2017

Case studies of metamorphic tesing

Wenjuan Ma
University of Wollongong

University of Wollongong 

CASE STUDIES OF METAMORPHIC TESTING

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Award of the Degree of

Master of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Wenjuan MA

School of Computing and Information Technology
Faculty of Engineering and Information Sciences

2017

Table of Contents

List of Tables

List of Figures

ABSTRACT

Metamorphic testing (MT) is a property-based software testing method which alleviates the oracle problem and enables test automation. The oracle problem refers to the difficulty or high cost of deciding whether the output program of test cases is correct. In this thesis we use MT to alleviate the oracle problem by testing code obfuscators which perform code obfuscation tasks. Code obfuscation is a popular and effective technique for protecting software code. Its key function is to transform a Program Being Protected (PBP) to a Semantically-Equivalent Version (SEV), such that the SEV is difficult to reverse (de-compile) or understand; SEV and PBP must have equivalent behaviour. Like compilers, obfuscators are critical applications because incorrectly obfuscated code not only compromises the confidentiality of software, it also cause serious and unexpected problems during the execution of SEV. However, the oracle problem makes it difficult to test the functional correctness of obfuscators, such deciding the equivalence between PBP and SEV. Although a lot of research into testing compilers has been carried out, very little has been done on testing obfuscators. In this research we use MT to test obfuscators in an automated fashion. The results of experiments show that MT detected a number of previously unknown bugs in 4 real world obfuscators, including open source software, free software, and commercial software.

In the second part of this thesis we conduct a case study on the characteristics of good metamorphic test cases that are very good at detecting faults. MT is a property-based testing strategy with properties known as metamorphic relations (MRs). For a target program many MRs are usually identified, and many test cases are usually generated for each MR. Testers must know how to prioritise the MRs and test cases so that faults can be detected earlier. In this study we investigate the relationship between fault-detection effectiveness and (dis)similarity between the initial and follow-up test case executions which constitute a metamorphic test. The results of these experiments confirm Cao et al.'s finding [10] [11] that the higher the dissimilarity, the better the chance that a metamorphic test will detect a fault. This finding can be used to prioritise MRs and test cases where test case coverage data are available or can be estimated, such as in the context of regression testing.

Acknowledgements

I am grateful to A/Prof. Zhiquan Zhou and Prof. Willy Susilo for their guidance of this research. Part of the thesis has been published in *Computer* [17].

Chapter 1

Introduction

1.1 Background

As technology continues to develop, more and more intelligent software is now used in industries such as bank payments and space exploration. Software plays an important role in our life but there are still negative effects such as vulnerabilities in the financial system which increase the risk of payment and may cause inestimable costs. This is why the quality of software is important and why software testing is receiving more attention in current software development.

The goal of software testing is to "provide information about the quality of the test item and any residual risk in relation to how much the test item has been tested; to find defects in the test item prior to its release for use; and to mitigate the risks to the stakeholders of poor product quality" [1]. Based on different phases of the software cycle, a reasonable and effective test plan and test cases can be designed and used to detect any fault in the program under test in order to verify the correctness of the software. Thus, the effectiveness of software testing will determine the quality of the software.

It is advisable to choose appropriate methods to test software because testing is a

complex process and many testing methods are currently being applied. Depending on the different aspects of testing there are many classifications, but based on the structure and algorithm of the particular software, two basic approaches are currently used to test software: black-box testing and white-box testing.

While both are useful, they do have different features. Black-box testing is also called "specification-based testing in which the principal test basis is the external inputs and outputs of the test item, commonly based on a specification, rather than its implementation in source code or executable software" [1]. It is commonly used to test software which focuses on its behaviour rather than its internal structure. Here the possible inputs are tested and the actual output is compared to the output expected from the specification to verify whether the actual output is correct. White-box testing is used to validate whether there is any fault in the algorithms, the internal structure, or the overall efficiency. White-box testing is also known as structural testing because it is only concerned with the internal perspectives of the software. Black-box testing and white-box testing should be designed to complement rather than replace each other, and therefore the choice of which to apply should be based on the testing requirements.

Testing can be done manually or automatically, each of which has benefits and disadvantages. In manual testing there are no tools or scripts, whereas with automation testing, test cases are carried out using tools, scripts, and software. Manual testing is carried out by humans, which is better when the requirements are not clear or there are no guidelines, as with new products for example. However, manual testing is not always reliable due to the limitations of human knowledge. When test cases are to be used iteratively, automation testing is considered to be used because automation testing is faster and less time is required to verify the outputs while the test cases and scripts take longer to write. Automation testing is better when frequent repetition is

required, although the maintenance costs are high. So the method of testing should be based on their particular features and on the test resources.

Software testing can be classified as functional testing, performance testing, usability testing, security testing, and compatibility testing, etc., but it can also include isolated components, unit testing, and integration or system testing. However, regardless of the method used, the primary aspects of software testing are test cases, respective designs, selection, and execution. Here, testers verify correctness by comparing the actual output to the expected output, and if there is any difference, then a failure is detected.

Normally the test cases for black-box testing can derive from the requirements or specifications of the software. Many methods of test case design have been proposed and applied [58] [67], such as Boundary Value Analysis (BVA), Equivalence Partitioning (EP) and Decision Table Testing, for white-box testing the Statement Coverage, Branch Coverage, and Path Coverage. While testing is generally supported by large numbers of test cases, the testing resources are not infinite to support every possible test case, which is why the design and selection of test cases will be considered. It is better to select some superior test cases from existing test cases because most of them are used in the regression test stage [37] [54]. The goal is to prioritise the test cases and choose the more effective test cases for further testing.

Testers will often use a test oracle [6] to decide whether a program has failures, by comparing the actual output to the expected output, but the test oracle may not always be available or cannot be practically applied. It is assumed that an oracle is available for most situations, but when complicated numerical simulations, compilers, or search engines are to be tested, the oracle is not always valid. Since it is difficult and/or expensive to verify correctness, it is also called an oracle problem [6] [21] [44]. The reliability of existing test cases also pose a problem because designing an absolutely

reliable test suite for a program is difficult, so its correctness can only be ensured by executing all the test cases. Therefore we need better methods to understand the test oracle problem and determine the reliability of test cases.

Metamorphic testing (MT) was introduced by Chen et al. [21] to alleviate the oracle problem, but in test cases where failure cannot be detected, it is still useful for future testing because it can be recombined to generate a new test case to test the target program [49] [55]. The expected output of these new test cases can be checked using the metamorphic relations (MRs) which exist between two or more test cases and their expected outputs. If the expected output cannot be obtained in advance, testers can then compare the outputs against the metamorphic relation to obtain a result, but if the result does not satisfy the metamorphic relation, then the implementation is incorrect. In MT, the program can be tested many times in order to validate its correctness. MT is useful at alleviating the oracle problem and it is also a technique for generating automation test cases. In practice, MT can also be used as a complementary testing method in conjunction with conventional testing, which will lead to faster fault detection [57]. Along with the development of software testing techniques, more research results in the area of metamorphic testing are also presented. In the ICSE International Workshop on Metamorphic Testing, more testing in this area has been discussed to present novel ideas about metamorphic relations, the application domain, and test case selection [7] [27] [61].

For instance, to test the *sine* function many metamorphic relationships can be defined based on the domain knowledge of the trigonometric function method. For $sin(x)$, the metamorphic relations are defined as follows: $sin(x) = sin(\pi - x)$, $sin(x) = sin(x + 2\pi)$, ..., etc.; the function is tested more than once based on these metamorphic relations but they are generated in the knowledge domain. Another example is calculator testing, but calculating some complex expressions is very expensive, so

MT can be used for testing and some simple MRs can be constructed. Moreover, the exchange property can also be used to design a metamorphic relation to test target functions such as $a + b = b + a$. The outputs on either side of this equality should be the same; otherwise there is a failure, although more MRs can be designed according to the properties of the calculator.

Like the oracle problem, selecting which test cases to carry out is another problem because to increase coverage, many cases must be prepared, and sometimes testers do not have enough time to run every case so they must priorities key cases, and this in turn means knowing the fault-detecting capacity of each test case. Similarly, when MT is used for testing, many test cases have the same metamorphic relation or more than one metamorphic relation has been created for the same property. This is why software testing has become an expensive but necessary activity in the development of modern software and why software testers or software engineers must design test inputs and oracles based on their specifications or requirements. Furthermore, expected outputs are also used to determine whether a test passed or failed, but as the software industry has developed, some parts cannot be tested or the tests are very expensive because the expected outputs such as big data modules, machine learning, compilers, and obfuscators are difficult to find.

In industry testing, testing "non-testable" programs [66] and is subject to limitations using conventional testing methods such as compiler testing. Moreover, identifying the oracle problem is difficult and expensive, as is automation testing, and since human intelligence and experience is needed to validate the non-testable programs, it can be unreliable and prone to error. This is why many techniques are used to alleviate these testing problems [43] [68] .

Code transformation software is a type of non-testable program which aims to transform source code based on specific methods. The common software used here

are compilers and code obfuscators. A source code is compiled by a compiler to an executable binary file where the code obfuscator can transform the source code into a confused condition, and while they have different features, they both change the original source code. Several methods have been proposed to validate the correctness of compilers [43] [68], and likewise, the quality of code obfuscators is also receiving more attention.

Code obfuscation is an important part of software development, so it is used in many domains [36] [38], although extensive testing is needed to ensure that the code obfuscation software and code obfuscators are correct [52]. This means the quality and correctness of the obfuscator is important because any fault may result in a faulty source code. If errors exist in a code obfuscator, the original semantics of a program can be changed and may even result in a non-executable program. I Obfuscated code is difficult to check because it is unreadable, so potential errors can cause the programmer a lot of angst; this is why having a correct code obfuscator is crucial to software development. Finally, how to verify the correctness of code obfuscators will also be discussed in this thesis.

1.2 Research Goals

This research has two main goals:

The first goal is to investigate whether a metamorphic testing technique is effective at code obfuscator testing and attempt to adopt MT into this application domain so that . Based on the properties of this application domain, these metamorphic relations are supposed to be used for further testing in the family of code obfuscators.

Because testing resources are not infinite, testers need intelligence and guidance to understand which pair(s) of test cases should be prioritised. Research into the effectiveness and priority of metamorphic test case pairs will be assessed as the second goal

in this thesis. Some distance metrics, coverage Manhattan distance (CMD), frequency Manhattan distance (FMD) and frequency Hamming distance (FHD) [69], are used to accomplish these research goals.

1.3 Contributions of the Thesis

In this thesis 6 bug reports were generated in 4 real world code obfuscators using the metamorphic testing method, and a failure was found in a commercial code obfuscator. These code obfuscators are already on the market, but had they been tested beforehand, more attention would have been given to conventional testing methods.

In this thesis, two sets of metamorphic relations were constructed to test code obfuscators, both effectively detected failures in code obfuscators so they will be used to test other code obfuscators, regardless of the platforms or languages.

The ability and effectiveness of pairs of metamorphic test cases were investigated, and several distance metrics were used to quantitatively measure any dissimilarity between them. The results confirm Cao et al.'s findings [10] [11] that good metamorphic tests yield large distances.

1.4 Organisation of the Thesis

The rest of this thesis is organised as follows: in Chapter 2, code transformation and metamorphic testing techniques are introduced in the literature review. Chapter 3 presents the metamorphic testing technique used to test code obfuscators. The subject programs, metamorphic relations, and the results of experiments will be shown. The effectiveness and priority of metamorphic test case pairs will be reported in Chapter 4, and the conclusions of this research are summarised in Chapter 5.

Chapter 2

Literature Review

2.1 Code Obfuscation

2.1.1 The Basic of Code Obfuscation

Since software needs more protection for intellectual property pertaining to software, more effective methods of protection for all software providers have been proposed. However, increasing the intellectual property while including the source codes of programs faces the risk of illegal encroaching and tampering [22] [32]. For example, decompilation is a technique that attackers often use to produce the source code from a binary code; a common decompiler for JAVA is Jad [41]. Java class files can easily be decompiled so that the source code can be retrieved by similar decompilers. Although some languages are now more complex, more and more tools have been proposed to decompile and reverse the software.

To prevent the source code from disclosure, Collberg et al. [23] proposed a code obfuscation technique. In the code obfuscation process, a Program Being Protected (PBP) is transformed into a difficult-to-read code which hides the real internal logic of PBP to confuse an attack. Here, even if a program is decompiled, attackers can

only obtain the obfuscated source code, so the real semantic meaning of the program is still difficult to read.

