Mutation-Based Genetic Improvement of Software

Fan Wu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

July 6, 2017

I, Fan Wu, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

The following publications [1] are associated with the candidate's PhD work at University College London:

1. Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1375–1382, New York, NY, USA, 2015. ACM (Cited by 28)

2. (Best Paper Award) J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke. Mutation testing of memory-related operators. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10, April 2015 (Cited by 7)

3. Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Information and Software Technology*, 81:97 – 111, 2017 (No cites yet)

4. (Best Paper Award) Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*, pages 18–33. Springer, 2016 (Cited by 1)

[1] All citation counts are from Google Scholar (`https://scholar.google.co.uk/citations?user=p8z2_usAAAAJ&hl=en`), accessed in May 2017.

In addition, the following publications are works that were authored by the candidate, during his PhD study at University College London, but do not directly contribute to this thesis.

1. Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic improvement for adaptive software engineering (keynote). In *Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2014 (Cited by 21)

2. Haitao Dan, Mark Harman, Jens Krinke, Lingbo Li, Alexandru Marginean, and Fan Wu. Pidgin crasher: searching for minimised crashing gui event sequences. In *International Symposium on Search Based Software Engineering*, pages 253–258. Springer, 2014 (Cited by 1)

3. Yue Jia, Fan Wu, Mark Harman, and Jens Krinke. Genetic improvement using higher order mutation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 803–804, New York, NY, USA, 2015. ACM (Cited by 3)

4. Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. Sbselector: Search based component selection for budget hardware. In *International Symposium on Search Based Software Engineering*, pages 289–294. Springer, 2015 (No cites yet)

5. David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 330–341, New York, NY, USA, 2016. ACM (Cited by 1)

6. L. Li, M. Harman, F. Wu, and Y. Zhang. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016 (Cited by 1)

Abstract

Genetic Improvement (GI) of software is a recent field that has drawn much attention from Software Engineering researchers. It aims to use search techniques to automatically modify and improve existing software. The drawback in previous GI approaches is scalability of these approaches, due to the large search space formed by the code base in real-world systems. To overcome the scalability challenge, more recent studies have confined the granularity of code modification at the statement level and applied a prior sensitivity analysis to further reduce the search space. However, some software improvements may require code changes at a finer level of granularity.

This thesis demonstrates that, by combining with Mutation Testing techniques, GI can operate at this finer granularity while preserving scalability. The thesis applies Mutation Operators to automatically modify the source code of the target software. After a prior sensitivity analysis on First Order Mutants, "deep" (previously unavailable) parameters are exposed from the most sensitive locations, followed by a bi-objective optimisation process to fine tune them together with existing ("shallow") parameters. The objective is to improve both time and memory resources required by the computation.

Since this approach relies on the selection of Mutation Operators and traditional Mutation Operators are not concerned with memory performance, the thesis proposes and evaluates Memory Mutation Operators in the Mutation Testing context. Using both traditional and Memory Mutation Operators, the thesis fur-

ther seeks to improve the target software by searching for Higher Order Mutants (HOMs). The thesis presents the result of a code analysis study, which reveals that, among all the code modifications that contribute to the improvement, more than half of them require a finer control of the code, which our approach is better at than previous GI approaches.

Acknowledgements

I would like to thank my supervisor Professor Mark Harman for his understanding, guidance, endless support and advice and for providing me with the opportunity to undertake my PhD. I would like to thank my second supervisors, Dr. Jens Krinke and Dr. Yue Jia, and my external advisor, Professor Westley Weimer, for their support and all the interesting discussion.

I am very grateful to Dr. Bill Langdon, Dr. Federica Sarro and Dr. Justyna Petke for their friendly, insightful guidance in my work and patient discussions on new ideas.

I would like to thank all my colleagues at the Centre for Research in Evolution, Search and Testing (CREST) for their generous feedback and discussions on my research. I would also like to thank the various anonymous referees for their comments on papers submitted for publication; their comments not only give useful suggestions to improving my papers, but also help me see new perspectives and directions from my old work.

I gratefully acknowledge Chinese Scholarship Council, for the generous financial support that helps me focus on my PhD study.

Last but not least, I am thankful for my parents' unconditional support. I thank them for continuously sharing my difficult and happy times during my PhD, and their full respect of my decisions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimising software for better performance (such as speed and memory consumption) can be demanding, especially when the resources in the running environment are limited. Manually optimising such non-functional properties while keeping or even improving the functional behaviour of software is challenging. This becomes an even harder task if the properties considered are competing with each other [11]. Search-Based Software Engineering (SBSE) [12] has demonstrated many potential solutions, for example, to speed up software systems [13, 14], or to reduce memory consumption [1] and energy usage [15].

Previous studies have applied different search-based techniques to automate the optimisation process [16–19]. However, scalability of these approaches remains a challenge. To scale up and optimise real-world programs, recent studies use a so-called 'plastic surgery' [20] Genetic Programming (GP) approach. To reduce the search space, it represents solutions as a list of edits to the subject program instead of the program itself [15, 21]. Each sequence of edits consists of inserting, deleting or swapping pieces of code. To ensure scalability, this approach usually modifies programs at the 'line' level of granularity (the smallest atomic unit is a line of code). As a result, it is not possible for 'plastic surgery' to optimise subject programs at a finer granularity.

On the other hand, many software systems can reap significant performance benefits from workload- or runtime-specific configurations or optimisations. Software developers often expose a set of parameters for users to re-configure such

software systems adaptively. However, manual parameter tuning is a demanding challenge because users are usually required not only to have extensive knowledge about the system and the workload, but also to balance many competing objectives (such as memory consumption and execution time, which are the primary concerns of this thesis).

Many studies have reported on the challenges of automated parameter tuning [16, 19, 22–26]. Early work focused on finding optimal values with mathematical approaches [22, 24–26], while Search-Based Software Engineering [27] has been used in more recent research [16, 19, 23] on this problem. Although these approaches can automatically re-configure a system, their improvements are limited to changes to existing, explicit parameters.

Listing 1.1: Code snippet for managing dynamic arrays in C language

```
1 if (dynamic_arr == NULL && length < THRESHOLD){
2     dynamic_arr = calloc(THRESHOLD, sizeof(T));
3 }
4 else{
5     dynamic_arr = calloc(10 * THRESHOLD, sizeof(T));
6 }
```

In this thesis, our goal is to propose and evaluate novel approaches that can improve software's non-functional properties at a fine granularity, and to implement and demonstrate it for C language programs. We focus on two non-functional properties in this thesis, memory consumption and execution time, because they are important objectives for many applications, especially for those running on a platform with limited computation resources. Also memory consumption and execution time are often naturally conflicting with each other, thereby yielding an interesting and rewarding multi-objective solution space. We aim to apply Mutation Operators, originally used in Mutation Testing, to modify the target program at a finer level, and adopt equivalent mutants to keep the functionalities. As a motivating example, Listing 1.1 shows a code snippet of dynamic array management in C Language. In

this example, `dynamic_arr` is a dynamic array of a self-defined structure `T`, the size of which can dynamically grow as needed. To avoid repeatedly re-allocating memory when the array grows (which is a time-consuming operation), it allocates more memory than needed when initialised, so that it only needs to re-allocate when the size exceeds this initial size. However, if the initial size of the array turns out to be excessively large, it will waste some memory because only a small part of the array is used. In our example, the initial size of the array is determined according to the user-request size `length`: if the user requests less than a pre-defined threshold and the array haven't been allocated before, the initial size is set as the same as the threshold, otherwise set the initial size to 10 times of the threshold. The parameter `THRESHOLD` is a developer-defined parameter ('shallow' parameter) for the program so that users can tune according to their needs before running the program.

Though users can tune the `THRESHOLD` parameter, the ratio of the initial sizes for small and large arrays is fixed at 10. Therefore users may be forced to compromise and set sub-optimal initial sizes for small and large arrays. We propose a novel approach (described in Chapter 3) that can automatically discover such situations using Mutation Operators and expose additional 'deep' parameters for further tuning. For example, by applying our approach, Line 5 in Listing 1.1 may become `dynamic_arr = calloc(10 * THRESHOLD + DEEP_PARAMETER, sizeof(T))`, and `DEEP_PARAMETER` is a 'deep' parameter that is exposed for user to finer tune the program.

Moreover, in Line 1 of the example, only when both of the sub-conditions are met can the array be set a smaller initial size. However, this constraint may be loosened so that the smaller initial size is set more often, in order to save memory. This can be done by applying Mutation Operators to change the logical operator `&&` (and) to `||` (or). There may be multiple places in the program that only need simple but effective mutations to improve the performance with respect to a particular non-functional property. In Chapter 5, we propose a Higher Order Mutation based approach to search for the optimal combinations of this kind of mutations, in order to improve the non-functional properties of the program.

Since our approaches use Mutation Operators to modify the subject programs, the selection of Mutation Operators can have a big impact on the effectiveness of our approaches. Existing Mutation Operators have limited influence on memory management functions, so that the efficiency of the memory management calls in the program can be hardly improved by applying existing Mutation Operators. For example, in Line 2 of the same example, it uses `calloc()` to allocate memory, which also initialise the memory to 0s. However, this may not be necessary. Therefore, changing `calloc()` to `malloc()`, which does not initialise allocated memory, can improve the time performance of the program. However, this operation cannot be done by existing Mutation Operators. To address this limitation, we propose Memory Mutation Operators in Chapter 4 that mutate memory management calls to potentially gain additional improvement on the target program. Not only changing `calloc()` to `malloc()` can improve time performance, but also it raises an interesting question of whether a test suite is able to detect such difference if this mutation introduces memory vulnerabilities. Therefore in Chapter 4, we also evaluate whether these Memory Mutation Operators can be used to reveal weaknesses in test suites for detecting memory vulnerabilities.

Mutation testing is an effective fault-based testing technique that aims to identify whether a codebase is vulnerable to specific classes of faults [28]. In mutation testing, faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called mutants, each containing a different syntactic change. By carefully choosing the location within the program and the types of faults, it is possible to detect vulnerabilities that are missed by traditional testing techniques [29, 30], to simulate any test adequacy criteria [31] whilst providing improved fault detection [32, 33]. A survey [29] has also shown that this mutation methodology is gaining traction and is being used in a growing number of large-scale commercial and experimental projects.

Memory errors are one of the oldest classes of software vulnerabilities that can be maliciously exploited [34]. Despite more than two decades of research on memory safety, memory vulnerabilities have still been ranked in the top 3 of the CWE

SANS top 25 most dangerous programming errors [35]. Recent work on memory vulnerability detection [36–38] in C applications has shown the existence, in published code, of a wide range of vulnerabilities such as uninitialised memory access, buffer overruns, invalid pointer access, beyond stack access, free memory access and memory leaks. Moreover, these vulnerabilities are highly prone to exploitation. For example, vulnerabilities such as buffer overflows when using `malloc()` facilitate exploits that overwrite heap meta-data, gain access to unavailable function/data pointers, overwrite arbitrary memory locations, and create "fake" chunks of memory that may contain modified pointers.

Traditional mutation operators only simulate simple syntactic errors, based on the Competent Programmer Hypothesis [39]. Mutants generated using these operators may drive testers to generate test suites that primarily target simple syntactic errors. Semantic mutation operators, on the other hand, seek to mutate the semantics of the language [40]. Semantic mutants can capture possible misunderstandings of the description language and thus capture the class of semantic faults. However, both traditional and semantic mutation operators are not designed to find test cases revealing memory faults, thereby creating a weakness in traditional Mutation Testing.

To mitigate this limitation, there has been an attempt to design mutation operators for a specific type of memory vulnerabilities, Buffer Overflow vulnerabilities [41]. This work proposed 12 mutation operators that seek to simulate Buffer Overflow by making changes to the related vulnerable library functions and program statements. However, the proposed operators do not consider other general memory vulnerabilities, such as uninitialised memory access, `NULL` pointer dereferencing nor memory leaks caused by faulty heap management.

To address this problem we design 9 Memory Mutation Operators, including the one we illustrated in the motivating example above, simulating three classes of common memory faults. We also introduce two additional weak killing criteria, i.e. Memory Fault Detection and Control Flow Deviation for memory mutants. Because memory faults do not necessarily propagate to the output, making the strong killing

criterion, which is widely adopted in traditional Mutation Testing, inadequate to detect such faults. An easy-to-use Mutation Testing tool was developed using both of the traditional and Memory Mutation Operators with the traditional strong killing criterion and the proposed weak killing criteria also incorporated.

The content of this thesis is organised as follows: Chapter 2 summarises the recent development and related work on Genetic Improvement, Mutation Testing and Memory Management, Chapter 3 describes a novel mutation-based approach to expose and optimise "deep" parameter to improve software, Chapter 4 introduces new Memory Mutation Operators that can support mutation-based GI approaches on improving memory performance, and evaluates these operators in the sense of traditional Mutation Operator, Chapter 5 introduces a Higher-Order-Mutation-based GI approach that uses both Selective Mutation Operators and Memory Mutation Operators, and compare the results with Deep Parameter Optimisation; finally, general conclusions are drawn in Chapter 6.

Chapter 2

Literature Survey

In this chapter, recent works that are related to this thesis are reviewed and summarised. Search-Based Software Engineering is a general technique that we use in this thesis. Its development and applications are introduced in Section 2.1. Genetic Improvement is a more recent topic that uses SBSE to improve software. We review and summarise the recent techniques in Section 2.2. Software Testing is important to make sure the correctness of software, therefore is reviewed in Section 2.3. Some Mutation Testing works that are related to this thesis are reviewed in Section 2.4, whilst memory management strategies and memory-related testing are reviewed in Section 2.5 for us to better understand and work on memory performance of software.

2.1 Search-Based Software Engineering

In Software Engineering, many problems can be seen as optimisation problems with respect to one or more evaluating functions that measure how good a solution is. Therefore, Search-Based Software Engineering (SBSE) [12] seeks using search algorithms such as Hill-Climbing or Genetic Algorithm to solve Software Engineering problems. In recent years, it can be seen that SBSE has been applied to problems throughout the software life cycle, from requirements and architecture design to testing and maintenance [42].

In requirement optimisation problems, a subset of requirements needs to be selected to minimise the cost under the budget and maximise the revenue. The

selection of requirements can be seen as a search problem, thus SBSE can be applied to solve this problem. By using SBSE, not only more robust solutions that are stable to small changes in requirements can be obtained, but also the trade-offs between cost and revenue can be revealed to help decision-makers design the next release [43]. In the design phase of Software Engineering, SBSE has also been applied for architecture design, software clustering and software refactoring [44], with different objectives and formulations.

When a set of related software products share some core functionalities, but differ in some specific features, it is possible to describe and extract the feature models from them and use the models to construct similar software products. This is also referred to as Software Product Line (SPL). Haman et al. [45] surveyed recent work that applied SBSE to Software Product Line problems, such as feature model selection and SPL testing, as well listed some potential SPL problems that SBSE can be applied on.

SBSE has also been commonly applied in the testing and maintenance phase of Software Engineering [46–48]. For example, test case generation can be difficult due to a large pool of possible software inputs and some certain coverage criteria to be met. SBSE is good at finding near-optimal solutions in a very large search space, thus can be applied to test case generation problems. Though sometimes the test cases are given, it may not be possible to execute all of them. In this case SBSE can help prioritise some test cases such that the cost of testing is minimised while the loss in coverage criteria is minimised. In order to improve the readability, adaptability and extensibility of software, refactoring of the software may be required. SBSE can as well be applied to automatically search for optimal refactoring solutions to improve the quality metrics of the software.

J. T. de Souza [49] et al. investigated human competitiveness of SBSE works. They reviewed multiple SBSE works on Next Release Problem, workgroup formation problem and test case selection problem with a variety of sizes. They compared the results with the solutions given by both professional programmers and senior students to study the quality of machine-generated solutions. Their results showed

that the quality of the solutions given by SBSE approaches is similar to, sometimes even better than human-provided solutions, and machine-generated solutions are generally more consistent when compared with human-provided solutions.

2.2 Genetic Improvement of Software

Automatically improving software with respect to one or more of its properties has been one of the most interesting topics of Software Engineering. Genetic Improvement (GI) is a recently emerging area that studies this topic. It regards the software improvement problems as search problems and uses Genetic/Evolutionary algorithms to solve them.

Harman et al. highlighted this area by posing the so-called GISMOE challenge [11]. They argued that optimising non-functional properties in a multi-objective formulation was demanded and it is feasible to use Search-Based Software Engineering (SBSE) [12] techniques to achieve this goal with automation or semi-automation. They also outlined some open challenges such as "human in the loop" optimisation and visualisation of high dimensional Pareto surfaces. In another paper, Harman et al. proposed a framework [5] that separates the optimisation process into two phases: online phase and offline phase. In the online phase, the framework collects usage data such as test cases and non-functional properties to be improved. In the offline phase, the framework uses the collected data as a guidance in the optimisation process. They also propose an approach that exposes implicit parameters from the software and tunes them by SBSE approaches to achieve better performances in the offline phase. In addition, they illustrated a use case of this framework called Dreaming Devices, which automatically collect usage data in the days and optimise the performance at nighttime (dreaming).

Genetic Improvement can be applied to improve not only software's non-functional properties such as execution time, memory consumption and energy consumption, but also it's functionalities or "correctness". There are many works that used SBSE approaches to automatically fix software bugs, therefore enhancing the functionality and reliability of the software. Arcuri proposed to use Genetic Pro-

gramming (GP) to automatically evolve a piece of code guided by test cases to fix bugs [50]. Additionally, in order to improve the quality of the code and test cases, a co-evolution formulation was proposed to improve both the source code and the test cases in a prey-predator like relation. More related work regarding this topic can be found later in this section.

Genetic Improvement has been applied in many different kinds of applications, but according to the approaches used in those works, they can be categorised into three classes: improving software by parameter tuning and Genetic Programming are described in Section 2.2.1 and Section 2.2.2 respectively, patch-based Genetic Improvement is described in Section 2.2.3.

2.2.1 Search-Based Parameter Tuning

There are usually some parameters exposed by the developers that can be tuned to improve software's performance. Therefore, one straightforward approach to improving the software is to search for the optimal values for these parameters. In addition, as can be seen soon in this section, some researcher also discovered more implicit parameters to control the behaviour of the software and then later optimised them.

Tantithamthavorn et al. [51] discovered that the parameter settings of classifiers can be crucial to the performance of defect prediction models, which are Machine Learning models trained to identify defects in software modules. They used automated parameter optimisation technique to adjust the parameter settings of previous defect prediction models and discovered that the performance of these models can be improved by up to 40 percent.

Compilers usually apply different optimisations to improve the non-functional properties of a program. However, users suffer from a large amount of choices when they use compilers. Boussaa et al. proposed a framework called NOTICE [52], which automatically tunes the compiler optimisation options to achieve the optimal performance on user-specified properties. The framework also works on multi-objective optimisations, in which case it provides insights to the optimal trade-offs between several competing non-functional properties.

Manotas et al. [53] focused on the energy efficiency of Java collection APIs and proposed SEEDS, a framework that automatically chooses the most energy efficient Java collection API in different use cases. Though it does not directly tune the parameters of a program, it creates a parameter for each collection API usage to represent which interchangeable collection should be used, and later tunes the parameter with respect to energy efficiency. Using this approach, Manotas et al. found up to 17% energy usage improvement across 7 real-world application.

When tuning the existing parameters does not meet the performance requirements or those parameters are not available for tuning, researchers discovered ways to expose implicit parameters from the programs to be improved. In previous works, the Software Tuning panel for Autonomic Control (STAC) [18] automated the exposure of a limited form of the "deep" parameters considered in this thesis. STAC first generates a design graph for a subject program under optimisation. The design graph represents data reference transition flows in the subject. It then uses the reference patterns of existing ("shallow") parameters to discover deep parameters, whose reference pattern is the same as one of the shallow parameter reference patterns. Although STAC can discover some deep parameters effectively, it suffers from two limitations: firstly, STAC requires initial human effort to characterise shallow parameters, and secondly, STAC can only find a subset of deep parameters, those that have similar data transition patterns to the known shallow parameters.

Hutter et al. [54] tuned the parameters of a SAT solver, SPEAR, by adjusting not only the explicit parameters but also many implicit parameters. They exposed almost all possibly tunable variables, including some redundant or unnecessary ones. Therefore, a very large search space is formed and searched in the optimisation process, which makes the approach vulnerable to scalability issues.

Hoffmann et al. [23] proposed *PowerDial*, a system which dynamically adjusts an application's behaviour to make it adaptable to fluctuating workloads. It first transforms some configuration parameters to non-constant variables residing in the application's memory, so that the behaviour of the application can be changed by controlling these variables at runtime. It then pre-runs the application with each

possible configuration to discover how these parameters influence the application and memorizes the Pareto-best candidates in terms of application's non-functional properties and the quality of the output. Whenever *PowerDial* detects a resource shortage, it sacrifices output quality by changing the values of these variables, allowing the application to "survive the crisis". However, the approach uses exhaustive search on the configuration variables, thus the number of variables and the search space formed are limited, by the scalability of exhaustive search.

2.2.2 Improvement by Genetic Programming

Many GI works target the source code of the software itself, and use Genetic Programming to modify the source code as an Abstract Syntax Tree (AST) or an array of statements. The advantage of GP is that it can evolve any arbitrary code that meets the requirement. However, without any heuristics or constraints, it usually suffers a very large search space.

Langdon et al. [55] applied Genetic Programming to several real-world programs ranging from image compression to machine learning benchmark problems, to improve the quality of the programs while minimising the size of them. In their research, they showed that seeding the initial population with existing programs can focus the search, thereby improve the efficiency of the approach. To further reduce the cost of the approach, they also showed that sometimes using a very small set of the test cases is sufficient to generate the optimal solutions.

Since energy efficiency is crucial to many embedded systems, Schulte et al. [56] focused on the optimisation of energy consumption of eight embedded benchmark applications, and applied Genetic Programming to automatically improvement the performance while maintaining the functionality. According to their experimental results, their approach achieved 20% energy reduction on average and maintained the functionality on 7 out of 8 benchmarks.

Gao et al. [57] showed that Genetic Programming can be used to fix specific types of bugs in software. In this work, they targeted memory leak defects in the programs and applied Genetic Programming to automatically discover and fix the defects. Since they only targeted the memory behaviour of the programs, their

approach used a Control Flow Graph to represent the memory allocation and usage of a program, so that the problem is simplified by this specific type of bugs. Across 15 benchmarks with 89 detected memory leak defects, their approach successfully fixed 28% of them.

Goues et al. [17] introduced a generic method for automatic bug fixing, named *GenProg*. Instead of using the Control Flow Graph of the subject programs, it makes no simplification of the programs but regards the programs as an array of statements, then use GP to evolve them, guided by their test cases. A hill-climbing clean-up process is conducted to remove redundant changes to the programs once a fix is found by GP. Across 16 benchmark programs, *GenProg* successfully fixed 77% of the bugs. In a follow-up study [58], Goues et al. showed that this approach can scale up to very large real-world applications, and by using cloud-computing resources, they estimated $7.32 cost on average for fixing a bug.

White et al. [59] applied GP optimisation in a bi-objective formulation on pseudo-random number generators. They targeted the quality of the pseudo-random number generators and the power consumption of them. In this work, they also showed that using only one fourth of the test cases is sufficient to form the Pareto-optimal solutions.

Arcuri and White et al. [13, 60] demonstrated how to use GP with other techniques to optimise the functional and non-functional properties of subject programs/functions in a multi-objective formulation. In addition to seeding the population to focus the search, they also used a set of co-evolved test cases to encourage the preservation of the programs' semantics. They applied their approach with different settings to eight benchmarks to show that they were able to find non-trivial optimisations that the compilers could not.

Co-evolution is also used by Arcuri et al. to automatically fix software bugs [61]. They used Genetic Programming to evolve an existing buggy program, while co-evolving a set of test cases. They claimed that their approach only needs the source code of a buggy program and a formal specification of it, and has no restrictions on the type of bugs.

2.2.3 Patch-Based Genetic Improvement

Genetic Programming sometimes is not a very efficient approach for software improvement, and most of the time the changes needed to improve the software are just a few statements. Similarly to human-created patches, GI approaches can simply evolve patches affecting a few statements instead of evolving a whole program as in Genetic Programming. Moreover, some researchers argue that many of these patches can be automatically generated from existing code bases [20].

Ackling et al. [62] used Genetic Algorithm (GA) to evolve a list of modifications to the target buggy programs. The generated patches are applied to the AST representation of the program, then the patched program is run against a set of predefined test cases. The possible modifications are limited to a modification table that is defined prior to the evolutionary process. Their results showed that their GA approach is able to find a fix much faster than random search and the fix is composed of 1 to 8 modifications on average.

Langdon et al. [63] used Evolutionary Computation to automatically improve the execution time of a highly complex system involving 50000 lines of code. Instead of evolving pre-defined modifications to the AST representation, they treat the target program as an array of statements and evolved more straight forward modification sequences such as inserting, deleting a statement or swapping two statements. After evolving the editing sequences, they also used a simple hill climbing process to clean up unnecessary modifications. The optimised version of the program is 70 times faster than the original, and surprisingly also achieves a small semantic improvement.

Petke et al. [64] used a similar approach to specialise a SAT solver to a class of problems, using other versions of SAT solvers as code donors, meaning the code from other SAT solvers were used to replace the code in the target solver. The result showed that automatic improving a program can be even faster than the best one written by human developers, and increasing the code base contributed to the final improved version of the SAT solver.

Bruce et al. [15] also used a similar framework to evolve patches that improve

the execution time and energy consumption of the target programs as a bi-objective optimisation problem. Their results showed that the execution time and the energy consumption of a program has a strong positive correlation, and on three medium size programs, their approach found up to 25% improvement on the energy consumption.

2.3 Software Testing

Software testing is an important part of the software life cycle. The goal of software testing is to provide information about the quality of the system under test (SUT) and how well it meets the specifications. Software testing methods can be generally classified as static testing methods and dynamic testing methods. Static testing methods usually involve static analysis of the source code without executing it, whilst dynamic testing methods execute the software with different inputs (test cases) and verify the outputs. However, there are some techniques that are considered a combination of both. For example, Symbolic Execution that can be found in many recent works [65] uses symbolic values to represent program inputs and propagate them throughout the execution paths of the program by those expressions involving them. The output of the program can then be expressed by symbolic values and be verified against the program specifications.

At different stages of software life cycle, different levels of testing may be involved, such as Unit Testing, Integration Testing, System Testing and Acceptance Testing. Due to the requirements of different levels of testing, a variety of techniques have been applied to the generation, execution and verification of tests [66–68], including SBSE techniques [69] introduced above.

In this thesis, we use Regression Testing to verify the correctness of the program under optimisation after we make changes to the program. Regression Testing is a testing activity when changes are made to the software to provide confidence that the existing behaviour of the software is not broken by the changes. Regression tests are usually comprehensive tests with high coverage, therefore they are sometimes costly to execute and techniques have been applied to minimise or pri-

oritise the tests [70]. Since Regression Testing has been demonstrated effectiveness in many software testing practices [71], by adopting Regression Testing, the threat to the correctness of a program after mutation is minimised.

