Undefined Behaviour in Mutation Testing

by

Mehrnoosh Ebrahimipour

B.Sc., Sharif University of Technology, 2014

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

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2016

Approval

Name:	**Mehrnoosh Ebrahimipour**
Degree:	**Master of Science**
Title:	***Undefined Behaviour in Mutation Testing***
Examining Committee:	**Chair:** Ryan Shea University Research Associate

William (Nick) Sumner
Senior Supervisor
Assistant Professor _____

Arrvindh Shriraman
Supervisor
Assistant Professor _____

Anoop Sarkar
Internal Examiner
Professor
School of Computing Science _____

Date Defended: 13 September 2016

Abstract

Mutation testing is used to evaluate the quality of a test suite by measuring how well the test suite detects systematically seeded faults (mutants). However, in the C programming language, the created mutants may introduce undefined behaviour. Such a mutant has no meaning and thus cannot meaningfully be reported as either detected or undetected by mutation testing. This introduces two problems. First, it increases the number of mutants that must be considered for mutation testing. Second, it creates a potential for bias in the mutation score when mutants with undefined behavior count toward the number of detected or undetected mutants. This thesis makes contributions toward identifying the ways in which traditional mutation testing mechanisms may lead to undefined behavior. It furthermore introduces automated analyses for statically detecting mutants that cause undefined behavior so that they can be filtered out ahead of time. A proof of concept implementation using Clang and LLVM validates that these techniques work for real world programs in the C programming language.

Keywords: Undefined Behaviour; Mutation Testing; Program Analysis

Dedication

To my loving mother, Mina.

Acknowledgements

I would like to thank my senior supervisor, Dr. Nick Sumner. His knowledge, support and insightful guidance has been of great value for me.

I would also like to thank Dr. Arrvindh Shriraman, Dr. Anoop Sarkar, and Dr. Ryan Shea for serving as my thesis committee.

Finally, I would like to thank my mother, Mina Hamidirad, for all her support and love throughout my life and a special thank you to my partner, Zephyr Corr, for all the encouragement and love.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Software applications potentially touch millions of people, enabling them to do their jobs effectively and efficiently [14]. Software testing is a process that is designed to make sure software applications do what they are supposed to do. In this process, the software under test is executed on a set of test cases. Then the results are evaluated to see if they match the expected results. If the results are not as expected, the software is faulty. Otherwise, better tests need to be developed to reveal errors. Adequate program testing is a necessary part of software development. One of the challenges in testing is to measure the adequacy (quality) of the test suite. There are several criteria that may be used to specify adequacy [16], and evidence suggests mutation testing to be an effective one [4].

Mutation testing evaluates the quality of a test suite by measuring how well it detects seeded faults. This is done by inserting numerous small defects into a program, one at a time. Each new seeded program is called a *mutant*. The quality of a test suite is measured by the number of seeded faults that the test suite can detect. A fault is detected, and the corresponding mutant *killed*, when the test suite fails on a test for the mutant but passes on that test for the original program. Otherwise, the mutant is live, and the seeded fault could not be detected. This establishes a base for calculating a *mutation score*, the ratio of killed mutants to seeded mutants. The higher the mutation score, the better the quality of the test suite. Mutation testing is based on two hypotheses: the Competent Programmer Hypothesis (CPH) and Coupling Effect [8]. The CPH states that programmers are competent, meaning they tend to develop programs close to the correct version and as a result, even though there might be faults in the developed program, they are simple faults that can be corrected by a few syntactic changes [8]. The coupling Effect, states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [8]. In the context of mutation testing, it means that complex faults are coupled with small faults, and a test suite that can detect the small ones can also detect the complex ones.

The main advantage of mutation testing is that the mutation operators can be described precisely and therefore provide a well-defined, fault-seeding process [4]. However, applying these operators to programs may create mutants with undefined behaviour. These mutants have no meaning defined by a language specification, and reporting them as either killed or alive by a mutation testing framework is not meaningful. Mutants with undefined behaviour (invalid mutants) have the potential to affect the mutation score of programs and also increase the number of mutants that need to be tested for computing it. In this work, we have studied the presence of invalid mutants for programs in the C programming language. We have proposed automated analysis techniques to detect such mutants. By measuring the number of invalid mutants, our results show that undefined behaviour in mutation testing does exist in real world C programs and has the potential to affect their mutation scores.

Chapter 2 provides the background information for this work. We have identified cases where mutation operators result in undefined behaviour and discussed them in chapter 3. The proposed automated analysis techniques to detect such cases automatically is described in chapter 4. In addition, an implementation of the proposed analysis shows that it works in real world projects as discussed in chapter 5. Finally, conclusions and future work are presented in chapter 6.

Chapter 2

Background

Software is an essential component in much of the infrastructure that exists in our society today such as airplanes, cell phones, cars, and more. Software testing is an important cycle of software engineering and quality assurance of any software. Testing observes the execution of a software system in order to validate whether it behaves as expected and to identify potential bugs in the system [6]. It is used in industry for quality assurance of software products. As software applications become more complex and critical, ensuring their desired quality becomes more crucial, difficult and expensive. Studies estimate that testing can comprise even more than half of development costs [6]. One of the challenges in testing is how to qualify and evaluate the effectiveness of testing criteria. The attributes of a good test are [2]:

(a) Having a high probability of finding bugs

(b) Not being redundant

(c) Should be neither simple nor complex

Mutation analysis is a test adequacy criterion for evaluating test suites. Mutation testing has been shown to be a potentially valuable testing technique, capable of simulating real bugs [17]. Mutants are versions of a program that differ from the original version by a small syntactic change. Each of these mutants is run against the test suite. If the output differs from the original program's results, then it means that the test suite is able to identify the bug and has killed the mutant. Otherwise the mutant stays alive. The higher the ratio of killed mutants to all mutants, the better the quality of the test suite.

The remainder of this chapter explains mutation testing frameworks and then provides background information for undefined behaviour.

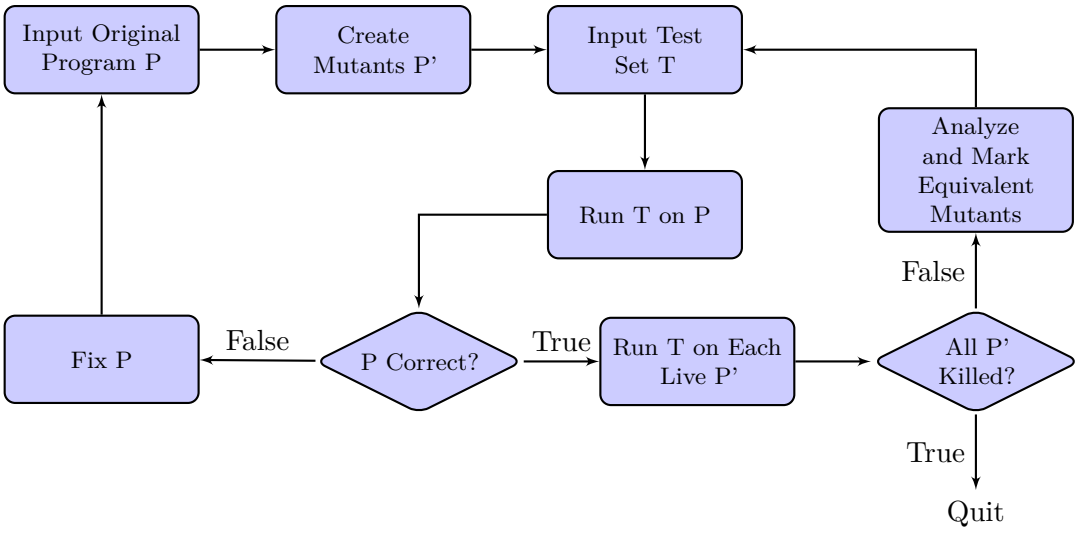

Figure 2.1: Process of Mutation Testing [8]

Original Program P	Mutant Program P'
`1 if (a && b)`	`1 if (a || b)`
`2 ...`	`2 ...`

Table 2.1: Example of a Simple Mutation

2.1 Mutation Testing

The process of mutation testing is shown in Figure 2.1. For a program P and a set of test cases T, first a set of mutants are generated. Each mutant is generated by a single syntactic change to the original program P. This transformation is known as mutation and a group of related syntactic changes (e.g., replace one arithmetic operator with another) is called a *mutation operator*. For example, Table 2.1 shows an original program and its mutant. By changing the `&&` operator that is used in an `if` statement to the `||` operator, a new program P' is generated that is a mutant of the original program. By applying an operator only once, a first order mutant is generated. Table 2.2 shows a set of mutation operators in C-like languages. Mutation operators can be categorized into replacement operators, insertion operators, and deletion operators. The first category, replacement operators, are designed to modify and replace a variable or an expression in a program. Insertion and deletion operators are used to add and delete statements in a program respectively [8].