In 1984, the first **I**nternational **O**bfuscated **C** **C**ode **C**ontest (IOCCC) was held. The stated goals of this contest include demonstrating the importance of programming style and illustrating "some of the subtleties of the C language" [24]. The candidates were required to develop interesting and "hard-to-understand" code, which they did, even with the simplest C code example. One anonymous entry accomplished the goal in an obfuscated C program (code A) (Figure 2.1) that can play on the conventional "hello, world!" program as follows [48] in first IOCCC:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Figure 2.1: Code A

Another program (code B) can also print "hello, world", which is a simple code that is generally used as an example when learning a programming language. Code A and B have the same behaviour, but code A is more difficult to read than code B.

```
#include <stdio.h>
main()
{
printf(''hello,world\n'');
}
```

2.1.2 The Techniques of Code Obfuscation

To make a *PBP* that cannot be understood, many different code obfuscation techniques are used to create a more complex *SEV*. The introduction to code obfuscation was presented by Campbell and Philip L. [23], and then Balakrishnan et al. [4] inves-

tigated general code obfuscation techniques. Byummarising previous works enables code obfuscation transformations to be classified into three main classes where several methods can be used to change the internal part of the original program for each transformation [3] [4] [5] [22] [23] [31] [32] [48] [60].

1) Layout transformation: this includes obscuring the code format; for example, the identifiers name is obfuscated, the comments or the space and tab are removed, and some information can be converted or removed so that the source code is changed into another confused format.

Layout transformation focuses on changing the look of the source code of a program rather than changing the program semantics. One method of obfuscation methods is to remove unnecessary blank lines or white spaces to break up the original look. Removing white spaces makes a program more compact, while another similar transformation approach is to remove the comments developed by a programmer with an explanatory source code. Likewise, more bogus comments and white spaces which cannot affect compiling can be inserted into the original source code to create confusion and misunderstanding.

Another method of transformation is to obfuscate the names of identifiers, such as the names of variables, class and function, because while identifier names should be meaningful, they are easy to identify, so this transformation can confuse the context by renaming it; for example, the name of count *count_num* can be changed to *ab789*. There are two ways to rename, either change the name of the identifiers randomly, or rename based on predefined rules such as the opposite meaning or certain induced names.

This type of obfuscation does not change the semantics of the program, it only confuses the syntax of the source code, and while it is easily defined and can be used automatically, it is generally strong enough for code protection. The variables

and constants of a source code are simple to transform and the original logic is not changed.

This means the structure of a source code between *PBP* and *SEV* is still equivalent whereas the others aim to obfuscate the original source code of PBP to SEV by varying extents.

2) Data transformation: In this case the structure of data in a source code can be converted to a variable representation such that data transformation changes the variables by techniques like splitting, merging, or converting static data to procedure data [23].

Data transformation is also divided into storage & encoding data, aggregation data, and ordering data [9]. A common method of transformation is to change the encoding of the integer variables to non-standard data representation so as not to reveal the real value of the variable. However, the values of the intermediate expressions must correctly compute the real value [31]; such that an integer variable x could be represented with a random integer a and another random integer b. That is $x' = a + b$.

Another method of data transformation is to change the lifetimes of the variables such that a variable x is not globally defined, and after obfuscation it will be changed to a global variable and used in different functions.

Collberg et al. [23] also discussed splitting up one Boolean or another variable into more than one variable, so that a variable x will be rewritten to textity variables (y_1, y_2, ..., y_n); while the number of new variables will affect the potency of obfuscation, cost has also increased. As well as splitting up one variable, merging more than one variable into one variable is also used to transform data.

Another popular method is static data convention; since static data is easy to identify in decompiling, it can be useful to make another program (or function) to dy-

namically produce the results in order to make the computation process more difficult to understand.

3) Control transformation: this uses techniques such as adding a dead code and extending the loop conditions to complicate the source code; whilst the inline method and the clone method can also be used to affect aggregation, ordering or computations [23].

These related methods are summarised by Majumdar et al. [46] and are known as a semantic transformation for the program. There are three main types of control transformations, aggregation, ordering, and computations.

A common method of aggregation is cloning which is used to make the process more difficult to understand. Obfuscating a method's call sites means that different routines will appear as if they are being called; several different versions of the same method can generally be constructed with different obfuscations.

There are also many ways of applying loop transformation in aggregation obfuscation to increase complexity; for instance, the "loop unrolling" method duplicates the body of the loop more than once, and with *for (i = 0; i < 3; i++)* it will be duplicated for three statements to represent the original loop. Another method called "loop fission" can change the same loop into several loops with the same iteration space.

These methods for ordering transformations are concerned with the locality of the source code, programmers generally keep the source code clean and logical so it is easy to read, but that makes it easy to understand the original semantic structure. Therefore, some methods applied in obfuscation such as expressions, statements or loops are designed to confuse the order of the source code.

Computation methods aim to confuse the semantic logic, but they cannot affect the results of the actual computation, however, there are methods such as dead or irrelevant code insertion and loop condition extension where some useless code snippet

can be inserted into the original source code to hide the real semantic purpose of control-flow.

More researchers are focusing on more complex construction of obfuscation methods [5] [31] [50] [52], but apart from obfuscation methods, obfuscation techniques are used to transform the different program stages so that the binary code file and the source code can be confused [33]. For example, a subject program called "Obfuscator-LLVM" can output an obfuscated binary code file, whereas others will create an obfuscated source code from the original source code. Both of these levels of obfuscation can protect the real source code by transforming original source code structure.

In summary, all the code obfuscation techniques focus on confusing a *PBP* to make it hard to read, but after being obfuscated, the *PBP* will be transformed into a version that is more difficult to understand and the comprehensibility of the code hides the real purpose of *PBP*. Several papers indicate that code obfuscation can make an unintelligible program while maintaining the functionality the same as the original code. Collberg et al. [23] defined code transformation such that Program P is confused to P' by transformation T: $P \xrightarrow{T} P'$ produces the same output as P. More precisely, the obfuscated program P' should behave the same as the original program P. Although there are several kinds of obfuscation, their performance has no effect on their original behaviour.

2.1.3 The Quality of Code Obfuscators

While code obfuscation has been widely applied to protect software, a type of software that can confuse and transform software has been developed; it is called a code obfuscator. Code obfuscators are used in different platforms to support programming languages such as C, C++ and JAVA. They now form a family of code obfuscators known as ProGuard for JAVA code programs, PHPprotect for PHP code programs,

and Cobfusc for C programs, as well as other commercial software such as Stunnix for multiple languages. As a tool for obfuscating codes, they have the same goal of preventing the source code from unwrapping, while hiding the real logic of *PBP* with some transformations and keeping the semantic correctness of *SEV*.

As an obfuscated version, *SEV* also a publicly released program that represents the complete executive behaviour of *PBP*. Therefore, any error of SEV involving code obfuscation may cause software companies, especially financial companies, to lose a lot of money, which means code obfuscators must be correct and verify the equivalence between *PBP* and *SEV*.

Technically, code obfuscators create a semantic equivalent version (SEV) according to transformation rules from a given input *PBP*, as shown in Figure 2.2, and then *SEV* shares the same set of all possible inputs as *PBP*. After this, the set of inputs is called the input domain, and therefore a correct code obfuscator should satisfy two properties. Firstly, *PBP* and its *SEV* perform the same in the whole input domain, and secondly, a code obfuscator should apply the predefined transformation rules correctly to create *SEV* from *PBP*.

Many famous software providers use code obfuscators to prevent malicious decompilations from attacking the intellectual property of their products. Here it is assumed that *PBP* is qualified for the company and then an *SEV* transformed by a code obfuscator will be released onto the market. Any error in *SEV* may cause heavy losses for the company, and since the correctness of *SEV* is affected by code obfuscators, their quality must be critically verified. To validate a code obfuscator involves human intelligence, especially when automatically verifying the correctness of non-deterministic transformation rules, and since judging the semantic equivalence between *PBP* and *SEV* is difficult due to constraints in the testing resources, no software can be assumed, and therefore code obfuscators must be tested systematically.

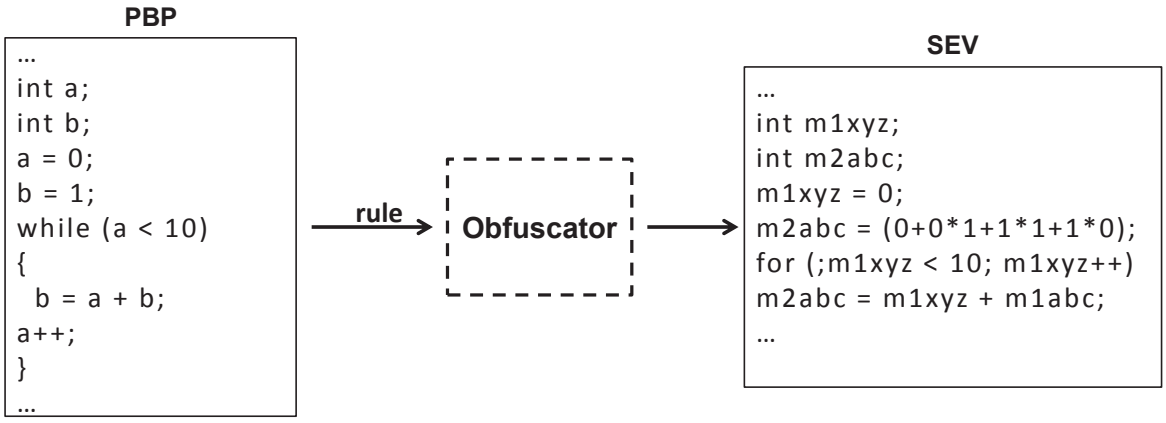

Figure 2.2: The executed process of code obfuscator

2.2 Metamorphic Testing

2.2.1 Basic Concepts of Metamorphic Testing

Testers normally use the expected outputs to verify the correctness of the program under test and generally assume that the expected outputs actually exist. However, in some situations there are no the expected outputs or it is very expensive to verify whether the actual output of a test case is correct; this is called the oracle problem [21]. Verifying this manually or empirically is very time consuming and is prone to error [47].

One famous oracle problem is the shortest path problem, which finds the shortest path between a start node A and an end node B in an undirected graph [16]. When a graph is large and complex, there are many paths between A and B and it will be very expensive and difficult to validate the results. Nevertheless, one property of this problem can be found, the shortest distance between A and B; there should only be one, and it should remain unchanged and therefore this property is a metamorphic relation where the distances of the initial input (A, B) and the follow-up input (B, A) are checked against the metamorphic relation. If this metamorphic relation has been

violated, a failure has been found.

Another example is if the oracle problem of an online search engine has been considered, such as the number of results returned or ranking [13] [70] [71].

The metamorphic testing technique is a property-based approach used to verify the program against the properties between the inputs and outputs of the multiple executions. By using the properties of the subject program, testers can automatically generate follow-up test cases from the initial test cases [8] [19].

Metamorphic testing can be carried out using the following approach; if we assume *p(x)* is a program under test, then let us assume T is an initial set of test cases that include x_1, x_2, ..., x_n, where $n > 0$. We first run p on T and get the corresponding outputs: p(x_1), $p(x_2)$, ..., $p(x_n)$, and to test this further, T is used to generate the follow-up set of test cases by a metamorphic relation r as the initial set of test cases, namely $T' = x'_1, x'_2, ..., x'_n$. Then p is run on T' to get the outputs $p(x'_1)$, $p(x'_2)$, ..., $p(x'_n)$. We can then check$p(x_i)$ and $p(x'_i)$ against r, and if there is any element to violate r, a failure is detected. Further follow-up test cases can be generated based on different metamorphic relations and then the target program is validated by comparing the outputs of the initial and follow-up test cases.

A common example is the trigonometric function problem where for the *sine* function, MRs can be constructed based on trigonometric domain knowledge; for example, $sin(x) = sin(x+360)$ and $sin(x) = sin(180-x)$. To take this example further, let us take the first MR, so when the *sine* function is implemented by *p(x)* the follow-up test T' can be generated from T by *r:* $x = x + 360$. The outputs of the *sine* function would be equal on T and T', but if the outputs are not the same for any test case, then a failure has been detected.

A metamorphic test should involve several steps; first, identify the metamorphic relations based on the properties of the program under test, and then generate follow-

up test cases from the initial test cases by the existing metamorphic relations and execute them. Note that there are different executions [26] [29] [30]. Lastly, check the correctness of the results of the initial and follow-up test cases against the metamorphic relations [14].

2.2.2 Metamorphic Relations

Note from the *sine* function testing that there may be more than one metamorphic relation based on the properties of the program under test. However, the fault-detection capabilities of diverse metamorphic relations differ, so constructing a metamorphic relation is a critical step in metamorphic testing. Metamorphic relations are generally designed on the basis of the program under test or are property based [16] [18]; some methods proposed to construct metamorphic relations will be reviewed.

Liu et al [45] presented a method called the "Composition of Metamorphic Relation" for constructing new metamorphic relations. Existing metamorphic relations can be combined by some specific rules to construct more of them, for instance, there are two independent metamorphic relations MR1 and MR2. MR1 and MR2 would be "compositable" if the follow-up test cases of MR1 can be used as the initial test cases of MR2; this extends the construction of metamorphic relations, but it is more cost effective than the original metamorphic relation.