2.4 Mutation Testing

Mutation Testing [28] is a white box testing technique that measures the quality/adequacy of tests by examining whether the test set (test input data) used in testing can reveal certain types of faults. A mutation system defines a set of rules (mutation operators) that generate simple syntactic alterations (mutants) of the program under test (PUT), representing errors that a "competent programmer" would make, known as the Competent Programmer Hypothesis (CPH) [39]. It states that programmers are competent and tend to develop programs close to the correct version. As a consequence, although there may be faults in the program developed by a competent programmer, it is assumed that these faults are simple faults which can be corrected by a few small syntactical changes. Table 2.1 shows an example of a mutation. In the example, a relational operator '$>=$' is mutated to '$<$' in the mutant, while all the other statements remain the same. According to different Mutation Operators, an operator, a variable or constant, a predicate or a statement can be mutated in different ways.

Table 2.1: Example of a Mutation

Original Program (P)	`int i=0;` `if(i+8>=0){...}`
Mutated Program (P')	`int i=0;` `if(i+8<0){...}`

To assess the quality of a given test suite, the set of generated mutants are executed against the input test suite to determine whether the injected faults can be detected. If a test suite can identify a mutant from the PUT (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is said to have *survived* (or live). A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test suite is inadequate to kill the mutant. The

Mutation Score (MS) is used to quantify how adequate a test suite is in detecting the artificial faults. It is calculated as the following formula:

$$MS(P,T) = \frac{Killed}{All}$$

P is the program under test and T is the set of tests. $Killed$ and All represents the number of killed mutants and the total number of all generated mutants respectively. This metric is traditionally used as an estimation of test suite effectiveness.

2.4.1 Challenges and Related Studies in Mutation Testing

Among those generated mutants, there are some mutants that are semantically equivalent to the original program despite syntactic changes, they are called equivalent mutants. Since they are semantically equivalent to the original program, they cannot be killed, and in fact, they should be excluded from the formula of Mutation Score since they do not contribute in revealing the effectiveness of a test suite.

The equivalent mutant problem is a major impediment to large-scale wide spread use and whether a mutant is equivalent has been proven to be undecidable [72, 73]. Although it has been shown that the problem of detecting equivalent mutants cannot be completely automated, approaches to partially solve this problem have been introduced. They consist of applying compiler optimization techniques [73, 74] and detecting infeasible paths using static analysis [75]. Other work combines mutants to generate HOMs (Higher Order Mutants) followed by using the number of unit tests that killed FOMs (First Order Mutants) that make up a HOM to identify equivalent mutants [76].

Co-evolution has also been proposed to achieve tailored selective mutation to partially evaluate mutants [77]. Due to the undecidable nature of this problem and the requirement of a human in order to solve it, it can be considered as a Human Intelligence Task (HIT) [78] such that the cost of the human oracle associated increases with the scale of the program under test. Madeyski et al. [79] conducted a Systematic Literature Review on recent techniques for the equivalent mutant problem.

Another problem associated with mutation testing is that the increase of computational cost is also positively correlated with the scale of the program under test [29]. Empirical studies have shown that even for small programs a very large number of mutants can be generated, this makes the process of evaluating all the generated mutants against a set of test cases very computationally expensive as the application has to be recompiled and executed each time. Techniques such as Mutant Sampling [39,80], Mutant Clustering and Selective Mutation have been used in recent studies in order to reduce the number of mutants that are generated without inflicting a significant loss in their effectiveness [81].

The Mutant Clustering approach reduces mutant count by applying clustering algorithms based on the killable test cases. As mutants in a given cluster are guaranteed to be killed by the same test cases, a representative proportion of mutants are selected from each cluster and the rest are discarded. As an instance, Devroey et al. [82] used Featured Mutant Model inspired by the software product line paradigm to reduce the generation and execution of mutants.

This approach is similar to the approach used by Jia et al. [76] to generate Higher Order Mutants. In their work, those first order mutants that are killed by at least one test but at the same time not killed by all test cases are deemed "non-trivial" and are selected as candidates to generate HOMs.

The Selective Mutation approach [83] achieves reduction in the number of mutants generated by reducing the mutation operators that are applied. As each mutation operator generated a different number of mutants, this data along with the performance of the mutants generated can be generalised and lead to convincing evidence to omit certain mutation operators.

Moreover, some may argue that the artificial faults introduced by Mutation Operators are not good representatives of real faults. Just et al. [33] investigated this problem by studying whether a test suite's ability to detect artificial faults is correlated with its ability to detect real faults. On 5 open source applications, their study found a strong correlation between artificial fault detection and real fault detection, therefore concluded that mutants are reasonable substitutes for real faults.

In Jia and Harman's survey on Mutation Testing [29], a comprehensive study on Mutation Testing approaches, tools and empirical results was conducted. From the survey, the authors concluded that Mutation Testing is reaching its maturity, with increasing number of publications and large scale applications.

2.4.2 Mutation Operators

For Mutation Testing systems, how effective the Mutation Operators are in representing real faults usually determines how the system performs. Though there are works committed to select a subset of the Mutation Operators to reduce the cost of Mutation Testing, there are also works that introduced more Mutation Operators that satisfy specific scenarios.

Some traditional Mutation Operators [84] are not very effective in revealing real faults, or some operators may generate overlapping mutants. Offutt et al. [83] investigated this problem for traditional Mutation Operators. Their research confirmed that with only a subset of Mutation Operators, the Mutation Testing system would only loss negligible effectiveness but reduce a large number of generated mutants, thereby reducing the computational cost. Therefore, the authors proposed Selective Mutation Operators, a subset of the traditional Mutation Operators, that can provide the same effectiveness in revealing real faults in Mutation Testing systems. In recent work, Selective Mutation Operators have been widely adopted as the standard operators for Mutation Testing.

Selective Mutation Operators are statement level or primitive level operators, meaning they are applied to a single statement or a primitive. Ma et al. [85] proposed class level and inter-class level Mutation Operators for Object-Oriented language Java. From their experimental results, they showed that these specially designed operators were able to test new features introduced by Object-Oriented languages, such as inheritance and polymorphism.

There are also many other Mutation Operators proposed to deal with other situations. One closely related work is that, Shahriar [41] proposed 12 mutation operators that primarily focused on Buffer Overflow vulnerabilities, vulnerable library functions and program statements. The tool they developed uses Mutation

Testing to generate test cases that expose vulnerabilities in the program under test. Their results suggested that the proposed mutation operators are designed mainly for identifying faults due to programmers' lack of fundamental knowledge about the programming language instead of identifying trivial syntactic faults.

2.4.3 Higher Order Mutation

Mutants that are generated from applying a Mutation Operator once to the original program are called First Order Mutants (FOM). Applying Mutation Operators to FOMs will generate so-called Higher Order Mutants (HOM). According to Competent Programmer Hypothesis (CPH) [39], test cases that kill First Order Mutants are likely to kill the Higher Order Mutants that contain the same changes. Therefore it is not necessary to include Higher Order Mutants in Mutation Testing studies. In addition, the number of HOMs is much larger than the number of FOMs, including HOMs in Mutation Testing were considered impractical.

Harman et al. [86] argued that by carefully selecting Higher Order Mutants, the number of mutants can be reduced since one Higher Order Mutant may be sufficient to represent several First Order Mutants, therefore these FOMs can be excluded from the mutant pool. In their study, they found examples of such Higher Order Mutants on ten benchmarks, and proposed both single-objective approach and multi-objective approach to finding those HOMs. In a follow-up study [87], Langdon et al. adopted a multi-objective approach and Genetic Programming to search for Higher Order Mutants. Their study also revealed that by combining First Order Mutants, the artificial faults may interact with each other. This makes the mutants harder to kill and corresponds to more complex real-world faults.

Harman et al. [88] used Higher Order Mutation to guide test data generation. On 17 benchmark programs, their approach found that using HOMs in test data generation, the test suite was able to kill 8% to 38% of the FOMs survived from the test suite generated from using FOMs only.

2.4.4 Mutation Analysis in Other Areas

Though Mutation Operators and mutants were originally used in Mutation Testing to represent real faults and evaluate the quality of test suites, they are also widely used in many other areas and applications for different purposes.

Similar to Harman's work [88] introduced in Section 2.4.3, Fraser et al. [89] used mutants as guidance to automatically generate unit-test suites for Java programs. In their approach, the test suites are optimised to kill more mutants (assumed good representatives of real faults), so that they have a high chance to detect real faults. They also compared machine-generated test suites with manually written test suites and found that machine-generated suites killed more mutants and reached a higher Mutation Score than manually written test suites. Haga et al. [90] conducted a similar study for C programs. Instead of comparing with manually written tests, they validated the machine-generated tests by their branch coverage, and evaluated that it cost only 130 ms on average to generate a test suite.

Andrews et al. [32] used mutation-based analysis to assess and compare four common testing coverage criteria: Block, Decision, C-Use and P-Use. For a test suite, they calculated corresponding coverage criteria and used Mutation Testing to calculate it's Mutation Score. Since Mutation Score to some degree represents the quality of the test suite (confirmed by their results as well), the coverage criteria can then be evaluated by their correlation with Mutation Scores. In addition, their study also included the computational cost for calculating each coverage criterion.

Staats et al. [91] used mutation analysis to provide support for automatic oracle creation. Their approach searches for a set of monitored variables such that using these variables as killing criteria, the number of killed mutants or the Mutation Score is maximised. After ranking these variables by how much they contributed to identifying non-equivalent mutants, it can provide insights to which variables are more important in oracle data selection.

Svajlenko et al. [92] used mutants to evaluate clone detectors. Since mutants are inherently similar to each other and to the original programs, they themselves can be the subjects of clone detectors. Therefore, they proposed a framework to

generate benchmark mutants and to compare the quality of different clone detectors. Additionally, mutation analysis has been applied for compiler testing [93], software specification extraction [94] and searching for program invariants [95].

Though equivalent mutants are considered a challenge in Mutation Testing, there are works that took advantage of equivalent mutants being semantically equivalent to the original program.

Schulte et al. [96] studied software's robustness to mutational changes by assessing what proportion of the mutants are equivalent to the original program with respect to a given test suite. Across 22 benchmark programs, their study found 30% of random mutations did not affect the output of the programs. Arcaini et al. [97] defined *static anomalies* as the non-behavioural properties of a program that can be improved by mutation, such as redundancy of the code or runtime performance. They proposed to apply Mutation Operators to the target programs and search for equivalent mutants that are improved with respect to a certain static anomaly.

2.5 Memory Management

In this thesis, we focused on the improvement of the memory performance of software. Therefore it is necessary to review the memory management strategies (Section 2.5.1), as well as some recent related work (Section 2.5.2).

2.5.1 Memory Allocation Strategies

When an application asks for some memory from the operating system, the simplest way is to give exact size of memory it requests, and to return that memory back to the system when the application frees it. However, getting memory from the system and returning it to the system require invoking some system calls, which are much slower than other operations. Due to the fact that memory allocation and deallocation are very common and are used frequently in many applications, frequently allocating memory directly from the system is impractical. Memory allocators have risen to solve this problem using different allocation strategies. They act as a cache for memory management in different languages. They request memory from the system when necessary, hold some extra available memory and manage them to

serve later memory requests from the application as efficiently as possible. The freed memory is not returned to the system directly, but held by memory allocators in case of any similar memory requests in the near future. Extra memory are only returned back to the system when certain criteria are met according to the allocators. Different ways of memory management significantly influence the performance of an application and researchers have proposed and studied the performance of them.

In terms of allocation strategies, many researchers have proposed many different strategies for memory allocation and deallocation, which have been well studied and have their own strengths and disadvantages [98]. The data structure that most of the allocation strategies use is a linked list of free chunks of memory, which costs a small overhead to store some basic information such as the size of the chunk and whether it is in use, whilst using the memory of the free chunks to store the linking pointers.

The linked list is also called free list since it only links the unoccupied chunks. Whenever the application sends a memory request, the allocator searches the free list to find a free chunk that meets the request and removes it from the list, this chunk is then given to the application to fulfil the request. When a chunk is freed by the application, it will not be returned to the operating system immediately but instead inserted to the free list, in case the application may soon request a chunk in the same size. A simple example of a free list is depicted in Figure 2.1, in which the shaded regions represent occupied memory chunks.

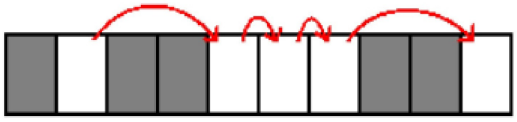


Figure 2.1: Linked List

Sequential fit is a category of allocation strategy, including first fit, next fit and best fit. They all use no more than a double linked list to manage the free chunks. When a memory request is received, first fit starts searching the list from the beginning and stops at the first time it finds a free chunk that fulfils the request. It splits the free chunk into two pieces, one of which is in the requested size and

is returned to the application whilst the remainder is inserted back to the list. One major disadvantage of first fit is poor locality, which increases the possibility of cache miss, thus increases the reference time as well as the total running time. Next fit also searches the free list one by one, but starting from where it found the free chunk for the previous request. The first and next fit both continuously split large free chunks to smaller ones, which cannot be used for large request later, therefore they both suffer from severe fragmentation. The best fit, on the other hand, goes through the whole free list before it makes a decision. Therefore, it guarantees to return the smallest free chunk in the list that meets the request, sometimes even the exact fit, therefore it reduces fragmentation when compared with first fit and next fit [99]. However, because it goes through the whole list for every request, it costs more time then first fit and next fit, especially when the free list is very long.

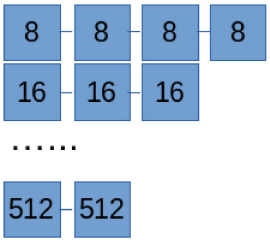


Figure 2.2: Segregated free lists

Segregated fit keeps multiple free lists at the same time (Figure 2.2) instead of managing one free list as in sequential fit. Each of the free lists kept by segregated fit policy contains free chunks in the exact same size which are always a multiple of 8 bytes or a size in power of 2. Whenever there is a request of size n, it only needs to fetch a free chunk from one of the free lists that corresponds to size n. In this case it saves the allocation time by diminishing the searching time for a best fit whilst keeping a low fragmentation. However, it slightly increases the deallocation time since it has to find the exact free list in the deallocated size before the freed chunk can be inserted. What's more, it is possible that there is no available free chunk in the free list that corresponds to the request size, in which case the allocator will retrieve a larger chunk from the next non-empty free list and split the chunk to meet the request. A variant of segregated fit differs only in returning a free chunk

without splitting it even if the chunk is much bigger than the request size. In this allocation strategy, the memory consumption is much higher because each chunk is in a fixed size and may be serving a much smaller request so that the extra size in that chunk can not be reused by other requests. This has also been known as the internal fragmentation (see below). On the other hand, the simplified segregated fit strategy is much faster since it never spends time on splitting memory chunks. This also shows that there is clearly a trade-off between allocation time and memory consumption.

Coalescing is a strategy which can be combined with or applied to the above allocation strategies, as well as the following ones. In order to save memory, most of the policies above choose to split a chunk before returning it to the application. This leads to more and more small chunks that cannot be used for big requests. But some of these small chunks could be contiguous in the address space so that they can be concatenated to serve large requests, so that the memory is used more efficiently.

Coalescing policies try to concatenate contiguous free chunks into a bigger one, but when to apply coalescing remains adjustable in different allocators. This is because if an application requests a chunk with a size of n right after it frees a chunk in the same size, the allocator may have coalesced the freed chunk and may need to split it out from the concatenated chunk to serve the same-size-request, which is a waste of computation. Therefore, instead of coalescing each time the application frees a chunk, some allocators apply coalescing in a fixed frequency or once there is no big enough chunk that meets the request. In summary, whether the coalescing policy improves or diminishes the performance of an allocator usually depends on the allocation strategy used by the allocator and the allocation pattern of the application.

Searching strategy refers to how an allocator manages and searches the free list(s). Some allocators keep the free list(s) in ascending or descending order by the physical address or size. It is quick for searching but may cause poor locality. Good locality refers to allocating chunks that may be accessed subsequently in the

application on the same physical page, in order to minimise the cache misses in the operating system as a means of saving execution time. Searching strategy is quite influential to locality since it determines where subsequent requests should be located. On the other hand, when a chunk is freed and returned to the allocator, there are several ways to place it back in the free list(s), two of which are first-in-first-out (FIFO) and last-in-first-out (LIFO). FIFO strategy gives the most recently released chunk the least priority and places it at the end of the searching queue, whilst LIFO gives the most recently released chunk the highest priority so that it is more likely to be reused in a short time. Which searching strategy is better usually depends on the allocation strategies and the application itself.

Bitmap is another combinable strategy that uses boolean flags to keep the status of chunks or free lists. When it is used for the status of chunks, one bit is used to represent whether a chunk is in use, and when it is used for the status of free lists, it usually means whether a free list is empty. The bitmap is designed to help the allocator to find an available chunk more efficiently. It narrows the range down by searching the bitmap before searching a free chunk that meets the request.

Buddy system is a well-known algorithm which involves a special splitting mechanism. It always allocates memory in several fixed sizes and whenever it needs to split a chunk into two, it splits it in a fixed ratio repeatedly until it meets the smallest chunk that bigger than the request size. In this special splitting mechanism coalescing is fast because finding a chunk's "buddy" only requires a few simple mathematical computations, according to the ratio defined for this buddy system. Even though the splitting ratio is adjustable, buddy systems still suffer from internal fragmentation because they almost always serve a larger chunk than requested.

Internal fragmentation: Most of the memory allocators keep all the chunks in the size of a multiple of 8 bytes due to the operating system requirements. In these cases, all the memory requests from applications are rounded up to an alignment of 8 bytes before they are allocated. Therefore, the returned chunk is always equal to or slightly bigger than the request size without the applications being aware of it. Then the little extra padding memory is neglected and cannot be used by the

applications. This is referred to as internal fragmentation.

External fragmentation: On the other hand, at any point of a running application, the memory allocator always hold some free chunks instead of returning them to the operating system directly, so that it can quickly respond to some memory request from the application using these free chunks. And some of these free chunks may be too small to serve bigger request and for some reason they cannot be coalesced neither (e.g., not next to any other free chunks). The number of these small chunks may increase when the application proceeds, which causes the application occupying more memory than it really needs. This is also known as external fragmentation.

Allocators always introduce fragmentation, in different degrees. Despite many attempts to minimise memory fragmentation in efficient ways, there is clearly a trade-off between running time and memory fragmentation. For instance, allocators with coalescing policies reuse small free chunks by concatenating them, but they introduce extra computation time for coalescing. On the other hand, both segregated fit and buddy systems suffer from high fragmentation, but they are usually faster in responding time than other strategies.

From the allocation strategies introduced above, many researchers have combined some of these strategies and developed complex dynamic memory allocators for different purposes. *Dlmalloc* is a general purpose memory allocator that has been widely adopted for C programs. It almost takes the advantages of most of the strategies introduced above, aiming at catering the needs of all programs fairly efficiently. Berger et al. [100] showed that most of the custom memory allocators perform no much better than *dlmalloc*, some are even worse. Therefore, we use *dlmalloc* as a concrete example in our studies in this thesis.

2.5.2 Memory Testing

Work done by Vilela [101] was an inspiration behind the mutation operators that mutate the parameters of `malloc()`, `calloc()` and other memory allocation/deallocation library functions as well. Static memory allocations (MSMA) and Dynamic memory allocations (MDMA) are proposed in this paper, each of which

mutates the buffer size in order to identify buffer overflow and buffer underflow vulnerabilities. Although their proposed mutation operators mutate memory related operations, they do not expose vulnerabilities that are caused due to uninitialised memory faults, as well as buffer vulnerabilities that can be caused due to an incorrect argument to the `sizeof()` function, which is generally used in memory allocation calls.

Although Zhivich [102] used a mutation approach to identify memory based faults as well as integrate dynamic memory analysis tools in their work, they primarily focus on buffer overflow vulnerabilities and do not consider the other classes of vulnerabilities. Also, their work is mainly concerned with using mutation to mutate test data and use code instrumentation along with dynamic memory analysers to identify vulnerabilities. Moreover, due to the fact they do not perform any source code transformation and much of their work is based on dynamic analysis of the program under test, their work can be classified as taking a more black-box testing approach in comparison to the white-box testing approach.

Kosmatov [103] developed a runtime memory monitoring library for runtime assertion checking in Frama-C [104], which is a platform for analysis of C code. Their work aims to detect dynamic memory related faults such as invalid pointers, out-of-bounds memory accesses, uninitialised variables and memory leaks. *Valgrind* is used in this work, though only as a benchmark to compare results. This study does not use specialised mutation operators to inject such vulnerabilities. Instead, mutation operators that mutate numerical arithmetic operators, pointer-arithmetic operators and comparison operators are used to generate faults that may or may not introduce memory based vulnerabilities to be picked up by the runtime memory analyser. Their work also acknowledges the fact that memory related faults are more likely to occur in C programs due to the lack of infrastructure available to detect them.

Interesting research such as a detailed evaluation of seven modern dynamic buffer overflow detection tools was carried out by Zhivich [102]. They ended up using *CRED* (C Range Error Detector) in order to analyse buffer overflow faults

over *Valgrind*'s *Memcheck*, which is the tool we use in our study, mainly because *Memcheck* uses sampled bound checking to detect buffer overflow errors instead of more precise bound checking techniques in *CRED*. Zhivich also found that *Valgrind* runs 25-50 times slower than *gcc* because it simulates program execution on a virtual x86 processor. Despite the speed issue with *Valgrind*, *Valgrind*'s *Memcheck* is used as the primary memory analysis tool in many ways due to its ability to analyse a wide range of other memory faults including uninitialised memory faults, when compared with other alternative tools.

Chapter 3

Mutation-Based Deep Parameter Optimisation

Many software systems contain undocumented internal variables or expressions that also affect the performance of the systems [105]. Thus, these elements could also be good candidates for automated parameter tuning. However, many of these elements are 'private', undocumented, or otherwise unexposed. Moreover, some internal values may not even be stored in variables, private or otherwise, but may merely exist as fleeting sub-expression evaluation outcomes. Identifying these variables and expressions is very difficult for general users, as it requires a deep understanding of the source code of the system.

In this chapter, we propose an automatic technique to discover internal variables and expressions that normally cannot be accessed directly, but impact non-functional properties of interest. Our goal is to expose new parameters that can directly influence the values of these internal variables and expressions. To distinguish from parameters exposed by software designers (which we call 'shallow parameters'), we call these exposed parameters 'deep parameters' [5]. Modifying shallow parameter values does not necessarily change the internal code elements represented by deep parameters. Therefore, deep parameters provide additional opportunities for subsequent automated parameter tuning.

We illustrate the approach by re-configuring a general purpose memory allocator, *dlmalloc*. We choose memory allocators because they are critical to the memory

consumption of many programs and can account for up to 60% of the total execution time in some scenarios [106]. As a result, memory optimisation is a widely-studied topic [107, 108]. We evaluate our approach using four specimen systems drawn from benchmarks for *dlmalloc* and real world applications. Our approach neither touches the source code of the application itself nor requires any knowledge about the application under optimisation, instead only tuning the parameters for *dlmalloc* library, making it applicable to other C applications with little effort.

This chapter presents evidence that deep parameter optimisation targeting *dlmalloc* is an effective approach for improving program's non-functional properties. The experimental results suggest that deep parameter optimisation competes favourably with both shallow optimisation and default configurations. For four subjects, deep parameter optimisation reduces memory consumption by 21% or execution time by 12% in the best cases. The contributions of this work are summarised as follows:

1. We introduce an automated approach to discover and expose deep parameters. The discovery of these parameters enhances search-based parameter tuning.

2. We report the results of an empirical study comparing the traditional shallow parameter tuning approach with our approach. On four applications totalling over 70,000 lines of code and guarded by over 700 tests, the results show that our approach can reduce memory usage by 21% and execution time by 12%, whereas shallow tuning alone achieves only a 16% and 10% corresponding reduction.

3. Furthermore, we evaluate the offline optimisation-time cost of our approach. For example, in our experiments, deep parameter tuning can improve memory savings by 14%, at the cost of 13% longer offline optimisation time. When deep parameter tuning is not helpful, this extra optimisation-time cost reduces to a mere 0.7%, compared to shallow parameter tuning.

3.1 Motivating Example

We illustrate the idea of deep parameters with an example found by our approach for *dlmalloc* (version 2.8.6) [109].

```
1 static void* sys_alloc(mstate m,size_t nb) {
2 ...
3         if (ss == 0){ //check if first time through
4                 char* base = (char*)CALL_MORECORE(0);
5         ...
6 }
```

Figure 3.1: `sys_alloc` function in *dlmalloc*

Figure 3.1 shows a part of the `sys_alloc` function in *dlmalloc*. We explain its internal operation here to give the reader a feeling for the opportunities for optimisation. Of course, our parameter exposing and search-based tuning are general purpose techniques that have no knowledge of how *dlmalloc* operates. *Dlmalloc* maintains an internal structure to organise the heap for memory reuse. Only when *dlmalloc* cannot find a suitable chunk of memory for a memory request does it call `sys_alloc()` to extend the current heap.

Our approach begins with a form of mutation analysis that evaluates subexpressions in the program to determine their utility as candidate deep parameters. A subexpression is evaluated by mutating it, running the resulting program variant against a test suite, and evaluating the results in terms of functional and non-functional properties. A subexpression that can be profitably mutated to optimise a non-functional property while retaining functional correctness can serve as a deep parameter.

In this example, the mutation analysis finds that mutants generated from mutating Line 4 have a notable effect on the memory consumption and the execution time of *dlmalloc*. We take a close took at Line 4. It calls the `CALL_MORECORE()` function, which takes an integer as input. `CALL_MORECORE()` is a macro wrapping the system call that extends or shrinks the current heap and returns the beginning address of the newly allocated region of the heap. Specifically, `CALL_MORECORE(0)` neither extends nor shrinks the heap but simply returns the current address of the

heap, which is the original purpose of Line 4 mentioned above.

Changing the input value for `CALL_MORECORE()` in Line 4 allows us to control the amount of memory pre-allocated. However, although *dlmalloc* provides several tuneable parameters to programmers, allowing them to adjust behaviours (see Section 5 for details), none of these shallow parameters can affect the `CALL_MORECORE()` function directly. Our algorithm exposes this as a new deep parameter by transforming Line 4 into the code below, where *D* is the deep parameter exposed that controls the pre-allocated heap.

```
char * base = (char*)CALL_MORECORE(0 + D);
```

The optimal size of pre-allocated memory depends on the specific program using this tunable memory allocator. Too much pre-allocation may result in waste. On the other hand, too little means that later requests must call `CALL_MORECORE()` again to extend the heap, increasing runtime. By tuning the deep parameter *D*, an SBSE approach can balance time and space consumption. This is just one example of a potential deep parameter. In our mutation analysis experiments, our tool 'discovers' that by changing the value of this deep parameter, it can achieve a modest (2.5%) time reduction without increasing heap space in one of our subjects.