A set of mutants is generated by applying mutations to the original program (one at a time). This set of mutants is shown as P' in Figure 2.1. In the next step, all the test cases in set T are executed against the original program P. If any test cases fail, the bugs indicated by those tests need to be fixed. After all the test cases pass for program P, the test suite is

then executed against every mutant in set P'. If all the test cases pass when executed for a mutant, that means the bug in that mutant could not be detected by T, and therefore the mutant has survived. Otherwise, in the case of any failing test cases, the mutant is killed by the test suite. At the end, when all mutants P' have been executed against T, some of them may have survived. This could happen for two reasons. (1) Either the test suite was not strong enough to detect the bug, or (2) the mutant was functionally equivalent to the original program and therefore should have passed all the test cases. In the first case, the programmer can add new test cases to T in order to increase the quality of the test suite and kill the surviving mutants. However, in the second case mutants are functionally equivalent with the original program and can never be killed. An example of syntactically different but functionally equivalent mutants is illustrated in Figures 2.2 and 2.3. Both programs will have the same output regardless of the test suite used because their meaning is the same. Automatically detecting these mutants is impossible because in general this problem is proved to be undecidable [8]. However, many solutions have been proposed in the literature for equivalent detection problem [8].

```
1 counter = 0
2 while(flag){
3    if (counter > 10){
4       break;
5    }
6 }
7 print(counter);
```

Figure 2.2: Original Program P

```
1 counter = 0
2 while(flag){
3    if (counter >= 10){ // Mutation: >=
4       break;
5    }
6 }
7 print(counter);
```

Figure 2.3: Equivalent Mutant P'

Finally, at the end of the process, a *mutation score* is reported. This score shows the quality of the test suite T when used for testing program P. Equation 2.1 shows that the mutation score is computed by the number of mutants killed by test suite T divided by the total number of non-equivalent mutants for program P [8].

$$Mutation\ Score(P,T) = \frac{Number\ of\ killed\ mutants}{Total\ number\ of\ non\text{-}equivalent\ mutants} \tag{2.1}$$

Mutation testing has been shown to effectively measure the quality of a test suite. However, it is considered to be expensive, and this is for two main reasons [17]. First, the number of mutants for a program can be very large, and this results in high computational costs for executing all the test cases against the enormous number of mutants. This problem has been addressed in the literature by applying various methods such as mutant sampling, mutant clustering, selective mutation, runtime optimizations and other techniques [8]. The second reason that makes mutation testing expensive is the equivalent mutant detection problem. As mentioned earlier, the problem of detecting whether a program and a mutant of that program are functionally equivalent or not is undecidable. Since the number of non-equivalent mutants is used for computing the mutation score, it is important to address this problem in order to achieve accurate results. In other words, if the equivalent mutants are not detected, the mutation score may be arbitrarily lowered by mutants that do not reflect faulty behavior. Because of the undecidable nature of this problem, one can never find a complete solution; however, finding fast and effective ways to detect equivalent mutants can help with overcoming the practical aspects of the problem [17].

One solution for detecting equivalent mutants is using compiler optimizations. This idea was first suggested by Baldwin and Sayward [17]. The intuition for this approach is the fact that optimization rules form transformations on equivalent programs. Therefore, if program P can be transformed to its mutant using optimization rules, then the two programs are functionally equivalent and have the same meaning. In paper Trivial Compiler Equivalence (TCE) [17], authors have used this intuition to build a tool to automate the process of equivalent mutant detection. TCE simply identifies two programs as equivalent if their compiled object codes are identical. Their experimental results show that TCE can detect approximately 30% of all the existing equivalent mutants [17].

2.1.1 Mutation Operator Types

Applying mutation operators to a program creates mutants of the original program. For each mutation operator, if the program contains several entities that are in the domain of that operator, then the operator is applied to each entity, one at a time. For example, consider the mutation operator that deletes a statement from the program. All the statements in the program are in the domain of this operator and therefore it will be applied to every statement. Each mutant will have all statements in the program except the one deleted by the mutant operator. These mutants are considered to be fault induced versions of the original program. Table 2.2 shows a detailed description of these selected mutation operators.

1. **Absolute Value Insertion**
   ABS mutates statements containing scalar references. It provides domain coverage for scalar variables. The domain is partitioned into three sections: negative, positive

Table 2.2: Mutation Operator Types [17]

Name	Description
ABS: *Absolute Value Insertion*	$\{(e, \texttt{abs}(e)), (e, \texttt{-abs}\ (e))\}$
AOR: *Arithmetic Operator Replacement*	$\{(x,y) \mid x,y \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\} \wedge x \neq y\}$
LCR: *Logical Connector Replacement*	$\{(x,y) \mid x,y \in \{\texttt{\&\&},\texttt{||}\} \wedge x \neq y\}$
ROR: *Relational Operator Replacement*	$\{(x,y) \mid x,y \in \{\texttt{>},\texttt{>=},\texttt{<},\texttt{<=},\texttt{==},\texttt{!=}\} \wedge x \neq y\}$
UOI: *Unary Operator Insertion*	$\{(v, \texttt{--}\ v), (v, v\texttt{--}), (v, \texttt{++}\ v), (v, v\texttt{++})\}$
CRCR: *Integer Constant Replacement*	$\{(c_i, x) \mid x \in \{\texttt{1}, \texttt{-1}, \texttt{0}, c_i\texttt{+1}, c_i\texttt{-1}, \texttt{-}\ c_i\}\}$
OAAA: *Arithmetic Assignment Mutation*	$\{(x,y) \mid x,y \in \{\texttt{+=},\texttt{-=},\texttt{*=},\texttt{/=},\texttt{\%=}\} \wedge x \neq y\}$
OBBN: *Bitwise Operator Mutation*	$\{(x,y) \mid x,y \in \{\texttt{\&},\texttt{|}\} \wedge x \neq y\}$
OCNG: *Logical Context Negation*	$\{(e, \texttt{!}(e)) \mid e \in \{\texttt{if(e)}, \texttt{while(e)}\}\}$
SSDL: *Statement Deletion*	$\{(s, remove(s)\}$

and zero [1]. ABS is designed to make sure each scalar variable will have a negative value, a positive value, and value zero [3]. For example `x = 5 + (*y)++` is mutated to the following statements:

```
x = 5 + abs(*y)++
```
(2.2)

```
x = 5 + -abs(*y)++
```
(2.3)

Instead, if we had `x = 5 + *(y++)` the mutated statements would be the following[1]:

```
x = 5 + abs(*(y++))
```
(2.4)

```
x = 5 + -abs(*(y++))
```
(2.5)

2. **Arithmetic Operator Replacement**
   Let S be equal to set {`+`,`-` ,`*`,`/`,`%`}. AOR replaces each occurrence of elements in set S by the other operators in the set [3]. For example, `c = a + b` is mutated to create the following:

```
c = a - b
```
(2.6)

```
c = a * b
```
(2.7)

```
c = a/ b
```
(2.8)

```
c = a % b
```
(2.9)

3. **Logical Connector Replacementt**
   This mutation operator simply replaces `&&` with `||` and the other way around. An example of this mutation was illustrated earlier in Table 2.1.