Identifying a metamorphic relation is a common relationship that makes extensive use of the identity relationships expected of the target program. However, some other relationships are also used to construct metamorphic relations. Chen et al. [19] designed beyond identity metamorphic relations and verified the program under test by comparing the differences between the results. When Zhou et al. [71] investigated to employ metamorphic testing in online search services testing, they constructed a set of non-identity metamorphic relations.

Since there are many methods for construct metamorphic relations, more than one can be involved in testing, in fact the more diverse the metamorphic relation is, the more effectively would the program be tested [2] [16] [39]. However, testing resources are not infinite, so while it is impossible to use all the metamorphic relations, their prioritisation is important for metamorphic testing [15] [18] [40]. This means that selecting the most effective ones will effectively test and also increase the chances of detecting failures.

Asrafi et al. [2] assessed the relationship between coverage and fault detection by monitoring the coverage of the initial and follow-up test cases. After that, more distance metrics were also used by Cao et al. [10] [11] to measure the fault detection capacity. Manhattan distance metrics [69] were employed to measure the distance between the initial and follow-up test case executions, as well as investigating the relationship between the effectiveness of metamorphic relations and difference between the initial and follow-up metamorphic test cases. They also found a strong correlation between the distance and the fault detecting capability of metamorphic relations which can be used in their selection.

2.2.3 The Application Domains of Metamorphic Testing

MT has been widely applied in various application domains such as web services and applications [56]. Metamorphic testing methodology was used in Service Oriented Applications (SOA) testing [12], WSDL description of web services [20], and online web search applications [13] [35] [62] [70] [71]. MT has also been adopted to generate automatic random testing and verify the results; the results reveal that metamorphic testing is effective at web services and applications testing. MT was also applied in machine learning algorithms. Murphy et al. [51] constructed the 6 metamorphic relations which exist in most machine learning applications and found there are a adopted lot

of real bugs in the three machine learning tools. In the decision support domain, MT has also been used for testing purposes. Metamorphic testing approach was presented by Kuo et al to test the decision support systems [42], and in which they found a bug that had not been detected during conventional testing. MT technology has also been applied in computer graphics, numerical programs, and other domains such as compiler testing. Tao et al [63] presented a metamorphic relation called "equivalence preservation" to generate equivalent test cases (programs). The correctness of compilers can be validated by checking the behaviour of the executable results. A testing tool called Mettoc was developed to automatically construct equivalent variants. Le et al [43] also presented two test compilers by using a new method called "equivalence modulo input" which deletes dead codes and generates a new equivalent input, and also validates the consistency of the original and new outputs. It is useful for testing a compiler, and indeed it confirmed 147 bugs in GCC and LLVM. In 2014, Vu et al. [43] proposed a method of removing dead code from test compilers to alleviate the oracle problem. A C program was compiled as the initial test case and then a new c program was created as a follow-up test case from the original c program by randomly removing the unexecuted lines and running the new c program on the same input. The outputs of the two c programs should be the same otherwise there is a failure in the compiler. The relationship between the new c program and the original c program was constructed as a metamorphic relation that can detect bugs in a variety of compilers. Code obfuscators such as compilers, are also an important tool in code transformation because they transform the source code into an unreadable file and binary file, while code obfuscators ship the source code into a file that is difficult to read, although with some transformation rules. With more and more attention being given to metamorphic testing technology, MT will increasingly be used in the code transformation domain.

2.3 Summary

This chapter has reviewed the literature on code obfuscation and metamorphic testing. There are several different methods for code obfuscation and there are many code obfuscators on the market.

Metamorphic testing can be used in situations where there is either no or very few test oracles, and it can also be integrated with conventional software testing to become an effective and complementary testing method. Firstly, we can define some metamorphic relations based on the properties of the target program, even without an oracle [21]. Secondly, metamorphic testing can automatically generate follow-up inputs for further testing. Thirdly, metamorphic relations can be reused after being defined with similar properties of a product family because they can save costs; and finally, metamorphic testing is independent of any programming language.

Chapter 3

Metamorphic Testing of Code Obfuscators

3.1 Subject Programs

In these experiments we chose four real life code obfuscators as a pilot, namely Cobfusc, Stunnix, Tigress and Obfuscator-LLVM because they are very good at protecting intellectual software property and at preventing malicious attacks on the source code. They can all obfuscate C programs and make them difficult to understand for illegal reuse, but the obfuscators must remain *PBP* and *SEV* equivalent. Each program has its own features as the following descriptions will show.

3.1.1 Cobfusc

Cobfusc is a Linux utility which is an open source code in the package cutils [25]; it only makes the C source code difficult to reuse. It mainly works on converting the layout, including every change in the identifier, comment removing and compacting the white spaces. For instance, option *a* can change the string to an octal form while

the variable name can be confused by option *c*. More options are then used to make the source code more complicated and prevent it from being reused. However, Cobfusc does not support the function that affects the internal logic, so it cannot realise the need for more complicated obfuscation, but it is effective and easy to use for elementary code obfuscation.

3.1.2 Stunnix

The Stunnix C++ Obfuscator is the only commercial tool involved in these experiments. Stunnix is an advanced tool to solute code obfuscation for languages such as C, C++ and Perl [59]. In this thesis the CXX-Obfus of Stunnix is involved because it can obfuscate the C and C++ source code. To make *PBP* more difficult to understand and reuse, Stunnix supports more obfuscation functions. A sample of code converting on the Stunnix website [59] is shown in Figure 3.1 and Figure 3.2, which represent the original and confused code respectively. Obviously the identifier names become more complicated and the layout is also changed by removing the spaces.

As a commercial code obfuscator, Stunnix has been a partner with many famous companies such as Cisco, Motorola, Siemens and DELL. The code obfuscators supported by Stunnix work for companies who develop software on many platforms, and since it is commercial software, a trial version was applied in the experiments.

3.1.3 Tigress

Tigress is another important tool for making the C language more intricate [64]. It was developed by Christian Collberg, a professor in the Department of Computer Science at the University of Arizona. Tigress is mainly used in related research work and while it is not an open-source program, we can download the binary code for different platforms. Tigress is a virtualiser for C/C++ source code and supports many types

```
main( int argc,  char * argv[ ] )
{
  int j;
  char version[ 80] ;
  while ( ( j =  getopt_helper( argc,  argv,  "n:o:vV:",  ((char)(0x2053+885-0x2360)), ((char)
(0x2368+457-0x24db))) )  !=  - 1)  {
switch ( j)  {
case ((char)(0x8f4+4043-0x1851)):
  name_wide =  MYMIN( atoi( optarg) , 0xff) ;
  break;
```

Figure 3.1: *PBP* for Stunnix

```
main( int za82b547bcb,  char * argv[ ] )
{
  int z2d29194d43;
  char version[ (0x138f+2785-0x1e20)] ;
  while ( ( j =  zefd3fb4f7d( argc,  z6965940303,  "n:o:vV:",  ((char)(0x2053+885-0x2360)),
  ((char)(0x2368+457-0x24db))) )  !=  - (0xc4+9243-0x24de))  {
switch ( j)  {
case ((char)(0x8f4+4043-0x1851)):
  z1c0ab7cf0c =  z048b31e7a8( atoi( optarg) , (0xfec+5036-0x2299)) ;
  break;
```

Figure 3.2: *SEV* for Stunnix

of code transformations, especially control and data transformation. It differs from other subject programs in that the obfuscations focus on changing the layout such as the variable name, space and comments. Tigress also supports more complicated transformations for data-flow and control flow, which means that a failure detected is related to data transformation by a metamorphic relation. Tigress can also generate different transformations based on different seeds, and therefore it can generate more variants for the same transformation.

3.1.4 Obfuscator-LLVM

The last code obfuscator is called Obfuscator-LLVM. LLVM is an important compiler which has been applied in famous companies such as Apple Inc. and Adobe Systems Incorporated, etc. The Obfuscator-LLVM is the part of LLVM developed by the information security group at the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains [53]. It is based on the Clang compiler and supports code transformation. It differs from the other obfuscation tools applied in this thesis because of the final output. Other obfuscators such as Stunnix for example, generate an obfuscated code for a further compiling process, but the transformations of Obfuscator-LLVM occur in the compiling process and the binary output is the obfuscated file. The obfuscated code cannot be outputted, which means the Clang compiler is necessary. It is also useful for code obfuscation, but it is difficult to validate correctness because the obfuscation is invisible. Obfuscator-LLVM currently supports some main function transformations, including substitution, flatting, and bogus control flow, and in the future it will support even more features.

3.2 Metamorphic Relations and Experimental Results

3.2.1 Metamorphic Relations

A code obfuscator should apply the transformation rules correctly, which means the input (PBP) to a code obfuscator and the output (SEV) generated by the code obfuscator should be semantically equivalent and denoted by $PBP \equiv SEV$. Semantic equivalence also implies that they should "consistently" produce the same outputs in the whole input domain. We call this the, "Relation of Behaviour Equivalence(RBE)", as shown in Figure 3.3. To simplify the discussion we will use $PBP(t)$ and $SEV(t)$ to denote the behaviour after executing an input t and $\overline{PBP}$ and $\overline{SEV}$ to describe the behaviour of PBP and SEV for every input in the whole input domain. In this process we simply assume that the compiler used to compile PBP and SEV is correct, even though the Compiler also has bugs [66]. Based on these two properties, we constructed two sets of MRs.

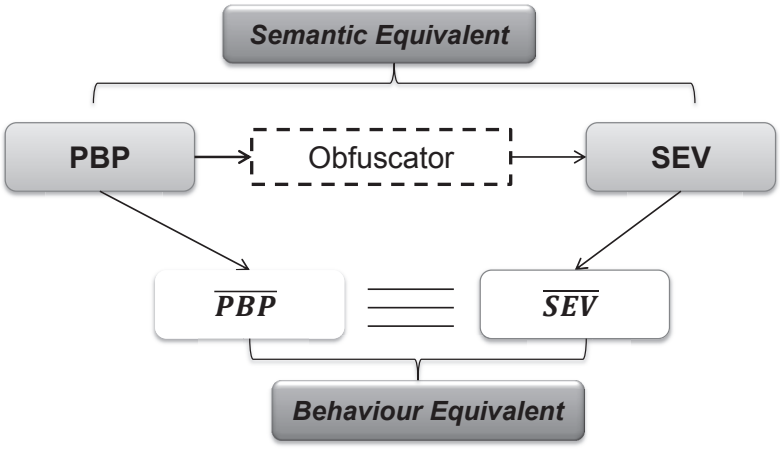

Figure 3.3: Relation of Behaviour Equivalence

Behaviour Equivalence Relations

RBE should hold for any given input program to obf. If $PBP \equiv PBP'$, then $SEV \equiv SEV'$, implying that *RBE* also hold in *SEV* and *SEV′*, which is $\overline{SEV} \equiv \overline{SEV'}$. If $\overline{SEV}$ and $\overline{SEV'}$ behave differently for some inputs, i.e. we can find any t which will trigger $SEV(t) \neq SEV'(t)$, and a failure was detected.

Three rules are purposed to generate PBP' in this thesis, MR1, MR2 and MR3.

MR1: The way to generate PBP' in this MR is to use another software S, which can change *PBP* but distort its semantic meaning; for example, C preprocessors of GCC can be used in PBP' generation. A program is expanded to an equivalent program after preprocessing by the inclusion of header files and Macro expansion. PBP' generation can also be implemented by scripts; for example, "If (condition) {do A} else {do B}" is equivalent to "If (not(condition)) {do B} else {do A}". Based on this feature of S, S is also called an "Alternative obfuscator *(AO)*" (Figure 3.4).

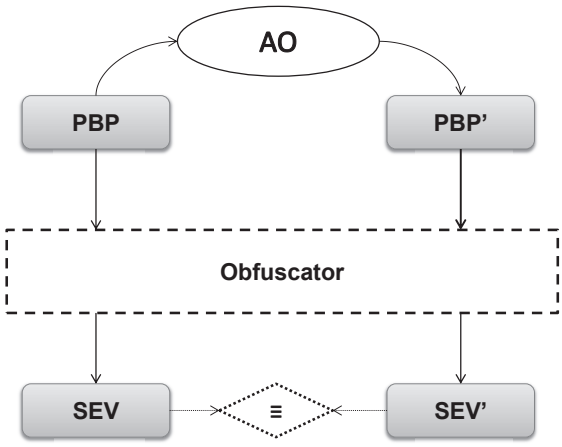

Figure 3.4: MR1

Because more the third party tools are involved in the testing process of MR1,

including AO and the compiler, more time is needed to diagnose the real reason for detecting failure, so a more dependable way of improving MR1 was tried.