3.2 Deep Parameter Optimisation

Figure 3.2 shows the work flow of our deep parameter optimisation. The approach takes the source code of the program, a set of test data and a set of non-functional properties of interest. It first applies mutation analysis and a non-dominated rank algorithm to discover potential locations for deep parameters, as explained in Section 3.2.1. It then exposes deep parameters based on the type of expressions found at the locations (Section 3.2.2). Finally, to tune the program, a multi-objective search algorithm is used to search for optimised values for both shallow and deep parameters (Section 3.2.3).

3.2.1 Discovering Locations for Deep Parameters

The first step is to identify potential locations at which we could expose deep parameters. In our approach, we represent the input program as an Abstract Syntax

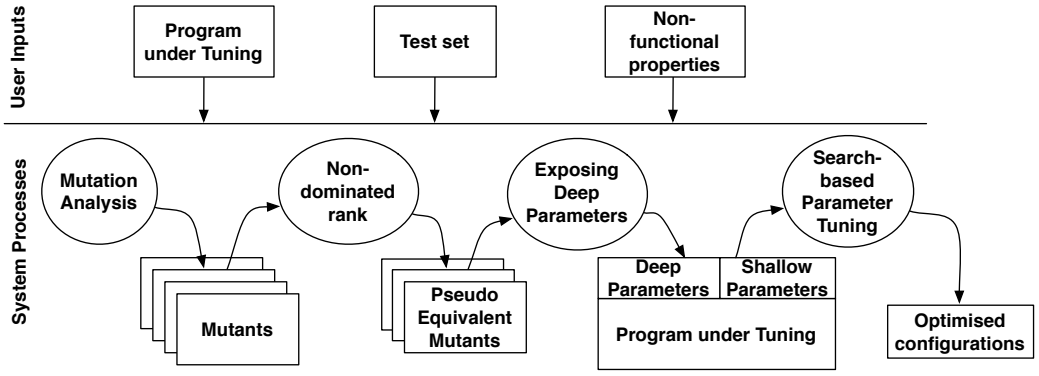


Figure 3.2: Deep parameter optimisation workflow. Given a program, test suite and non-functional properties, our approach applies mutation analysis, exposes deep parameters, and optimises them.

Tree (AST) and a potential location L is an expression node of the AST. We want to find a set of locations L_D such that, when we tune the value of the expression at L_D, some non-functional properties of the program could be improved while the program retains identical functional behaviour. We use a suite of regression test data to validate the correctness from the optimisation, following other established Genetic Improvement approaches [14, 110, 111]. We assume the presence of a test suite with each target application.

We use mutation analysis to automate the process of searching for locations L_D. Mutation analysis deliberately makes simple syntactic changes to the input program, to create a set of various versions of a program called mutants, each containing a different syntactic change [112]. A transformation rule that generates a mutant from the input program is known as a mutation operator. By carefully choosing mutation operators, we can use mutants to simulate the effect of making changes at those locations Ls from which a potential deep parameter may be exposed. We currently use a Mutation Testing tool to automatically mutate all potential locations, this may be improved by static or dynamic code analysis, such as using expression patterns to identify more impacting expressions or only mutating locations that are covered by test cases. Table 3.1 lists the operators we used to generate mutants, covering locations of constants, relational, logical and arithmetic expressions. These so-called 'selective' mutation operators have been widely used in mutation analysis

Table 3.1: Selected mutation operators

Mutation Operators	Changes Between
CRCR – Constant replacement	constants, 0, 1, −1
OAAN – Arithmetic operator	+, −, ⋆, /, %
OAAA – Arithmetic assignment	+=, −=, ⋆=, /=, %=
OCNG – Logical context negation	*expr*, !*expr*
OIDO – Increment/decrement	++x, −−x, x++, x−−
OLLN – Logical operator	&&, ||
OLNG – Logical negation	*x* op *y*, *x* op !*y*, !*x* op *y*, !(*x* op *y*)
ORRN – Relational operator	>, >=, <, <=, ==
OBBA – Bitwise assignment	&=, |=
OBBN – Bitwise operator	&, |

experiments [112].

To assess the quality of a mutant, we test each mutant against the input test set and record the values of the non-functional properties. If the functional result of running a mutant is different from the result of running the original program for any test data in the input test set, then the mutant is said to be 'killed', otherwise it is said to have 'survived'. After all mutants are executed, we first filter out the killed mutants which fail to retain the functional behaviour. A mutant is called **pseudo equivalent** with respect to a given test suite T iff. it passes the regression test of T. Thus we only select pseudo-equivalent mutants which preserve the functional behaviour of the original program while potentially changing non-functional behaviour.

In practice, there is a large number of pseudo-equivalent mutants [96,113,114] generated. We desire a subset of them that represents the locations that could have the greatest impact on the non-functional properties of interest while also maintaining a diversity of choices. We achieve this by ranking the mutants based on their non-functional properties using the non-dominated sorting approach of the NSGA-II algorithm [115]. Each mutant is assigned a Pareto Level value and a Crowd Distance value, where Pareto Level n means a mutant will be on the Pareto Front after all the mutants with Pareto Level less than n are removed, while Crowd Distance indicates how close a mutant is to its neighbours on the same Pareto Level.

For example, a mutant with Pareto Level 1 is on the Pareto Front among all the mutants and has the priority to be considered first. A mutant is better than another in terms of non-dominated sorting if its Pareto Level is smaller or if their Pareto Levels are the same but the former is less crowded (larger Crowd Distance) than the latter. After sorting all the mutants in terms of their non-functional properties, we apply a greedy algorithm to pick the first k locations that could best influence the non-functional properties of the original, where k is the desired number of deep parameters one wants to expose.

3.2.2 Exposing Deep Parameters

The second step is to expose deep parameters that allow users to modify the value of the expression at selected locations. Based on the type of mutation, we first classify the selected mutants into two sets. Set 1 contains mutants generated from CRCR, OAAN, OAAA and OIDO operators, which simulate locations with non-logical expressions. Set 2 contains mutants generated from the OCNG, OLLN, OLNG and ORRN operators, which simulate locations with logical expressions (Table 3.1). Given a location L, E_L is the expression at the location L, we use the following transformation rules to rewrite E_L with a new parameter v_L.

$$E_L \rightarrow \begin{cases} (E_L + v_L) & \text{if } L \in \text{Set 1} \\ (E_L)\ xor\ v_L & \text{if } L \in \text{Set 2} \end{cases} \tag{3.1}$$

We use addition to affect the value of non-logical expression and `exclusive or` to affect the logical ones. Finally, we expose v_L as a 'public' parameter so that users can assign a value to v_L through parameter passing or APIs.

3.2.3 Search-based Parameter Tuning

Although the exposed deep parameters can provide additional 'knobs' [23] to tune the program, a set of k deep parameters need not necessarily subsume the existing shallow parameters of the program. Thus, in this work, we propose to use both shallow parameters and deep parameters and tune them together using SBSE [27]. Because we are interested in multiple conflicting properties, we consider this as

a multi-objective optimisation problem, thus a multi-objective Genetic Algorithm, NSGA-II [115], is applied to search for optimal values for both shallow and deep parameters.

We use an integer vector to represent the tuning parameters. Each integer stores a solution value for one parameter. At each generation, our NSGA-II implementation first applies tournament selection, followed by a uniform crossover and a uniform mutation operation. In our experiments, our fitness functions are designed to capture two non-functional properties: execution time and memory consumption, while preserving the functionality by assigning the worst value to both non-functional properties. To measure execution time, *Glibc*'s `wait4` system call is used to calculate the CPU time (mean of 10 evaluations). For memory consumption, we instrumented the program to record the high-water mark of the virtual memory consumption. We chose this instrumentation approach because the physical memory reported by the OS is not always deterministic but depends on the workload and the OS, and because the virtual memory requirement is an upper bound of the physical memory actually needed. For a subject program with configuration c, we measure the execution time $t_i(c)$ and the high-water-mark of memory consumption $m_i(c)$ of each test case i. Then the fitness functions for the configuration c regarding execution time and memory consumption can be formulated as:

$$f_t(c) = \sum_i t_i(c) \qquad f_m(c) = \sum_i m_i(c).$$

After fitness evaluation, a standard NSGA-II non-dominated selection creates the next generation. Finally, all non-dominating solutions in the final population are returned.

3.3 Experiments

To assess the improvement of our Deep Parameter Tuning approach, we compared it with Shallow Parameter Tuning:

RQ1 How much performance improvement, with respect to the unmodified program, can be obtained by Shallow Parameter Turning using random

search or NSGA-II?

We consider RQ1 to provide a baseline result against which we compare the results from Deep Parameter Tuning. We used NSGA-II algorithm (described in Section 3.2.3) and Random algorithm to search for better values for the shallow parameters in *dlmalloc*, then compare the performance with *dlmalloc*'s default configuration.

RQ2 How much additional improvement can be achieved by our Deep Parameter Tuning algorithm compared with Shallow Parameter Tuning alone?

We ask RQ2 to evaluate how useful our approach is at finding better configurations for the given non-functional properties. In these experiments, our Deep Parameter Tuning approach uses a custom mutation analysis to identify the most sensitive parts of the program, followed by an application of NSGA-II to optimise both explicit and implicit parameters for *dlmalloc*.

RQ3 What are the optimisation-time costs for these approaches to find their solutions?

Since our Deep Parameter Tuning approach exposes additional parameters which are then optimised in conjunction with the baseline shallow parameters, it may require extra resources at optimisation time. We thus measure the baseline cost of Shallow Parameter Tuning, as well as the extra computation required by our Deep Parameter Tuning. The user may use this gain/cost ratio to decide whether to employ Shallow or Deep Parameter Tuning.

3.3.1 Experiment Target

Many memory allocation strategies and managers have been proposed and studied by many researchers to efficiently manage dynamic memory. Among them, Doug Lea's malloc (*dlmalloc*) is "among the fastest, most space-conserving, tunable, and portable general purpose allocators" [109]. A study of Berger et al. [116] shows that many other custom memory allocators do not perform significantly better than

dlmalloc, and are sometimes worse. We focus on *dlmalloc* as an indicative starting point and optimise its configuration to each of the subject applications.

Dlmalloc is a general memory allocator for C programs. Some statistics about the library are given in Table 3.2. Although it provides a number of configuration parameters, it is usually used with its default values. We call these configurable parameters provided by the original author shallow parameters. In these experiments, we consider the nine shallow parameters that are more relevant to the tradeoff between runtime and memory high-water-mark. Table 3.3 briefly describes these shallow parameters.

Table 3.2: *dlmalloc* statistics

	dlmalloc
LoC	3649
# functions	45
# system calls	21
# mutatable locations	3056

Table 3.3: *dlmalloc* selective shallow parameters made available by the developers and used in our experiments

Name	Default	Range
MALLOC_ALIGNMENT	$2 * sizeof(void*)$	(1 – 16)$*sizeof(void*)$
FOOTERS	`false`	`true` or `false`
INSECURE	`false`	`true` or `false`
NO_SEGMENT_TRAVERSAL	`false`	`true` or `false`
MORECORE_CONTIGUOUS	`true`	`true` or `false`
DEFAULT_GRANULARITY	0	4 KB – 512 KB or 0
DEFAULT_TRIM_THRESHOLD	2048 KB	64 KB – 16 MB
DEFAULT_MMAP_THRESHOLD	256 KB	16 KB – 2 MB
MAX_RELEASE_CHECK_RATE	4095	1000 – 10000
Name	Type	Description
MALLOC_ALIGNMENT	$2^n * sizeof(void*)$	Alignment unit
FOOTERS	boolean	Additional information of each chunk
INSECURE	boolean	Secure check
NO_SEGMENT_TRAVERSAL	boolean	Traversal of chunks before coalescing
MORECORE_CONTIGUOUS	boolean	Contiguous heap extension support
DEFAULT_GRANULARITY	2^n KB or 0	Unit of heap extension
DEFAULT_TRIM_THRESHOLD	2^n KB	Threshold of trimming
DEFAULT_MMAP_THRESHOLD	integer	Threshold of direct memory mapping
MAX_RELEASE_CHECK_RATE	integer	Frequency of coalescing

3.3.2 Experiment Setup

For our evaluation, we selected four applications: *espresso*, *gawk*, *flex* and *sed*. *Espresso* is a fast application for simplifying complex digital electronic gate circuits. We use the *espresso* benchmark source code and test cases from the *DieHard* project [117]. *Gawk* is the GNU *awk* implementation for string processing. We collect Version 4.1.0 of this application, as well as its test suite, from the GNU archives. *flex* is a tool for generating scanners, programs which recognise lexical patterns in text, and *sed* is an editor that automatically modifies files given a set of rules. We obtain these last two programs and corresponding test suites from the SIR repository [118]. Summary data for these subject programs is listed in Table 3.4. We used *gcc* optimisation option -O3 to optimise off the trivial changes made by mutation, where the changes are considered equivalent to the original by the compiler and do not affect the non-functional behavior of the program.

Table 3.4: Subject applications

Name	Loc	# Tests	Description
espresso	13256	19	Digital circuit simplification
gawk	45241	334	String processing
flex	9597	62	Fast lexical analyzer generator
sed	5720	362	Special file editor

3.3.3 Experiment Procedures

We first used the shallow parameters only and applied the NSGA-II algorithm with a population of 50 for 300 generations, using 5000 randomly generated chromosomes as seeds. These standard values were chosen after a few trial experiments to have the best performance and ensure a convergent result for the algorithms. A random search was also applied with the same computation budget in optimising the shallow parameters.

We used the open source C mutation testing tool MILU [88, 119] to automatically generate mutants from the selective operators shown in Table 3.1. This mutation based pre-analysis finds the equivalent mutants that are sensitive to the non-functional properties under optimisation. These equivalent mutants are transformed

and exposed into 9 deep parameters (the same number as the provided shallow parameters for a fairer comparison) for each subject program separately, as described in Section 3.2. Combining shallow and deep parameters, we again applied NSGA-II and random search with other identical experimental settings. All experiments were repeated for 20 runs to admit statistical analyses.

All experiments were carried out on desktop machines with a quad-core CPU and 7.7 GB memory running 64-bit Ubuntu 14.04. We used *dlmalloc* version 2.8.6, which was compiled with *gcc* 4.8.1 with -O3 option. To capture the execution time and memory consumption precisely, we developed our own performance tool to measure the CPU time and the high-water-mark virtual memory consumption (see Section 3.2.3). The tool is publicly available at `https://github.com/FanWuUCL/memory`.

3.4 Results

We formalise the metrics we use to compare multi-objective optimisation approaches in this section. The results are presented in Section 3.4.2, and are used to answer the **RQ**s.

3.4.1 Metrics

To investigate RQ1 and RQ2, we collect the non-dominated set of solutions from each algorithm for 20 runs, and report it in an attainment surface as introduced by Fonseca [120]. To quantitatively compare the quality of each algorithm, we calculate Hypervolume and Contribution indicators to assess the multi-objective Pareto Front.

Hypervolume: The Hypervolume indicator [121] measures the space dominated by the solutions. It is defined as the hypervolume of the union of hypercubes dominated by each solution on the Front. The bigger the Hypervolume is, the larger the area dominated by the Pareto Front in the objective space is, and thus the better the performance is.

Contribution: Since there is no way to know the true Pareto Front, we use the non-dominated set of joint solutions from all experiments to approximate the true

Pareto Front, forming a 'reference' front. The Contribution indicator represents the ratio of solutions on the reference front that are found by a given algorithm. A higher ratio indicates a more successful search.

To allow comparison across subject programs, objectives are normalised to the original performance of each subject.

3.4.2 Answers to RQs

For brevity we use *Sha* to refer to shallow parameters and *All* to refer to all parameters including shallow and deep parameters, followed by *Rand* or *NSGA* to indicate the search method used (random search or NSGA-II). For example, *ShaNSGA* refers to using NSGA-II to search for better values for shallow parameters.

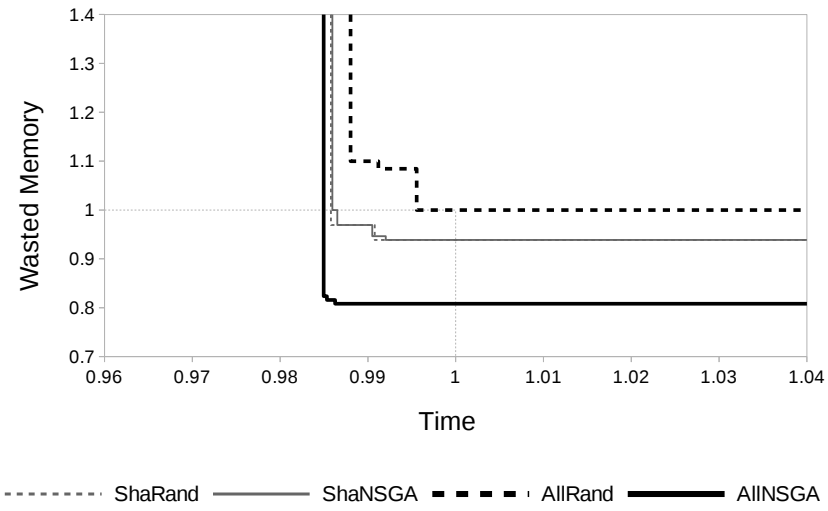


Figure 3.3: Combined best solutions from the results of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* over 20 runs for *espresso*. Lower and lefthand solutions dominate high and righthand solutions. 'Wasted' memory is memory that is used but not needed.

To answer RQ1 and RQ2, we first report the 0%-attainment surfaces (the 'reference front' that combines best solutions over all runs) of the results of *ShaRand*, *ShaNSGA*, *AllRand* and *AllNSGA* on all subjects in Figure 3.3-3.6. The solutions are plotted according to their execution time and memory usage (at the 'high-water-mark') compared to the original performance. Specially, the original always lies at (1, 1) and is pinpointed by light grey dashed lines. The high-water-mark is our primary target since the remaining non-wasted memory is needed and thus cannot be

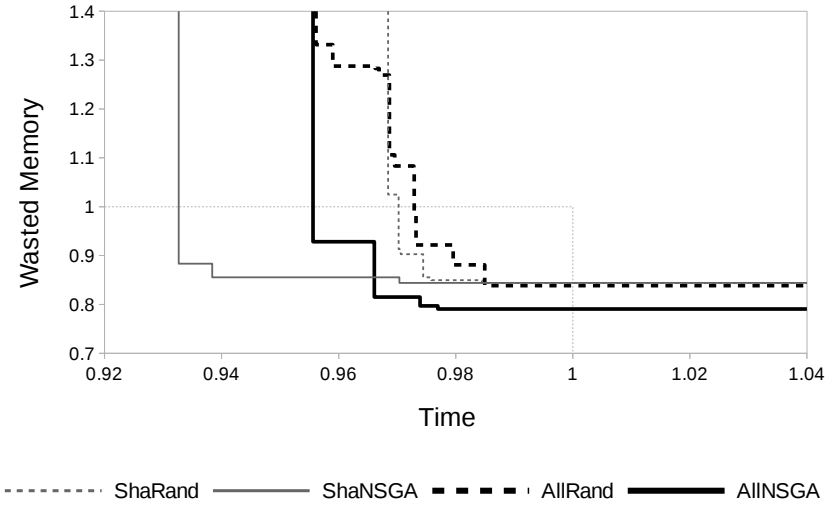


Figure 3.4: Combined best solutions from the results of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* over 20 runs for *gawk*. Lower and lefthand solutions dominate high and righthand solutions. 'Wasted' memory is memory that is used but not needed.

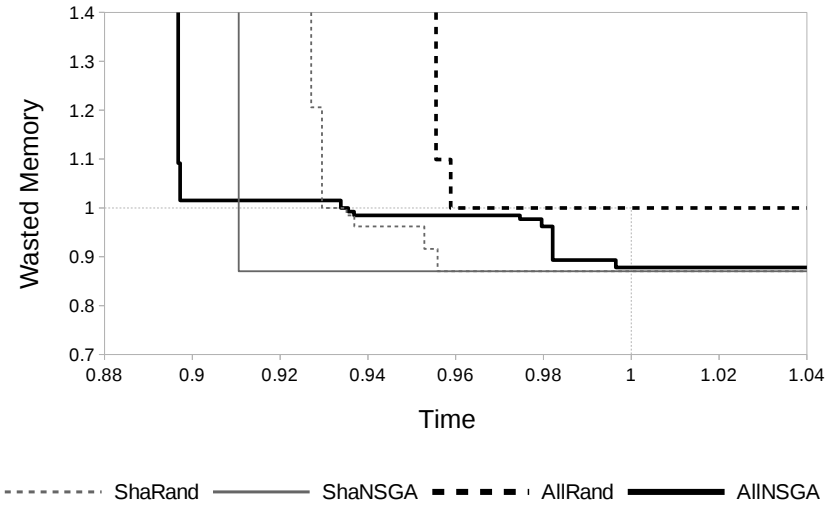


Figure 3.5: Combined best solutions from the results of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* over 20 runs for *flex*. Lower and lefthand solutions dominate high and righthand solutions. 'Wasted' memory is memory that is used but not needed.

reduced. The figure shows that all algorithms can reduce time or memory consumption without reducing the other objective, implying that the default configuration of *dlmalloc* is not optimal for any application considered. This finding motivates the use of SBSE for tuning memory allocators. In three subjects (*espresso*, *gawk* and *sed*), *AllNSGA* outperforms the other three on memory objective. In terms of time, no algorithm is strictly better and each has its own strengths on different subjects.

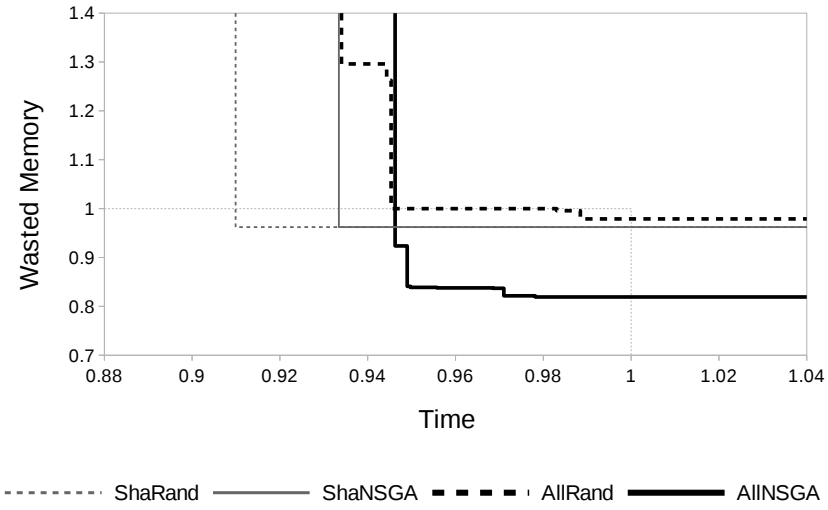
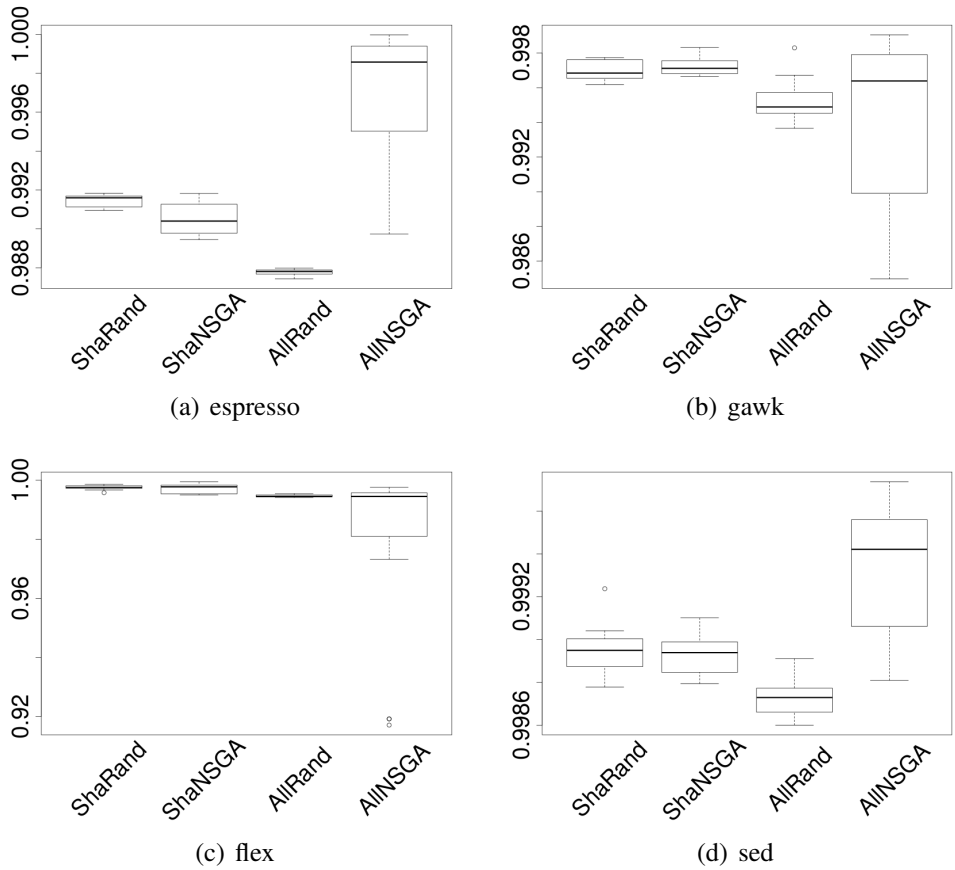


Figure 3.6: Combined best solutions from the results of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* over 20 runs for *sed*. Lower and lefthand solutions dominate high and righthand solutions. 'Wasted' memory is memory that is used but not needed.

We calculated the Hypervolume and Contribution indicator of each algorithm on every subject, and report them in Figure 3.7 and 3.8 respectively for all 20 runs. In Figure 3.7, all the values are normalised to the hypervolume of the 0% attainment reference front, and the closer the value to 1 is, the better the result is. It is clear that *AllNSGA* outperforms the others on subject *espresso* and *sed* while it performs poorly on subject *flex*, and on subject *gawk* the best value reached by *AllNSGA* is better than that of the others. In terms of Contribution, the performance of all algorithms is similar to that of Hypervolume. In general *AllNSGA* is no worse than other algorithms on all subjects but *flex*, where *ShaNSGA* has the highest Contribution value.

Since *AllNSGA* is good at finding better performance on memory consumption, we report the most memory-saving performance found by each algorithm of each of 20 runs in Figure 3.9. On subject *espresso* and *sed*, *AllNSGA* finds more memory reduction than the other approaches. On *gawk*, it does not perform as consistently, but can also find more memory reduction than other approaches in the best case.