4. **Relational Operator Replacement**
   ROR replaces each occurrence of one of the relational operators that belong to set { `>`,`>=`,`<`,`<=`,`==`,`!=`} with other elements in the set [3]. For example, applying ROR to `if(a>b)` produces the following mutants:

```
if(a>=b)        (2.10)
if(a<b)         (2.11)
if(a<=b)        (2.12)
if(a==b)        (2.13)
if(a!=b)        (2.14)
```

5. **Unary Operator Insertion**
   UOI replaces each reference v by one of the elements in {`--` v, v`--` , v`++`,`++` v}. Values of variables can be off the desired value by `+1` or `-1` . This operator is designed to model such errors and is useful for checking boundary conditions [1]. Consider statement `c = a + b`. Applying UOI to variable `b` in this statement creates the following mutants:

```
c = a + (b++)   (2.15)
c = a + (++b)   (2.16)
c = a + (b--)   (2.17)
c = a + (--b)   (2.18)
```

6. **Integer Constant Replacement**
   Let S denote the set {`1`,`-1` , `0`, c_i`+1` , c_i`-1` ,`-` c_i}, where c_i is a constant in the program. CRCR replaces each constant with one of the elements of set S. This mutation operator aims to ensure that one constant is not mistakenly used for another [1]. Applying CRCR to `n |= (n >> 4)` will result in following mutants:

```
n |= (n >> 1)   (2.19)
n |= (n >> -1)  (2.20)
n |= (n >> 0)   (2.21)
```

```
n |= (n >> 5)
```
(2.22)

```
n |= (n >> 3)
```
(2.23)

```
n |= (n >> -4)
```
(2.24)

Note that the number of bits to shift in Equations 2.20 and 2.24 are negative and therefore these statements are not meaningful in C. We shall discuss this more in the next chapter.

7. **Arithmetic Assignment Mutation**
   OAAA replaces any occurrence of elements of set { `+=` ,`-=` ,`*=` ,`/=`,`%=`} with the rest of its elements.

8. **Bitwise Operator Mutation**
   OBBN is for mutating bitwise operators. It replaces `&` with `|` and vice versa.

9. **Logical Context Negation**
   In conditional statements, often the condition is reversed. OCNG aims to model such types of errors [1]. This operator negates the controlling condition inside conditional statements such as `if` and `while`. It also replaces the condition with constant `true` and `false` boolean values. For instance, `if(expr)` will be mutated to:

```
if(!expr)
```
(2.25)

```
if(true)
```
(2.26)

```
if(false)
```
(2.27)

   This operand may cause infinite loops.

10. **Statement Deletion**
    SSDL removes statements from the program. It is designed to ensure that each statement in the program is executed and has an effect on the output [1].

Based on previous research on mutation operator selection [17], we have used the union of two sets of operators. The first set includes ABS, AOR, LCR, ROR, and UOI. This set has been used in the literature extensively. The second set consists of CRCR, OAAA, OBBN, OCNG and SSDL. This set has been shown to provide accurate predictions of the real fault detection ability of the test suite [17].

2.2 Undefined Behaviour

According to the C Standards, undefined behaviour is "behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements" [7]. It further explains that even though a segment of source code may be an erroneous program construct, a translator is not required to issue a diagnostic message for it. The erroneous data can happen either during compilation or execution. For instance, division by zero may occur by dividing an expression by a constant zero (during compilation) or by a division operator whose right operand is calculated to zero at run time (during execution).

When *executing* code that contains undefined behaviour, one of the following may happen in practice:

(a) A signal is raised. For instance `SIGFPE` on divide by zero.

(b) The defined behaviour of the processor occurs. For example, two's complement modulo rules for signed integer overflow.

(c) The machine code generated as a result of a translation time decision is executed. For instance, `x = x++` may be translated to increment first and then assign or maybe translated to assign first and then increment or any other combination of instructions.

The C Standard summarizes the above cases: "Possible undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with issuance of a diagnostic message)." [7].

Some programming languages define many constructs as undefined behaviour, whereas others have less undefined behaviour. The trade-off is that on one hand, having more undefined behaviour enables the compiler to generate efficient code, and on the other hand it might generate code that may behave differently on various hardware platforms.

For instance, undefined behaviour exists in C-based programming languages because it simplifies the job of the compiler. According to the language design principles, a C program should be "made fast, even if it is not guaranteed to be portable" [7]. This helps the compiler to be able to generate efficient low-level code, since it is not required to add either complex static checks or dynamic checks that may slow down compilation and execution time [7]. In contrast, programming languages like Java that are considered to be safe have less undefined behaviour and the cost for that is sacrificing performance [9].

One way that compilers exploit undefined behaviour is by assuming that programmers never invoke undefined behaviour [**?**]. Under this assumption, compilers make optimizations that can lead to surprising results for developers. More precisely, a program may work as

expected when compiled with no optimization but the same code with a higher level of optimization may fail to execute. The reason behind this is that the compiler may remove some parts of the code based on undefined behaviour because it can infer that it is dead code.

2.2.1 Categories of Undefined Behaviour

This section introduces categories of undefined behaviour and explains how the compiled versions of these programs might not behave as the developers may expect.

1. **Division by Zero**

```
1 if(size == 0){
2     size = 1 / size ; // Provoke a signal
3 }
```

Figure 2.4: Division by Zero Example [18]

Division by zero is undefined behaviour and the compiler can assume it never happens. In example 2.4 the programmer intends to provoke a signal in case the variable `size` is zero. When compiling this code with Clang, since `size` is used as a divisor in line 2, the compiler can infer that `size != 0`. Therefore the condition in line 1 is always evaluated to `false` and is considered to be dead code. The compiler will optimize away the code because it can never happen [18].

2. **Logical Shift Overflow**

```
1 groups_per_flex = 1 << sbi->s_log_groups_per_flex;
2 /* There are some situations, after shift the
3 value of groups_per_flex can become zero and
4 division with 0 will result in fixpoint
5 divide exception */
6 if (groups_per_flex == 0)
7     return 1;
8 flex_group_count = ... / groups_per_flex;
```

Figure 2.5: Oversized Shift Example [18]

Shifting an n-bit integer by n or more bits is undefined behaviour. For instance left-shifting a 32-bit one by 32 bits produces one on x86, but zero on ARM and PowerPC; however, left-shifting a 32-bit one by 64 bits produces zero on ARM, but one on x86 and PowerPC [18]. Because of these hardware differences, the behaviour is undefined by C. Therefore, the compiler can assume that size of shifting an n-bit integer is at most $n - 1$ bits. Based on this assumption, result of shifting `one` is always non-zero regardless of the amount of shifting.

Example 2.5 is a fragment of code from the file system in Linux. The programmer intends to prevent a security vulnerability from happening by guarding a division by zero in line 8; however, the guarded variable, `groups_per_flex`, is the result of shifting constant value `one`. As discussed earlier, the compiler makes the assumption that this value is never zero and therefore removes the guard as dead code.

3. **Out of Bounds Pointer**

```
1  int vsnprintf(char *buf, size_t size, ...){
2     char *end;
3     /* Reject out-of-range values early.
4     Large positive sizes are used for
5     unknown buffer sizes. */
6     if (WARN_ON_ONCE((int) size < 0))
7        return 0;
8     end = buf + size;
9     /* Make sure end is always >= buf */
10    if (end < buf) { ... }
11    ...
12 }
```

Figure 2.6: Out of Bounds Pointer Example [18]

If an integer is added to or subtracted from a pointer, the result should point to an object with the same type of the original pointer or just one past the end of that object; otherwise, it is undefined behaviour in C [18]. However, some programmers wrongly assume that pointer arithmetic wraps around and use this assumption to check for overflows. Example 2.6 illustrates one such bug. In line 10 the programmer is checking for overflow in case of `size` being large, but the compiler can optimize away the check since it is undefined behaviour and should not happen. More precisely, `buf+size<size` will be translated to `size<0` and since `size` is not negative (according to check in line 6), the guard is dead code.

4. **Null Pointer Dereference**

```
1  unsigned int
2  tun_chr_poll(struct file *file, poll_table * wait){
3     struct tun_file *tfile = file->private_data;
4     struct tun_struct *tun = __tun_get(tfile);
5     struct sock *sk = tun->sk;
6     if (!tun)
7        return POLLERR;
8     ...
9  }
```

Figure 2.7: Null Pointer Dereference Example [18]

Dereferencing a null pointer is undefined behaviour in C [18]. Therefore, whenever a pointer is dereferenced in the code, the compiler can assume that the pointer is not null. This can result in a bug if programmers check for a pointer being null after they have already dereferenced it. For example in the code fragment of Figure 2.7, the programmer checks for `tun` being not null at line 6, however, this pointer has already been dereferenced in line 5. The compiler removes the check as dead code.