MR2: The environment variables (EVs) are involved in MR2, but (PBP, EVs) and (PBP', EVs') are unchangeable. Code obfuscators are used to generate SEV and SEV' respectively based on the different EVs shown in Figure 3.5. If SEV and SEV' are not equivalent, then a failure has been detected. This MR can be applied in conditions such as system setting and multiple platforms, especially when the rule with randomness is triggered.

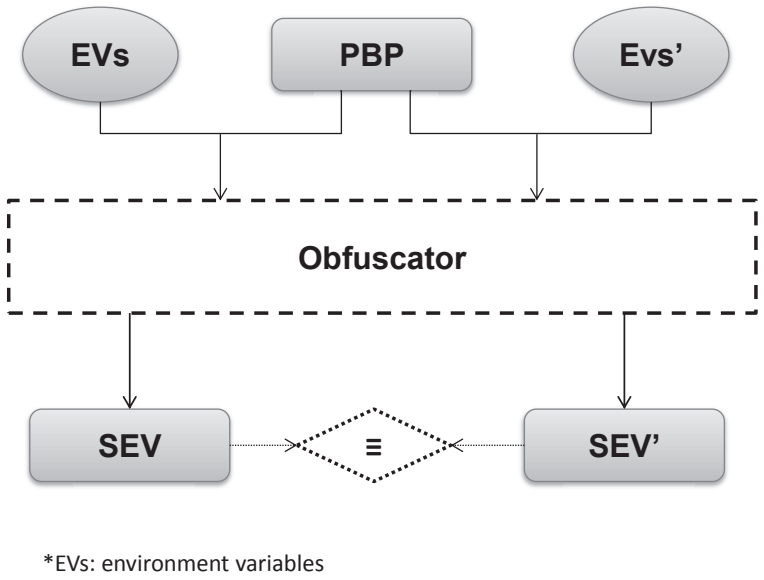

Figure 3.5: MR2

In MR2, environment variables are used to construct MRs and PBP' is generated with $EV(s)$. This diminishes the effects from third party techniques during the testing process.

MR3: The code obfuscator under test was used as AO to generate PBP' from PBP. In MR3, the first output SEV^1 of the code obfuscator was used as PBP' because the code obfuscators remain PBP and SEV equivalent, and SEV^1 is also equivalent to

PBP. We can repeat the process to generate SEV^2 from SEV^1, and then obtain the final output SEV^n after n time recursion. This MR is also called a "recursive relation". If SEV^n is not equivalent to SEV^1, there is a possible failure in the code obfuscator. MR3 is shown in Figure 3.6.

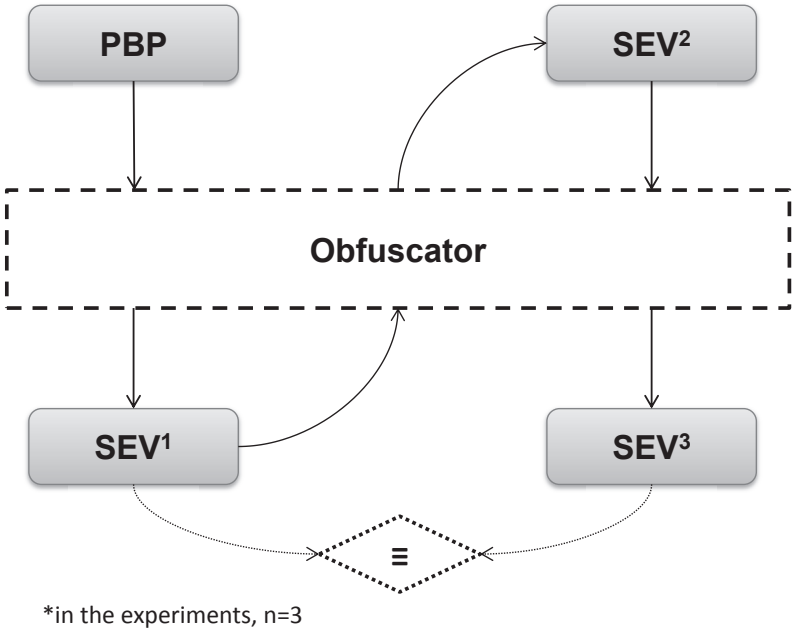

Figure 3.6: MR3

In MR3, the code obfuscator under test can be used to generate *PBP′*.It can be generated without third party techniques and can reduce the effect of the testing result. However, this transformation can be repeated, so we can get a different SEV respectively in each recursion. Therefore, any SEV^n (n>2) can be used as a follow-up output to compare with SEV^m (where m>1 & m≠n) against MR.

Transformation Rule Relations

The second set of MRs is used to validate the transformation rules by comparing the source codes, especially for non-deterministic rules. This type of MRs is executed without comparing the behaviour and saves time in compiling and executing. MR4

is similar to the MR2 involved EVsused in PBP to construct PBP'. For the number constants transform function, all the number constants are transformed into random equivalent expressions. For example, $i = 5$; is transformed into a random expression $i = (1*1+1+1*1+2/2)$. The expressions for $i = 5$; are different for each transformation because of randomness, but $i = 5$; should be transformed with each EV.

3.2.2 Experimental Results

In these experiments we chose four real world code obfuscators for the C program, Cobfusc, Stunnix, Tigress and Obfuscator LLVM respectively, and found 6 failures. Moreover, the two sets of MRs based on two different properties of code obfuscators were effective at code obfuscators testing. These failures are as follows.

MR1

In MR1, one failure was detected in Tigress (version: Linux x86_64-unstable revision 1676) when PBP' was constructed by PBP from AO. Then the data transformation will result in an error executing on SEV' because when the program is run, the *if* condition will always be run regardless of any input. Excerpts of PBP, PBP', SEV, SEV' and the output are shown in Figure 3.7, Figure 3.8, Figure 3.9, and Figure 3.10.

PBP has two integer variables i and j, each of which is assigned an initial value. If i is greater than j then i is set to $i - 10$, otherwise i is set to $i + 10$, and then the value of i is printed. In MR1, SEV and SEV' were compiled into executable programs and then executed, which were then run on the same input, and then their outputs were compared. Figure 3.10 shows that the outputs were different and therefore a failure was detected in Tigress; it was found that Tigress incorrectly transformed the PBP statement $if(i > j))$ into an SEV statement $if((int)((i > (long)j+116)-116))$. The if condition should be evaluated as a true or false value in the C language, so if its

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
      int i = 1000;
      int j;
      j = atoi(argv[1]);
      if (i > j)
           i -= 10;
      else
           i += 10;

      printf("%d\n", i);
}
```

(a) *PBP*

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
      int i = 1000;
      int j;
      j = atoi(argv[1]);
      if (i <= j)
           i += 10;
      else
           i -= 10;

      printf("%d\n", i);
}
```

(b) *PBP′*

Figure 3.7: *PBP* and *PBP′* for Tigress

value is 0, an if branch will be executed, but with a non-zero value, an else branch will be executed. In *SEV* the if condition is always evaluated to -115 or -116, so the if statement of *SEV* will always take the true branch and the false one will not be executed. Consequently, when *SEV* and *SEV′* were compiled and executed on the same input, they had different outputs, as Figure 3.10 shows.

There is still an arguable issue that failure would be detected by comparing the outputs of *PBP* and *SEV* in the conventional testing method, but it should be found when taking certain specific inputs and running the false branch. For *PBP*, only the i value is less than the j value, and then failure can be found. Thus, MT can still be applied with the oracle and will be better than conventional testing methods because it emphasizes the need to test from diverse perspectives.

In the experiments, some simple implementations used as AO were a first attempt; there are other methods which can be used to generate the equivalent programs, and they may be involved in further experiments.

```
int main(int argc , char **argv )
{
   long i ;
   int j ;

   {
   megaInit();
   i = 1000L + 116;
   j = atoi((char const      *)*(argv + 1));
   if ((int )((i > (long )j + 116) - 116)) {
      i = (i - (10L + 116)) + 116;
   } else {
      i = (i + (10L + 116)) - 116;
   }
   printf((char const      */* __restrict    */)"%d\n", (int )(i - 116));
```

Figure 3.8: SEV for Tigress

```
int main(int argc , char **argv )
{
   long i ;
   int j ;

   {
   megaInit();
   i = 1000L + 116;
   j = atoi((char const      *)*(argv + 1));
   if ((int )((i <= (long )j + 116) - 116)) {
      i = (i + (10L + 116)) - 116;
   } else {
      i = (i - (10L + 116)) + 116;
   }
   printf((char const      */* __restrict    */)"%d\n", (int )(i - 116));
```

Figure 3.9: SEV' for Tigress

```
tigress - unstable $gcc PBP.c
tigress - unstable $./a.out 15
990
tigress - unstable $gcc PBP\'.c
tigress - unstable $./a.out 15
990
tigress - unstable $gcc SEV.c
tigress - unstable $./a.out 15
990
tigress - unstable $gcc SEV\'.c
tigress - unstable $./a.out 15
1010
```

Figure 3.10: The outputs of *PBP*, *PBP′*, *SEV* and *SEV′*

MR2

A failure was found in Obfuscator-LLVM (obfuscator-clang version 3.4). Some programs have different outputs after multiple transformations, as Figure 3.11(b) and Figure 3.12 shows. As MR2 states that the outputs should be equivalent when obfuscating at different times for the same *PBP*. *PBP* was obfuscated with the same command line parameters of Obfuscator-LLVM, so the *SEVs* should have the same behaviour.

When the outputs were different, a failure was detected. This issue was confirmed when Clang was used to verify the output of *PBP*, and the outputs were the same regardless of the number of times it took to compile *PBP*. After further investigation, it was discovered that failure was caused by an uninitialised variable. If there is an uninitialised variable, the behaviour of *SEVs* may be different. It was also shown that the software must have the capability of various inputs, including unusual ones.

MR3

Three failures were detected based on MR3, in Cobfusc and Tigress respectively.

Two failures were also found in Cobfusc (package cutils version 1.6); the first one

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
int i;
int j;
int n = 694;
i = atoi(argv[1]);
if( i>n)
{
i = i + j;
}
printf("%d\n", i);
}
```

(a) *PBP*

```
build $./bin/clang PBP.c
build $./a.out 10000022
14195494
build $./bin/clang PBP.c
bui ld $./a.out 10000022
14195494
build $./bin/clang PBP.c
build $./a.out 10000022
14195494
```

(b) outputs of *PBP*

Figure 3.11: *PBP* and outputs of *PBP* for Obfuscator-LLVM

```
build $./bin/clang –mllvm –bcf –mllvm –boguscf -loop=3 PBP.c
build $./a.out 10000022
10000022
build $./bin/clang –mllvm –bcf –mllvm –boguscf -loop=3 PBP.c
build $./a.out 10000022
14195494
build $./bin/clang –mllvm –bcf –mllvm –boguscf -loop=3 PBP.c
build $./a.out 10000022
10000022
```

Figure 3.12: The outputs of *SEV*

```
#include <stdio.h>
#include “parse.h”
int j = 1729;
```

(a) *PBP*

```
%:include <stdio.h>
%:include “\160\141\162\163\145\56\150”
int j = 1729;
```

(b) SEV^3

Figure 3.13: *PBP* and SEV^3 for Cobfusc

was where a header file name using (*include “ ”*) in the code was changed to the octal format with MR3. SEV^3 cannot be compiled successfully after that because the header file could not be found. For example, the header file name (*include “version.h”*) was transformed, as shown in Figure 3.13, and when SEV^3 was compiled, the header files defined in the project should be loaded. Because the name of header file was changed, it cannot refer to the header file in the project, so it could not pass through the compiler. A failure was also detected in Cobfusc.

A second failure was also detected when testing Cobfusc. After multiple transformations of numerical variants, a whole line of source code was split into a new line. As Figure 3.14 shows, the length of the original line in the source code was longer after numerical variant transformation because the numbers were transformed into expressions. In MR3, the second line of SEV^3 was separated into three lines, so the new lines were not valid for the source code and it could not be compiled; a failure was also found.

A third failure was found in Tigress, as shown in Figure 3.15. After multiple transformations the function names clashed, which means Tigress added some a new functions to construct new SEV^3, and the new function name is the same as the existing one. A redefinition of the function names will result in unsuccessful compiling.

