–5.B.3–13.

[50] Hierons, R., Harman, M., and Danicic, S. Using program slicing to assist in the detection of equivalent mutants. *STVR 9*, 4 (1999), 233–262.

[51] Hong, S., Kwak, T., Lee, B., Jeon, Y., Ko, B., Kim, Y., and Kim, M. MUSEUM: debugging real-world multilingual programs using mutation analysis. *IST 82* (2017), 80–95.

[52] Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., and Kim, M. Mutation-based fault localization for real-world multilingual programs (t). In *ASE* (2015), pp. 464–475.

[53] Inozemtseva, L., Hemmati, H., and Holmes, R. Using fault history to improve mutation reduction. In *ESEC/FSE* (2013), pp. 639–642.

[54] Jia, Y. Milu: A higher order mutation testing tool. https://github.com/yuejia/Milu.

[55] Jia, Y., and Harman, M. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART* (2008), pp. 94–98.

[56] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *TSE 37*, 5 (2011), 649–678.

[57] Just, R. The major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA* (2014), pp. 433–436.

[58] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. Are mutants a valid substitute for real faults in software testing? In *FSE* (2014), pp. 654–665.

[59] Just, R., Schweiggert, F., and Kapfhammer, G. M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE* (2011), pp. 612–615.

[60] Kurtz, B., Ammann, P., Offutt, J., Delamaro, M. E., Kurtz, M., and Gökçe, N. Analyzing the validity of selective mutation with dominator mutants. In *FSE 2016* (2016), pp. 571–582.

[61] Kusano, M., and Wang, C. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *ASE* (2013), pp. 722–725.

[62] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *PLDI* (2012), pp. 193–204.

[63] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO* (2004), pp. 75–86.

[64] Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., and Marinov, D. An extensive study of static regression test selection in modern software evolution. In *FSE* (2016), pp. 583–594.

[65] Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. G. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS* (2011), pp. 213–224.

[66] Lou, L., Nguyen, P., Lawrence, J., and Barnes, C. Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *SIGGRAPH 35*, 5 (2016), 153:1–153:14.

[67] Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., and Zhang, L. How does regression test prioritization perform in real-world software evolution? In *ICSE* (2016), pp. 535–546.

[68] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. An empirical analysis of flaky tests. In *FSE* (2014), pp. 643–653.

[69] Ma, Y.-S., Offutt, J., and Kwon, Y. R. MuJava: An automated class mutation system. *STVR 15*, 2 (2005), 97–133.

[70] Ma, Y.-S., Offutt, J., and Kwon, Y.-R. MuJava: a mutation system for Java. In *ICSE* (2006), pp. 827–830.

[71] Madeyski, L., and Radyk, N. Judy-A mutation testing tool for Java. *IET software 4*, 1 (2010), 32–42.

[72] Marinescu, P. D., and Cadar, C. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE* (2012), pp. 716–726.

[73] Mathur, A. P. Performance, effectiveness, and reliability issues in software testing. In *COMPSAC* (1991), pp. 604–605.

[74] Misailovic, S. Exploring the Effectiveness of Loop Perforation for Quality of Service Profiling. Tech. rep., MIT, 2010.

[75] Misailovic, S., Carbin, M., Achour, S., Qi, Z., and Rinard, M. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA* (2014), pp. 309–328.

[76] Misailovic, S., Sidiroglou, S., Hoffmann, H., and Rinard, M. Quality of service profiling. In *ICSE* (2010), pp. 25–34.

[77] Mitra, S., Gupta, M. K., Misailovic, S., and Bagchi, S. Phase-aware optimization in approximate computing. In *CGO* (2017).

[78] of Illinois, U. Clang: A C family language frontend for LLVM. `http://clang.llvm.org/`.

[79] of Illinois, U. The LLVM compilation infrastructure. `http://llvm.org/`.

[80] of Illinois, U. llvm-cov - emit coverage information. `https://llvm.org/docs/CommandGuide/llvm-cov.html`.

[81] of Illinois, U. Tail call optimization. `http://llvm.org/docs/CodeGenerator.html#tail-call-optimization`.

[82] Offutt, A. J., and Craft, W. M. Using compiler optimization techniques to detect equivalent mutants. *STVR 4*, 3 (1994), 131–154.

[83] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. An experimental determination of sufficient mutant operators. *TOSEM 5*, 2 (1996), 99–118.

[84] Offutt, A. J., and Pan, J. Detecting equivalent mutants and the feasible path problem. In *COMPASS* (1996), pp. 224–236.

[85] Offutt, A. J., and Pan, J. Automatically detecting equivalent mutants and infeasible paths. *STVR 7*, 3 (1997), 165–192.

[86] Offutt, A. J., Rothermel, G., and Zapf, C. An experimental evaluation of selective mutation. In *ICSE* (1993), pp. 100–107.