5. **Uninitialized Read**

```
1 struct timeval tv;
2 unsigned long junk; /* XXX left uninitialized
3                       on purpose */
4 gettimeofday(&tv, NULL);
5 srandom((getpid() << 16)
6         ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Figure 2.8: Uninitialized Read Example [18]

Using an uninitialized variable in C is undefined behaviour [18]. In the C programming language, variables are neither initialized to zero by default nor contain random values. Compilers can eliminate computation derived from uninitialized variables [9]. In example 2.8, the programmer aims to use the variable `junk` as a random value in computation of `srandom`. But since it is uninitialized the compiler can ignore the entire expression used in `srandom`'s argument.

6. **Type-Punned Pointer Dereference**

In C programming languages, programmers are allowed to cast pointers of one type to another. However, pointer casts are often mistakenly used to reinterpret a given object with a different type. In this case, programmers expect that two pointers of different types, point to the same location in memory (known as aliasing). However the C standard has strict rules for aliasing and two pointers of different types do not alias and have their own memory address. Therefore, misusing such behaviour leads to undefined behaviour [18].

In example 2.9, first `iwe->len` is updated and then `memcpy` is called in line 14 to copy contents of `(char*)iwe` to `stream`. The Linux kernel contains its own version of `memcpy` and the code is expanded as follows:

This code first writes the new `len` value to `iwe->len` and then reads `iwe`, which points to the same memory location of `iwe->len`, using type `int`. GCC assumes that the read and write do not happen in the same memory location because they are using different pointer types and therefore point to different memory locations. As a result, GCC can reorder the operations and copies an old version of `iwe->len` [18].

```
struct iw_event {
  uint16_t len; /* Real length of this stuff */
  ...
};
static inline char * iwe_stream_add_event(
  char * stream, /* Stream of events */
  char * ends, /* End of stream */
  struct iw_event *iwe, /* Payload */
  int event_len ) /* Size of payload */
{
  /* Check if it is possible */
  if (likely((stream + event_len) < ends)) {
    iwe->len = event_len;
    memcpy(stream, (char *) iwe, event_len);
    stream += event_len;
  }
  return stream;
}
```

Figure 2.9: Type-Punned Pointer Dereference Example [18]

```
iwe->len = 8;
*(int *)stream = *(int *)((char *)iwe);
*((int *)stream + 1) = *((int *)((char *)iwe) + 1);
```

Figure 2.10: `memcopy` expansion in Linux [18]

7. Signed Integer Overflow

```
int do_fallocate(..., loff_t offset, loff_t len){
   struct inode *inode = ...;
   if (offset < 0 || len <= 0)
      return -EINVAL;
   /* Check for wrap through zero too */
   if ((offset + len > inode->i_sb->s_maxbytes)
   || (offset + len < 0))
      return -EFBIG;
   ...
}
```

Figure 2.11: Signed Integer Overflow Example [18]

One may assume that signed integer operations wrap around on overflow using two's complement. However, if an arithmetic operation overflows on an integer, it is undefined behaviour in C. As a result of this, compilers can make many optimizations. For example, `x+1 > x` is always evaluated to `true` because `x+1` cannot overflow. In the example 2.11, the programmer intends to check whether `offset + len` exceeds a

number but then realizes that the sum may overflow and bypass the guard. In an attempt to prevent this, a check is added for the case that the sum overflows. However, since integer overflow is undefined behaviour, the compiler makes the conclusion that the check for overflow always evaluates to false and therefore removes it. Thus, the code remains vulnerable and may not behave as expected if the sum overflows.

Chapter 3

Motivation

In this section, we explore how different mutation operators can lead to undefined behaviour. In such cases, the resulting mutants can have undefined behavior and might unduly impact the mutation score. However, exploring this full range of possible undefined behaviors for even a single version of a program is costly. Exploring such behaviors for each individual mutant is prohibitive. Instead, we examine how each mutation operator might lead to undefined behavior while considering only instructions local to the syntactic change. Table 3.1 presents a summary of the results discussed in this chapter.

Table 3.1: Undefined Behaviour in Mutation Testing

Undefined Behaviour Category	**Mutation Operator Type**
DBZ: *Division By Zero*	**CRCR:** *Integer Constant Replacement* **ROR:** *Relational Operator Replacement*
LSO: *Logical Shift Overflow*	**ABS:** *Absolute Value Insertion* **CRCR:** *Integer Constant Replacement* **ROR:** *Relational Operator Replacement*
OBP: *Out of Bounds Pointer and Array Access*	**UOI:** *Unary Operator Insertion* **ABS:** *Absolute Value Insertion* **CRCR:** *Integer Constant Replacement* **ROR:** *Relational Operator Replacement*
NPD: *Null Pointer Dereference*	**OCNG:** *Logical Context Negation* **ROR:** *Relational Operator Replacement* **LCR:** *Logical Connector Replacement* **SSDL:** *Statement Deletion*
UIR: *Uninitialized Read*	**OCNG:** *Logical Context Negation* **ROR:** *Relational Operator Replacement* **SSDL:** *Statement Deletion*
TPD: *Type-Punned Pointer Dereference*	-
SIO: *Signed Integer Overflow*	-

3.1 Division By Zero (DBZ)

Assume the value of a variable evaluates to zero. If this variable is used as a divisor then it is undefined behaviour (division by zero). There are two cases when this can happen in mutation testing. First, a constant integer may be used as a divisor. Replacing this constant integer with zero results in DBZ. Second, a variable is guarded not to be zero and then used as a divisor in the following block. In this case, flipping the condition of the guard results in DBZ. Example 3.1 illustrates both cases.

```
1 x = y / 2;          // CRCR: x = y / 0
2 x = y % 2;          // CRCR: x = y % 0
3
4 if( y != 0)         // ROR: if( y == 0)
5     x = x/y;
```

Figure 3.1: Division by Zero (CRCR and ROR)

3.2 Logical Shift Overflow (LSO)

Shifting an integer variable by a negative value is undefined behaviour. Any mutation operator that changes the sign of values can result in this behaviour. For example, in the case of a constant c used as a shift value, CRCR makes c negative by replacing it with either `-1` or `-` c. Also, in the case of a non-constant variable, v, the ABS operator changes the sign of v to negative by applying `-abs` (v).

```
1 x = y >> 1;         // CRCR: x = y >> -1
2 x = y >> count      // ABS:  x = y >> -abs(count)
```

Figure 3.2: Logical Shift Overflow (CRCR and ROR)

3.3 Out of Bounds Pointer and Array Access (OBP)

Assume pointer `arr` points to the first element of an array. In this case, mutating `index` in `arr[index]` such that it is replaced with a negative value, results in out of bound array access. This can happen in three cases. First, replacing a constant by a negative value using CRCR, second, applying `-abs()` on a non-constant variable, and third, changing guards that are used for sign checking. Also, in general, if `arr` points to the start of a block of memory, applying UOI may cause OBP.

Example 3.3 is a real world code fragment from `utimens.c` file in gzip. In this example, we can infer that `tt` is a pointer to start of an array of memory and therefore accessing

negative indices of `tt` is undefined behaviour. This means that the constant index in lines 7 cannot be mutated to negative values by CRCR.

```
1 struct timeval *tt = NULL;
2 struct timeval truncated_timeval[2];
3 truncated_timeval[0] = ...
4 truncated_timeval[1] = ...
5 if (abig && adiff == 1 && get_stat_atime_ns (&st) == 0){
6     tt = truncated_timeval;
7     tt[0].tv_usec = 0;     /* CRCR: tt[-1].tv_usec = 0; */
8 }
```

Figure 3.3: Code Fragment from lib/utimens.c of gzip-1.6 Project (CRCR)

Example 3.4 is another code fragment from the gzip project. In this example, `leaves` is a pointer to the starting location (memory address) of an array of memory. As a result, every array access to this memory, cannot be mutated to negative values using `-abs()` .