```
1 #include <stdio.h>
2 int p = 20;
3 int main(int argc, char * argv[]) {
4 int i;
```

```
1 #include <stdio.h>
2 int p = (((2*(2*1+0)+1)*((1*(1*1+0)+0)*(1*(1*1+0)+0)+0)+(1*(2*1+0)+0))*((1+1)
3 *((0+1)*(1*1+0)+0)+0)*((1*(1*1+0)+0)*((1+0)**(1*1+0)+0)+0)+0)+((1+2)*(
4 (2*(1*1+0)+0)*(0+1)+0)+0));
5 int main(int argc, char * argv[]) {
6 int I;
```

Figure 3.14: The failure of Cobfusc

```
2.c:89:6: error: redefinition of `m_i $nit'
 void m_i$nit(void)
      ^
2.c:73:6: note previous definition of `m_i $nit' was here
 void m_i$nit(void)
      ^
2.c:116:6: error: redefinition of `j_i $nit'
 void j_i$nit(void)
      ^
2.c:73:6: note previous definition of `j_i $nit' was here
 void j_i$nit(void)
      ^
```

Figure 3.15: The failure of Tigress

MR4

In the experiments a failure was detected in the numerical transformation of Stunnix (the trial edition of Stunnix CXX-Obfusc 4.2). (Because the output of Obfuscator-LLVM is executable file rather than source code, this type of MRs was not run on it.) It was found that some numbers cannot be changed successfully for each EV because they go against the specifications. The number constants can be transformed into expressions in which the value is computed the same as the original number. For more complex obfuscations, a different expression can be generated randomly which means that a number constant which can be transformed should be changed every time. The PBP, SEV, and SEV' are shown in Figure 3.16 and Figure 3.17.

Stunnix was also tested for other research purposes [65], but no faults were found. In this thesis the anomaly situation cannot be defined as a fault, but it should still receive due attention and since Stunnix is commercial software, any failure might result in costs.

```
#include <stdio.h>
int j = 1908;
int k = 1662;
int m = 1734;
int n = 468;
int p = 1046;
int q = 613;
int main() {
int i = 1000;
if (i > q)
{
i -= 10;
}
else if (i < k)
{
i += 10;
}
printf("%d\n", i);
}
```

Figure 3.16: *PBP* for Stunnix

```
#include <stdio.h>
int j = (0x1eb7+3040-0x2323);
int k = (0x129b+1584-0x124d);
int m = (0x791+6578-0x1a7d);
int n = (0xec4+227-0xdd3);
int p = (0x7b4+3551-0x117d);
int q = (0x16fc+819-0x17ca);
int main() {
int i = (0x186f+1261-0x1974);
if (i > q)
{
i -= (0x6cc+3610-0x14dc);
}
else if (i < k)
{
i += (0x221+8486-0x233d);
}
printf("%d\n", i);
}
```

(a) *SEV*