Inferential statistical tests were applied to the Hypervolume, Contribution and Best-Memory-Reduction results over all subjects. We used the Mann-Whitney-Wilcoxon U-test since we make no assumptions about results distributions and



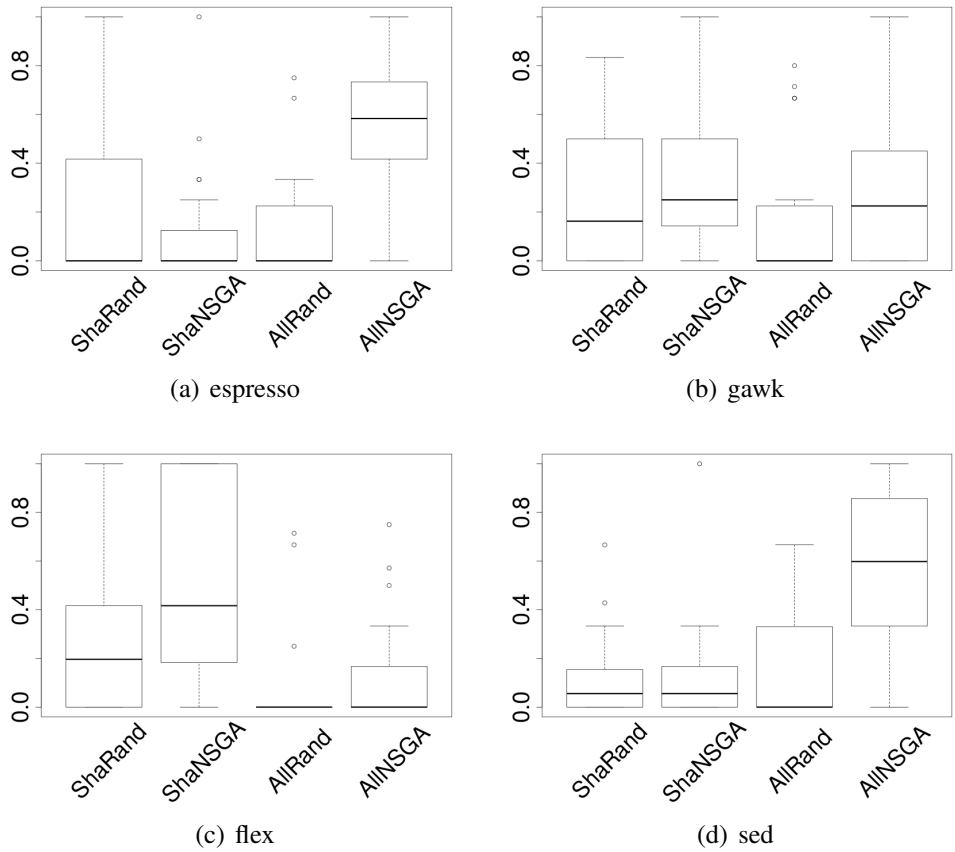
(a) espresso (b) gawk

(c) flex (d) sed

Figure 3.7: Hypervolume indicator of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* on all subjects. Larger values are better.

apply a Bonferroni Correction (catering for 16 total statistical tests) to draw conservative conclusions with no risk of Type 1 error. For those p-values less than $0.05/16 = 0.003125$, we apply the Vargha-Delaney ($\hat{A}_{12}$) effect size measure (see Table 3.5). The effect sizes are all large (either above 0.79 or below 0.21).

In all experiments involving *All** we generated and evaluated invalid configurations (i.e., those that cause the program to crash). However, this issue is not specific to our deep-parameter approach: surprisingly, even by just tuning the programmer-specified shallow parameters (*ShaRand* and *ShaNSGA* optimisations) we *also* encounter (and discard) some configurations that crash the program. This suggests that SBSE memory allocator tuning can be used as a search based testing technique [122]. Without any guidance, *AllRand* finds valid configurations less often than *ShaRand*, and thus requires more optimisation time than *ShaRand*. Holding



(a) espresso (b) gawk

(c) flex (d) sed

Figure 3.8: Contribution indicator of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* on all subjects. Larger values are better.

the searches to the same budget means that *AllNSGA*, which must explore a higher search space, will exhibit a higher variance. Despite this more challenging search space, exposing and optimising deep parameters still allows *AllNSGA* to find better configurations than *ShaNSGA*.

To enable a more quantitative look at maximal time and memory savings, we examine the extreme performance observed in our experiments. We report those that have the best performance on one objective, even at the cost of reducing performance on the other objective, found by each algorithm on each subject and summarise them in Table 3.6. Some of these results are significant departures from the original and are thus not plotted in Figure 3.3-3.6.

To answer RQ3, we provide the average optimisation computation time for each of the approaches in Table 3.7. Recall that *AllRand* generates and evaluates

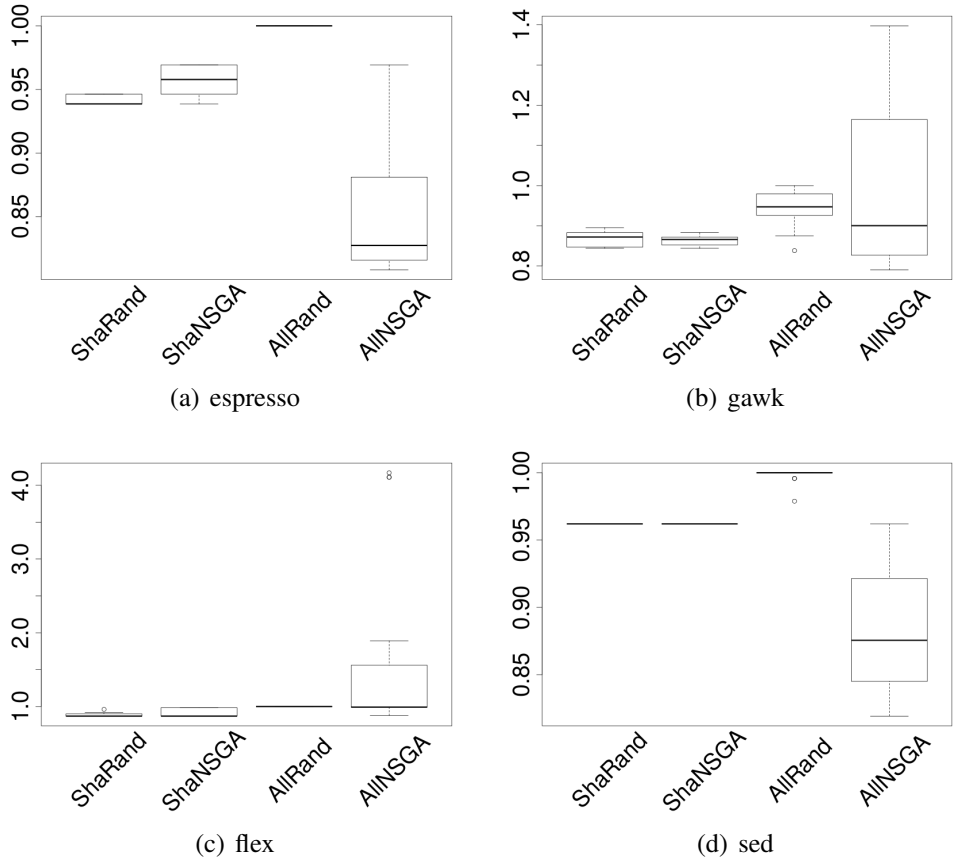


Figure 3.9: The least memory consumption found by each algorithm. Smaller numbers are better.

numerous invalid configurations. However, since crashing or incorrect mutants can be discarded immediately, the computation time of *AllRand* is the lowest among all approaches (given a fixed budget in terms of mutants considered). Similarly, *AllNSGA* generates invalid configurations more often than *ShaNSGA*, so it costs less computation time than *ShaNSGA*. Taking the deep parameter discovery time into account, *AllNSGA* requires slightly more time than *ShaNSGA* does, and the percentage of the extra computation time is reported in the last column of Table 3.7. Ultimately, *AllNSGA* requires at most 18% more computation time than *ShaNSGA* (on *espresso*), but requires only 0.7% more computation time on *flex*, on which *AllNSGA* does not perform as good as *ShaNSGA*. Overall, since this optimisation step is a compile-time rather than run-time cost and can be done before deployment, we view the benefits of deep parameter optimisation as significantly outweighing

Table 3.5: Vargha-Delaney effect sizes of Hypervolume, Contribution and Best Memory Reduction for any two of the approaches on all subjects. Only the effect sizes of tests with p-value less than $5\%/16 = 0.3125\%$ are reported.

Comparing Approaches		Hypervolume			
		espresso	*gawk*	*flex*	*sed*
AllNSGA	*AllRand*	1.000	–	–	0.975
	ShaNSGA	0.935	–	0.105	0.808
	ShaRand	0.900	–	0.035	0.785
AllRand	*ShaNSGA*	0.000	0.053	0.038	0.045
	ShaRand	0.000	0.070	0.000	0.040
ShaNSGA	*ShaRand*	0.198	–	–	–
Comparing Approaches		Contribution			
		espresso	*gawk*	*flex*	*sed*
AllNSGA	*AllRand*	0.859	–	–	0.835
	ShaNSGA	0.868	–	0.191	0.868
	ShaRand	0.814	–	–	0.875
AllRand	*ShaNSGA*	–	–	0.144	–
	ShaRand	–	–	–	–
ShaNSGA	*ShaRand*	–	–	–	–
Comparing Approaches		Best Memory Reduction			
		espresso	*gawk*	*flex*	*sed*
AllNSGA	*AllRand*	0.000	–	–	0.000
	ShaNSGA	0.063	–	0.950	0.050
	ShaRand	0.100	–	0.979	0.050
AllRand	*ShaNSGA*	1.000	0.940	1.000	1.000
	ShaRand	1.000	0.928	1.000	1.000
ShaNSGA	*ShaRand*	0.800	–	–	–

their slight additional optimisation time cost.

3.4.3 What are the Deep Parameters

To have a better understanding of what are those Deep Parameters and how they contributed in the improvement, we manually inspect the Deep Parameters exposed from each subject program and analyse why these parameters could affect the time/memory performance of a program. For 36 Deep Parameters exposed from all four subjects (9 parameters each), they have overlaps, meaning the same parameter could be discovered and exposed for multiple subjects. This is because different subject may have very different memory usage profiles, and the performance usually depends on the test scenarios as well. In the end, there are in total 23 distinct

Table 3.6: Best reduction of time or memory (separately) found by each algorithm

Subject	Time Original (s)	Time Reduction (%)			
		ShaRand	*ShaNSGA*	*AllRand*	*AllNSGA*
espresso	7.24	1.4	1.4	1.5	1.5
gawk	3.43	3.2	6.7	4.4	4.4
flex	0.13	7.9	10.0	6.2	11.6
sed	0.25	9.4	7.0	7.0	5.4
Subject	Memory Original (Peak/Wasted KB)	Wasted Memory Reduction (%)			
		ShaRand	*ShaNSGA*	*AllRand*	*AllNSGA*
espresso	3500/521	6.1	6.1	0	19.2
gawk	29680/3552	15.6	15.6	16.2	20.9
flex	10816/525	13.0	13.0	0	12.2
sed	7048/948	3.8	3.8	2.1	17.9

Table 3.7: Computation Cost in Time

Subject	Optimisation Time (h)				Exposing Time (h)	Extra Time Needed for **NSGA* (%)
	ShaRand	*ShaNSGA*	*AllRand*	*AllNSGA*		
espresso	39.7	46.4	9.0	39.3	12.5	18.5
gawk	22.7	18.4	13.9	16.4	5.4	11.7
flex	7.7	6.3	5.3	5.0	1.3	0.7
sed	9.4	7.6	5.9	6.6	1.9	12.6

Deep Parameters. Among them, five parameters are exposed for strictly two different subjects, one parameter is exposed for three subjects and two parameters are exposed for all four subjects. It is worth mentioning that we observed that, usually the top 4 or 5 locations in each subject are considerably more sensitive than the rest. Therefore, by picking the top 9 most sensitive locations to expose Deep Parameters, we can make sure that the exposed Deep Parameters can have a great impact on the performance.

We manually go through all 23 Deep Parameters and try to explain why they could affect the execution time and/or memory consumption, based on our knowledge to *dlmalloc*. Our observation is as follows: for 12 out of all 23 distinct parameters, we are able to explain why they could affect the performance, and for those 8 parameters that are exposed for more than one subject, 7 of them can be explained.

Listing 3.1: Deep Parameter Examples

```
1 static void* sys_alloc(mstate m,size_t nb)
2 {
```

```
3  ...
4      /* Subtract out existing available top
5         space from MORECORE request. */
6      ssize = granularity_align(nb - m->topsize
7         + SYS_ALLOC_PADDING + EXPOSE_4334);
8  ...
9      if (is_global(m))
10         init_top(m, (mchunkptr)tbase, tsize
11             - TOP_FOOT_SIZE + EXPOSE_4425);
12 ...
13 }
```

To give a better illustration on this, we explain why the two parameters exposed for all subjects can affect the performance in the rest of this section. The examples are given in Listing 3.1. Both of these two Deep Parameters are exposed from the function `sys_alloc()`, which manages most of the allocation from the system. The fist expression containing `EXPOSE_4334` is executed when the size of the top chunk on the heap is smaller than a pre-defined threshold, and is about to be extended in system's dynamic memory. The value of this expression decides how much memory should be applied for from the system this time. Smaller values tend to save memory, but may cause the program to extend the top chunk more frequently, which is a waste of time. Larger values tends to save time by allocating a big chunk of memory at once, but may waste memory if it turns out to be not necessary. How much memory should be applied for at this point may vary from applications to applications, therefore we expose a Deep Parameter from here so that we can control the size to be extended accordingly. The second example concerns when the memory manager initialises the heap. The value of the expression containing `EXPOSE_4425` is for determining how much memory should be pre-allocated at the first time. For similar reasons, bigger values tend to waste memory but save time, and smaller values the other way around. Both of these parameters can be any integer in our approach, so that the optimal values for them are precise

at integer's granularity.

3.5 Threats to Validity

We summarise some potential internal and external threats to validity in this section.

Internal Validity When exposing deep parameters, we used a mutation-based sensitivity analysis because of its advantages in terms of efficiency and automation. Whether it is the best way to expose deep parameters remains to be proven. In addition, we have not formally investigated the relative merits of the Mutation Operators used. Intuitively, our Mutation Operators change a constant or an operator in an expression, and thus are likely to change the values of expressions to different degrees, allowing us to capture the sensitivity of that program's non-functional behaviour to the value of that expression. Any lack of efficacy of these Mutation Operators at capturing sensitivity information introduces a threat to the effectiveness of our approach. A formal evaluation of mutation operators for deep parameter tuning remains as future work.

Another threat to the internal validity is that the execution time measured may depend on the workload of the machine. We mitigate this threat by averaging the execution time of 10 trials on an otherwise-unloaded machine.

External Validity Our choice of benchmark programs and their associated test suites influences the generality of our results. Even a good test suite that achieves high branch coverage, for example, could still differ from real world inputs, in which case the optimised configuration over this test suite may neither achieve the best performance nor retain required functionality. We attempt to mitigate this threat by including two subjects (*flex* and *sed*) from the SIR repository [118]. These subjects come with sets of high-quality test suites, which achieve multiple adequacy testing criteria.

Another aspect of generality is whether these results hold on other applications. We attempt to mitigate this threat by selecting subject applications from different fields, but our results may not generalise beyond these benchmarks.

3.6 Conclusions

In this chapter, we propose a mutation-based method for discovering and optimising deep parameters to tune a C-language library, *dlmalloc*, to adapt to different programs and test cases, with respect to its memory and time performance. Our approach combines mutation analysis to discover sensitive deep parameters as well as an SBSE approach which subsequently optimises these parameters, while retaining the functionality expressed in a test suite. In a series of experiments involving over 70,000 lines of code and 700 test cases we found that our deep parameter approach outperformed baseline optimisations (which use only the programmer-provided shallow parameters), ultimately improving execution time by 12% and memory consumption by 21% in the best cases. In addition, despite the larger search space considered, the additional optimisation time cost of our approach is acceptably low. Overall, we feel that deep parameter tuning approaches show much promise for the automated improvement of software with respect to non-functional properties.

Chapter 4

Memory Mutation Testing

Though by using traditional Mutation Operators, Deep Parameter Optimisation approach can find promising improvement, but these Mutation Operators were not designed to target software's memory performance. In order to directly affect the memory performance by mutation, we proposed new operators that directly target on memory management statement. In this chapter, our objectives are to propose a comprehensive set of Memory Mutation Operators targeting dynamic memory operations, to assess their effectiveness in revealing memory vulnerability, and to compare them with traditional Mutation Operators in terms of the quality of generated mutants and the number of equivalent mutants. These Memory Mutation Operators can be used in future Mutation Testing studies or applications, especially for memory intensive or memory critical programs. Additionally, we shall see their application on optimising memory performance of programs in the next chapter.

we design 9 Memory Mutation Operators, simulating three classes of common memory faults. We also introduce two additional weakly killing criteria, i.e. Memory Fault Detection and Control Flow Deviation for memory mutants. Because memory faults do not necessarily propagate to the output, making the strong killing criterion, which is widely adopted in traditional Mutation Testing, inadequate to detect such faults. A single Mutation Testing tool was developed using both of the traditional and Memory Mutation Operators with the traditional strong killing criterion and the proposed weakly killing criteria also incorporated.

We compare the effectiveness of Memory Mutation Operators against tradi-

tional mutation operators using 18 subject programs with a variety of sizes. Our results show that our memory mutants introduced memory faults that cannot be simulated by traditional mutation operators. We also study the difference between traditional strongly killing criterion with the proposed weakly killing criteria. The results show that, among 1536 generated memory mutants (with 90 TCE-equivalent or duplicate mutants excluded), traditional strong killing criterion killed only 43% of the mutants, leaving 869 mutants unkilled. We also find the two new memory killing criteria introduced are more effective at distinguishing memory mutants, killing up to 80% of those survived mutants across all subject programs.

The primary contributions of this chapter are as follows:

1. The design of 9 Memory Mutation Operators to mimic several categories of memory faults. The mutants generated from these operators can be used to select tests that mitigate memory vulnerabilities.

2. A comprehensive empirical study exploring the characteristics of memory mutation operators and a further empirical study to compare them with the traditional operators. On 16 subject programs, Memory Mutation Operators successfully insert memory faults and generate 368 mutants, 94% of them cannot be simulated by traditional mutation operators. A case study using 2 large programs demonstrates that Memory Mutation Operators are feasible to scale to large programs.

3. The introduction of Memory Fault Detection (using *Valgrind* for precise assessment of memory faults) and Control Flow Deviation as additional killing criteria. This is the first weak killing criteria proposed for memory mutation and the results show that up to 80% of surviving mutants are killed by these additional criteria.

4. An open source C mutation testing tool [1] that features both traditional and Memory Mutation Operators. The tool also supports traditional strong killing

[1] https://github.com/jaysnanavati/Mutate

criteria as well as the Memory Fault Detection and Control Flow Deviation killing criteria.

4.1 Background

Mutation Testing [28] is a white box testing technique that measures the quality/adequacy of tests by examining whether the test set (test input data) used in testing can reveal certain types of faults. A mutation system defines a set of rules (mutation operators) that generate simple syntactic alterations (mutants) of the Program Under Test (PUT), representing errors that a "competent programmer" would make, known as the Competent Programmer Hypothesis (CPH) [39].

To assess the quality of a given test suite, the set of generated mutants are executed against the input test suite to determine whether the injected faults can be detected. If a test suite can identify a mutant from the PUT (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is said to have *survived* (or to be live). A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test suite is inadequate to kill the mutant. The Mutation Score (MS) is used to quantify how adequate a test suite is in detecting the artificial faults. It is calculated as the following formula:

$$MS(P,T) = \frac{\text{number of mutants killed}}{\text{total number of non-equivalent mutants generated}}.$$

P is the program under test and T is the set of tests. However, it is very hard and generally undecidable to determine the exact number of non-equivalent mutants, therefore, the total number of generated mutants is used as an approximation for the number of non-equivalent mutants in practice.

The equivalent mutant problem is a major impediment to large-scale widespread use. Whether a mutant is equivalent has been proven to be undecidable [72, 73]. Although it has been shown that the problem of detecting equivalent mutants cannot be completely automated, approaches to partially solve this prob-

lem have been introduced. They consist of applying compiler optimisation techniques [73] and detecting infeasible paths using static analysis [75]. Other work combines mutants to generate HOMs (Higher Order Mutants) followed by using the number of unit tests that killed FOMs (First Order Mutants) that make up a HOM to identify equivalent mutants [76]. Co-evolution has also been proposed to achieve tailored selective mutation to partially evaluate mutants [77].

To reduce the potential effect of equivalent mutants, we use the Trivial Compiler Equivalence (TCE) technique to automatically detect equivalent mutants in our experiments [74]. TCE is a new approach that finds equivalent mutants by simply compiling each mutant and comparing its machine code with that of the original program. The idea underpinning TCE is that there are several transformation phases when compiling a program into machine code. Many syntactic changes that do not affect the semantics of the program will be ignored by the compiler during these optimisation phases. Though TCE was applied mainly on traditional mutants, the fundamental idea behind TCE does not concern the type of the mutants. The changes to memory operations that do not change the semantics of the program will also be ignored during the compiler optimisation phases, like any other types of changes. Therefore, TCE should be as effective for memory mutants as for traditional mutants. TCE-equivalent mutants are those mutants that have the same machine code as the original after compiled, while TCE-duplicate mutants have the same machine code as other mutants. We use TCE to detect both equivalent mutants and duplicated mutants in our experiments.

4.1.1 Memory Mutation

Though there are many Mutation Testing engines [123–132] for different programming languages, they do not address the issue of evaluating test suite effectiveness with regards to memory-based faults adequately. Using this as a basis for our investigation, we propose a set of mutation operators that aim to model such faults and developed a Mutation Testing tool for C programs that integrates these Memory Mutation Operators.

In traditional Mutation Testing, only the mutant's output is compared against

the original program's output to kill the mutant, while much more information is generated during testing, such as implicit memory faults and control flow graphs. The problem of equivalent mutants may be alleviated by using a richer source of other information to eliminate some seem-to-be equivalent mutants. This is essentially a process of oracle enrichment [133]. In this paper, we propose two more killing criteria for memory-related mutation testing, Memory Fault Detection and Control Flow Deviation. These two techniques enrich the sensitivity of the oracle we can use to distinguish the behaviour of memory faults. The number of detected memory faults in a mutant is used to suggest whether the mutant is potentially equivalent to the original program, while Control Flow Deviation is used to determine whether a mutant executes a different path to the original. More details about Memory Fault Detection and Control Flow Deviation criteria can be found in Section 4.2.2 and Section 4.2.3 respectively.

4.2 Methodology

In this section, we first present 9 Memory Mutation Operators we designed to simulate memory faults, as well as Memory Fault Detection and Control Flow Deviation killing criteria. Then we list the research questions that we aim to answer in this chapter.

4.2.1 Memory Mutation Operators

A set of 9 **Me**mory **M**utation **O**perators (**MeMO**s) are proposed in this project. Each mutation operator mutates calls to memory related function calls (e.g. `malloc()`, `calloc()`, or `free()`), their arguments, or assignments of `NULL`. We divide these mutation operators into three categories based on the types of faults they inject into the code base: *uninitialized memory access*, *faulty memory allocation* and *faulty heap management*. (Faulty memory accesses and faulty static memory management can be mostly simulated by traditional Mutation Operators, thus not included in these proposed operators.)

Table 4.1 lists the proposed operators together with brief descriptions. All of the mutation categories evaluate the inherent vulnerabilities of memory related

Table 4.1: Memory Mutation Operators

Type of faults	Operator	Brief Description
Uninitialized Memory Access	REC2M	Replace `calloc()` with `malloc()`
	RMNA	Remove `NULL` character assignment statement
Faulty Memory Allocation	REDAWN	Replace dynamic memory allocation calls `malloc()`, `calloc()`, `alloca()` and `realloc()` with `NULL`
	REDAWZ	Replace size of the requested block with 0 for dynamic memory allocation functions
	RESOTPE	Replace the arguments of the `sizeof()` unary operator with the pointer type equivalent if a non-pointer type is specified
	REMSOTP	Replace the arguments of the `sizeof()` unary operator with the non-pointer type equivalent if a pointer type is specified
Faulty Heap Management	RMFS	Remove `free()` statement
	REM2A	Replace `malloc()` with `alloca()`
	REC2A	Replace `calloc()` with `alloca()`

functions in C programs which concerned previous works [104, 117, 134].

Each MeMO is detailed in one of the three categories below as well as a rationale behind its choice.

4.2.1.1 Uninitialized Memory Access

The proposed operators in this category generate mutants that can cause uninitialized memory to be accessed in the program. As defined in the C specification, memory allocated by `malloc()` is not guaranteed to be initialized in comparison to `calloc()`, which initialises the memory to 0. **REC2M** replaces instances of `calloc()` with `malloc()` in order to inject uninitialized memory usage faults into the program (Table 4.2).

Table 4.2: Example of **REC2M**

Original Program (P)	`int *array;` `array = calloc(n, sizeof(int));`
Mutated Program (P')	`int *array;` `array = malloc(n * sizeof(int));`

If such a fault were to be left in a program, it may cause the program to exhibit arbitrary behaviour. For example, the code snippet in Listing 4.1 contains such a fault. Specifically, when `flag` is evaluated as `false`, the value of `*array` is not initialised and can be any arbitrary number. Therefore, the subsequent conditional statement may take either branch depending on the arbitrary value of `*array`.

Listing 4.1: Example of the fault represented by REC2M

```
int *array = malloc(sizeof(int));
if(flag){ *array = 1; }
if(*array > 0){ ... }
else { ... }
```

RMNA removes `NULL` assignment statements. Depending on the usage of the mutated pointer, faults such as dangling pointers or dereferencing uninitialized pointer are injected. Table 4.3 shows an example application of the **RMNA** operator, where the mutated program P' is an example of how this mutation operator can inject a dangling pointer fault and introduce undefined behaviour into the application.

Table 4.3: Example of **RMNA**

Original Program (P)	`char *str = calloc(n, sizeof(char));`
	`...`
	`free(str);`
	`str = NULL;`

Mutated Program (P')	`char *str = calloc(n, sizeof(char));`
	`...`
	`free(str);`
	`str;`

As an example, in Listing 4.2, because the pointer `array` was not assigned with `NULL` after `free()`, when the `flag` is evaluated as false, `array` still points to a piece of freed memory which may be allocated to another pointer in the program. In this case, the value pointed by `array` can be any number, causing the subsequent conditional statement to take an arbitrary branch. Operator **RMNA** can simulate such a fault by removing the `NULL`-assignment after a `free()` call.

Listing 4.2: Example of the fault represented by RMNA

```
free(array);
...
if(flag){
    array = calloc(n, sizeof(int));
    array[0] = 0;
```

```
}
if(array != NULL && array[0] == 0){ ... }
else { ... }
```

4.2.1.2 Faulty Memory Allocation

The proposed operators in this category generate mutants that mutate the way memory is allocated in order to measure the effectiveness of test suites at detecting faults such as buffer overflow, underflow and undefined behaviour. Variable length arrays (VLAs) are a class of C arrays that can be declared with a size that is not a constant integer expression, where the size expression is evaluated at runtime. According to the C specification, if size arguments of VLAs are not in a valid range, this could result in undefined behaviour. Moreover, a violation of this constraint does not stop code from being compiled and no compiler warning will be generated.