```
1 local int leaves [MAX_BITLEN+1]; /* Number of leaves for each bit
    length */
2 for (len = 1; len <= max_len; len++) {
3     leaves[len] = read_byte();
4     if (max_leaves - (len == max_len) < leaves[len])
5         gzip_error("too many leaves in Huffman tree");
6     max_leaves = (max_leaves - leaves[len] + 1) * 2 - 1;
7     n += leaves[len];      /* ABS: n += leaves[-abs(len)] */
8 }
```

Figure 3.4: Code Fragment from lib/utimens.c of gzip-1.6 Project (ABS)

In the following example by applying ROR the comparison operator changes from `!=` to `<`. The compiler can then infer that index `i` is negative in the following block of the condition in line 3. Therefore accessing the element `i` of `arr` will be undefined in line 4.

```
1 int arr[3] = {1, 1, 1};
2 for (int i = 0; i< size; i++){
3     if(i != 0){           /* ROR: if( i < 0 ) */
4         arr[i] += 5;
5     }
6     ...
7 }
```

Figure 3.5: Out of Bounds Array Access (ROR)

Finally, in line 1 of the following code, one block of memory is allocated for pointer `pat_y` and therefore this pointer cannot be incremented or decremented in line 2, because it will exceed its allocated memory and will access an invalid address in memory. In line 4, two

blocks of memory are allocated for `pat_x`, so it can be incremented but not decremented in line 5.

```
1 struct a_pattern *pat_y = xmalloc(sizeof(struct a_pattern));
2 pat_y->str = p;      /* UOI: ((pat_y)++)->str = p; */
3
4 struct a_pattern *pat_x = xmalloc(2*sizeof(struct a_pattern));
5 pat_x->str = p;      /* UOI:  ((pat_x)--)->str = p;*/
```

Figure 3.6: Out of Bounds Pointer (OBP)

3.4 Null Pointer Dereference (NPD)

There are two cases where mutation operators cause null pointer dereference. The first case is omitting definition statements. The SSDL mutation operator maintains the syntactic validity of a mutant; however, if the deleted statement is a definition statement, a null-initialized pointer variable may remain null. Any access to it results in a null pointer dereference. Second, the mutation operators OCNG, ROR, LCR, and UOI can change null pointer guards. More precisely, assume a conditional statement is guarding a pointer to not be null and then the guarded code dereferences the same pointer in the following block. Mutation operators that negate conditions of such guards, result in NPD.

In example 3.7, applying SSDL to statement in line 6, causes `tt` to remain null. As a result, accessing this variable in line 7 is an undefined behaviour of type NPD.

```
1 struct timeval *tt = NULL;
2 struct timeval truncated_timeval[2];
3 truncated_timeval[0] = ...
4 truncated_timeval[1] = ...
5 if (abig && adiff == 1 && get_stat_atime_ns (&st) == 0){
6    tt = truncated_timeval; /* SSDL deletes this statement */
7    tt[0].tv_usec = 0;
8 }
```

Figure 3.7: Code Fragment from lib/utimens.c of gzip-1.6 Project (SSDL)

The code in example 3.8 simply checks whether struct `ctx != NULL` and then reads field `i` of `ctx`. A mutation operator that changes the condition of the `if` statement to `ctx == NULL`, results in NPD in line 4. It can happen either by applying ROR to statement in line 1 or by applying OCNG to the condition in line 3. If the guarded condition is a pointer as shown in line 8, applying UOI to the pointer will change the condition to be always `true` and causes NPD in line 9.

```
1  bool cond = ctx != NULL; /* ROR: bool cond = ctx == NULL */
2  unsigned int ctx_i = 0;
3  if (cond) {               /* OCNG: if(!(cond))  or if(true) */
4      ctx_i = ctx->i;
5      ...
6  }
7
8  if (ctx) {               /* UOI: if(--ctx) or if(++ctx) */
9      ctx_i = ctx->i;
10     ...
11
12 }
```

Figure 3.8: Null Pointer Dereference (OCNG)

3.5 Uninitialized Read (UIR)

Uninitialized reads happen when the mutation operator modifies the code in a way that a variable is not initialized before it is read from. Two cases can cause this behaviour. The first case is SSDL, it can delete the initialization statement of a variable. The second case is mutation operators that change the flow of the program in a way that variables remain uninitialized. Most of the mutation operators can be counted toward the second case, such as OCNG, ROR, and others.

3.6 Type-Punned Pointer Dereference (TPD)

The mutation operators in Table 2.2 do not change the type of pointers and therefore do not result in this type of undefined behaviour.

3.7 Signed Integer Overflow (SIO)

Mutation operators in table 2.2 may result in SIO, but such cases are not local to the mutation and require specific conditions to hold. For example, if an integer variable contains the maximum integer value possible, incrementing it results in SIO; however, reasoning about the ranges of values in variables is a non-trivial task and is not in the scope of our study.

Chapter 4

Approach

In this chapter, we introduce three approaches for identifying cases where a mutation would lead to undefined behavior.

4.1 Data Flow Analysis (DFA)

Data flow analysis is a framework for gathering facts about programs. DFA algorithms were originally developed in the field of compiler construction, used to detect opportunities for optimization [20]. They also have many other applications in software development to solve problems in verification, debugging, testing and more [12]. Their applications range from selecting test cases based on dependence information to detecting patterns that indicate programming errors such as reads of potentially uninitialized variables or null pointer dereferences [20].

DFA algorithms take a program as input, capture static information and return derived information as a solution [12]. A control flow graph (CFG) is a representation of a program that makes data flow analysis easier. It is a directed graph in which nodes represent statements of a program and edges represent the control flow. A data flow solution assigns a value to each node in the CFG. The assigned values can be a set of facts or relations that can flow to other nodes in the graph. Data flow problems can be forward or backward. In a case of forward analysis, the information flows in the same direction as CFG edges, and in a backward analysis, it flows in the opposite direction [12]. In addition, they can be interprocedural or intraprocedural. An interprocedural analysis takes into account the procedures or functions of a program and their calling relationships, while an intraprocedural analysis only considers one procedure at a time [12].

A formal definition of DFA frameworks is presented in section 4.1.1. Then our approach using DFA to find potential undefined behaviour in mutation testing is described in the following sections.

4.1.1 Data Flow Analysis Framework

Data flow analysis framework is defined as a tuple $\tau = ((D, \sqcap, \sqsubseteq, \top, \bot), T)$, where the first element of τ, defines a semi-lattice and its second element, T, is a set of transfer functions. The semi-lattice consists of a set of data flow values, D, and a *meet* operator. Each element of τ is described as follows [19]:

(a) D**:** A set of data flow values which abstractly represent states of a program. This abstraction is based on characteristics and facts that DFA is searching for.

(b) $\sqcap$**:** A meet operator that operates from a DxD domain to D. It takes two elements of D and maps it to another element in D.

(c) $\sqsubseteq$**:** A partial order relation on domain D ($x \sqsubseteq y$ iff $x \sqcap y = y$).

(d) $\top, \bot$**:** Top and bottom elements of the lattice respectively, which are part of domain D.

(e) T**:** A set of transfer functions that defines for each side effect, the impact on the data flow values.

For instance, assume the data flow problem is to approximate signs of integer variables in a program. In this case, D is $\{+, -, 0\} \cup \{\top, \bot\}$ representing all the possible signs. The meet operator is applied when multiple paths in the program join and is used to approximate the sign of variables at the joint point. For example, if variable `a` is known to be positive in one path and negative in another path, when the two path join, the meet operator is applied and the state of variable `a` will be *unknown* at that point. Transfer functions show how to evaluate the approximated behaviour. For example, in case of `c = a + b`, if the analysis have computed facts about signs of `a` and `b` then it can infer sign of `c` (adding a positive integer to another positive integer results in a positive value and etc.).

Given a program P, and a DFA framework τ, a data flow solution computes a mapping between nodes of the control flow graph of P and abstract states of set D. This map is called the *abstract state* of P. To compute this mapping a fixpoint computation is used, which is based on a *worklist* algorithm. The complexity of this computation is bounded by the height of the semi-lattice in τ. For each node n, the algorithm iteratively computes the abstract state $D(n)$ for that node using the following equations:

$$D_0(n) = \top \qquad \forall n \in \mathrm{CFG}(P) \tag{4.1}$$

$$D_i(n) = \bigsqcap_{n_p \in pred(n)} T_f(n_p, n) \qquad \forall i \geq 0 \tag{4.2}$$

$$D_i(n) = \bigsqcap_{n_p \in succ(n)} T_b(n_p, n) \qquad \forall i \geq 0 \tag{4.3}$$

In the first iteration, all the nodes are initialized to the top element, $\top$, as shown in equation 4.1. Then depending on the flow of the analysis, forward or backward, either equation 4.2 or 4.3 is used. For instance, in the case of forward analysis, for each iteration i of node n, the transfer function uses the current state of n, which is $D_{i-1}(n)$, along with the states of all its predecessors to compute a new state for n, defined as $D_i(n)$ in equation 4.2. The backward analysis is very similar to forward analysis and the difference is that the successors of each node are used for computation instead of its predecessors [19].

4.1.2 Data Flow Analysis Algorithm

Function *computeAbstractState* presented in algorithm 1 shows the process described earlier (for the forward analysis). It takes in a function as input and returns the *abstract state* for that function. The algorithm uses a standard worklist based approach to compute a fixpoint solution. First, a new *abstractState* is created which is a mapping between nodes of the program to their state in the domain D. After all the nodes are added to a list in topological order, at each step, node n is removed from the *worklist*. Then for each of its predecessors, a call to function $T_f(n_p, n)$ is made. This function computes a new incoming abstract state for n in relation to n_p. In the next step, node n is analyzed to collect facts of interest. Finally, if the state of n, $D(n)$, has changed in this iteration, then successors of n are added to the *worklist* using function *addToWorkList*. This algorithm is guaranteed to terminate if the lattice is well-defined and the height of the lattice is finite. After the abstract state is computed for *func*, it can be postprocessed to find facts of interest.

Algorithm 1 computeAbstractState