```
#include <stdio.h>
int j = (0xcd2+7981-0x248b);
int k = 1662;
int m = (0x1372+1577-0x12d5);
int n = (0x1fba+91-0x1e41);
int p = (0x1572+4135-0x2183);
int q = (0x514+7106-0x1e71);
int main() {
int i = (0xc25+6597-0x2202);
if (i > q)
{
i -= (0x97f+4918-0x1cab);
}
else if (i < k)
{
i += (0x93a+5435-0x1e6b);
}
printf("%d\n", i);
}
```

(b) *SEV′*

Figure 3.17: *SEV* and *SEV′* for Stunnix

Chapter 4

Effectiveness of Metamorphic Test Case Pairs

In the previous experiments with code obfuscators, some test case pairs were better at detecting failures within an MR, whereas failures could not be found when the other obfuscators were run. In a real world situation, it is possible that only a few test cases will find the failure. These pairs detecting failure(s) are called "violating Metamorphic Group" (violating MGs), while the others are called "non-violating Metamorphic Group" (non-violating MGs). Further research is needed to investigate whether there are different characteristics between the violating and the non-violating MGs.

4.1 Distance Metrics

We studied the distance metric of test case pairs as a metric to analyse any differences between the test case pairs. Following Cao et al. [10] [11], the following 3 distance metrics (initially proposed by Zhou et al. [69]) were used to measure the (dis)similarity between the initial and follow-up executions: the coverage Manhattan distance (CMD), the frequency Manhattan distance (FMD), and the frequency Ham-

ming distance (FHD).

Suppose x is an initial test case and let $X = x_1,\ x_2,\ ...,\ x_n$ be the execution information of the initial test case x, for $i = 1,\ 2,\ 3,\ ..,\ n$, where n is the total number of statements (or function or branch). If a statement or a branch has been executed once or more times, the value of x_i is set to 1; otherwise it is 0. Let $X' = x'_1,\ x'_2,\ ...,\ x'_n$ be the execution information of the follow-up test case x'. We then have $CMD(X, X')$ $= \sum_{i=1}^{n} \mid x_i - x'_n \mid$, where the values of x_i and x'_i $(i = 1,\ 2,\ ...,\ n)$ are either 1 or 0. Then, given $X = (0, 1, 1, 0)$ and $X' = (1, 1, 0, 0)$. X means that statement(or function or branch) 1 is not executed, statement(or function or branch) 2 is executed, statement(or function or branch) 3 is executed, and statement(or function or branch) 4 is not executed. The information given indicates that the number means the statement (or function or branch) is or is not covered. We do not care whether it is covered once or more times, as long as it is "covered". Base on the definition of CMD, we have: $CMD(X, X') = |0\text{-}1| + |1-1| + |1-0| + |0+0| = 1 + 0 + 1 + 0 = 2$. When the branches are executed, it is named BCMD, but when the statements are executed it is named SCMD. The FMD metric is based on frequency, so let $X = x_1,\ x_2,\ ...,\ x_n$ and $X' = x'_1,\ x'_2,\ ...,\ x'_i$ be the execution information of the initial test case x and the follow-up test case x' where n is the total number of statements(or function or branch). x_i and x'_i are the execution number (frequency) of statement i executed by the corresponding test case, for $i = 1,\ 2,\ 3,\ ...,\ n$. Then FMD compares each $(x_i,\ x'_i)$ and sums up the differences: $FMD(X, X') = \sum_{i=1}^{n} \mid x_i - x'_i \mid$. When branches are executed, it is named BFMD, but when statements are executed, it is named SFMD. Another metric concerned with the frequency of how many $(x_i,\ x'_i)$ pairs are not identical is FHD: $FHD(X, X') = \sum_{i=1}^{n} \mid k_i \mid$, where $k_i = 0$, if $x_i = {}'_i$, otherwise $k_i = 1$. x_i and x'_i are the number of times that a statement or branch has been executed, $i = 1,\ 2,\ ...,\ n$. When branches are executed, it is named BFHD, but when statements are executed

it is named SFHD.

4.2 Subject Programs

4.2.1 Cobfusc

Cobfusc is described in Chapter 3.

4.2.2 Gzip

Gzip (GNU zip) is a common utility designed for compression; it is frequently used to compress, and it is also used to uncompress. A file (or some files) should be compressed correctly and should be restored to a whole and infallible file (or files). Therefore, the quality of Gzip should be verified.

The options of Gzip which are copied from Gzip(http://www.gnu.org/) [34] are as follows:

"*–stdout/–to-stdout/-c* Write output on standard output; keep original files unchanged. If there are several input files, the output consists of a sequence of independently compressed members. To obtain better compression, concatenate all input files before compressing them."

"*–decompress/–uncompress/-d* Decompress."

"*–force/-f* Force compression or decompression even if the file has multiple links or the corresponding file already exists, or if the compressed data is read from or written to a terminal. If the input data is not in a format recognized by gzip, and if the option –stdout is also given, copy the input data without change to the standard output: let zcat behave as cat. If -f is not given, and when not running in the background, gzip prompts to verify whether an existing file should be overwritten."

"*–help/-h* Print an informative help message describing the options then quit."

"*–keep/-k* Keep (dont delete) input files during compression or decompression."

"*–list/-l* For each compressed file, list the following fields: compressed size, uncompressed size, ratio and uncompressed_name."

"*–license/-L* Display the gzip license then quit."

"*–no-name/–n* When compressing, do not save the original file name and time stamp by default."

"*–name/-N* When compressing, always save the original file name and time stamp; this is the default."

"*–quiet/-q* Suppress all warning messages."

"*–recursive/-r* Travel the directory structure recursively. If any of the file names specified on the command line are directories, gzip will descend into the directory and compress all the files it finds there (or decompress them in the case of gunzip)."

"*–fast/–best/-n* Regulate the speed of compression using the specified digit n, where -1 or –fast indicates the fastest compression method (less compression) and –best or -9 indicates the slowest compression method (optimal compression). The default compression level is -6 (that is, biased towards high compression at expense of speed)."

4.3 Metamorphic Relations

4.3.1 MRs of Cobfusc

The MRs of Cobfusc are described in Chapter 3.

4.3.2 MRs of Gzip

In the metamorphic testing literature, the diversity of metamorphic relations is effective for the program under test. Therefore, the identity and non-identity are both considered in constructing the MRs of Gzip. MR1, MR2, MR3, MR4 and MR5 rep-

resent an equivalent relation between the initial and follow-up test cases. MR6 and MR7 compared the initial and follow-up test cases for the less than and greater than. These MRs were constructed based on the knowledge of Gzip options, as shown below.

MR1: *gzip file* and *gzip* -c *file* should have the same output file.

MR2: When a file A is compressed to the file $B.gz$. When $B.gz$ is decompressed to file C, C should be same as file A.

MR3: -6 is the default compression level. The outputs with -6 and without this option should be same.

MR4: $-q$ is to control if the waring messages will be printed. The outputs with or without this option should be the same.

MR5: 1 equals to *fast* which indicates the fastest method of compression, so their outputs should have the same ratio.

MR6: With the default compression level, the output file should have a compression ratio, and then the output file should have a lower compression ratio with a random number selected from 1 to 5.

MR7: With the default compression level, the output file should have a compression ratio, and then the output file should have a higher compression ratio with a random number selected from 7 to 9.

4.4 Experimental Results

4.4.1 Experimental Results of Cobfusc

An independent t test was conducted to compare the means of the two different test case pair sets for each distance metric, shown in Table 4.1. An independent t test's Sig value that is less than or equal to 0.05 indicates a significant difference between the two variables. The cells with values lower than 0.05 are highlighted, thus indicating that

	violating MGs		non-violating MGs		t	Sig
	mean	SD	mean	SD		
BCDM	5.714	2.22	4.576	2.57	-5.268	< 0.05
BFHD	89.43	1.67	89.40	1.26	-0.245	> 0.05
BFMD	7421.33	2359.23	7310.44	2970.20	-0.463	> 0.05
SCMD	8.95	4.067	7.08	4.63	-4.767	< 0.05
SFHD	129.29	1.396	129.22	1.397	-0.536	> 0.05
SFMD	11014.07	3424.86	11021.05	4425.74	0.020	> 0.05

Table 4.1: Independent t test results of Cobfusc

the difference between the violating MGs and the non-violating MGs was statistically significant for BCMD and SCMD. The distributions of distance metrics are shown as follows.

Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5 and Figure 4.6 presents the distribution of violating MGs and non-violating MGs for the distance metrics in Cobfusc testing (0:non-violating MGs; 1:violating MGs). It was observed there was no large difference between the violating MGs' distribution and non-violating MGs' distribution for BFMD and SFMD, whereas BFHD and SFHD also had a similar distribution. However, the violating MGs' distribution was greater than the non-violating MGs' for BCMD and SCMD.

4.4.2 Experimental Results of Gzip