Table 4.4: Example of **REDAWN**

Original Program (P)	`char *str = malloc(n*sizeof(char));` `strcpy(str, "hello");`
Mutated Program (P')	`char *str = NULL;` `strcpy(str, "hello");`

REDAWN replaces instances of memory allocation calls (`malloc()`, `calloc()`) with `NULL` in order to generate mutants which measure the effectiveness of test suites in identifying faults that occur due to unchecked return value of memory allocation functions. Table 4.4 shows an application of this mutation operator. In the program P in the table, `malloc()` may return `NULL` depending upon the requested size. The subsequent uses of `str` (without checking for any `NULL`-pointer) may cause the program to crash. **REDAWN** can reveal such a bug by forcing the allocation to fail (always return `NULL`).

REDAWZ replaces the request size passed to memory allocation calls with 0 in order to inject `zero allocation` faults. The C specification states that for any of the memory allocating functions, if a memory block of size 0 is requested, the behaviour is implementation-defined, i.e. the value can be a `NULL` pointer or a

unique pointer. One of the problems this can cause is that, for those implementations where allocation functions return a unique pointer, `NULL` checks, which are considered adequate when receiving pointers through dynamic allocation, will pass. This assumption can also lead to tests not detecting such faults. Subsequent uses of such a pointer may cause buffer-overflow problems and contaminate other memory, thus lead the program to undefined behaviours. Table 4.5 shows an application of **REDAWZ** where in P' although allocation failed to return the request size, the program will still trust the allocated pointer and return *true*, unless the programmer wrote special checks to account for such behaviour.

Table 4.5: Example of **REDAWZ**

Original Program (P)	`int num = malloc(n*sizeof(int));` `return num != NULL;`
Mutated Program (P')	`int num = malloc(0);` `return num != NULL;`

RESOTPE and **REMSOTP** mutate the data type of the `sizeof()` operator that is typically used by programmers when dynamically allocating memory. This mutation aims to generate faults that model the incorrect use of the `sizeof()` operator on pointer data types. On some architectures it may be possible that for a given data type T, `sizeof(`T`)` is equal to `sizeof(`T`*)`. This may lead programmers that lack understanding of the C programming language to believe that this is indeed the case. However, the effect of this is: `sizeof(`T`)` returns the size of the data type T itself, while `sizeof(`T`*)` returns the size of a pointer (to T). Moreover, the C standard allows pointers to different types to have different sizes, e.g. `sizeof(char*)` is not necessarily the same as `sizeof(int*)` which implies that `sizeof(`T`*)` is not guaranteed to always be the same regardless of the type of T. Faults of this nature can cause incorrect memory allocations and lead to buffer-overflows. Table 4.6 shows applications of these mutation operators. For instance, `sizeof(char)` returns 1 and `sizeof(char*)` returns 4 on a 32-bit Linux system. In the mutated program P' of **REMSOTP** in Table 4.6, the allocated memory is less than desired. As a result, the subsequent uses of this memory may cause buffer-overflows.

Table 4.6: Examples of **RESOTPE** and **REMSOTP**

	RESOTPE
Original Program (P)	`char *str;` `str = malloc(n*sizeof(char));`
Mutated Program (P')	`char *str;` `str = malloc(n*sizeof(char*));`
	REMSOTP
Original Program (P)	`char **str;` `str = malloc(n*sizeof(char*));`
Mutated Program (P')	`char **str;` `str = malloc(n*sizeof(char));`

4.2.1.3 Faulty Heap Management

The proposed operators in this category generate mutants that model faults that can occur due to improper memory management and also to test the effectiveness of test suites in handling events where allocation functions may fail due to a lack of free memory.

RMFS mutates instances of the `free()` standard C library function by removing instances of `free()` in order to inject memory leaks into the program (Table 4.7). As an example, the `free()` statement is removed in the mutated program in Table 4.7. Because the `free()` statement and the memory allocation for the same pointer are nested in a loop, removing the `free()` statement will cause the memory pointed by `data` be lost, thereby leading to memory leaks.

Table 4.7: Examples of **RMFS**

Original Program (P)	`T *data;`
	`for (int i = 0; i < m; i++){`
	`  data = malloc(n*sizeof(T));`
	`  ...`
	`  free(data);`
	`}`

Mutated Program (P')	`T *data;`
	`for (int i = 0; i < m; i++){`
	`  data = malloc(n*sizeof(T));`
	`  ...`
	`}`

REM2A and **REC2A** replaces instances of `malloc()` and `calloc()` with `alloca()` (Table 4.8). Mutants generated by these operators dynamically allocate

Table 4.8: Examples of **REC2A** and **REM2A**

	REC2A
Original Program (P)	`int *nums;` `nums = malloc(8*sizeof(int));`
Mutated Program (P')	`int *nums;` `nums = alloca(8*sizeof(int));`
	REM2A
Original Program (P)	`int *nums;` `nums = calloc(8, sizeof(int));`
Mutated Program (P')	`int *nums;` `nums = alloca(8*sizeof(int));`

memory on the stack instead of the heap. If a pointer to that memory is dereferenced after the function containing the allocating call finished, the pointer is dangling and the memory may have meanwhile been overwritten. Another problem with using `alloca()` is the fact it is not guaranteed to return `NULL` if it fails to find enough space on the stack and, depending on the implementation, it may cause some parts of the stack to be overwritten. On the other hand, as memory allocated using `alloca()` is automatically freed once the function that called it returns to its caller, it is possible that this mutation may introduce double-free fault in the program. For instance, the program snippet shown in Listing 4.3 uses `alloca()` to allocate a piece of memory. However, this piece of memory is automatically freed at the end of `func` and the memory pointed by `data` may be overwritten. Therefore, the subsequent conditional statement may depend on an arbitrary value. Furthermore, the following `free()` statement will fail and may cause the program to crash.

Listing 4.3: Example of the fault represented by REC2A/REM2A

```
void func(T *data){
        data = alloca(sizeof(T));
}
void test(){
        T *data;
        func(data);
        ...
```

```
        if(*data){ ... }
        free(data);
}
```

4.2.2 Memory Fault Detection

Traditional Mutation Testing only uses the test output of the mutants and the test output of the original to (strongly) kill the mutants. However, executing the mutants against a test suite generates much richer information that is neglected in traditional Mutation Testing. Memory Fault Detection (MFD) is a weakly killing criterion we propose that uses one kind of such additional information: memory faults. If a memory fault is detected in a mutant but is not detected in the original, one can be sure that the memory fault must have been introduced into the mutant by the mutation operator, thus the mutant can not be equivalent to the original in every sense of "equivalent". Memory Fault Detection can reveal presence of memory faults that do not always propagate to the output, so some of the memory faults may not reveal themselves in the test outcome, yet we still have other ways to detect them during the execution.

Listing 4.4: Example of memory fault propagation

```
1  int foo(int* array, int size){
2      if(size == 0){
3          free(array);
4          array = NULL; // RMNA will remove this line
5      }
6      if(array == NULL || array[0] == 0){
7          return 0;
8      }
9      return 1;
10 }
```

Listing 4.4 gives an example of a memory fault introduced by **RMNA**. After operator **RMNA** is applied, the `NULL` assignment will be removed, which will introduce a dangling pointer and a use-after-free defect. Depends on the values of the arguments, the bug may or may not propagate to the output. Specifically, when `size` is not 0, the defected block is not reached. If `size` is 0 and `array` is a `NULL`-pointer, the defected block is reached but the value of `array` is not infected, therefore the defect will not propagate to the output. On the other hand, if `size` is 0 and `array` is not a `NULL`-pointer, the second conditional statement may rely on the value of a freed space and the return value can be arbitrary, therefore the defect can propagate to the output and subsequently be detected by tests. When we apply *Valgrind* to this mutant, a memory fault will be detected and reported, therefore we can weakly kill this mutant even if the defect is not propagated to the output.

In this chapter, we propose using *Valgrind* [104] to detect memory faults as an additional weak mutation killing criterion. This criterion requires the memory fault detection tool to have consistently deterministic results, independent of the workload and running Environment of the platform. *Valgrind* uses a simulated environment to execute the subjects, independent of the environment of the host machine, therefore it always provides consistent results. Despite other tools that can also detect memory faults, we choose *Valgrind* because it is a stable tool and widely used for memory fault detection. *Valgrind* can easily be changed to other memory fault detection means.

For each mutant and each test, we evaluate the number of memory faults occurred during execution using *Valgrind*, written $\text{MFD} : P \times T \mapsto \mathbb{N}$. We say a mutant M_i is killed by Memory Fault Detection criterion if $\text{MFD}(M_i,t) \neq \text{MFD}(P_{\text{UT}},t)$ for some test case(s) t, where $\text{MFD}(P,t)$ is the number of memory faults found when executing program P against test case t. Notice that the original program may also contain some memory faults, we only care whether a mutant contains 'different' memory faults from that in the original. However, it is very hard to precisely identify and distinguish newly introduced memory faults from the old ones. We simplify this by only looking at whether the number of memory faults from the mutant

is different from that from the original, based on the assumption that if new memory faults are introduced by an operator, the mutant will have a different number of memory faults compared with the original.

Since MeMOs are designed to simulate real memory faults, we expect to see a large number of mutants generated from MeMOs that can not be killed by the traditional criterion but are killed by MFD criterion.

4.2.3 Control Flow Deviation

A program can be considered to be a network of nodes which represent branches in the program and each test as a possible entry point into the network. If we consider the execution of each mutated program as a path from a starting node (the point in the program execution at which the mutant is executed) to a sink node (the point at which the program terminates), then the set of mutants which generate a path that is different in comparison to the path generated by the PUT can be classified as weakly non-equivalent mutants.

Similar to the representation of control flow graphs [135], in Figure 4.1, test cases are represented by circles labelled $t_1, \ldots, t_7$, the mutation execution point is labelled as M, the set of program branching states that make up the control flow for a given program are labelled $s_1, \ldots, s_7$ and the sink state (program termination) is labelled s_8. Figure 4.1 also shows an example execution of test case t_4 (two graphs to the left) where the control flow of the mutated program M $\langle t_4, s_1, s_3, s_7, s_8 \rangle$ is able to deviate from the control flow of the original program P $\langle t_4, s_1, s_3, s_5, s_6, s_8 \rangle$, whereas in the execution of test case t_7 (the graph to the right), the execution paths are the same $\langle t_7, s_1, s_2, s_3, s_7, s_8 \rangle$. We consider this kind of deviation as a distinguishing factor that can help reduce the set of survived mutants. We also consider this deviation as a sign of test suite weakness, since the deviation shows that a mutant was able to execute undesired code and pass all the test cases regardlessly.

Let $CFG(P)$ be the control flow graph of program P and let $E(G,t)$ be the edge set of graph G for the execution of test case t. We say a mutant M is killed by the

Control Flow Deviation (CFD) criterion, iff

$$\begin{aligned} \exists t \in T : \quad & (E_O = E(CFG(P_{\mathrm{UT}}), t)) \\ \wedge \quad & (E_M = E(CFG(M), t)) \\ \wedge \quad & ((E_O \cup E_M) - (E_O \cap E_M) \neq \emptyset) \end{aligned}$$

On the other hand, if $E(CFG(M),t)$ is the same as $E(CFG(P_{\mathrm{UT}}),t)$, the mutant survives. For example, in Figure 4.1 the original program is mutated at state s_3, and test case t_4 drives the execution path to different states from s_3 in the mutated program (graph in the middle) and the original program (graph to the left). In such case, $(E_O \cup E_M) - (E_O \cap E_M) \neq \emptyset$ hence the mutant is killed regardless of other test cases.

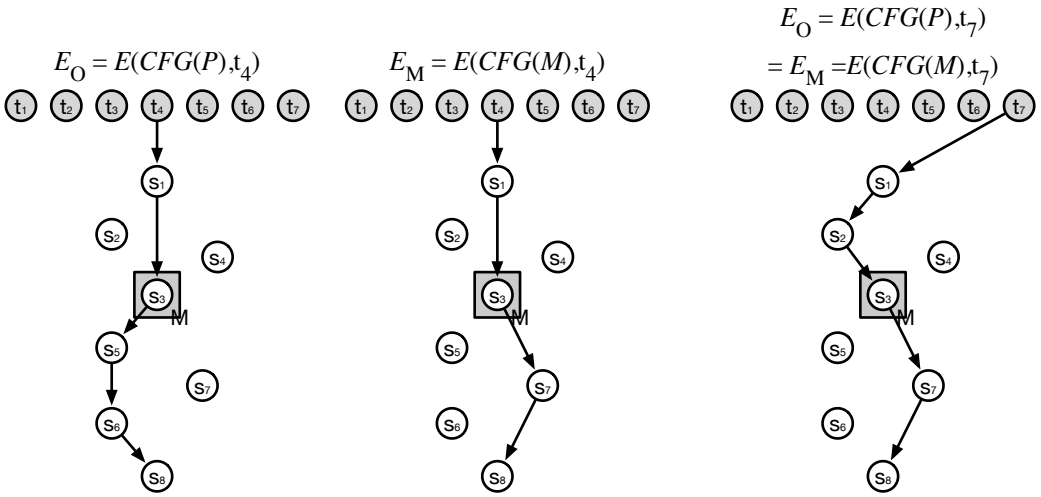


Figure 4.1: Control Flow Deviation. Test t_4 generates different control flow before and after the mutation at state s_3 (the two graphs to the left). Test t_7 generates the same control flow (the graph to the right).

In our experiments, we use *GCov* at the branch level to determine the execution path of each test case. Therefore, we require the subjects to be single-threaded and deterministic, otherwise *GCov* may fail to provide accurate information on some executions. We use CFD criterion to further reduce the number of survived memory mutants. However, CFD criterion is not specially designed to detect memory faults, thus can be adopted by any Mutation Testing framework.

4.2.4 Research Questions

This section presents the research questions concerning memory mutation operators and the memory weak killing criteria, for which Section 4.4 provides the answers.

RQ1 What is the prevalence of memory and traditional mutants overall and per program studied?

RQ2 What is the contribution of each memory mutation operator to the proportion of memory mutants and TCE equivalent mutants found?

Our memory mutation operators are the first set of mutation operators designed to simulate a wide range of memory faults. Therefore, the natural first research question we consider concerns whether they can effectively inject artificial memory faults. We answer this question by comparing the prevalence of the memory mutants and mutants generated by traditional mutation operators. In RQ2, we take a close look at each individual Memory Mutation Operator and analyse the contribution of each memory mutant operator to the proportion of memory mutants and TCE equivalent mutants. Since it is not trivial to determine whether a mutant is equivalent to the original program, we adopt the TCE technique [74] to estimate a lower bound of the number of equivalent mutants.

RQ3 What is the quality of memory mutations compared with traditional mutants? What proportion of memory mutants are TCE-duplicates of the mutants generated by traditional operators?

RQ3 studies the quality of the memory mutants generated. We use mutation score to approximate the quality of mutants. Given a test suite and two sets of mutants, we say the set of mutants yielding a lower mutation score has a higher quality, because the faults simulated by the mutants are more difficult to detect. To answer this question we run both memory mutants and traditional mutants against a set of good quality unit tests and compare their mutation score. In RQ3, we also investigate how many memory mutants can be emulated by traditional mutation operators. Memory mutants will be redundant if many of them are duplications

of the mutants generated by the traditional operators. Again, we adopt the TCE technique to assess whether a mutant is a duplicate of another. Because memory faults are different from other kinds of faults, we expect to see a low percentage of memory mutants that are TCE-duplicate to traditional mutants.

RQ4 How effective are the memory weakly killing criteria? What is the reduction rate of survived memory mutants?

RQ5 What is the contribution and unique contribution of strongly killing, MFD and CFD criteria?

RQ4 investigates the effectiveness of the memory weakly killing criteria. When only the traditional strongly killing criterion is applied, some of these memory mutants may be not equivalent, yet still survive. This is due to the fact that some memory faults do not propagate to the output. After weakly killing criteria are applied, we expect to see that most of these mutants are identified and killed. To answer RQ4 we compare the Mutation Scores before and after weakly killing criteria are applied, and what is more important, how much we can reduce the survived mutants such that we can leave less survived mutants for further investigation of their equivalence.

From another perspective of view, we are interested in how much each of the strongly killing criterion, MFD and CFD criteria contributes in killing those non-equivalent mutants. Specially, if we find all of the mutants that are killed by one criterion, can be killed by another criterion, then the first criterion is totally subsumed by the second one, thus can be abandoned without affecting the results. More formally, the relation between the mutants killed by each criterion can be easily understood by a Venn Diagram in Figure 4.2. Sets T, M and C contain the mutants killed by a test suite when the traditional (T), Memory Fault Detection (M) or Control Flow Deviation criterion (C) is applied respectively. The Mutation Score with respect to a specific criterion can be calculated as ($S \in \{T, M, C\}$):

$$MS = \frac{|S|}{|\text{AllMutants}|}$$

In the final research question, we are interested in the sizes of the sets T, M and C in the proportion of the region $K = T \cup M \cup C$. They correspond to how much each criterion contributes to the identification of those non-equivalent mutants (K). For example, the bigger the proportion of set T in K is, the greater will be the contribution that the traditional criterion makes in identifying non-equivalent mutants. Therefore, the more thoroughly we can kill non-equivalent mutants by applying the traditional criterion alone. Furthermore, we would like to know the unique contribution of each criterion, i.e., the area covered by one criterion but not covered by any other criterion. The unique contribution corresponds to how many non-equivalent mutants can only be identified by the corresponding criterion. If a criterion has zero-unique-contribution, or, in another words, all non-equivalent mutants that are identified by this criterion can also be identified by other criterion/criteria, then we can completely drop this criterion without losing Mutation Score accuracy. Formally, we define the contribution c and unique contribution u metrics as follows.

$$c_S = \frac{|S|}{|K|}, u_S = \frac{|S - \bigcup_{I \in \{T,M,C\}, I \neq S} I|}{|K|}, \text{for } S = T, M, C$$

For example, c_T is calculated as the size of T divided by the size of K, or the union of T, M and C, u_M is calculated as the size of the area covered by M but not covered by T or C, divided by the size of K. Specially, if there is a criterion S having $u_S = 0$, then the criterion is subsumed by other criteria, thus there is no need to apply this criterion when other criteria are applied.

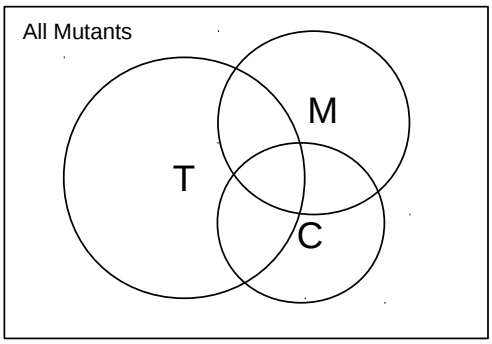


Figure 4.2: Venn Diagram of the relation between mutants killed by each criterion. Sizes of the circles do not correspond to the real data.

4.3 Experiments

We built a Mutation Testing tool targeting C programs which supports Memory Fault Detection and Control Flow Deviation killing criteria. Some key components of the framework are introduced in Section 4.3.1 and the experiment setup is described later in Section 4.3.2.

4.3.1 Mutation Testing Framework

The overall framework of our Mutation Testing tool is illustrated in Figure 4.3. It takes a program under test (PUT) and associated test suite as input, as well as a configuration file containing paths to the source files to be mutated and other parameters. After generating mutants and testing them against the test suite, the mutation report for each mutant is generated, which contains the mutated point with the Mutation Operator applied, whether the mutant has survived or been killed, and a list or criteria that kill it if it is killed.

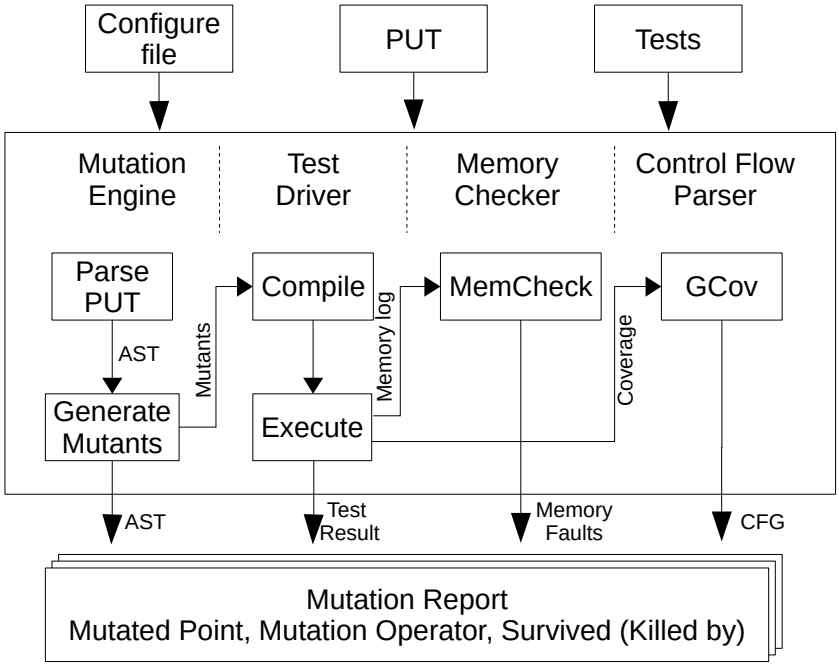


Figure 4.3: Mutation Testing Tool work flow. It takes a program under test, a test suite and a configure file as input, and outputs the mutation report for each mutants generated.

The mutation engine in our tool was developed using *TXL* source transformation language [136]. *TXL* is a functional language, which provides a rule-based

way of traversing and mutating the generated abstract syntax tree of a source file. It has been used by many of previous works for Mutation Testing [123–128]. One big advantage of using *TXL* is that the mutations are guaranteed to be safe, due to the fact that the transformational rules are confined by a grammar which guarantees the transformations will always be compilable and abide by the grammar of the language.

After the mutants are generated, some basic mutation information including the mutation point and the Mutation Operator applied are stored as part of the mutation report for each mutant. The test driver then instruments the mutants and compiles them. After the compilation, all the mutants are executed against the test suite, which generates three independent results: the test result indicating whether a mutant passes the regression test, the memory log including potential memory faults and the coverage information.

To detect memory faults, the mutants are executed by *Valgrind*'s *memcheck* tool [104] which generates the memory fault report for each mutant. The control flow information is gathered using *Gcov* [137] to generate the control flow graph for each mutant. All of this information is summarised in a single mutation report for each mutant as the output of our tool.

4.3.2 Experimental Setup

With the framework introduced in Section 4.3.1, we applied MeMOs to 18 "real world" C programs that implement unit tests using CuTest C testing framework [138] (Table 4.9). We separate them into two groups: the first 16, relatively small, PUTs are fully studied to answer the Research Questions, and 2 large PUTs are analysed as a case study to verify the results of smaller subjects and to demonstrate the scalability of our overall approach.

Libksi, the largest subject used, is an API library for *Keyless Signature Infrastructure (KSI)* developed by the security company *Guardtime*. The other large subject, *libdssl*, is a toolkit for network capture and SSL decryption. *PeerWire-Protocol* is a C-based implementation of the *BitTorrent* peer wire protocol. This program facilitates the exchange of file blocks between peers over the wire in a

Table 4.9: Subject Programs under Test

PUT NO.	Program	LoC	# of Tests	Statement Coverage(%)	# of Memory Management Calls (`malloc()`, `calloc()`, `free()`)
1	PeerWireProtocol	1547	54	82.57	27
2	Craft	731	71	85.96	12
3	CfixedArraylist	497	1	26.19	12
4	ChashMapViaLinkedList	488	21	94.58	8
5	CAVLTree	405	9	56.94	4
6	CpseudoLRU	384	16	98.68	5
7	CHashMapViaQuadraticProbing	391	21	98.27	5
8	CtextureAtlas	301	7	97.65	4
9	Csplaytree	319	14	99.20	5
10	CstreamingBencodeReader	371	30	91.11	4
11	CSparseCounter	328	30	91.67	10
12	Cheap	207	16	98.13	2
13	CcircularBuffer	118	12	77.05	2
14	ClinkedListQueue	200	7	72.63	7
15	CbipBuffer	118	12	86.00	2
16	Cbitfield	87	4	75.81	6
L1	libdssl	6600	48	86.05	69
L2	libksi	19445	130	62.02	201

P2P BitTorrent system. *CRaft* is a C-based implementation of the Raft Consensus protocol which implements the Raft Consensus Algorithm in order to manage multiple fault-tolerant distributed systems. *CpseudoLRU* implements the Least Recently Used (LRU) caching algorithm and *CstreamingBencodeReader* is a library for reading and manipulating encoded data, which is the encoding used by the peer-to-peer file sharing system *BitTorrent* for storing and transmitting loosely structured data. The rest of the programs are implementations of common memory intensive data structures whereas.

The main reason for choosing these programs is the fact that they are non-trivial real-world programs ranging from 87 to 19445 lines of code, making them of manageable size for our simple proof-of-concept research tool. Moreover, they were used in previous studies as memory intensive benchmark subjects, and were known to make extensive use of memory-based operations. Additionally, these programs come with real-world test suites and half of them achieve over 90% of statement coverage. In many cases, the regression test suite is well designed for testing a program's functional behavior, therefore it may not necessarily be good at exploiting and revealing memory vulnerabilities of the program under test. For instance, special memory tests should be designed to reach a certain potential memory defect and

the input value should be adjusted to trigger the defect. Though in this study, we did not use a specially designed memory test suite for testing, the default regression tests of these subjects can be used for memory testing, because they are memory-intensive benchmarks that are well tested for their memory behaviors. Additionally, memory defects may not propagate to the output of the program, in which case the traditional oracle, validating the output, may fail. Therefore, in this study, we introduced Memory Fault Detection and Control Flow Deviation along with traditional oracle for memory testing. The number of tests, statement coverage and the number of memory management calls for each subject are also reported in Table 4.9. All of the subject programs can be found in GitHub repositories.

After we generated mutants from these subject programs, we use traditional strongly killing criterion and both strongly and weakly killing criteria (MFD/CFD) to kill the mutants. For each mutant, we collect the information of whether it is killed and under which criterion if it is killed, and summarise this information to answer research questions. In order to compare memory mutants with traditional mutants, we adopted traditional selective mutation operators to generate traditional mutants. These mutation operators are list in Table 4.10. We use *gcc* to compile all the traditional and memory mutants, then use *diff* to check their TCE-equivalence to each other and the original.

Table 4.10: Traditional Selective Mutation Operators

Operator	Description
ABS	replaces expressions `expr` with `abs(expr)` or `-abs(expr)`
AOR	replaces `+`, `-`, `*`, `/`, `%` with each other
LCR	replaces `&&` and `||` with each other
ROR	replaces >, >=, <, <=, == and != with each other
UOI	replaces variable `v` with `v++`, `++v`, `v--` or `--v`

All experiments are run in a Ubuntu 14.04.3 system with 8GB RAM. *TXL* version 10.6a is used to transform programs, *gcc* version is 4.8.4 and *-O3* is used to optimise programs.

4.4 Results

In this section, we answer the Research Questions, one by one, using the experimental results of the first 16 subjects, then compare and discuss the results of 2 large subjects in Section 4.4.6.