```
function computeAbstractState(function func)
    add all nodes of func to worklist in topological order
    new abstractState
    while worklist ≠ ∅ do
        Remove node n from worklist
        D(n): current state for n
        for each predecessor n_p of n do
            D'(n) = D'(n) ⊓ T_f(n_p, n)
        end for
        D'(n) = D'(n) ⊓ Transfer(n)
        if D'(n) ≠ D(n) then
            D(n) = D'(n)
            for each successor n_p of n do
                addToWorkList(n_p)
            end for
        end if
    end while
    return abstractState
end function
```

4.2 Null Guard Analysis (NGA)

Null guard analysis is a forward data flow analysis that aims to identify a particular type of null pointer dereference in mutation testing. This type of NPD happens if the mutation operator changes the condition of a guard that is making sure a pointer is not null in its following block. This can be caused by OCNG and ROR mutation operators and was discussed earlier in chapter 3 with examples. The goal of this analysis is to find conditional statements that are guarding pointer variables such that negating them causes NPD. In this case, the analysis abstracts facts about pointers being null or not null at each statement of the program. By analyzing this abstract state, a statement s that dereferences a pointer p while its abstract state contains the fact $p \neq$ null can be identified. Such statements indicate that mutating the program in a way that switches the state of that fact to be null, can potentially generate a meaningless mutant. In example 4.1, pointer `p` is guarded by condition `flag` to not be null in line 4 and then is dereferenced in line 5. Therefore, negating this condition causes undefined behaviour as a result of dereferencing a null pointer.

```
1  int foo(int* p){
2      int a = 0;
3      bool flag = (p != null);      /* NULL check */
4      if (flag){                    /* Guarding pointer p */
5          a = *p;                   /* Dereferencing pointer p */
6      }else{
7          a = boo();
8      }
9      return a;
10 }
```

Figure 4.1: NGA Example

4.2.1 Null Guard Analysis Instantiation

The NGA data flow framework instantiates the simple DFA framework by choosing a particular abstract domain D, transfer functions and meet operator. The following provides their detailed definitions:

(a) **Domain:**
NGA computes a set of facts over a program's pointer variables. The following equations describe the facts that are tracked by the analysis:

$$fact \in \{pointerVar\} \times \{ubInfo\} \tag{4.4}$$

$$ubInfo \in \{guardState\} \times \{actPoint\} \tag{4.5}$$

$$guardState \in \{= \varnothing, \neq \varnothing, \top, \bot\} \tag{4.6}$$

Each *fact* consists of a pointer variable, *pointerVar*, and a corresponding *ubInfo* that contains both a *guard state* asserting the (non)nullness of a pointer and an *activation point* where that assertion is used as a guard. Initially, the *guardState* for all pointers is set to $\top$, denoting that no information about the pointer has been computed. If a pointer is proved to be null or not null, $= \varnothing$ and $\neq \varnothing$ are used respectively. Finally, if contradictory states are observed, *guardState* will be set to $\bot$, denoting that the value is overconstrained and unknown. The activation point is the node that contains the guard for the null comparison. For instance, in example 4.1, the null comparison for pointer `p` is done in line 3 and then is used (activated) in the `if` condition in line 4. Thus, the `if` node is set as the *actPoint* in *ubInfo* for `p`.

(b) **Transfer Functions:**
Transfer(n): If node n is a conditional node with guarding condition *cond* such that *cond* is comparing a pointer variable p to null, then the following *fact* is generated for p:

$$fact =< p\,,\ ubInfo > \tag{4.7}$$

$$ubInfo =< cond\,,\ n > \tag{4.8}$$

$T_f(n_p, n)$: If n_p is not a conditional node, all the facts in n_p flow to n directly. Otherwise, two cases are possible for each fact in n_p. In the first case, the fact is not activated in n_p, meaning n_p is merely propagating those facts. These facts also flow directly to n. In the second case, the fact is activated by the guard condition in n_p, meaning that its *actPoint* is equal to n_p. In this case, if n is the `then` block of n_p, the new facts flow directly to n, but if n is the `else` block of n_p, the negation of these facts flow to n. Fact a is the negation of fact b, if their *guardStates* are the negation of each other ($\neq \varnothing$ and $= \varnothing$).

(c) **Meet Operator:**
In order to model the abstract state of each node, n, in relation to all possible execution paths in the program, states of all the predecessors of n should be merged using the meet operator. The following constraint is used for computing the meet:

$$D_i(n) = \bigsqcap_{n_p \in pred(n)} T_f(n_p, n) \qquad \forall i \geq 0 \tag{4.9}$$

The lattice for NGA is shown in figure 4.2 with the top element as *undefined* and the bottom element as *unknown*.

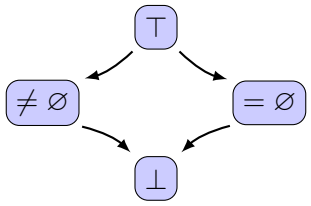

Figure 4.2: Lattice of NGA

4.2.2 Applying Null Guard Analysis

Our approach is a forward intraprocedural analysis. In other words, the analysis propagates a set of abstract state facts forward in the program one function at a time. First, a preprocessing algorithm is performed on the program to find pointer comparisons to null, called *cmpFact*s. Equation 4.10 describes the *cmpFact*s that we track. If node *cmp* is comparing pointer variable p to be equal to null, then *cmpFact*$=< p, = \varnothing >$ is generated, and if *cmp* compares p to be not equal to null, *cmpFact*$=< p, \neq \varnothing >$ is generated. In the next step, the null guard analysis is performed. A *cmpFact* can potentially be activated in

this step and generate a *fact* if used in a guard node. For example, in code fragment 4.1, in the preprocessing step $cmpFact{=}< p, \neq \varnothing >$ is generated in line 3. Since it is activated in a guard in line 4, the corresponding fact $factP{=}< p, ubInfo >$ such that $ubInfo{=}< \neq \varnothing, 4 >$, is generated during the null guard analysis.

$$cmpFact \in \{pointerVar\} \times \{guardState\} \tag{4.10}$$

This process is shown in figure 4.3 for code fragment 4.1 . Initially, the state is empty. As mentioned earlier $factP =< p,\ ubInfo >$ is generated and added to the state in line 4. Node 4 is a conditional node, and the *actPoint* for *factP* is equal to node 4 (it is a newly learned fact), therefore, *factP* flows directly to node 5 (`then` block of node 4) and the negation of it flows to node 7 (thbased on line numberse `else` block of node 4). Line 5 dereferences pointer p and the current state for this line contains a *fact* for p with $ubInfo =< \neq \varnothing, 4 >$. This indicates that a mutation that switches the *guardState* of this fact will cause a null pointer dereference. Next, two contradictory facts from node 5 and 7 flow to node 9. Thus, the *guardState* for *factP* is set to $\perp$ at this node.

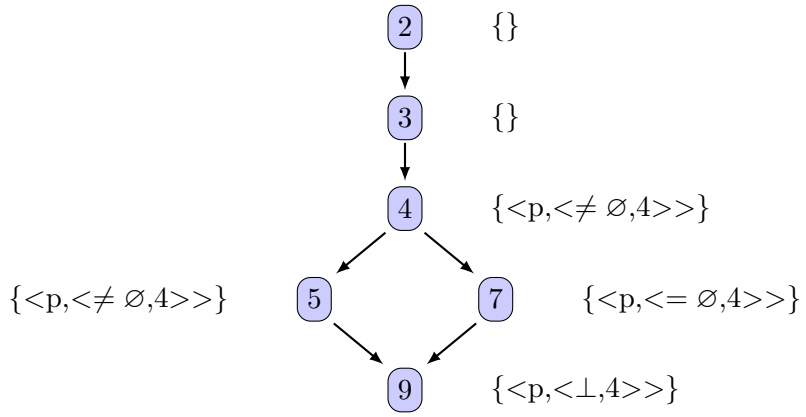

Figure 4.3: Control Flow Graph of code fragment 4.1

4.2.3 Null Guard Analysis Algorithm

Algorithm 2 presents the intraprocedural analysis used for NGA. It takes a program as input and returns as output a set of guard statements that should not be negated in the mutation testing framework. For each function *func* in the program, it first collects a set of *cmpFacts* that hold for each node in CFG(*func*), called $cmpFactMap$. This map is then passed to the *computeAbstractState* function, which was defined earlier in algorithm 1. Functions $Transform(n)$ and $T_f(n_p, n)$ are used in algorithm 1 as transfer functions. These functions are defined for NGA as follows:

(a) $Transfer(n)$

Algorithm 3 defines this transfer function. If node n is a conditional node and its condition has a corresponding *cmpFact* set in *cmpFactMap* then, those *cmpFacts* are activated and new corresponding *facts* are generated and added to the state of n, replacing any existing facts for the pointers to which they pertain. The new *facts* have n as their *actPoint* and their *guardState* matches the one in *cmpFact*.

(b) $T_f(n_p, n)$

This function is defined in algorithm 4. It controls which facts from the predecessor should flow to n. As described earlier, depending on whether n_p is conditional or not and whether a fact is a newly activated fact or an old one, the flow of facts to n can be different.

After the *abstractState* is computed for *func*, *collectPotentialUB* makes a call to *collectGuards*. This function finds all statements that contain a pointer dereference while their *state* has a *fact* indicating the pointer is not null at that point. The *actPoint* of that *fact* is then added to the result set.

Algorithm 2 collectPotentialUB for NGA