A set of 500 initial test cases were used in the experiments, including 50 test cases in the Gzip package and 450 test cases generated randomly. In these experiments, the same number of follow-up test cases were generated from the initial test cases to test Gzip for each MR. The Gzip package used in the thesis was downloaded from the Software-artifact Infrastructure Repository (SIR, http://sir.unl.edu) [28] and the size of the program was approximately 5,680 LOC. Some default faults had been activated

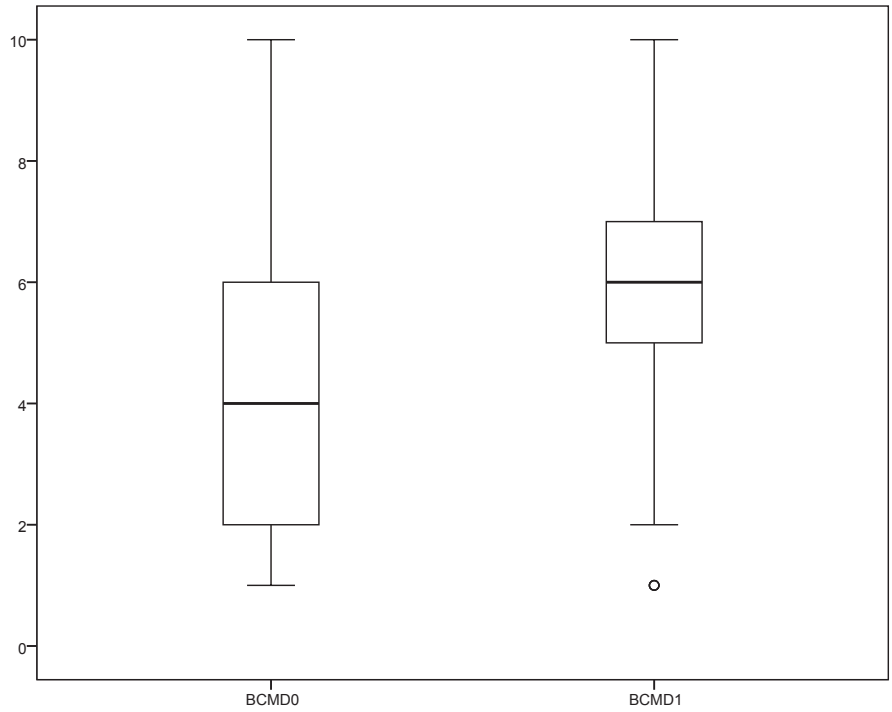

Figure 4.1: BCMD of Cobfusc

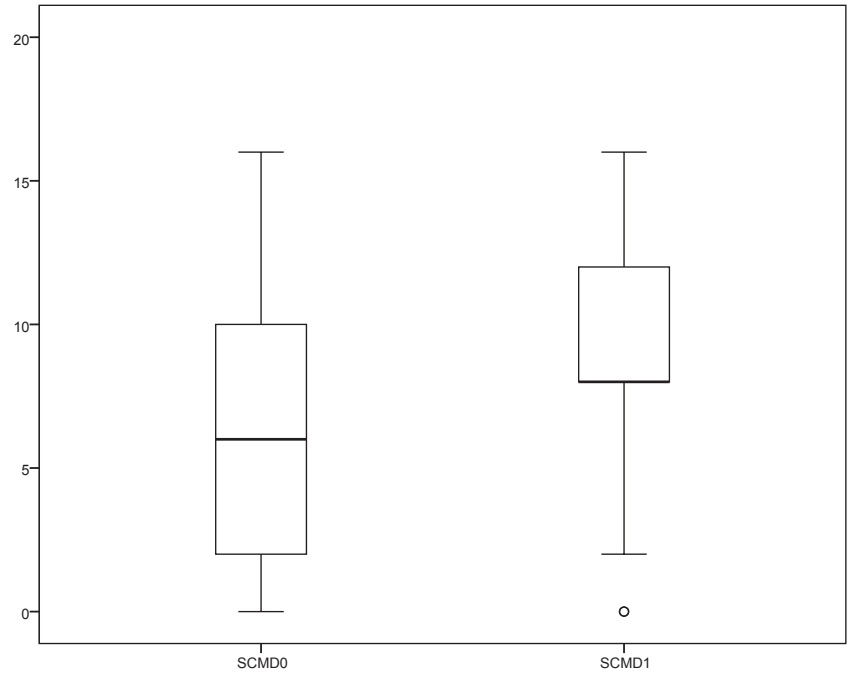

Figure 4.2: SCMD of Cobfusc

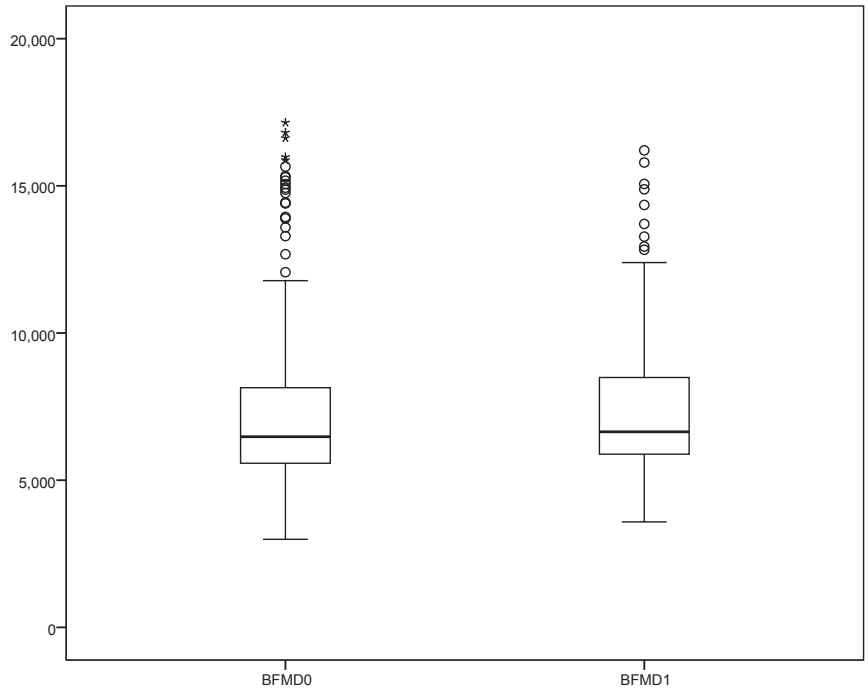

Figure 4.3: BFMD of Cobfusc

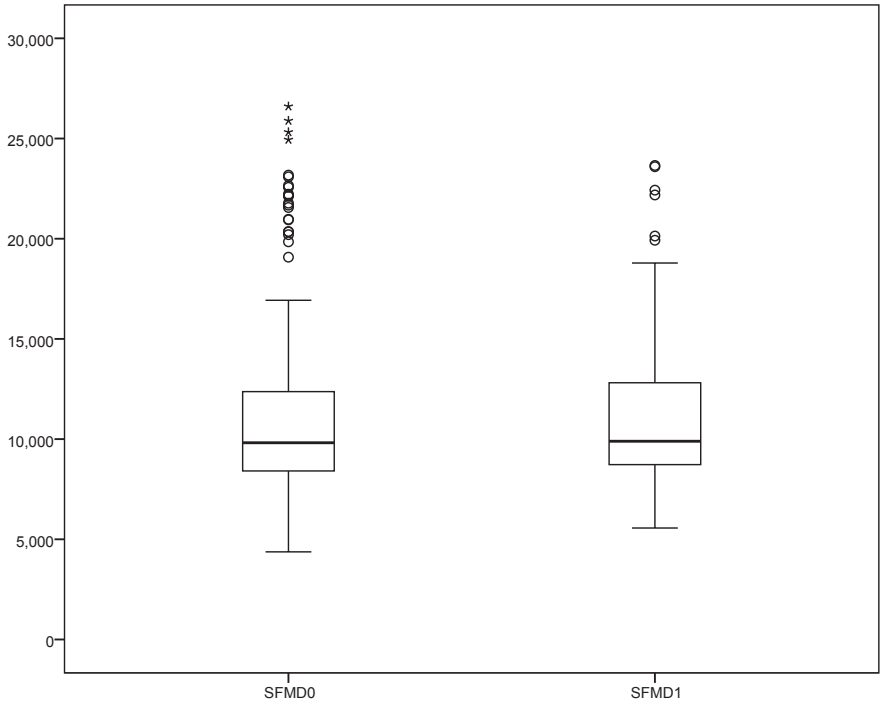

Figure 4.4: SFMD of Cobfusc

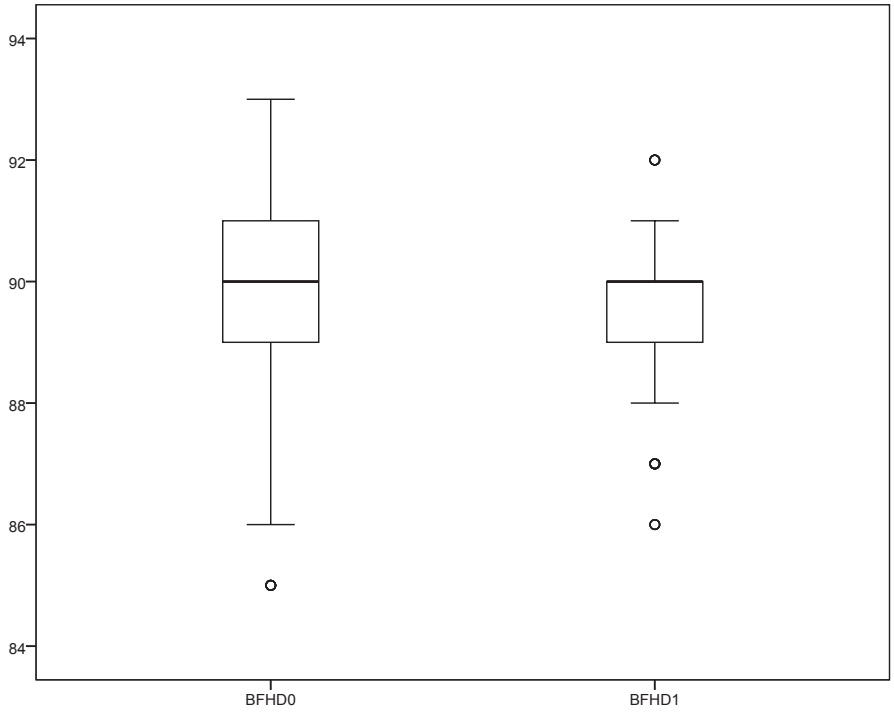

Figure 4.5: BFHD of Cobfusc

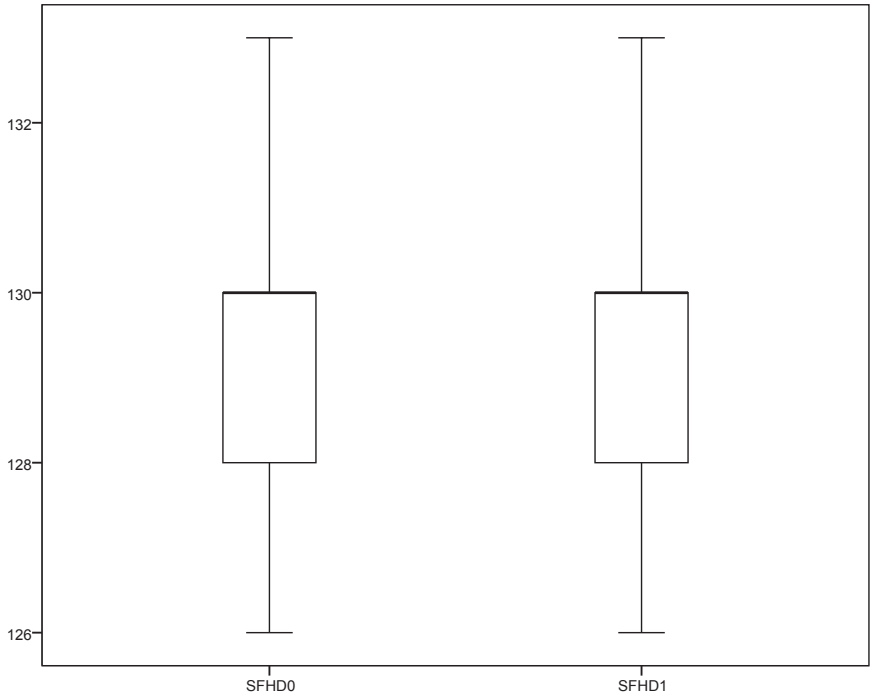

Figure 4.6: SFHD of Cobfusc

to create the 5 faulty programs used in the thesis. Furthermore, an independent t test was carried out to compare the mean value and sig value of violating MGs and non-violating MGs for each distance metric in the Gzip experiments. The results are shown in Table 4.2 and Table 4.3.

In Table 4.2, MR1, MR2, ..., MR7 represent 7 MRs, and *F* and *NF* refer to the violating MGs and the non-violating MGs. The value of each table cell indicates the mean value of the corresponding distance metric. The Sig values are shown in Table 4.3.

Table 4.2 shows that the mean values of the violating MGs were greater than the non-violating MGs for BCMD and SCMD, whereas there was no regularity between the two MGs for the other distance metrics. This finding is consistent with previous observations that the BCMD and SCMD are often better than other distance metrics in Cobfusc testing. An independent t test's Sig value that is less than or equal to 0.05 indicates a significant difference between the two variables. In Table 4.3 the cells with values lower than 0.05 are highlighted, thus indicating that the difference between the violating MGs and the non-violating MGs was statistically significant.

The distribution of violating MGs and non-violating MGs of Gzip are also presented in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11 and Figure 4.12 for MR1. Although the difference for BCMD and SCMD between violating MGs distribution and non-violating MGs distribution for Gzip was not greater than Cobfusc, there were significant differences from the other distance metrics distribution. Indeed Figure 4.7 and Figure 4.8 shows that BCMD is better than SCMD at representing fault-detection and is also found that coverage based metrics are more effective than frequency-based metrics [69].

It was also observed that all the MRs detected failure, but this does not mean that the MRs constructed during testing can find all the failures and that all the MRs

		BCMD	BFHD	BFMD	SCMD	SFHD	SFMD
MR1	F	98.03	90.12	85553	568.14	325.80	76125
	NF	94.18	90.16	99295	494.35	326.01	87173
MR2	F	113.56	89.56	85162	520.16	320.11	87173
	NF	103.08	88.99	83729	510.98	220.19	97276
MR3	F	106.14	92.60	91743	532.18	318.77	101123
	NF	88.72	93.14	91635	532.10	322.87	98972
MR4	F	120.36	92.16	58961	487.90	318.90	87276
	NF	97.03	92.50	99135	488.01	319.11	93817
MR5	F	100.31	95.88	93618	499.54	365.11	73763
	NF	96.08	96.01	87361	487.15	366.00	87261
MR6	F	97.14	93.42	107361	500.33	366.00	98271
	NF	89.00	94.00	99732	499.52	338.00	98272
MR7	F	102.13	90.04	72718	511.37	324.97	108287
	NF	97.77	90.03	87261	510.27	325.09	117272

*F: violating MGs
*NF: non-violating MGs

Table 4.2: Mean value of distance metrics of violating MGs and non-violating MGs

can be effective in detecting failure. Therefore, a diversity of MRs should be used to improve the testing coverage.

It can be concluded that distance metrics can describe the effectiveness of MGs and there is a difference between violating MGs and non- violating MGs, and this difference was statistically significant across all the subject programs. In other words, the coverage metric can reflect MGs fault-detection capability and can also be used to select metamorphic test case pairs.

	MR1	MR2	MR3	MR4	MR5	MR6	MR7
BCMD	0.019	0.011	0.029	0.031	0.038	0.041	0.022
BFHD	0.154	0.225	0.657	0.118	0.357	0.227	0.226
BFHD	0.716	0.592	0.847	0.691	0.277	0.321	0.305
SCMD	0.059	0.031	0.037	0.064	0.028	0.043	0.036
SFHD	0.881	0.602	0.595	0.091	0.189	0.226	0.158
SFMD	0.216	0.411	0.381	0.115	0.989	0.441	0.343

Table 4.3: Mean sig value of violating MGs and non-violating MGs

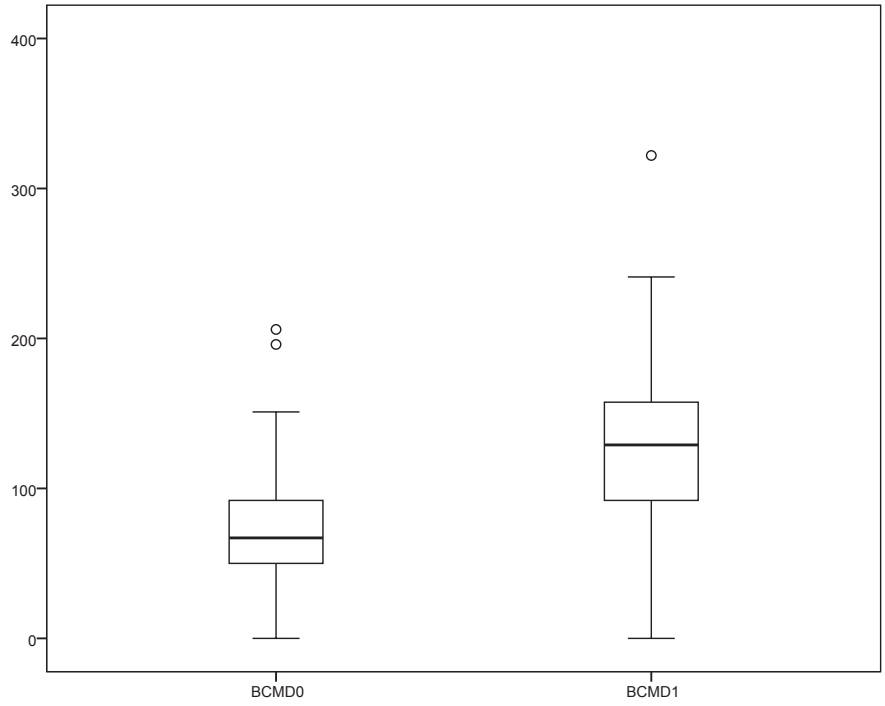

Figure 4.7: BCMD of Gzip

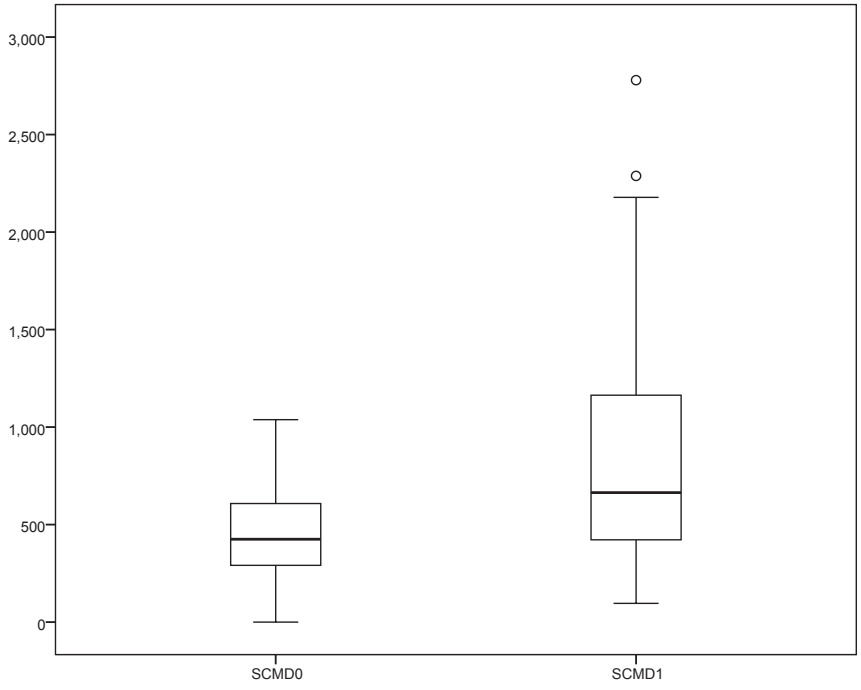

Figure 4.8: SCMD of Gzip

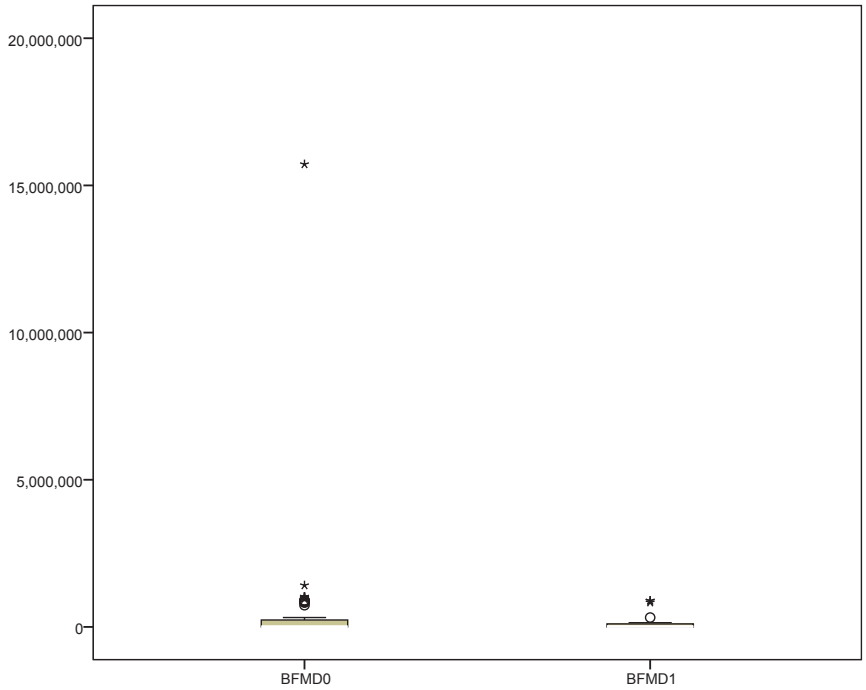

Figure 4.9: BFMD of Gzip

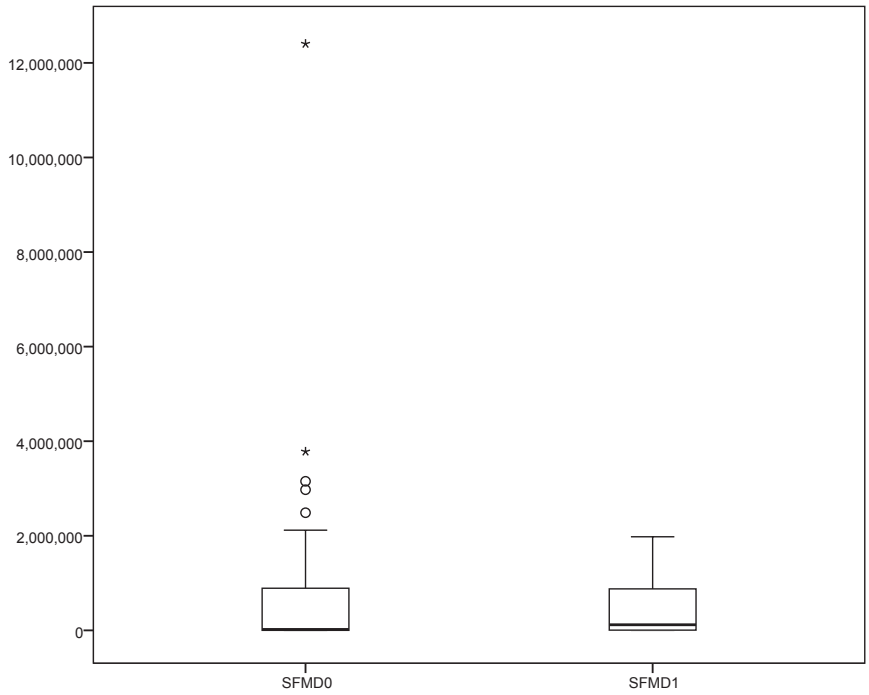

Figure 4.10: SFMD of Gzip

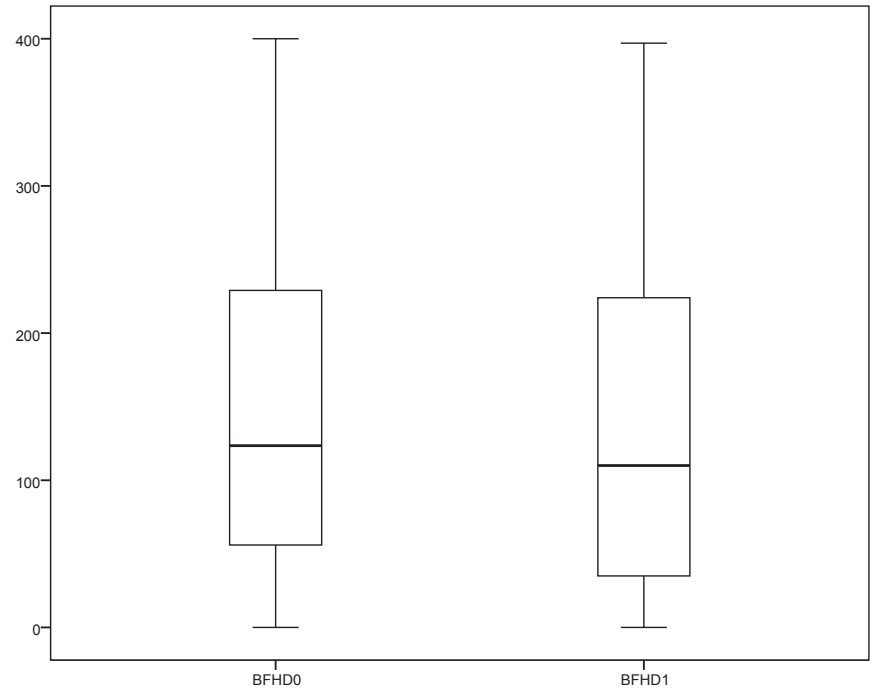

Figure 4.11: BFHD of Gzip

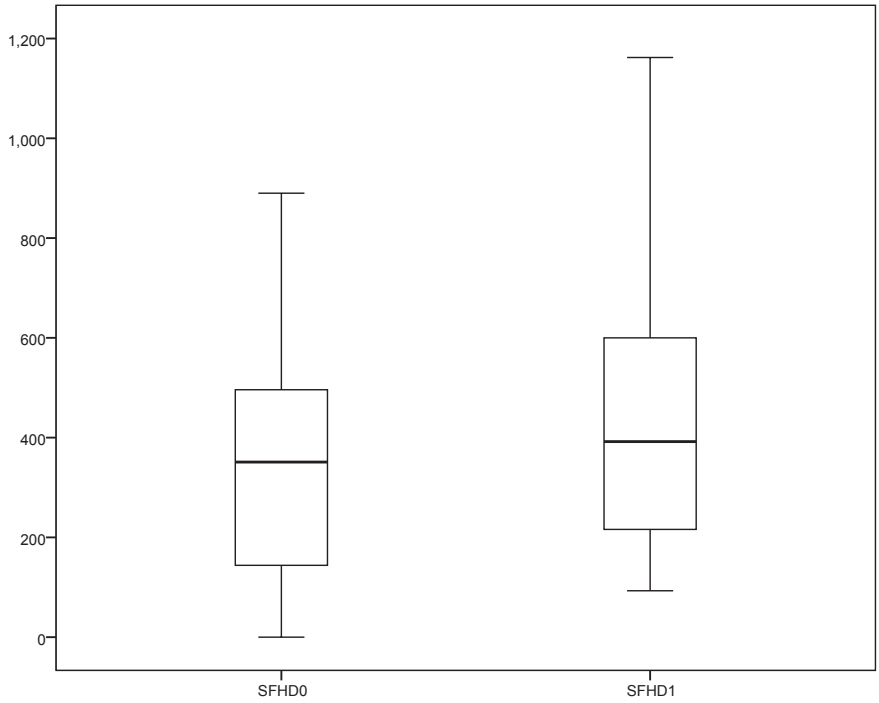

Figure 4.12: SFHD of Gzip

Chapter 5

Conclusion

Metamorphic testing is a practical approach to improve the cost effectiveness of testing and as such has been applied in various application domains, either with or without an oracle. MRs are identified from the expected properties of the target program, and the follow-up test cases for each MR can be generated automatically from the initial test cases.

Code obfuscation is a popular approach for protecting software and preventing malicious attacks for illegal purposes. Major companies use code obfuscation software, called code obfuscators, to protect their software. Like compilers, the correctness of code obfuscators is critical for the correctness of the final software products released onto the market, and therefore the quality of code obfuscators is a determining factor. In this thesis we focused on code obfuscator testing and used MT for this purpose. Two sets of MRs were constructed from the key properties of code obfuscators, and the behaviour equivalence and transformation rules were both verified. In the experiments, MT detected 6 previously unknown bugs in real world code obfuscators.

The first type of MR detected 6 bugs in the 4 real world obfuscators. These were implemented using the concept of *RBE*, which is the key property of code obfuscation. The behaviour of *PBP* and *SEV* should be equivalent to any obfuscation rule so it

can be used regardless of whether single or multiple obfuscation rules are applied.The second type of MR focused on the transformation rules. Although they could be verified manually, human intelligence is still needed and therefore manual verification is expensive and prone to error.

Code obfuscators have common functions so the systematic testing method is normally applicable for reuse in other related code obfuscation tools. The results reported in this thesis show that MT is effective at code obfuscator testing.

We also studied the nature of effective metamorphic test case pairs by using distance metrics to measure the (dis)similarity between the initial and follow-up test case executions. Our results provide hints for the selection and/or prioritisation of MRs and metamorphic test cases. Because testing resources are not infinite, a method for selecting or prioritising metamorphic test case pairs should be considered, i.e., it is important to know which metamorphic test case pairs should be given priority in software testing. In this thesis it was found that the pairs with a large coverage distance have a higher chance at detecting failure(s). This observation confirms the findings first reported by Cao et al. [10] [11]. Our findings can be used in situations where the initial and follow-up test case execution differences are known or can be estimated (such as in the context of regression testing). It is possible in future research to adopt other variables or metrics to further study the prioritisation and selection of MRs and metamorphic test cases.

References

[1] Software and systems engineering software testing part 1: concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)*, pages 1–64, Sept 2013.

[2] M. Asrafi, H. Liu, and F.-C. Kuo. On testing effectiveness of metamorphic relations: A case study. In *Proceedings of the 5th International Conference on Secure Software Integration and Reliability Improvemen (SSIRI'11)*, pages 147–156. IEEE Computer Society, 2011.

[3] V. Balachandran and S. Emmanuel. Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEEE Transactions on Information Forensics and Security*, 8(4):669–681, April 2013.

[4] A. Balakrishnan and C. Schulze. Code obfuscation literature survey, 2005. `pages.cs.wisc.edu/~arinib/writeup.pdf`

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference(CRYPTO 2001)*, pages 1–18, 2001.

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

[7] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, and H. W. Schmidt. The impact of source test case selection on the effectiveness of metamorphic testing. In *Proceedings of the IEEE/ACM 1st International Workshop on Metamorphic Testing (MET'16), in conjunction with the 38th International Conference on Software Engineering (ICSE)*, pages 5–11. ACM Press, 2016.

[8] G. Batra and J. Sengupta. An efficient metamorphic testing technique using genetic algorithm. In *Proceedings of the 5th International Conference on Information Intelligence, Systems, Technology and Management*, pages 180–188. Springer, 2011.