[87] Oliveira, R. A. P., Algroth, E., Gao, Z., and Memon, A. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *Mutation* (2015), pp. 1–10.

[88] Papadakis, M., Jia, Y., Harman, M., and Le Traon, Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE* (2015), pp. 936–946.

[89] Papadakis, M., and Malevris, N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Control 19*, 4 (2011), 691–723.

[90] Patrick, M., Oriol, M., and Clark, J. A. Messi: Mutant evaluation by static semantic interpretation. In *ICST* (2012), pp. 711–719.

[91] Rajan, A., Whalen, M. W., and Heimdahl, M. P. The effect of program and model structure on MC/DC test adequacy coverage. In *ICSE* (2008), pp. 161–170.

[92] Roy, P., Ray, R., Wang, C., and Wong, W. F. Asac: Automatic sensitivity analysis for approximate computing. In *LCTES* (2014), vol. 49, pp. 95–104.

[93] Rubio-González, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonious: Tuning assistant for floating-point precision. In *SC* (2013), p. 27.

[94] Sampson, A., Dietl, W., Fortuna, E., and Gnanapragasam, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI* (2011), pp. 164–174.

[95] Schkufza, E., Sharma, R., and Aiken, A. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices 49*, 6 (2014), 53–64.

[96] Schuler, D., and Zeller, A. Javalanche: Efficient mutation testing for Java. In *ESEC/FSE* (2009), pp. 297–298.

[97] Schuler, D., and Zeller, A. (Un-) Covering equivalent mutants. In *ICST* (2010), pp. 45–54.

[98] Schulte, E. llvm-mutate. http://eschulte.github.io/llvm-mutate/.

[99] Schulte, E. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, 2014.

[100] Shi, A., Gyori, A., Gligoric, M., Zaytsev, A., and Marinov, D. Balancing trade-offs in test-suite reduction. In *FSE* (2014), pp. 246–256.

[101] Shi, A., Gyori, A., Legunsen, O., and Marinov, D. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST* (2016).

[102] Shi, A., Yung, T., Gyori, A., and Marinov, D. Comparing and combining test-suite reduction and regression test selection. In *FSE* (2015), pp. 237–247.

[103] Siami Namin, A., Andrews, J. H., and Murdoch, D. J. Sufficient mutation operators for measuring test effectiveness. In *ICSE* (2008), pp. 351–360.

[104] Sidiroglou, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE* (2011), pp. 124–135.

[105] Singh, P. K., Sangwan, O. P., and Sharma, A. A study and review on the development of mutation testing tools for Java and Aspect-J programs. *International Journal of Modern Education and Computer Science 6*, 11 (2014), 1.

[106] Smith, B. H., and Williams, L. An empirical evaluation of the MuJava mutation operators. In *TAICPART-MUTATION* (2007), pp. 193–202.

[107] Sousa, M., and Sen, A. Generation of TLM testbenches using mutation testing. In *CODES+ISSS* (2012), pp. 323–332.

[108] Spoon. `http://spoon.gforge.inria.fr/`.

[109] Sui, X., Lenharth, A., Fussell, D. S., and Pingali, K. Proactive control of approximate programs. In *ASPLOS* (2016).

[110] Sung, A., Choi, B., Wong, W. E., and Debroy, V. Mutant generation for embedded systems using kernel-based software and hardware fault simulation. *IST 53*, 10 (2011), 1153–1164.

[111] Thomas, A., and Pattabiraman, K. LLFI: An intermediate code level fault injector for soft computing applications. In *SELSE* (2013).

[112] Untch, R. H., Offutt, A. J., and Harrold, M. J. Mutation analysis using mutant schemata. In *ISSTA* (1993), pp. 139–148.

[113] Vásquez, M. L., Bavota, G., Tufano, M., Moran, K., Penta, M. D., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. Enabling mutation testing for Android apps. In *ESEC/FSE* (2017), pp. 233–244.

[114] Venkatagiri, R., Mahmoud, A., Hari, S. K. S., and Adve, S. V. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO* (2016), pp. 1–14.

[115] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *MICRO* (2013), pp. 1–12.

[116] Yao, X., Harman, M., and Jia, Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ICSE* (2014), pp. 919–930.

[117] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: A survey. *STVR 22*, 2 (2012), 67–120.

[118] Zhang, L., Gligoric, M., Marinov, D., and Khurshid, S. Operator-based and random mutant selection: Better together. In *ASE* (2013), pp. 92–102.

[119] Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., and Mei, H. Is operator-based mutant selection superior to random mutant selection? In *ICSE* (2010), pp. 435–444.

[120] Zhu, H., Hall, P. A. V., and May, J. H. R. Software unit test coverage and adequacy. *ACM Comput. Surv. 29*, 4 (1997), 366–427.