```
function COLLECTPOTENTIALUB(program program)
    for each function func in program do
        cmpFactMap = findCmpFacts(func)
        abstractState = computeAbstractState(func, cmpFactMap)
        guardSet = collectGuards(abstractState, func)
    end for
    return guardSet
end function
```

Algorithm 3 Transfer(n) for NGA

```
function TRANSFER(node n, cmpFactMap)
    if n is conditional then
        cmp = condition(n)
        if cmp in cmpFactMap then
            for each pointerVar p in cmpFactSet do
                kill = {< p', u' >∈ D(n)|p' = p}
                gen = {new fact (actPoint = n, pointerVar = p)}
                D(n) = (D(n)\kill)∪gen
            end for
        end if
    end if
    return D(n)
end function
```

Algorithm 4 $T_f(n_p, n)$ for NGA

```
function T_f(node n_p, node n)
    D(n_p): state of n_p , D(n) : state of n
    for each fact in D(n_p) do
        if n_p is not conditional then
            D(n) = D(n)⊓ fact
        else
            if fact is old or (fact is new and n is then block of n_p) then
                D(n) = D(n)⊓ fact
            else if fact is new and n is else block of n_p then
                D(n) = D(n) ⊓ ¬ fact
            end if
        end if
    end for
    return D(n)
end function
```

4.3 Malloc Pointer Analysis (MPA)

Malloc pointer analysis is a forward data flow analysis for identifying another type of invalid pointer. This type can be caused by unary operator insertion mutation type (illustrated earlier in chapter 3). The goal is to find a set of statements that are dereferencing a pointer such that those pointers cannot be incremented or decremented because doing so will exceed

the allocated memory for the pointer. The analysis abstracts facts about pointers' allocated size in order to evaluate the validity of UOI operators. In example 4.4, struct `point` is defined and a pointer to `point p` is passed to function `foo`. Then depending on condition `cond`, one or two blocks of memory is allocated for `p` (each block is of size `point`). In the first case, `p` cannot be incremented nor decremented because only one block is allocated for it. In the second case, `p` can be incremented (point to the second block), but cannot be decremented.

```
1  struct point{
2      int x;
3      int y;
4  };
5
6  void foo(struct point* p, int a, bool cond){
7      if(cond){
8          p = malloc(sizeof(struct point));
9          p->x = a;
10     }else{
11         p = malloc(2*sizeof(struct point));
12         p->x = a;
13     }
14     return;
15 }
```

Figure 4.4: MPA Example

4.3.1 Malloc Pointer Analysis Instantiation

MPA instantiates the simple DFA framework by choosing the following abstract domain D, transfer functions and meet operator:

(a) **Domain:**
MPA computes a set of facts over a program's pointer variables. The following equations describe the facts that are tracked by the analysis:

$$fact \in \{pointerVar\} \times \{allocState\} \tag{4.11}$$

$$allocState \in \{\mathsf{no}\ {+}{+}, \mathsf{no}\ {-}{-}, \mathsf{neither}, \top, \bot\} \tag{4.12}$$

Each *fact* is a pair consisting of a pointer variable *pointerVar* and its allocation state *allocState*. Initially, the *allocState* is set to $\top$, meaning an allocation state has not been defined for this variable yet. If it is proved that *pointerVar* is the start/end point of a chunk of memory, then it cannot be decremented/incremented and its *allocState*

is set to no -- or no ++ respectively. In case *pointerVar* is pointing to only one block of memory as shown in example 4.4 (line 8), then it can be neither incremented nor decremented (neither). Finally, if any contradictory states are observed, then *allocState* is set to $\perp$ (unknown).

(b) **Transfer Functions:**
Transfer(n): If node n allocates memory to pointer variable p by making a call to an allocation function, then the following *fact* is generated:

$$fact =< p\,,\ allocState > \tag{4.13}$$

An allocation function (`malloc` for example), takes in an argument, *size*, that is the size of the memory block to be allocated. If the *size* argument is equal to the size of the type to which p points, then p points to the start of a single valid object in memory, and its *allocState* is set to neither (similar to example 4.4, line 8). Otherwise, p points to the start of the allocated memory block but not the end of it. In this case *allocState* is set to no --.

$T_f(n_p, n)$: All the facts in n_p flow to n.

(c) **Meet Operator:**
The meet operator for MPA is very similar to the one introduced for NGA. It combines the states of all the predecessors of node n to overapproximate the behaviour of the program in n using the following equation:

$$D_i(n) = \bigsqcap_{n_p \in pred(n)} T_f(n_p, n) \qquad \forall i \geq 0 \tag{4.14}$$

Figure 4.5 shows the lattice for MPA. The top element again represents *undefined* state, and the bottom element represents *unknown* state.

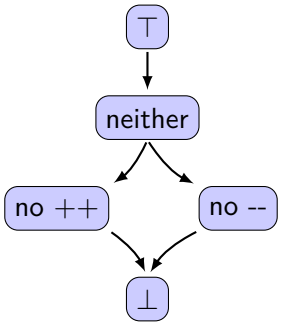

Figure 4.5: Lattice of MPA

4.3.2 Applying Malloc Pointer Analysis

Malloc pointer analysis is a forward intraprocedural analysis. It computes the abstract state for each function in the program separately. Initially, the state is empty and each node in the CFG can potentially transfer the state. The process is shown in figure 4.6 for code fragment 4.4 based on line numbers. In node 7, the state is empty. Node 8, makes a call to `malloc` with a *size* argument equal to the size of type `point` and therefore a new *fact* is generated and added to the state with *allocState* = neither. Node 9 will have the same state as its predecessor (node 8), and since it is dereferencing pointer `p`, a UOI mutation that increments or decrements `p` is undefined behaviour. Node 11, also makes a call to `malloc`, however this time the allocated memory is more than the size of type *point* and thus, a new *fact* is generated with *allocState* = no -- and then flows to node 12. This node is also dereferencing p and this time a UOI mutation that increments p is allowed, but decrementing p is undefined behaviour. Finally, two different *allocState* for p will flow to node 14 and the meet of them (no --) is set as the new *allocState* for this node.

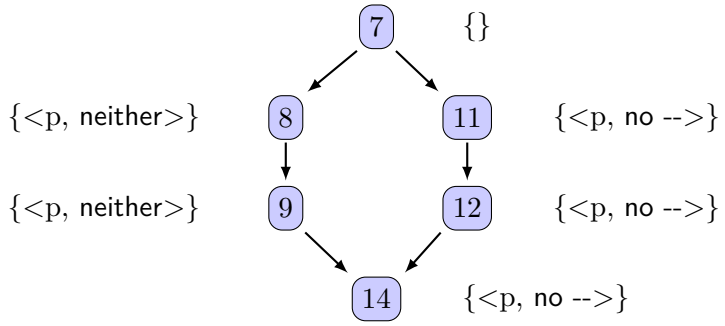

Figure 4.6: Control Flow Graph of code fragment 4.4

4.3.3 Malloc Pointer Analysis Algorithm

Algorithm 5 presents the intraprocedural analysis used for MPA. It takes a program as input and returns as output a set of dereferencing statements such that their pointers should not be mutated with unary operator insertion mutation operator. For each function *func* in the program, it first computes the abstract state for *func* by using algorithm 1. Transfer functions *Transfer(n)* and $T_f(n_p, n)$ used in algorithm 1 are designed as follows for MPA:

(a) *Transfer(n)*

Algorithm 6 defines *Transfer(n)*. Node n adds a new fact to the state only if it is allocating memory to a pointer, p, by making a call to a memory allocation function. The *allocState* for this *fact* depends on the size passed to the allocation function as described earlier.

(b) $T_f(n_p, n)$

Algorithm 6 defines this transfer function. Each *fact* in the state of n_p (a predecessor of n) flows to the state of n.

After the abstract state is computed for *func*, *collectPotentialUB* makes a call to *collectDerefs*. This function finds all statements that contain a pointer dereference while their *state* has a *fact* indicating the pointer cannot be incremented and or decremented. This statement is then added to the result set.

Algorithm 5 collectPotentialUB for MPA