4.4.1 RQ1. Prevalence of Memory and Traditional Mutants

To understand the prevalence of memory mutants and traditional mutants, answering RQ1, we report the number of each type of mutants generated per program, as shown in Table 4.11 (2nd and 6th column). Overall, the results indicate that Memory Mutation Operators generate much fewer mutants than traditional mutants. In total there are 368 memory mutants generated, only accounting for 2.8% of the mutants generated by traditional operators. This suggests that using memory mutation is practical as it only introduces a very small number of additional mutants.

We can see that all programs, no matter how small, do possess memory mutants, ranging between 5 and 68 mutants. The table also shows almost a quarter of the memory mutants come from the program *PeerWireProtocol* and *Craft*, while two buffer data structures *circularBuffer* and *bipBuffer* contribute the least memory mutants. To determine the relationship between the prevalence of the memory mutants and the traditional mutants, we computed the Spearman rank correlation. The result suggests there is a strong positive correlation between the two (ρ = 0.725, p = 0.001), and the correlation is statistically significant.

We also present the numbers of TCE-equivalent and duplicate mutants in the table. The numbers of TCE-equivalent mutants are presented in Column TCE-equivalent, and TCE-duplicate columns give the numbers of mutants that are duplicates of other mutants in the same type (memory or traditional), after ruling out TCE-equivalent mutants. Note that for each duplicate-group (within which the mutants are TCE-equivalent of each other), one and only one mutant need to be preserved as a representative of the group, while the others can be discarded. We report the number of TCE-duplicate mutants that can be discarded in the table. Among 368 generated memory mutants, there are 19 or 5.2% TCE-equivalent or duplicate mutants, while the ratio is 17.4% for traditional mutants. This shows MeMOs may

Table 4.11: Number of mutants and Mutation Scores for memory and traditional mutants by subject. The numbers of mutants that are TCE-equivalent of the originals are given in "TCE-equivalent". The column "TCE-duplicate" shows the number of mutants that can be discarded due to duplication of other mutants. (The mutant that is preserved from each duplicate-group is not included in the numbers)

PUT No.	Memory Mutants				Traditional Mutants			
	Generated	TCE-equivalent	TCE-duplicate	Mutation Score	Generated	TCE-equivalent	TCE-duplicate	Mutation Score
1	68	1	0	**0.433**	1598	229	61	0.659
2	49	3	0	**0.457**	1579	240	0	0.795
3	39	11	1	**0.111**	570	384	13	0.295
4	23	0	0	**0.652**	820	93	35	0.767
5	22	0	0	**0.364**	1542	268	56	0.550
6	14	0	0	**0.714**	927	81	14	0.731
7	20	0	0	0.800	927	110	66	**0.772**
8	20	0	0	**0.600**	573	73	8	0.679
9	14	0	0	0.786	802	91	33	**0.656**
10	20	0	0	**0.700**	944	119	28	0.764
11	31	2	0	**0.310**	1173	53	24	0.796
12	9	0	0	**0.667**	572	70	35	0.846
13	5	0	0	**0.400**	409	36	7	0.773
14	18	1	0	**0.353**	367	35	3	0.623
15	5	0	0	**0.400**	293	10	0	0.703
16	11	0	0	**0.182**	237	26	16	0.733
Total	368	18	1	**0.476**	13333	1918	399	0.712

introduce much less equivalent or duplicate mutants than traditional operators.

4.4.2 RQ2. Contribution of Memory Mutation Operators

To take a close look at each Memory Mutation Operator, we report the number of mutants generated from each MeMO, as well as the percentage of their contribution to all memory mutants (**RQ2**). Moreover, we report the numbers that are TCE-equivalent to the original as an estimated lower bound on the number of equivalent mutants. The results can be found in Table 4.12.

The numbers of generated mutants from all Memory Mutation Operators are comparable (except **REMSOTP**), contributing 7.9% to 17.7% of all generated mutants (1.4% for **REMSOTP**). Recall that **REMSOTP** only applies to the unary operator `sizeof()` where the argument is a data type pointer, which is not a common case. Therefore it is anticipated that **REMSOTP** generates fewer mutants than other operators. Among all 368 memory mutants, 18 (or 4.9%) of them are TCE-equivalent to their original programs. The number of TCE-equivalent mutants varies from 0 to 7, or 0.0% to 20% of generated mutants across all MeMOs. None

Table 4.12: Number of mutants and TCE-equivalent mutants by operator.

Category	Mutation Operator	Generated Mutants (% of all mutants)	TCE-equivalent Mutants	TCE-equivalence Rate (%)
Uninitialized Memory Access	REC2M	30 (8.1%)	0	0.0
	RMNA	39 (10.6%)	7	17.9
Faulty Memory Allocation	REDAWN	65 (17.7%)	2	3.1
	REDAWZ	63 (17.1%)	2	3.2
	RESOTPE	53 (14.4%)	5	9.4
	REMSOTP	5 (1.4%)	1	20.0
Faulty Heap Management	RMFS	54 (14.7%)	0	0.0
	REM2A	29 (7.9%)	1	3.4
	REC2A	30 (8.2%)	0	0.0
All		368 (100%)	18	4.9

of the mutants generated from **REC2M**, **RMFS** or **REC2A** are TCE-equivalent to their originals, implying they may generate less equivalent mutants as well.

On the other hand, about 20% of the mutants generated from **RMNA** or **REMSOTP** are TCE-equivalent to their originals. Recall that **RMNA** only removes `NULL` assignment, this makes it more likely to generate TCE-equivalent mutants if the variable being assigned `NULL` is never used later. **REMSOTP** replaces the argument of `sizeof()` from `type *` to `type`, which can also easily generate TCE-equivalent mutants if they are of the same size and can be determined at compilation. The answer to **RQ2** is that, all MeMOs contribute equally to all generated memory mutants, while **REMSOTP** contributes slightly less, and only 4.9% of all mutants are TCE-equivalent, while the ratio varies from 0.0% to 20.0% across all MeMOs.

4.4.3 RQ3. Quality of MeMO

We use Mutation Score as a quantitative estimation of the quality of memory and traditional mutants. Column 5 and Column 9 in Table 4.11 summarise the Mutation Scores of memory mutants and traditional mutants per program. All Mutation Scores are calculated after filtering out all TCE-equivalent mutants and preserving only one mutant from each TCE-duplicate group, to achieve a more precise estimation of Mutation Scores. Overall, the Mutation Score of memory mutants (0.476) is smaller than the Mutation Score of traditional mutants (0.712), indicating memory mutants are much more difficult to kill than traditional mutants. We can see that

for 14 out of 16 programs, the Mutation Score of memory mutants is lower than that of traditional mutants. For programs, *SparseCounter* (11th), *circularBuffer* (13th) and *bitfield* (16th), we observe a considerable difference between Mutation Scores of memory mutants and traditional mutants. It is also noticeable that traditional mutants yield a slightly lower Mutation Scores than memory mutants for 2 programs. In addition, we calculated the Spearman correlation coefficient ρ for Mutation Scores of memory mutants and Mutation Scores of traditional mutants, to see whether there is a correlation between them. Our result indicates that there is little evidence for any correlation between these two types of Mutation Scores ($\rho = 0.273, p = 0.304$).

Furthermore, we investigate the number of memory mutants that are TCE-duplicate of traditional mutants, as another assessment of the quality of Memory Mutation Operators. Memory mutants will be redundant if many of them are duplicates of the mutants generated from traditional operators. As before, we adopt TCE technique to assess whether a mutant is a duplicate of another. If the TCE technique shows the machine code of a memory mutant and a traditional mutant are identical, we say the memory mutant is a TCE-duplicate of the traditional mutant. The result of TCE-duplicates can be found in Table 4.13.

Table 4.13: Number of memory mutants that are TCE-duplicates of traditional mutants.

Category	Mutation Operator	TCE-duplicate Mutants	TCE-duplicate Rate (%)
Uninitialized Memory Access	REC2M	0	0.0
	RMNA	9	23.1
Faulty Memory Allocation	REDAWN	2	3.1
	REDAWZ	3	4.8
	RESOTPE	5	9.4
	REMSOTP	1	20.0
Faulty Heap Management	RMFS	0	0.0
	REM2A	1	3.4
	REC2A	0	0.0
All		21	5.7

According to the table, only 5.7% of memory mutants are TCE-duplicates, and for some of the MeMOs none of the mutants is TCE-duplicates. If a memory mutant is TCE-equivalent to the original, then it will be a TCE-duplicate because any traditional mutant that is TCE-equivalent to the original will be TCE-equivalent to

this memory mutant, making it a TCE-duplicate. After we filter out all the TCE-equivalent mutants, there remain only 3 TCE-duplicates, 2 from operator **RMNA**, and 1 from operator **REDAWZ**. In another words, only less than 1% of (over 300) TCE-inequivalent memory mutants can be generated from traditional operators. This suggests MeMOs will inject almost totally different kinds of artificial faults and make the test suite more comprehensive if Mutation-based test case generation techniques are applied. The answer to **RQ3** is memory mutants are much harder to kill than traditional mutants according to their Mutation Scores, demonstrating Memory Mutation Operators generate high quality mutants. Among all memory mutants, less than 1% are TCE-duplicates to traditional mutants, indicating Memory Mutation Operators are good complements to traditional operators, introducing very few duplicated mutants.

4.4.4 RQ4. Effectiveness of Weakly Killing Criteria

Since applying only strongly killing criterion results in low Mutation Score, we introduce Memory Fault Detection and Control Flow Deviation weakly killing criteria to further identify faulty memory mutants. In **RQ4**, we study the effectiveness of MFD and CFD weakly killing criteria by looking at the Mutation Scores of memory mutants, when strongly and weakly killing criteria are applied. More importantly, we want to know how many mutants that survived strongly killing criterion are killed by weakly killing criteria, thereby reducing mutants survived. The results are reported in Table 4.14 and Figure 4.4. In the table, Column 3 to Column 5 report the Mutation Scores for each operator when strongly killing, CFD or MFD criterion alone is applied, and the last column shows the Mutation Scores when all three criteria are applied together. These are the results after filtering out TCE-equivalent mutants and TCE-duplicate mutants. We only filter out TCE-duplicate mutants that are duplicates of the mutants generated from the same operator, because if there are duplicates from different operators, it would be unfair to favour one operator and discard all the other TCE-duplicate mutants from other operators.

From Table 4.14 we can see that, for 7 out of 9 operators the Mutation Scores with respect to strongly killing criterion are below 0.6. The highest Mutation Score

Table 4.14: Mutation Scores by operator while strongly killing, CFD or/and MFD criteria are applied

Category	Mutation Operator	Mutation Score (Strongly killed)	Mutation Score (CFD)	Mutation Score (MFD)	Mutation Score (Weakly killed)
Uninitialized Memory Access	REC2M	0.167	0.233	0.667	0.700
	RMNA	0.563	0.625	0.500	0.656
Faulty Memory Allocation	REDAWN	0.841	0.889	0.841	0.889
	REDAWZ	0.459	0.459	0.820	0.820
	RESOTPE	0.479	0.521	0.896	0.896
	REMSOTP	0.000	0.000	0.200	0.200
Faulty Heap Management	RMFS	0.019	0.151	0.019	0.151
	REM2A	0.464	0.429	0.607	0.643
	REC2A	0.833	0.800	0.900	0.900
All		0.476	0.516	0.653	0.702

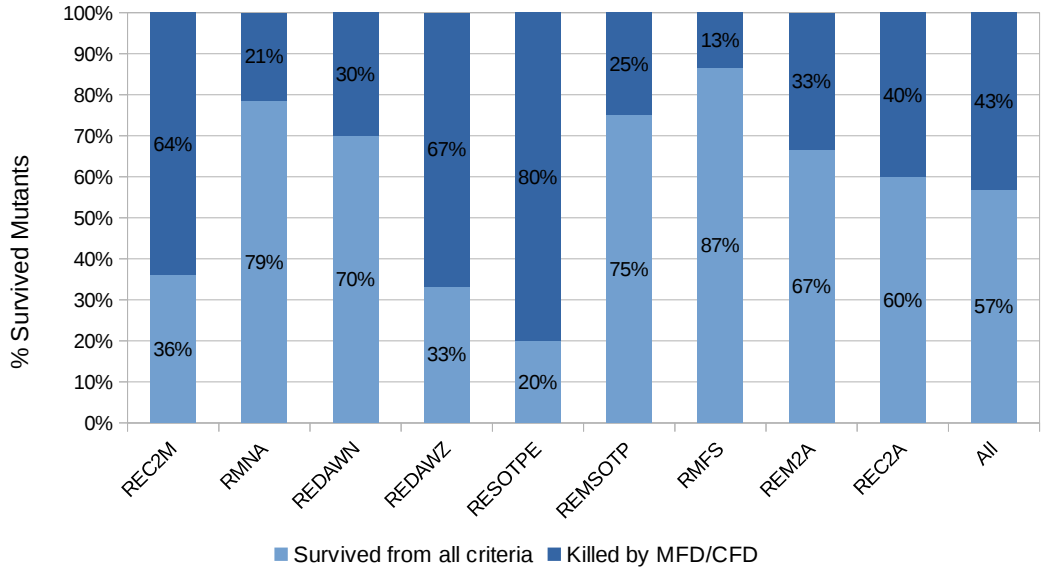


Figure 4.4: Reduction of equivalent mutants after introducing Memory Fault Detection and Control Flow Deviation killing criteria. The percentage of the darker bars is calculated as "number of mutants survived from strongly killing criterion but killed by MFD and/or CFD" divided by "number of mutants survived from strongly killing criterion".

is 0.833, while the lowest is 0. This means that none of the mutants generated by the operator (**REMSOTP**) was killed under strongly killing criterion. When both strongly and weakly killing criteria are applied, the Mutation Score for every mutation operator increases, with the biggest increase from 0.167 to 0.700 (**REC2M**). Overall, the Mutation Score increases from 0.476 to 0.702, indicating that there is a considerable amount of memory mutants can be weakly killed but cannot be strongly killed. Another interesting finding is that for some operators, applying

MFD (CFD) alone is sufficient to reach the same Mutation Scores as that achieved by applying all criteria. In these cases, applying other criteria does not kill any additional mutants, therefore the other criteria are completely subsumed by MFD(CFD).

From another perspective, the weakly killing criteria can help with reducing survived mutants, thus reducing the number of potential equivalent mutants that may require further investigation. We report the reduction rate of survived mutants in Figure 4.4. In the figure, each 100% bar represents all the survived mutants for each Memory Mutation Operator with respect to the strongly killing criterion, or $\overline{T}$ referring to the Venn Diagram in Figure 4.2. The darker part of each bar represents the ratio of these mutants killed by weakly killing criteria, or $R = \overline{T} \cap (M \cup C)$ referring to Figure 4.2, then the percentage number on the dark bars is calculated as $|R|/|\overline{T}| \times 100\%$. With 43% on average, the reduction rate varies from 13% to 80%.

This result shows that about two-fifths of the survived mutants are not equivalent and can be weakly killed. By weakly killing these mutants, we can save test effort when survived mutants need further investigation of their equivalence. The answer to **RQ4** is, Memory Fault Detection and Control Flow Deviation criteria are effective in reducing the number of survived mutants with the highest reduction rate of 80% (**RESOTPE**), while the average reduction rate is 43%.

4.4.5 RQ5. Contribution of Killing Criteria

To further understand the relationship between the strongly and proposed weakly killing criteria, we calculate the contribution, c, and unique contribution, u, metrics as defined in Section 4.2.4 for each criterion. The result, by subject, is given in Table 4.15, where the maximum contribution or unique contribution is highlighted for each subject. According to the table, there appears to be no obvious pattern in the contribution or unique contribution of a criterion across all subjects, meaning each criterion may have different contribution to killing non-equivalent mutants for different subjects. Also, on average, there is no criterion with 0 unique contribution. Therefore, each criterion is required in killing all these non-equivalent mutants.

The answer to **RQ5** is, different criteria perform differently across all subjects, furthermore, each criterion has a non-zero unique contribution, meaning that aban-

Table 4.15: Contribution and Unique Contribution of strongly and weakly killing criteria

PUT	Contribution			Unique Contribution		
No.	*T*	*M*	*C*	*T*	*M*	*C*
1	0.690	0.833	**0.881**	0.000	0.119	**0.167**
2	0.618	**0.971**	0.706	0.029	**0.265**	0.000
3	0.375	**1.000**	0.250	0.000	**0.625**	0.000
4	0.882	**0.941**	**0.941**	0.000	**0.059**	**0.059**
5	0.615	**0.769**	0.615	0.000	**0.385**	0.000
6	0.909	0.909	**1.000**	0.000	0.000	**0.091**
7	0.941	**1.000**	**1.000**	**0.000**	**0.000**	**0.000**
8	0.600	**1.000**	0.600	0.000	**0.400**	0.000
9	0.917	0.917	**1.000**	0.000	0.000	**0.083**
10	0.778	**1.000**	0.667	0.000	**0.222**	0.000
11	0.409	**1.000**	0.409	0.000	**0.591**	0.000
12	**1.000**	**1.000**	**1.000**	**0.000**	**0.000**	**0.000**
13	0.500	**1.000**	0.250	0.000	**0.500**	0.000
14	0.500	**0.833**	0.667	0.000	**0.333**	0.167
15	0.500	**1.000**	0.500	0.000	**0.500**	0.000
16	0.400	**0.800**	0.600	0.000	**0.400**	0.200
All	0.678	**0.931**	0.735	0.004	**0.245**	0.053

doning any criterion may result in inadequate testing.

4.4.6 A Case Study

The conclusions, so far, are drawn from relatively small subjects (87 – 1547 LoC). To investigate whether the conclusions can be generalised to larger subjects, we conducted the same experiments on two large subjects, libdssl (6600 LoC) and libksi (19445 LoC), respectively.

The number of generated mutants, TCE-equivalent mutants and TCE-duplicate mutants for both subjects are summarised in Table 4.16. Being consistent with the previous result, the number of memory mutants remains a small portion (1% – 3%) of the number of traditional mutants. Moreover, the proportion of TCE-equivalent or TCE-duplicate memory mutants remains at low (5.6%), while the number is 10.3% for traditional mutants. These numbers are 5.2% and 17.4% respectively for the first 16 smaller PUTs.

For understanding the prevalence of memory mutants by Memory Mutation Operators, we analyse the number of generated mutants and TCE-equivalent mu-

Table 4.16: Number of mutants and Mutation Scores of memory and traditional mutants for large subjects. The numbers of mutants that are TCE-equivalent of the originals are given in "TCE-equivalent". The column "TCE-duplicate" shows the number of mutants that can be discarded due to duplication of other mutants. (The mutant that is preserved from each duplicate-group is not included in the numbers)

PUT No.	Memory Mutants				Traditional Mutants			
	Generated	TCE-equivalent	TCE-duplicate	Mutation Score	Generated	TCE-equivalent	TCE-duplicate	Mutation Score
L1	288	69	0	**0.183**	20743	2573	821	0.709
L2	970	2	0	**0.475**	47375	1712	1910	0.817
1–16 Total	368	18	1	**0.476**	13333	1918	399	0.712
Total	1626	89	1	**0.434**	81451	6203	3130	0.775

tants for each MeMO, using the memory mutants from the two large subjects. This result can be found in Table 4.17. It is notable that, **RMNA** alone generated 70% of all the memory mutants. This is because both of the large subjects are security libraries, which usually contain a large number of `NULL`-assignments for security reasons. Despite a large number of **RMNA** mutants, the ratio of TCE-equivalent mutants regarding **RMNA** remains at a relatively high level (8.3%) as it is for smaller subjects, implying that **RMNA** is more likely to generate equivalent mutants than other MeMOs. For the rest of the mutation operators, the numbers of generated mutants are comparable, as they are for smaller subjects. Furthermore, the overall TCE-equivalence ratio (6.0%) is comparable with the previous result (4.9%).

Table 4.17: Number and proportion of TCE-equivalent mutants and mutants that are TCE-duplicate to traditional mutants, case study on two large subjects.

Mutation Operator	Generated Mutants (% of all mutants)	TCE-equivalent Mutants	TCE-equivalence Rate (%)	TCE-duplicate Mutants	TCE-duplicate Rate (%)
REC2M	19 (1.6%)	0	0.0	0	0.0
RMNA	833 (70.2%)	69	8.3	1	0.1
REDAWN	74 (6.2%)	0	0.0	3	4.1
REDAWZ	68 (5.7%)	0	0.0	4	5.9
RESOTPE	76 (6.4%)	2	2.6	0	0.0
REMSOTP	4 (0.3%)	0	0.0	0	0.0
RMFS	122 (10.3%)	0	0.0	0	0.0
REM2A	43 (3.6%)	0	0.0	0	0.0
REC2A	19 (1.6%)	0	0.0	0	0.0
All	1187 (100%)	71	6.0	8	0.7

Table 4.17 also reports the number of TCE-duplicate mutants that are TCE-duplicates of traditional mutants. As an indicator of the quality of the Memory

Mutants, the overall ratio of TCE-duplicate mutants remains at low (0.7%), indicating that most memory mutants cannot be simulated by traditional operators. The same ratio for smaller subjects is 5.7%. The fifth and the last column of Table 4.16 report the strongly killing Mutation Scores for memory mutants and traditional mutants respectively. For these two large subjects (and most of the smaller subjects), the Mutation Scores of memory mutants are consistently lower than the Mutation Scores of traditional mutants, implying that memory mutants are harder to kill using the same test suites.

Table 4.18 reports the Mutation Scores for each MeMO when strongly killing, CFD or MFD is applied alone (Columns 2–4) and when all criteria are applied (Column 5). More importantly, we are interested in how many non-equivalent mutants that survive the strongly killing criterion, but are captured by our weakly killing criteria. These numbers are reported in the last column, represented as the proportion of weakly killed mutants in all survived mutants (from strongly killing criterion). According to the table, by applying weakly killing criteria, we can reduce by up to 71%, the numbers of survived mutants (80% for the 16 smaller subjects), with an average of 25% reduction rate (43% for smaller subjects). These results are comparable with the previous results on smaller subjects.

Table 4.18: Mutation Scores by operator while strongly killing, CFD or/and MFD criteria are applied, and the reduction rate of survived mutants, case study on two large subjects.

Mutation Operator	Mutation Score (Strongly killed)	Mutation Score (CFD)	Mutation Score (MFD)	Mutation Score (Weakly killed)	Reduction Rate of Survived Mutants (%)
REC2M	0.105	0.105	0.105	0.105	0
RMNA	0.444	0.529	0.476	0.546	18
REDAWN	0.676	0.838	0.176	0.865	58
REDAWZ	0.456	0.662	0.471	0.765	57
RESOTPE	0.703	0.811	0.676	0.811	36
REMSOTP	0.250	0.750	0.250	0.750	67
RMFS	0.025	0.213	0.025	0.213	19
REM2A	0.279	0.512	0.651	0.791	71
REC2A	0.526	0.526	0.842	0.842	67
All	0.421	0.534	0.429	0.568	25

Lastly, we report the contribution and unique contribution of all three criteria for the large subjects in Table 4.19. Similarly, all criteria have non-zero unique contribution, indicating that abandoning any criterion will mistakenly label some

non-equivalent mutants as equivalent and, therefore, lead to less accurate Mutation Scores.

Table 4.19: Contribution and Unique Contribution of strongly and weakly killing criteria, case study on two large subjects.

PUT	Contribution			Unique Contribution		
No.	*T*	*M*	*C*	*T*	*M*	*C*
L1	0.245	0.209	**0.945**	0.006	0.049	**0.650**
L2	0.900	0.930	**0.940**	0.002	**0.057**	0.014
All	0.742	0.755	**0.941**	0.003	0.055	**0.168**

4.4.7 Overall Findings

Overall, memory mutants injected by Memory Mutation Operators are more difficult to kill than the traditional mutants, as suggested by their Mutation Scores. Moreover, MeMOs inject faults that are almost completely different from the kind of faults injected by traditional operators; only less than 1% of memory mutants are found to be TCE-duplicate of traditional mutants. Our results also show that the number of memory mutants accounts for only 2.0% of traditional mutants. Introducing MeMOs will therefore cost only a small portion of the overall testing effort.

A low Mutation Score of memory mutants also suggests that the traditional strongly killing criterion is inadequate for distinguishing memory faults. After the MFD and the CFD weakly killing criteria are applied, the Mutation Scores for memory mutants increase drastically, indicating that these newly introduced criteria effectively distinguish memory faults, giving more accurate Mutation Scores. Furthermore, we found up to 80% of the mutants survived from strongly killing criteria were determined to be non-equivalent by these weakly killing criteria, reducing the number of survived mutants that may require further investigation for their equivalence. By taking a closer look at which mutants are killed by each criterion, we found both strongly killing criterion and weakly killing criteria are required for killing non-equivalent mutants, while each criterion performs differently across subjects.

We used TCE technique to identify TCE-equivalent and TCE-duplicate mutants and filtered them out accordingly in our experiments. This is the first work

that we are aware of to handle duplicate mutants. The use of the TCE technique resulted in different values compared to a previous study that included them [2]. This confirms that the lack of handling equivalent and duplicate mutants may lead to inaccurate results in other studies. In extreme cases, it might even affect their conclusions, and so our findings also suggest that TCE should be considered in other Mutation Testing studies.

4.5 Threats to Validity

In this section, we summarise some potential threats to validity.

4.5.1 Internal Validity

We cannot inspect how many memory mutants are guaranteed to be equivalent to the original since the equivalence of mutants is undecidable [72, 73]. This could introduce bias to the Mutation Scores calculated. We sought to minimise the influence of equivalent mutants by adopting the TCE technique and estimating the lower bound of the number of equivalent mutants.

We use Mutation Score to indicate the quality of the mutants. However, low Mutation Scores do not necessarily indicate high-quality of the mutants, as it also may be caused by a low-quality test suite. To minimise this threat, we report the number of tests and their statement coverage for each subject. In fact, half of the subjects have a coverage level above 90%, indicating the quality of the test suites is reasonably adequate. In addition, we focus on the relative differences between Mutation Scores of traditional mutants and memory mutants, instead of the absolute value of Mutation Scores. Therefore, the quality of the tests has little impact on the overall conclusion.

In this work, we use *Valgrind* and *Gcov* to instrument the PUTs in order to gather memory fault and control flow coverage information during the execution. Whether these two instrumentations affect each other remains untested. This could be an internal threat to the validity of the reduction rate of survived mutants. We mitigate this threat by using branch coverage instead of statement coverage, so that *Valgrind*'s instrumentation would not likely to affect the coverage outcome. Also,

we only focus on the number of memory faults in *Valgrind*'s report after executing a program. Since it is very unlikely that *Gcov* would introduce additional and different memory faults in the original and a mutant, so we have confidence that *Gcov* does not affect the output of *Valgrind*.

Another threat to validity comes from the execution environment. The programs and the mutants are run in *Valgrind*'s virtual environment, which may differ from a real environment. This could bias the number of survived mutants with respect to each killing criterion. Since *Valgrind* uses a simulated processor, which well mimics the environment of a real processor, to execute the programs, it can hardly affect the execution path or the output of a program. Even though sometimes *Valgrind* prevents a program from crashing due to memory faults but continue executing it, it will record the memory faults so that we can still kill this mutant with Memory Fault Detection criterion, so that this threat is alleviated.