[9] P. L. Campbell. An introduction to software obfuscation, Sandia National Laboratories, United States Department of Energy, 2004.

[10] Y. Cao. On the selection of metamorphic relations in metamorphic testing. Master's thesis, School of Computer Science and Software Engineering, University of Wollongong, Australia, 2013.

[11] Y. Cao, Z. Q. Zhou, and T. Y. Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*, pages 153–162, July 2013.

[12] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, pages 470–476, Sept 2005.

[13] W. K. Chan, S. C. Cheung, and K. R.P.H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. In *Software Ap-*

plications: Concepts, Methodologies, Tools, and Applications., pages 2894–2914, 2009.

[14] L. Chen, L. Cai, J. Liu, Z. Liu, S. Wei, and P. Liu. An optimized method for generating cases of metamorphic testing. In *Proceedings of the 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, pages 439–443, Oct 2012.

[15] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[16] T. Y. Chen, F.-C. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *Proceedings of the 5th International Conference on Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD 2004)*, pages 128–134, 2004.

[17] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou. Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55, June 2016.

[18] T. Y. Chen, F.-C. Kuo, R. Merkel, and W. K. Tam. An empirical study on the selection of good metamorphic relations. In *Proceedings of the IEEE 14th International Conference on Engineering of Complex Computer Systems*, pages 23–29, 2009.

[19] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice*, pages 94–100, 2003.

[20] T. Y. Chen, C. Sun, G. Wang, B. Mu, H. Liu, and Z. Wang. A metamorphic relation-based approach to testing web services without oracles. *International Journal of Web Services Research*, 9(9):51–73, Jan 2012.

[21] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[22] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, Aug 2002.

[23] C. Collbergand, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, 1997.

[24] International Obfuscated C Code Contest. `http://www.ioccc.org`

[25] Cutils. `http://manpages.ubuntu.com/manpages/precise/man1/cobfusc.1.html`

[26] J. Ding, T. Wu, J. Q. Lu, and X. H. Hu. Self-checked metamorphic testing of an image processing program. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 190–197, June 2010.

[27] J. H. Ding, D. M. Zhang, and X. H. Hu. An application of metamorphic testing for testing scientific software. In *Proceedings of the IEEE/ACM 1st International Workshop on Metamorphic Testing (MET'16), in conjunction with the 38th International Conference on Software Engineering (ICSE)*, pages 37–43. ACM Press, 2016.

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

[29] G. Dong, T. Guo, and P. Zhang. Security assurance with program path analysis and metamorphic testing. In *Proceedings of the IEEE 4th International Conference on Software Engineering and Service Science (ICSESS)*, pages 193–197, May 2013.

[30] G. Dong, C. Nie, B. Xu, and L. Wang. An effective iterative metamorphic testing algorithm based on program path analysis. In *Proceedings of the 7th International Conference on Quality Software (QSIC 2007)*, pages 292–297, Oct 2007.

[31] S. Drape. Obfuscation of abstract data-types, PhD thesis, St Johns College, University of Oxford, 2004.

[32] S. Drape. Intellectual property protection using obfuscation, Technical Report CS-RR-10-02, Oxford University Computing Laboratory, 2010.

[33] P. Falcarin, S. Di Carlo, A. Cabutto, N. Garazzino, and D. Barberis. Exploiting code mobility for dynamic binary obfuscation. In *Proceedings of World Congress on Internet Security (WorldCIS)*, pages 114–120, Feb 2011.

[34] Gzip. `http://www.gzip.org`

[35] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, Apr 2015.

[36] M. Hataba and A. El-Mahdy. Cloud protection by obfuscation: Techniques and metrics. In *Proceedings of the 7th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 369–372, Nov 2012.

[37] H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, pages 141–150, Nov 2010.

[38] S. Hosseinzadeh, S. Hyrynsalmi, M. Conti, and V. Leppnen. Security and privacy in cloud computing via obfuscation and diversification: A survey. In *Proceedings of the IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 529–535, Nov 2015.

[39] P. Hu, Z. Zhang, W. K. Chan, and T. H .Tse. An empirical comparison between direct and indirect test result checking approaches. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, pages 6–13. ACM Press, 2006.

[40] Z. W. Hui, S. Huang, H. Li, J. H. Liu, and L. P. Rao. Measurable metrics for qualitative guidelines of metamorphic relation. In *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, volume 3, pages 417–422, 2015.

[41] Jad. `http://www.kpdus.com/jad.html`

[42] F.-C. Kuo, Z. Q. Zhou, J. Ma, and G. Zhang. Metamorphic testing of decision support systems: a case study. *IET Software*, 4(4):294–301, 2010.

[43] V. Le, M. Afshari, and Z. D. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226. ACM Press, 2014.

[44] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, Jan 2014.

[45] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software (QSIC)*, pages 59–68, 2012.

[46] A. Majumdar, C. Thomborson, and S. Drape. A survey of control-flow obfuscations. In *Proceedings of the 2nd International Conference on Information Systems Security*, pages 353–356. Springer, 2006.

[47] L. I. Manolache and D. G. Kourie. Software testing using model programs. *Softw. Pract. Exper.*, 31(13):1211–1236, Oct 2001.

[48] M. Mateasand and N. Montfort. A box, darkly: Obfuscation, weird languages, and code aesthetics. In *Proceedings of the 6th Digital Arts and Culture Conference*, pages 144–153. IT University of Copenhagen, 2005.

[49] J. Mayer and R. Guderlei. On random testing of image processing applications. In *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, pages 85–92, 2006.

[50] T. M. Meyers and D. Binkley. Slice-based cohesion metrics and software intervention. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 256–265, Nov 2004.

[51] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering*, pages 867–872, 2008.

[52] N. A. Naeem, M. Batchelder, and L. Hendren. Metrics for measuring the effectiveness of decompilers and obfuscators. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 253–258, 2007.

[53] Obfuscator-LLVM. `https://github.com/obfuscator-llvm/obfuscator/wiki`

[54] D. Rai and K. Tyagi. Estimating the regression test case selection probability using fuzzy rules. In *Proceedings of International Conference on Recent Trends in Information Technology (ICRTIT)*, pages 603–611, July 2013.

[55] G. Ralph and M. Johannes. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(06):757–781, 2007.

[56] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortes. A survey on metamorphic testing. *IEEE Transactions on Software Engineerin*, 42(9):805–824, 2016.

[57] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245 – 258, 2011.

[58] D. P. Shrivastava and R. C. Jain. Unit test case design metrics in test driven development. In *Proceedings of International Conference on Communications, Computing and Control Applications (CCCA)*, pages 1–6, Mar 2011.

[59] Stunnix. `http://stunnix.com`

[60] Q. Su, Z. Y. Wang, W. M. Wu, J. L. Li, and Z. W. Huang. Technique of source code obfuscation based on data flow and control flow tansformations. In *Proceedings of the 7th International Conference on Computer Science Education (ICCSE)*, pages 1093–1097, July 2012.

[61] C. Sun, Y. Liu, Z. Wang, and W. K. Chan. μMT: A data mutation directed metamorphic relation acquisition methodology. In *Proceedings of the IEEE/ACM 1st International Workshop on Metamorphic Testing (MET'16), in conjunction with the 38th International Conference on Software Engineering (ICSE)*, pages 12–18. ACM Press, 2016.

[62] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen. Metamorphic testing for web services: Framework and a case study. In *Proceedings of IEEE International Conference on Web Services (ICWS)*, pages 283–290, July 2011.

[63] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 17th Asia Pacic Software Engineering Conference*, pages 270–279. 2010.

[64] Tigress. `http://tigress.cs.arizona.edu`

[65] M. Velez. Finding and understanding bugs in obfuscators, 2013. `https://bitbucket.org/martinvelez/obfuscator_bugs_paper/downloads`

[66] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[67] H. Wu. An effective equivalence partitioning method to design the test case of the web application. In *Proceedings of International Conference on Systems and Informatics (ICSAI)*, pages 2524–2527, May 2012.

[68] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, pages 283–294, 2011.

[69] Z. Q. Zhou. Using coverage information to guide test case selection in adaptive random testing. In *Proceedings of the IEEE 34th Annual Computer Software*

and Applications Conference (COMPSAC'10), 7th International Workshop on Software Cybernetics (IWSC'10), pages 208–213, IEEE Computer Society Press, 2010.

[70] Z. Q. Zhou, S. Xiang, and T. Y. Chen. Metamorphic testing for software quality assessment: A study of search engines. *IEEE Transactions on Software Engineering*, 42(3):264–284, 2016.

[71] Z. Q. Zhou, S. Zhang, H. Markus, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243, 2012.