4.5.2 External Validity

All of the conclusions are drawn from the PUTs ranging from small sizes to fairly large sizes. However, they are not guaranteed to hold for very large programs. To mitigate the threat to validity, we conducted our research on 16 subject programs with a variety of sizes and different kinds of functionalities, then verified the results on 2 large real-world programs. The results show that the conclusions are consistent when the sizes of the programs scale. So we expect to see similar results from even larger subjects in the future.

Recall that most of the MeMOs apply only on `malloc`/`free` routines, the number of mutants generated must relate to how frequently a program uses the routines. This could lead to a threat to the validity of the conclusion that MeMOs effectively insert memory faults to the subject programs in terms of the number of generated mutants. From another perspective, if a program uses dynamic memory allocation less frequently, the program is less likely to have memory faults and needs memory related Mutation Testing less, thus the severity of this threat is reduced.

Memory Fault Detection and Control Flow Deviation killing criteria are originally designed for killing memory mutants, because they are hard to be killed by

comparing the outputs. So we only reported the contribution and unique contribution of criteria on memory mutants, but there is no preventing these weakly killing criteria from being applied on traditional mutants. However, their contribution on traditional mutants is likely to be different since there are much fewer memory faults in traditional mutants.

4.6 Conclusion

Mutation Testing has been proved to be an effective way to test programs. However, traditional Mutation Operators only mimic general faults while memory faults may be missed by these operators. In this chapter, we have proposed 9 Memory Mutation Operators (MeMOs) simulating general memory faults. By applying MeMOs to 18 real world programs, we found that these operators are not only effective in inserting memory faults but also produce memory mutants that are harder to kill than traditional mutants. We also found that memory mutants are very different from the traditional mutants, and only less than 1.9% of them can be generated using traditional operators directly.

In order to effectively detect these artificial memory faults and identify more non-equivalent mutants, we proposed Memory Fault Detection and Control Flow Deviation weakly killing criteria to aid traditional Mutation Testing. We collected the memory faults and the control flow graph coverage of mutants and of the original program and used this information to reduce the number of survived mutants. The experimental results showed that introducing these two killing criteria could further reduce the survived mutants by up to 80%. Finally, we investigated the contribution and unique contribution of strongly and weakly killing criteria. The results suggest that all criteria have a non-zero unique contribution, while there is no guarantee of which criterion alone performs the best in killing non-equivalent memory mutants.

Chapter 5

Higher Order Mutation for Software Improvement

In this chapter, we introduce the HOMI approach to improve non-functional properties of software while preserving the functionality, using both Selective Mutation Operators and Memory Mutation Operators introduced in the previous chapter. HOMI utilises search-based higher order mutation testing [139] to effectively explore the search space of varying versions of a program. Like other previous GI work [14, 15, 21], HOMI relies on high-quality regression tests to check the functionality of the program. Given a program p and its regression tests T. HOMI generates two types of mutants that can be used for performance improvement. A **GI-FOM** is constructed by making a single syntactic change to p, which improves some non-functional properties of p while passing all the regression tests T. Having the same characteristics as GI-FOMs, a **GI-HOM** is constructed from the combination of GI-FOMs.

By combining with Mutation Testing techniques, we specifically utilise equivalent mutants which are expressly avoided by mutation testers where possible [74]. We implemented a prototype tool to realise the HOMI approach. The tool is designed to focus on two aspects of software runtime performance: execution time and memory consumption. Time and space are important qualities for most software, especially on portable devices or embedded systems where the runtime resources are limited. Moreover, these two qualities are usually competing with each other,

yielding an interesting multi-objective solution space. Our tool produces a set of non-dominated GI-HOMs (thus forming a Pareto front). We evaluate our tool using four open source benchmarks. Since the tool requires no prior knowledge about the subjects, it can be easily applied to other programs.

The chapter presents evidence that using Higher Order Mutation is an effective, easy to adopt way to improve existing programs. The experimental results suggest that equivalent First Order Mutants (FOMs) can improve the subject programs by 14.7% on execution time or 19.7% on memory consumption. Further results show that by searching for GI-HOMs, we can achieve up to 18.2% time reduction on extreme cases. Our static analysis suggests that 88% of the changes in GI-HOMs cannot be achieved by 'plastic surgery' based approaches. The contributions of this work are as follows:

1. We introduce an automatic approach to improve programs via Higher Order Mutation, which explores program search space at a fine granularity while maintaining good scalability.

2. We evaluate our approach on four open source programs with different sizes. We report the results and demonstrate that our approach is able to reduce the execution time by up to 18.2% or to save the memory consumption by up to 19.7%.

3. The results of a manual analysis are reported to show that our approach works on a smaller granularity, such that 88% of the changes found by our approach cannot be achieved by line based 'plastic surgery' approaches.

4. We also show evidence that it is possible to combine the HOMI approach with Deep-Parameter-optimisation approach to further improve the performance.

5.1 The HOMI approach

We propose the HOMI approach, a higher order mutation based solution to GI. Figure 5.1 shows the overall architecture of the HOMI approach. Given a subject program with a set of regression tests, and some optimisation goals, HOMI applies

SBSE to evolve a set of GI-HOMs that improve the properties of interest while passing all the regression tests. To explore the search space efficiently, we follow the current practice of GI in separating our approach into two stages [63]. In the first stage, we apply first order mutation to find locations in the program at which making changes will lead to significant impact on the optimisation goals. In the second stage, we apply a multi-objective search algorithm at these program locations to construct a Pareto front of GI-HOMs.

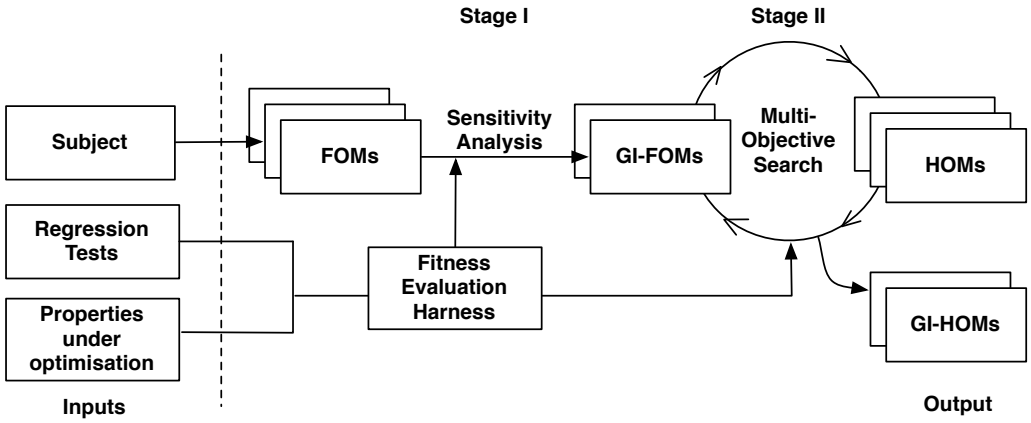


Figure 5.1: The overall architecture of the HOMI approach.

5.1.1 Stage I: Sensitivity Analysis

Sensitivity analysis has been shown to be an effective way to reduce the search space in previous GI work [15,17,63]. Given a subject program under optimisation, some code pieces may have a greater impact on the properties of interest than others. Sensitivity analysis seeks to find small portions of code that have the greatest impact on the properties of interest. Thus, the subsequent optimisation can focus on a manageable amount of code, effectively reducing the search space. We use the same first order mutation based sensitivity analysis approach as introduced in Chapter 3, to gather sensitivity information. We use this form of sensitivity analysis because it provides finer granularity than traditional statement or line-based sensitivity analysis.

As shown in Figure 5.1, HOMI first generates a set of FOMs of the subject program and then evaluates them using a fitness evaluation harness. The evaluation

harness is composed of regression tests and the measurement components for optimisation goals. It runs each FOM on all the tests and outputs the measurements of the optimisation goals as fitness values. After the fitness evaluation, HOMI removes FOMs that fail any regression tests and keeps only the survived ones. We do this because any mutants that pass all the regression tests are more likely to preserve the correctness of the subject. Finally, HOMI applies a non-dominated sorting [115] to rank all the survived FOMs by their fitness values.

The sensitivity analysis stage outputs a set of GI-FOMs. These FOMs are "potentially equivalent mutants" with respect to the regression test suites and have a positive impact on the properties of interest. We measure the sensitivity of code based on the FOMs' fitness values. A piece of code A is said to be more sensitive than another piece B, if a FOM generated from A dominates the FOMs generated from B on the Pareto front. The range of a code piece can be measured at different granularity levels by aggregating the results of FOMs, such as at the syntactic symbol, the statement level, or the nesting code block level. The GI-FOMs generated and their sensitive information are passed to the next search stage as inputs.

5.1.2 Stage II: Searching for GI-HOMs

In the second stage, HOMI applies a multi-objective algorithm to search for a set of improved versions of the original program in the form of HOMs. We use an integer vector to represent HOMs, which is a commonly used data representation in search-based Higher Order Mutation Testing [140]. Each integer value in the vector encodes whether a mutable symbol is mutated and how it is mutated. For example, given a mutant generated from the arithmetic operator '+', a negative integer value means it is not mutated while the integer 0, 1, 2, 3 indicate that the code is mutated to '$-$', '$*$', '/', '%' respectively. In this way, each FOM is represented as a vector with only one non-negative number and HOMs can be easily constructed by the standard crossover and mutation search operators.

The algorithm takes the GI-FOMs as input and repeatedly evolves HOMs that inherit the strengths of the GI-FOMs from which they are constructed and yield better performance than any GI-FOMs alone. The fitness function that guides the

search is defined as the sum of the measurement of each optimisation property over a given test suite. Given a set of N optimisation goals, for each mutant M, the fitness function $f_n(M)$ for the nth optimisation goal is formulated as follow:

$$\textit{Minimisation} \quad f_n(M) = \begin{cases} \sum C_i(M) & \text{if } M \text{ passes all test cases} \\ C_{\text{MAX}} & \text{if } M \text{ fails any test case} \end{cases}$$

The fitness function is a minimisation function where $C_i(M)$ is the measurement of the optimisation goal n when executing the test i. If the mutant M fails any regression tests, we consider it as a bad candidate and assign it with the worst fitness values C_{MAX}. The algorithm produces a Pareto front of GI-HOMs. Each HOM on the front represents a modified version of the original program that passes all the regression tests while no property of interest can be further improved without compromising at least one other optimising goal.

5.1.3 Implementation

We implemented a prototype tool to realise the HOMI approach. The HOMI tool is designed to optimise two non-functional properties (running time and memory consumption) for C programs. In the fitness evaluation harness, we use *Glibc*'s *wait* system calls to gather the CPU time, and we instrument the memory management library to measure the 'high-water' mark of the memory consumption. We choose to measure virtual instead of physical memory consumption because the physical memory consumption is non-deterministic. This means it depends on the workload of the machine. By contrast, the virtual memory used is always an upper bound of the physical memory actually used.

HOMI uses the open source C mutation testing tool, Milu [119] to generate mutants. Though we implemented our own Mutation Testing engine for the experiments in Chapter 4, it only supports First Order Mutation. On the other hand, Milu supports both First Order and Higher Order Mutation, and it provides an easy-to-use interface to customise mutant generation and restoration. Therefore, we choose

to use Milu as the Mutation engine in this chapter. By default, Milu supports only the traditional C mutation operators [141]. As memory consumption is one of the optimisation goals, we extended the original version of Milu to support Memory Mutation Operators proposed in Chapter 4. Table 5.1 lists the Mutation Operators used in HOMI and their brief descriptions. During the search stage, HOMI transforms the internal integer vector representation of the candidate HOM to the data format recognisable by Milu, then invokes Milu to generate the HOM.

Table 5.1: Mutation Operators used by HOMI

Category	Name	Description
Selective Mutation Operators	ABS	Change an expression `expr` to `ABS(expr)` or `-ABS(expr)`
	OAAN	Change between +, −, ⋆, /, %
	OLLN	Change between `&&`, `||`
	ORRN	Change between >, >=, <, <=, ==
	OIDO	Change between ++x, −−x, x++, x−−
	CRCR	Change a constant `c` to `0`, `1`, `-1`, `c+1`, `c-1`, `c*2`, `c/2`
Memory Mutation Operators	REC2M	Replace `malloc()` with `calloc()`
	RMNA	Remove `NULL` assignment
	REDAWN	Replace memory allocation calls to `NULL`
	REDAWZ	Replace allocation size with `0`
	RESOTPE	Replace `sizeof(T)` with `sizeof(*T)`
	REMSOTP	Replace `sizeof(*T)` with `sizeof(T)`
	REM2A	Replace `malloc()` with `alloca()`
	REC2A	Replace `calloc()` with `alloca()`
	RMFS	Remove `free()` statement

The HOMI tool employs a customised NSGA-II [115] to evolve GI-HOMs. During the search process, HOMI maintains a population of candidate HOMs. For each generation, the uniform crossover and mutation are performed to parent HOMs, generating offspring HOMs that are later evaluated using the fitness functions mentioned. A tournament selection is then performed to form the next generation. This process is repeated until a given budget of evaluation times is reached. Finally, HOMI will generate a set of non-dominating GI-HOMs that perform better than the original program on time and/or memory consumption.

5.2 Empirical Study

This section first discusses the research questions we address in our empirical evaluation of the HOMI tool, followed by an explanation of the chosen subjects, tests

and experiment settings.

5.2.1 Research Questions

Since the HOMI approach generates GI-HOMs from the combination of FOMs, a natural first question to ask is 'whether existing FOMs can be used to improve software'. This motivates our first research question.

RQ1: Can GI-FOMs improve program performance while passing all of its regression tests?

To answer this question, we run HOMI for sensitivity analysis only and report how much running time and memory can be saved by GI-FOMs. Of course, the answer also depends on the quality of the regression tests. All the tests used in our evaluation are regression tests generated by developers for real world systems. However, they may still not be sufficient to reveal the faults introduced by mutation. To make our experiment more rigorous, we carried out a pre-analysis in our evaluation. We analyse the function coverage of each subject using the GNU application *Gcov* and HOMI is set only to mutate the functions that are covered by regression tests.

RQ2: How much improvement can be achieved by GI-HOM in comparison with GI-FOMs?

If GI-FOMs alone can improve performance, we expect that GI-HOMs will inherit some strengths from the GI-FOMs and improve the performance further. To answer this question, we use HOMI to generate a GI-HOM Pareto front and investigate whether the GI-FOM solutions generated are on the Pareto front. Furthermore, it is interesting to see whether the new memory mutation operators help to improve the performance. This motivates our sub-question which studies the effect on mutation operators used.

RQ 2.1 How does the improvement achieved by applying the traditional mutation operators only compare to applying both of the traditional and memory mutation operators?

We answer this question by comparing the HyperVolume quality indicator of the Pareto fronts generated from HOMI using both sets of mutation operators. Given a Pareto front A and a reference Pareto front R, HyperVolume is the volume of objective space dominated by solutions in A. To take into account the stochastic nature of the search algorithms, we repeat both experiments 30 times. We use the non-parametric Mann-Whitney-Wilcoxon-signed rank tests to assess the statistical significance of the HyperVolume and untransformed [142] Vargha-Delaney effect size to further assess the magnitude of the differences [143].

RQ3: Can 'plastic surgery' GP based GI approach find edit sequences to construct the GI-HOMs found by HOMI?

We ask this question because we want to understand whether the granularity of mutation changes can be produced by the 'plastic surgery' GP approach. The 'plastic surgery' GP approach is a popular GI approach which searches for a list of edits from the existing source code. Typical changes generated by the GP approach are movements or replacements of different lines of code [63, 111]. To answer this question, we carried out a sanity-check experiment manually using all the GI-HOMs found. For each GI-HOM, we search the entire program to see if the mutated statement exists in the program. If it does, the GI-HOM can be constructed by the patches generated from the GP approach easily. Otherwise, we consider the line/statement based 'plastic surgery' GP might not able to generate the GI-HOM directly.

RQ4 Can HOMI be combined with Deep Parameter Optimisation to achieve further improvement?

Finally, we want to investigate whether the HOMI approach can be combined with other types of GI techniques. Deep Parameter Optimisation approach proposed in Chapter 3 is one of the parameter-tuning-based GI techniques. It seeks to optimise library code used instead of the source code of the subjects. We have obtained a set of memory allocation libraries that were optimised for the time and memory performance in Chapter 3. We answer this research question by evaluating the GI-

HOMs after linking them to Deep-Parameter-optimised libraries, then comparing them with their performance before the linking, and with the performance of the original program after linking to Deep-Parameter-optimised libraries.

5.2.2 Subject programs and tests

We optimise four subjects in our evaluation. Table 5.2 lists the subjects and their brief description. All tests used are regression tests, deemed to be useful and practical by their developers. *Espresso* is a fast application for simplifying complex digital electronic gate circuits. *Gawk* is the GNU *awk* implementation for string processing. *Flex* is a tool for generating scanners, programs which recognise lexical patterns in text, and *sed* is an editor that automatically modifies files given a set of rules. We use the *espresso* version as well as its test cases from *DieHard* project [117]. Version 4.1.0 of *gawk* is used in this work. The source code and the test cases can be found in the GNU archives. We obtain the last two programs and corresponding test suites from the SIR repository [118].

Table 5.2: Subject programs

Name	LoC	# of Tests	Description
espresso	13,256	19	Digital circuit simplification
gawk	45,241	334	String processing
flex	9,597	62	Fast lexical analyzer generator
sed	5,720	362	Special file editor

5.2.3 Search Settings

In the sensitivity analysis stage, we pick the top 10% most sensitive locations of the GI-FOMs and only search for GI-HOMs from these locations. We use a relative ratio instead of an absolute number because the search space can be adapted to the size of the subject. The choice of 10% is based on our observation that the locations in the first 10% are usually much more sensitive than the remaining locations since sensitivity seems to follow a power law, according to our informal observation. However, the ratio can be easily adapted as a parameter to our approach accordingly.

We repeat all HOMI experiments 30 times to cope with the non-deterministic nature of NSGAII and to facilitate inferential statistical analysis. The NSGAII per-

forms a tournament selection of size 2 and uniform crossover with a probability of 0.8. There are 50 HOMs in each generation and the algorithm stops at 100th generation. These numbers were chosen after initial calibration experimentation to determine suitable parameters for our search process. All of the experiments are carried out on a desktop machine with a quad-core CPU and 7.7 GB RAM running 64-bit Ubuntu version 14.04.

5.3 Results and Discussion

5.3.1 Improvement by GI-FOMs

We begin by looking at the time and memory performance of the GI-FOMs generated from the sensitivity analysis stage to answer the RQ1. We calculated the improvement of GI-FOMs relative to the original program, and reported them in Columns 2 and 5 in Table 5.3. These values are averaged from 10 repeated evaluations. By applying selective and memory mutation operators to generate FOMs, we found the improved versions of all four subjects, both on time and memory performance. More specifically, the improvement ranges from 0.9% to 14.7% on time and from 0.5% to 19.7% on memory performance. However, there might be a large gap between the memory and time improvement for some subject, for example, the GI-FOM of *sed* can run up to 14.7% faster but can only save 0.5% memory. We conclude that even with the simplest changes introduced by first order mutation, the HOMI approach is able to improve the execution time and memory consumption.

5.3.2 Improvement by GI-HOMs

We now turn to the improvement found by GI-HOMs. Since improvement was found on GI-FOMs, it is interesting to investigate whether we can improve the performance further by combining them to form GI-HOMs. We applied NSGA-II [115] to search for better performance in HOMs using Selective Mutation Operators (GI-HOMs-Sel) and using both Selective and Memory Mutation Operators (GI-HOMs-All) respectively. Each experiment was repeated for 30 times and the best time/memory performance found for each subject is reported in Table 5.3.

The results of GI-HOMs-Sel are reported in Columns 3 and 6, and those of

Table 5.3: Improvement on time and memory by GI-FOMs and GI-HOMs. GI-HOMs-Sel are found using only Selective Mutation Operators while GI-HOMs-All and GI-FOMs are found using both Selective and Memory Mutation Operators

		Time (%)	
Subject	GI-FOMs	GI-HOMs-Sel	GI-HOMs-All
espresso	5.2	6.5	**6.9**
gawk	2.3	6.7	**9.8**
flex	0.9	**2.3**	**2.3**
sed	14.7	**18.2**	**18.2**
		Memory (%)	
Subject	GI-FOMs	GI-HOMs-Sel	GI-HOMs-All
espresso	1.6	**1.7**	**1.7**
gawk	2.5	1.9	**4.3**
flex	**19.7**	**19.7**	**19.7**
sed	**0.5**	**0.5**	**0.5**

GI-HOMs-All are reported in Columns 4 and 7. We immediately observe that GI-HOMs achieve greater improvement than GI-FOMs on execution time for all subjects, also on memory consumption for two out of four subjects. The greatest time improvement found by GI-HOMs can be promoted to 18.2%, while the improvement can be up to four times better (on *gawk*) than the improvement yielded from GI-FOMs. We also observe one case *gawk*), on which the GI-HOMs-Sel achieve less memory improvement compared with GI-FOMs, because they are lack of some memory-related changes that can only be achieved by Memory Mutation Operators.

We combine the results of 30 runs for each experiment and plot the Pareto fronts of GI-HOMs using all Mutation Operators, GI-HOMs using Selective Mutation Operators and GI-FOMs in Figure 5.2-5.5. In the figure, time (x-axis) and memory (y-axis) are both normalised to the original performance. On all four subjects, we can see there is always an improvement from GI-FOMs to GI-HOMs, while the differences between GI-HOMs-Sel and GI-HOMs-All are less clear. To statistically demonstrate the difference, we calculated the HyperVolume [121] of the Pareto fronts of GI-HOMs-All and GI-HOMs-Sel over 30 runs, and applied Mann-Whitney-Wilcoxon U-test on the HyperVolume metric for these Pareto fronts. For subject *espresso* and *gawk*, the difference between HOMs (all) and HOMs (Selective) are significant ($p < 0.01$) with a large effect size ($A_{12} > 0.9$), while for the

other two subjects, the difference is not significant.

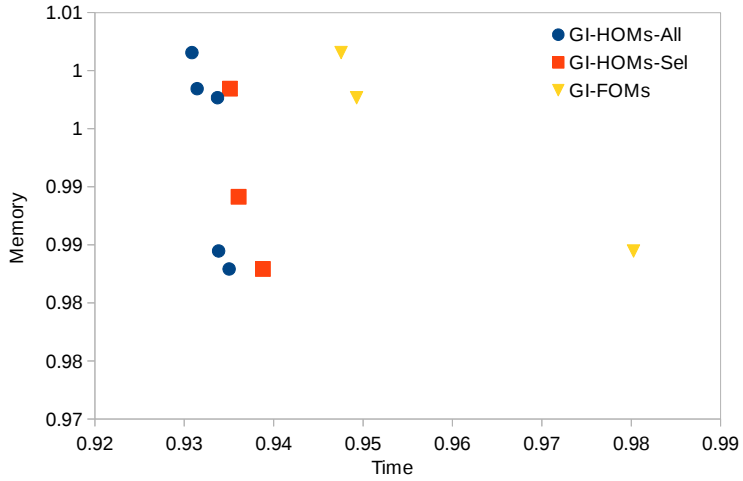


Figure 5.2: Pareto fronts of GI-HOMs and GI-FOMs for *espresso*. Lower and lefthand solutions dominate high and righthand solutions.

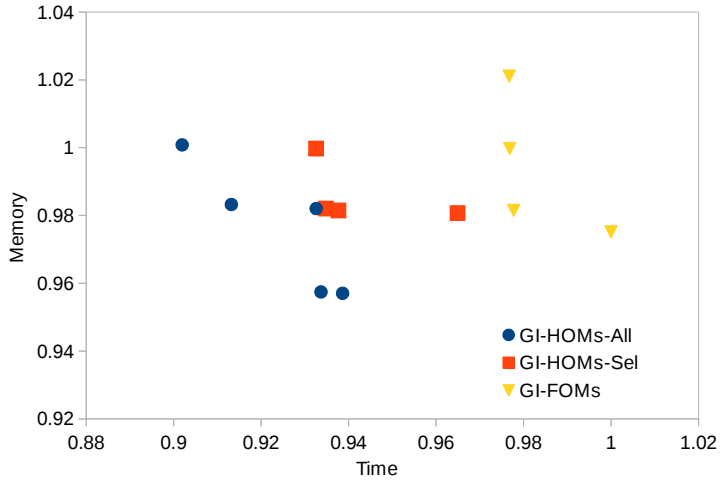


Figure 5.3: Pareto fronts of GI-HOMs and GI-FOMs for *gawk*. Lower and lefthand solutions dominate high and righthand solutions.

In summary, the answer to RQ2 is that GI-HOMs can improve the time performance by up to 18.2% or the memory performance by up to 19.7%. For the GI-HOMs using Selective Mutation Operators only, we found the same upper bound of the improvement, but only achieved sub-optimal solutions on two subjects. By including Memory Mutation Operators, further improvement on these subjects was found. Therefore, we can conclude that Memory Mutation Operators provide further improvement potentials for both time and memory optimisation.

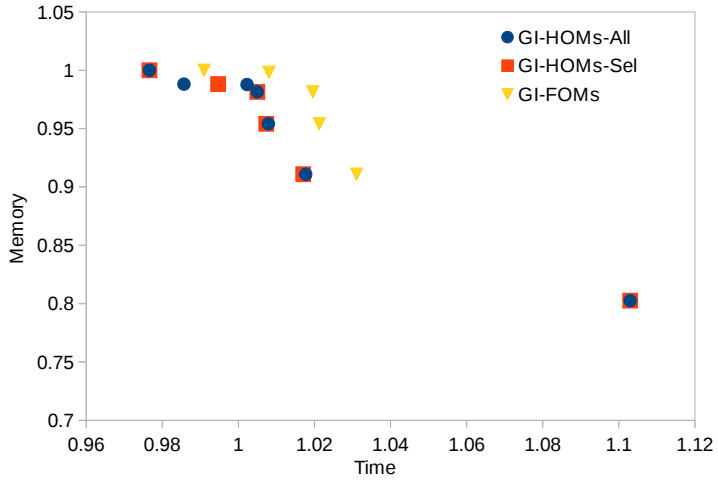


Figure 5.4: Pareto fronts of GI-HOMs and GI-FOMs for *flex*. Lower and lefthand solutions dominate high and righthand solutions.

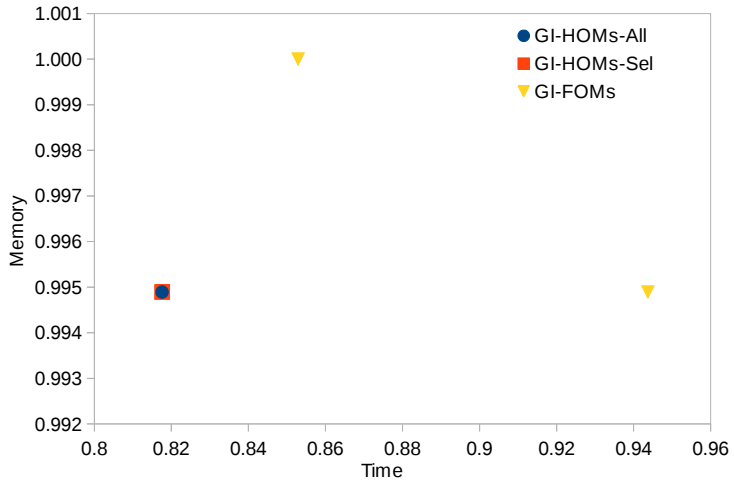


Figure 5.5: Pareto fronts of GI-HOMs and GI-FOMs for *sed*. Lower and lefthand solutions dominate high and righthand solutions.

5.3.3 HOMI vs 'plastic surgery' GP based GI

This RQ investigates whether the granularity of mutation changes can also be generated by the 'plastic surgery' GP approach. To answer this question, we investigated the GI-HOMs found in all experiments and manually reproduce them following the evolution rules used in the line/statement based 'plastic surgery' GP approach [63, 111]. All together HOMI found 273 mutations in the improved GI-HOMs across all subjects. We first applied a simple hill-climbing algorithm to clean up the mutations that do not contribute to the improvement. This step narrowed the

number of mutations down to 141. In total, there are 108 unique mutations identified (the same mutations may be found in several GI-HOMs).

For each of the unique mutation changes, we search the entire program to see if the mutated line/statement exists in the program. Because the typical changes generated by the 'plastic surgery' GP approach are movements or replacements of different lines of code [63, 111], if a mutation does not appear somewhere else in the original source code, it cannot be generated directly from this form of GP approaches. The result shows that 95 (88%) out of 108 mutations cannot be found in the original source code. Therefore, the answer to RQ3 is, there are 108 unique mutational changes found in the GI-HOMs, 88% of which cannot be generated from the line-based 'plastic surgery' GP approach directly.

This result can only demonstrate how differently these two approaches modifies the subject programs, but does not represent the semantic differences between the modifications, neither does it provide evidences of which approach is better. For instance, if the original code writes '`v = v + 2`', HOMI approach may mutate it to '`v = v + 1`', while Plastic Surgery approaches may replace the whole statement with '`v++`', which is grafted from the codebase. Though these two approaches yield syntactically different modifications, the semantics of the outcome is the same, and this kind of semantic equivalence is not accounted for in this study. However, with 88% of the changes being syntactically different, it motivates us to conduct a full comparison of these two approaches in the future.

5.3.4 HOMI combines with Deep Parameter Optimisation

In the last Research Question, we want to understand whether the improvement can be preserved or even promoted if we combine GI-HOMs with Deep-Parameter-optimised memory management library. To answer this question, we use the memory allocation libraries that were optimised for the time and memory performance for each subject in Chapter 3. We created four new optimised version of each subject by linking the most time/memory-saving GI-HOMs and libraries in pairwise.

The results are reported in Table 5.4. In the table, rows represent HOMI-improved programs and columns represent Deep-Parameter-optimised libraries,

where 'Original' indicates the original program or library, 'T' indicates it is the most time-saving ones and 'M' indicates the most memory-saving ones. All of the numbers are the improvement in percentage compared with the original version. If in a combination (that does not involve the Original program/library), the time/memory performance is not worse than that of any of the 'ingredient' program/library, it is highlighted in bold font. On the other hand, all the underlined performances are the ones that are worse than both of the 'ingredient' program and library. For subject *sed*, there is only one GI-HOM on the Pareto front, thus it is both the most time and memory-saving program.

Table 5.4: HOMI combines with Deep Parameter Optimisation. Each cell reports the time improvement followed by memory improvement in percentage. 'T' or 'M' indicates it is most time-saving or memory-saving GI-HOM/optimised library.

		Memory Management Library		
		Original	Deep(T)	Deep(M)
espresso	Original	0/0	0.8/0.1	0.7/0.2
	GI-HOM(T)	6.9/-0.2	4.8/**0.1**	4.7/**0.2**
	GI-HOM(M)	6.5/1.7	4.7/**1.8**	**6.7**/**1.7**
flex	Original	0/0	15.7/-2.6	-1.1/0.6
	GI-HOM(T)	2.3/0	14.4/-2.6	-Inf/-Inf
	GI-HOM(M)	-10.3/19.7	-3.5/**19.7**	-Inf/-Inf
gawk	Original	0/0	5.4/1.6	-0.2/2.3
	GI-HOM(T)	9.8/-0.1	5.6/**1.6**	5.4/**2.3**
	GI-HOM(M)	6.1/4.3	4.1/**5.8**	4.8/**5.5**
sed	Original	0/0	7.9/-1208	5.6/2.0
	GI-HOM(TM)	18.2/0.5	5.8/-1208	4.1/0.9

We observe that there are 10 out of 28 cases (bold numbers) when combining GI-HOMs with the Deep-Parameter-optimised library, the performance is at least the same as the best performance of the GI-HOM or library it is combined from, and is strictly better in four cases. However, there are three cases (underlined) that the combination makes their performance worse. In most of the cases, the performance lies between the performance of the GI-HOM and the library that it is combined from. In one extreme case (*flex*), we found that the most memory-saving library breaks the functionality of HOMs (indicated by '-Inf' in the table). Therefore, the answer to RQ4 is, when combining the HOMI approach with Deep Parameter opti-

misation, the GI-HOM programs can be either improved or jeopardised. This result motivates a future study that searches and optimises HOMI and Deep Parameters altogether.

5.4 Threats to Validity

We discuss the threats to validity in this sections, where the threats to internal validity are discussed in Section 5.4.1 and those to external validity are discussed in Section 5.4.2.

5.4.1 Internal Validity

We used the regression tests that come with the subjects to evaluate the correctness of mutants. All the subject programs used in this work were well tested in established works, and their tests used are regression tests, deemed to be useful and practical by their developers. However, passing the regression tests does not necessarily mean the semantics of the mutant is the same as the original program. This may pose a threat to the correctness of the GI-HOMs. To mitigated this threat, we set HOMI to apply mutation changes at the code that is covered by the regression tests.

After the sensitivity information is collected, we focus on 10% most sensitive locations only. This is based on an assumption that less sensitive code is less likely to affect the performance of the program. However, there are still chances that the interactions between multiple less sensitive code may lead to some significant improvement. This possible synergy, if there is any, requires a much larger search space, thus, will make the approach much less scalable. To make the HOMI approach scalable, we confine the search on the most sensitive locations, making the search more effective. Furthermore, we make the ratio of sensitive locations a parameter of our approach, such that it can be adapted to trade between exploration and exploitation.

Another threat to validity comes from the measurement of time and memory performance. We applied the same measurement approach as the one used in Chapter 3. To make the measurement accurate, we use CPU time and use the mean of 10

measurements to minimise the noise. For memory consumption, we instrument the memory management library to calculate the exact use of virtual memory. Therefore, the measurement noise is minimised.

5.4.2 External Validity

The approach can be easily applied to other subjects, but the conclusion may not generalise to larger scale systems. We use four subjects with varying sizes from 5,000 to 45,000 lines of code, and the results are consistent across all subjects. Therefore, we have confidence that the results may likely be generalised to larger scale systems, and the threat is thereby ameliorated.

We adopt Memory Mutation Operators in our approach because we are interested in time and memory performance. However, the same set of Mutation Operators does not necessarily lead to similar results when other software qualities are concerned. Since the selection of Mutation Operators is independent of the other parts of the approach, the choices of Mutation Operators can be easily adapted accordingly, thereby minimising this threat.

5.5 Conclusion

In this chapter, we have introduced, HOMI, a search-based higher order mutation approach to GI. HOMI uses mutation operators to automatically modify subject programs at a finer granularity. Using a multi-objective search algorithm, HOMI found GI-HOMs that improve subject programs by 18.2% on time performance or 19.7% on memory consumption without breaking any regression tests. In our empirical study, we also find that by including Memory Mutation Operators, HOMI can find GI-HOMs that achieve better performance than using just traditional Selective Mutation Operators on two subjects. Furthermore, we find that 88% of the mutational changes in our GI-HOMs cannot be generated from the currently widely-used line based 'plastic surgery' GP approach. Finally, by combining GI-HOMs with Deep-Parameter-optimised memory management libraries, we found further improvement than GI-HOMs or optimised libraries alone could achieve, which motivates a future research direction that searches and optimises GI-HOMs and Deep Parameters altogether.

Chapter 6

General Conclusion and Discussion

This chapter summarises the achievements of this thesis, draws general conclusions from the findings in this thesis, and lists several potential future works.

6.1 Summary of Achievements

Manually optimising non-functional properties while maintaining functionalities of a program is not practical for human programmers. Therefore automatically optimising the non-functional properties or even automatically improving the functionalities are demanding. Most recent works used Genetic Programming or patch-based Genetic Improvement to automatically modify the source code to achieve better performance. However, Genetic Programming has only previously been shown to work well on small programs, so the approach may become impractically expensive when the program size increases. On the other hand, patch-based Genetic Improvement approaches usually work on a coarse level of code granularity such as line-based or block-based, therefore some optimal solutions that need a finer level of modification, such as an operator or a variable, may be missed.

The goal of this thesis was to propose new mutation-based Genetic Improvement approaches that operate on a finer code granularity and are scalable to large programs. By focusing on two of the most important non-functional properties for all programs: execution time and memory consumption, we wanted to evaluate the newly proposed approaches when these two competing non-functional properties are to be improved together. To achieve this goal, we proposed two novel Genetic

Improvement approaches that use Mutation Operators to precisely operate on a finer level of code granularity, whilst keeping the search space size manageable by sensitivity analysis prior to the optimisation stage.

Deep Parameter Optimisation approach (Chapter 3) applies Mutation Operators to automatically generate different versions of the program, each of which only differs from the original program on a single syntax, and evaluates the mutants with respect to a given regression test suite to find out the most sensitive code to the property-to-be-improved. It only selects the most sensitive locations and exposes implicit ("deep") parameters that are later tuned to improve the performance of the program whilst keeping it faithful to the regression test. In our empirical study, we only modified the memory management library *dlmalloc* for C programs and used four subject programs to demonstrate the approach. The results showed a promising 20% reduction on memory consumption and 12% reduction on execution time in the best cases.

Though using only traditional Mutation Operators has shown promising improvement in our experiments, these operators were not designed to influence software's memory performance. Therefore we proposed Memory Mutation Operators that directly target memory management statements for C programs (Chapter 4). In order to show that these operators can also be used to reveal memory vulnerabilities, we analysed these operators in the context of traditional Mutation Testing and compared them with Selective Mutation Operators.

Our results on 18 real-world programs showed that not only Memory Mutation Operators generated fewer mutants, but also generated higher quality mutants that are harder to kill. In addition to traditional strongly killing criterion, we proposed two weakly killing criteria to more effectively distinguish non-equivalent mutants. Our study suggests that all three criteria contributed in killing non-equivalent mutants with no one completely subsuming another, therefore all three killing criteria should be applied in future Mutation Testing studies.

We did not apply Memory Mutation Operators in the Deep Parameter Optimisation approach because, these operators modify C language memory management

functions such as `malloc()` and `free()`, while our Deep Parameter Optimisation study focused on the source code of these functions themselves, where these functions were not regressively called. Therefore, in the corpus of our Deep Parameter Optimisation study there was no memory management functions and adding Memory Mutation Operators would not change the results. However, in the study of another mutation-based approach "HOMI" (Chapter 5), we applied Mutation Operators on the source code of the subject programs themselves, therefore both Selective Mutation Operators and Memory Mutation Operators were included. Similar to the Deep Parameter Optimisation approach, the HOMI approach applies Mutation Operators and conducts sensitivity analysis to find out which part of the code is more sensitive to the property-to-be-improved. In the second stage, HOMI applies search algorithms to find the optimal Higher Order Mutants (or combinations of multiple First Order Mutants) that keep the functionalities and outperform the original program on the target non-functional property(ies).

our results showed 20% memory reduction and 18% time reduction in the best cases on the same four subjects that were used in the Deep Parameter Optimisation study. In addition, we combined HOMI-optimised programs with Deep-Parameter-optimised libraries to see whether there was further improvement. According to the results, we found not only further improvement, but also some synergy between subjects and libraries, which motivates further study in the future.

In summary, the overall contribution of this thesis is a demonstration that mutation-based Genetic Improvement is not only effective in finding better performance, but also ensures a manageable search space through sensitivity analysis, so that the approach can easily scale up to large real-world applications. In addition, we also demonstrated that our Memory Mutation Operators generate fewer but higher-quality mutants, and that when used in mutation-based GI approaches, they are effective in providing more potential memory improvement.

6.2 Future Work

In this section, we summarise potential future research directions following this work.

We evaluated the Deep Parameter Optimisation approach on the source code of memory management library for C language, and linked the library with several real-world subjects to demonstrate the improvement. The advantage of modifying the library only is that it will never change the semantic of the subjects so that it is more likely to generate mutants that are faithful to the functionalities than other approaches. However, because it does not modify the source code of the subjects, it cannot improve the sub-optimal code in the subjects themselves. In fact, the Deep Parameter Optimisation approach itself does not have any limit on where it can be applied: source code of the subjects or libraries. Therefore, it could be applied to the source code of subjects themselves, or both the subjects and multiple libraries that affect performance. When Deep Parameter Optimisation approach is applied on the source code of subjects, Memory Mutation Operators can be included to potentially increase the memory/time reduction.

Though HOMI approach differs from Deep Parameter Optimisation, they do not necessarily conflict with each other. Deep Parameter Optimisation favours exploitation since it searches the optimal values for each deep parameter, while HOMI favours exploration because it combines many simple changes together. In future work, these two approaches are likely to be combined together to balance exploration and exploitation.

For instance, HOMI can be applied in the first step to explore potential beneficial changes, followed by a sensitivity analysis to determine which of them can be further exploited to gain more improvement. In the second step, Deep Parameter Optimisation can be performed on these places to find the optimal solutions. We can even dynamically perform Deep Parameter Optimisation during the process of HOMI approach, to optimise each Higher Order Mutant before they are compared with each other.

For different purposes, software may have requirements for different non-

functional properties. In this thesis, we focus on two of the most important non-functional properties as our objectives: execution time and memory consumption, but the approaches are not limited to these objectives. Energy consumption is crucial to portable devices such as laptops and cell phones, and response time can be crucial to web applications. A direction for future work may extend the current objectives to more non-functional properties or even functional properties. In the meantime, other types of Mutation Operators may be proposed to target corresponding performance vulnerabilities and to be applied in our approaches to have more impact on the objectives.

We used automated approaches to find modifications to the source code to gain improvement on the properties of interest. However, whether these modifications can be understood and accepted by human developers remains unknown. A future study may compare the machine-generated modifications with human-written patches, both of which can be assessed manually by a third party to determine the quality of them. The goal of this study is not to show how machine-generated patches are similar to human-written patches, but to investigate the quality of machine-generated patches compared with human-written ones, even though they may be completely different. In addition, a large empirical study may be conducted to find out how likely a machine-generated patch will be accepted by the software's developer, given the improvement gained from the patch.

Bibliography

[1] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1375–1382, New York, NY, USA, 2015. ACM.

[2] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke. Mutation testing of memory-related operators. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10, April 2015.

[3] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Information and Software Technology*, 81:97 – 111, 2017.

[4] Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*, pages 18–33. Springer, 2016.

[5] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic improvement for adaptive software engineering (keynote). In *Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2014.

[6] Haitao Dan, Mark Harman, Jens Krinke, Lingbo Li, Alexandru Marginean, and Fan Wu. Pidgin crasher: searching for minimised crashing gui event sequences. In *International Symposium on Search Based Software Engineering*, pages 253–258. Springer, 2014.

[7] Yue Jia, Fan Wu, Mark Harman, and Jens Krinke. Genetic improvement using higher order mutation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 803–804, New York, NY, USA, 2015. ACM.

[8] Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. Sbselector: Search based component selection for budget hardware. In *International Symposium on Search Based Software Engineering*, pages 289–294. Springer, 2015.

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

[10] L. Li, M. Harman, F. Wu, and Y. Zhang. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[11] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[12] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[13] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug 2011.

[14] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration cuda software with genetic program-

ming. In *Conference on Genetic and Evolutionary Computation*, GECCO '14, 2014.

[15] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[16] Andrea Arcuri and Gordon Fraser. *On Parameter Tuning in Search Based Software Engineering*, pages 33–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, Jan 2012.

[18] Nevon Brake, James R. Cordy, Elizabeth Dan Y, Marin Litoiu, and Valentina Popes U. Automating discovery of software tuning parameters. In *Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, 2008.

[19] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1), 2009.

[20] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[21] Justyna Petke, William B. Langdon, and Mark Harman. *Applying Genetic Improvement to MiniSAT*, pages 257–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[22] Cristian Ţāpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Conference on Supercomputing*, Supercomputing '02, 2002.

[23] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[24] Takahiro Katagiri, Kenji Kise, Hiroaki Honda, and Toshitsugu Yuba. *FIBER: A Generalized Framework for Auto-tuning Software*, pages 146–159. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[25] Richard Vuduc, James W. Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *International Conference on Computational Science (ICCS)*, 2001.

[26] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Conference on Supercomputing*, Supercomputing '98, 1998.

[27] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, 2007.

[28] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 220–233, New York, NY, USA, 1980. ACM.

[29] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, Sept 2011.

[30] A.Jefferson Offutt and RolandH. Untch. Mutation 2000: Uniting the orthogonal. In W.Eric Wong, editor, *Mutation Testing for the New Century*,

volume 24 of *The Springer International Series on Advances in Database Systems*, pages 34–44. Springer US, 2001.

[31] Nan Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 220–229, April 2009.

[32] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

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

[34] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In Davide Balzarotti, SalvatoreJ. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*, pages 86–106. Springer Berlin Heidelberg, 2012.

[35] Steve Christey, M Brown, D Kirby, B Martin, and A Paller. CWE/SANS top 25 most dangerous software errors, 2011. http://cwe.mitre.org/top25.

[36] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 223–234, New York, NY, USA, 2005. ACM.

[37] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory er-

rors). In *Computer Security Applications Conference, 2004. 20th Annual*, pages 82–90, Dec 2004.

[38] Emre C. Sezer, Peng Ning, Chongkyung Kil, and Jun Xu. Memsherlock: An automated debugger for unknown memory corruption vulnerabilities. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 562–572, New York, NY, USA, 2007. ACM.

[39] Allen Troy Acree Jr. On mutation. Technical report, DTIC Document, 1980.

[40] H. Dan and R.M. Hierons. Smt-c: A semantic mutation testing tools for c. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 654–663, April 2012.

[41] H. Shahriar and M. Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 979–984, July 2008.

[42] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[43] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. *Search Based Requirements Optimisation: Existing Work and Challenges*, pages 88–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[44] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203 – 249, 2010.

[45] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 5–18, New York, NY, USA, 2014. ACM.

[46] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, March 2011.

[47] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[48] M. O'Keeffe and M. O. Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–260, March 2006.

[49] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *2nd International Symposium on Search Based Software Engineering*, pages 143–152, Sept 2010.

[50] Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 1003–1006, New York, NY, USA, 2008. ACM.

[51] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 321–332, New York, NY, USA, 2016. ACM.

[52] M. Boussaa, O. Barais, B. Baudry, and G. Suny. Notice: A framework for non-functional testing of compilers. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 335–346, Aug 2016.

[53] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[54] Frank Hutter, Domagoj Babic, Holger H. Hoos, and A.J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007.

[55] W. B. Langdon and J. P. Nordin. *Seeding Genetic Programming Populations*, pages 304–315. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[56] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652, New York, NY, USA, 2014. ACM.

[57] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[58] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, June 2012.

[59] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1775–1782, New York, NY, USA, 2008. ACM.

[60] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. *Multi-objective Improvement of Software Using Co-evolution and Smart Seeding*, pages 61–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[61] A. Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168, June 2008.

[62] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1427–1434, New York, NY, USA, 2011. ACM.

[63] W.B. Langdon and M. Harman. Optimizing existing software with genetic programming. *Evolutionary Computation, IEEE Transactions on*, 19(1):118–135, Feb 2015.

[64] Justyna Petke, Mark Harman, WilliamB. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In Miguel Nicolau, Krzysztof Krawiec, MalcolmI. Heywood, Mauro Castelli, Pablo Garca-Snchez, JuanJ. Merelo, VictorM. Rivas Santos, and Kevin Sim, editors, *Genetic Programming*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer Berlin Heidelberg, 2014.

[65] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339, 2009.

[66] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, June 1982.

[67] Robert V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252, 1996.

[68] G. Suganya and S. Neduncheliyan. A study of object oriented testing techniques: Survey and challenges. In *2010 International Conference on Innovative Computing Technologies (ICICT)*, pages 1–5, Feb 2010.

[69] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *CoRR*, abs/1411.1154, 2014.

[70] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[71] Emelie Engström and Per Runeson. *A Qualitative Survey of Regression Testing Practices*, pages 3–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[72] TimothyA. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[73] A. Jefferson Offutt and W. Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[74] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946, May 2015.

[75] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[76] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[77] Konstantinos Adamopoulos, Mark Harman, and RobertM. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1338–1349. Springer Berlin Heidelberg, 2004.

[78] Edward Miller and William E Howden. *Tutorial: software testing & validation techniques*, volume 138. IEEE Computer Society, 1978.

[79] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jzala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, Jan 2014.

[80] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980. AAI8025191.

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

[82] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 655–666, New York, NY, USA, 2016. ACM.

[83] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 100–107, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[84] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J Martin, Aditya Mathur, and Eugene Spafford.

Design of mutant operators for the c programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[85] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 352–363, 2002.

[86] M. Harman, Yue Jia, and W.B. Langdon. A manifesto for higher order mutation testing. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 80–89, April 2010.

[87] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416 – 2430, 2010.

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

[89] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, March 2012.

[90] H. Haga and A. Suehiro. Automatic test case generation based on genetic algorithm and mutation analysis. In *2012 IEEE International Conference on Control System, Computing and Engineering*, pages 119–123, Nov 2012.

[91] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 870–880, Piscataway, NJ, USA, 2012. IEEE Press.

[92] J. Svajlenko, C. K. Roy, and J. R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 8–9, May 2013.

[93] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399, New York, NY, USA, 2015. ACM.

[94] A. C. Nguyen and S. C. Khoo. Discovering complete api rules with mutation testing. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 151–160, June 2012.

[95] Sam Ratcliff, David R. White, and John A. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1907–1914, New York, NY, USA, 2011. ACM.

[96] Eric Schulte, ZacharyP. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3), 2014.

[97] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Paolo Vavassori. A novel use of equivalent mutants for static anomaly detection in software artifacts. *Information and Software Technology*, 81:52 – 64, 2017.

[98] PaulR. Wilson, MarkS. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In HenryG. Baler, editor, *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer Berlin Heidelberg, 1995.

[99] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? *SIGPLAN Not.*, 34(3):26–36, October 1998.

[100] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. *SIGPLAN Not.*, 37(11):1–12, November 2002.

[101] P Vilela, M Machado, and WE Wong. Testing for security vulnerabilities in software. *Software Engineering and Applications*, 2002.

[102] Michael A Zhivich. *Detecting buffer overflows using testcase synthesis and code instrumentation*. PhD thesis, Massachusetts Institute of Technology, 2005.

[103] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 167–182. Springer Berlin Heidelberg, 2013.

[104] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[105] M. J. P. v. d. Meulen and M. A. Revilla. Correlations between internal software metrics and software dependability in a large population of small c/c++ programs. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 203–208, Nov 2007.

[106] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Not.*, 27(12), December 1992.

[107] José L. Risco-Martín, David Atienza, J. Manuel Colmenar, and Oscar Garnica. A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. *Parallel Computing*, 36(10 - 11), 2010. Parallel Architectures and Bioinspired Algorithms.

[108] José L. Risco-Martín, David Atienza, Rubén Gonzalo, and J. Ignacio Hidalgo. Optimization of dynamic memory managers for embedded systems using grammatical evolution. In *Conference on Genetic and Evolutionary Computation*, GECCO '09, 2009.

[109] Doug Lea and Wolfram Gloger. A memory allocator, April 2000. http://g.oswego.edu/dl/html/malloc.html.

[110] Mark Harman, Yue Jia, and WilliamB. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 247–252. Springer International Publishing, 2014.

[111] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. *Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class*, pages 137–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[112] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5), 2011.

[113] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering*, 15. To appear.

[114] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 919–930, 2014.

[115] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2), 2002.

[116] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, 2002.

[117] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[118] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), 2005.

[119] Yue Jia and Mark Harman. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Proceedings of the TAIC PART'08*, pages 94–98, Windsor, UK, 29-31 August 2008.

[120] CarlosM. Fonseca and PeterJ. Fleming. On the performance assessment and comparison of stochastic multiobjective optimizers. In *PPSN*. Springer Berlin Heidelberg, 1996.

[121] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4), Nov 1999.

[122] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing (keynote). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Graz, Austria, April 2015.

[123] H. Dan and R.M. Hierons. SMT-C: A semantic mutation testing tools for C. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 654–663, April 2012.

[124] C.K. Roy and J.R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 157–166, April 2009.

[125] BenW.Y. Kam and ThomasR. Dean. Linguistic security testing for text communication protocols. In Leonardo Bottaci and Gordon Fraser, editors, *Testing Practice and Research Techniques*, volume 6303 of *Lecture Notes in Computer Science*, pages 104–117. Springer Berlin Heidelberg, 2010.

[126] J.S. Bradbury, J.R. Cordy, and J. Dingel. Exman: A generic and customizable framework for experimental mutation analysis. In *Mutation Analysis, 2006. Second Workshop on*, pages 4–4, Nov 2006.

[127] Anna Dereziska and Anna Szustek. Object-oriented testing capabilities and performance evaluation of the C# mutation system. In Tomasz Szmuc, Marcin Szpyrka, and Jaroslav Zendulka, editors, *Advances in Software Engineering Techniques*, volume 7054 of *Lecture Notes in Computer Science*, pages 229–242. Springer Berlin Heidelberg, 2012.

[128] Songtao Zhang, Thomas Dean, and Scott Knight. A lightweight approach to state based security testing. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '06, Riverton, NJ, USA, 2006. IBM Corp.

[129] M. Kusano and Chao Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 722–725, Nov 2013.

[130] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: A mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[131] Heinz Riener and Grschwin Fey. FAuST: A framework for formal verification, automated debugging, and software test generation. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 234–240. Springer Berlin Heidelberg, 2012.

[132] Yue Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, pages 94–98, 2008.

[133] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, May 2015.

[134] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[135] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M Hierons, and Mark Harman. An analysis of the relationship between information squeeziness and failed error propagation in software testing. *RN*, 13:18, 2013.

[136] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04).

[137] Arthur Griffith. *GCC: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2002.

[138] NicholasI.M. Gould, Dominique Orban, and PhilippeL. Toint. Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, pages 1–13, 2014.

[139] Mark Harman, Yue Jia, Pedro Reales Mateo, and Macario Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 397–408, New York, NY, USA, 2014. ACM.

[140] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009. Source Code Analysis and Manipulation.

[141] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of mutant operators for the C programming language. techreport SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[142] Geoffrey Neumann, Mark Harman, and Simon Poulding. *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, chapter Transformed Vargha-Delaney Effect Size, pages 318–324. Springer International Publishing, 2015.

[143] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.