```
function COLLECTPOTENTIALUB(program program)
    for each function func in program do
        abstractState = computeAbstractState(func)
        derefSet = collectDerefs(abstractState, func)
    end for
    return derefSet
end function
```

Algorithm 6 Transfer(n) for MPA

```
function TRANSFER(node n)
    if n is allocation function then
        size = argument passed to malloc
        if size == size(Type of p) then
            gen = new fact (pointerVar = p, allocState = neither)
        else if size > size(Type of p) then
            gen = new fact (pointerVar = p, allocState = no --)
        end if
        kill= {< p', a' >∈ D(n)|p' = p}
        D(n) = (D(n)\kill) ∪ {gen}
    end if
    return D(n)
end function
```

Algorithm 7 $T_f(n_p, n)$ for MPA

function T_f(node n_p, node n)
  $D(n_p)$: state of n_p , $D(n)$: state of n
  for each *fact* in $D(n_p)$ **do**
    $D(n) = D(n) \sqcap$ *fact*
  end for
  return $D(n)$
end function

4.4 Sign Value Analysis (SVA)

Sign value analysis is designed to make sure variables maintain a valid sign in mutation testing. SVA uses the abstract syntax tree (AST) of a program to analyze its source code. An AST is a simplified syntactic representation of the source code that represents the parsed string of the program in a structured way [5]. Since SVA operates on the AST level, it has some limitations. This is because the AST is an abstract representation of a program and does not contain all the information. Therefore it is challenging to track the flow of information at the AST level.

Each value in the program is either a constant or in a form of an expression. In the first case, CRCR can change the sign of the value and in the second case, ABS can cause the sign to change.

SVA only analyzes the outermost expressions and does not take into account the inner expressions to reason about the sign of the outer expressions. For instance in code fragment 4.7, since variable `i` is used as the index of `arr`, SVA can infer that sign of `i` in line 3 cannot be negative. Variable `i` itself is a complex expression (`i = t + 5`). Since the inner expressions in `i` like `t` are not analyzed by SVA, `t` can be mutated to be negative even though it might cause a negative value for `i`. In addition to this limitation, SVA cannot analyze that `t+5` itself cannot be mutated to be negative. Because SVA does not keep track of the flow of information.

```
1 void bar(int* arr, int t){
2     int i = t + 5;
3      arr[i] = ...
4 }
```

Figure 4.7: SVA Example

Examples of sign related undefined behaviour were discussed in chapter 3. SVA identifies the following cases:

(a) **Array Index:** Values used as index of an array cannot be negative. For example in code fragment 4.7, since variable `i` is used as the index of `arr`, SVA reports that `i` in line 3 cannot be mutated with ABS operator.

(b) **Shift Value:** Values used as a shift width cannot be negative. SVA finds such values and reports that they cannot be mutated to have a negative sign.

(c) **Division By Zero:** Variables used as a divisor cannot be zero. SVA finds all the variables that are used in a divisor and reports that these variables cannot be zero.

In this chapter, we proposed three automated analyses for statically finding mutants with undefined behaviour. Null guard analysis and malloc pointer analysis operate on the control flow graphs and use dataflow analysis framework. Sign value analysis operate on abstract syntax tree of the source code. In the next chapter, we have used the teceshniques to find mutants with undefined behaviour in the real world projects.

Chapter 5

Evaluation

5.1 Implementation

We have used LLVM and Clang [10] to implement our approach. The LLVM (Low Level Virtual Machine) compiler infrastructure is a language and compiler system designed for static and dynamic compilation [11]. LLVM provides a fast, robust, and well-documented framework to support compilers and tools. It has three main components: first, the LLVM virtual instruction set, second, a collection of libraries for analysis, and third, tools built from those libraries. The LLVM virtual instruction set is the common intermediate representation (IR) shared by all the LLVM subsystems. The instruction set provides control flow graphs, data flow information in static single assignment (SSA) form and more [11]. Clang is a compiler front end for C-like programming languages and uses LLVM as its backend.

CEER is a mutation testing framework implemented using Clang and LLVM. It generates mutants for a given program and supports mutation operators listed in Table 2.2. CEER uses the Clang AST representation to find and generate the mutants. A preprocessing analysis that detects invalid mutations (undefined behaviour in mutants) would help CEER to be able to filter those out in advance.

We have implemented SVA using the Clang AST and passed its result to CEER. The result of SVA is a set of expressions that cannot be mutated with particular mutation operators. Since CEER and SVA both operate at the AST level, they can communicate and filter out the invalid mutants. NGA and MPA are implemented using LLVM IR because the IR provides a convenient infrastructure to do dataflow analysis, whereas the AST level is an abstract representation and does not support the flow of information. The result of this analysis is a set of instructions in IR that should not be mutated with a particular set of mutation operators at the AST level. Therefore, creating a communication channel between the Clang AST and the LLVM IR is necessary. Each expression at AST maps to one or multiple instructions at IR and they share the same debugging information. More precisely, each expression has debugging information containing the file name of the source file and

the corresponding IR instructions have the same debugging information. Our solution is to modify the source file name of each expression within the AST so that the file name contains a unique identifier for that expression. For example, changing the file name from *test.c* to *test_id.c* in which *id* is a unique identifier for an expression. This information can be retrieved in the IR level because each instruction in the IR has a corresponding expression in AST. The file name exists in the debugging information of the IR instruction and matches with the one from the AST expression. This way, a mapping between AST expressions and IR instructions is created. Thus, CEER can identify the invalid mutants and filter them out.

5.2 Experimental Results and Analysis

Table 5.1 presents the information about our benchmark. The benchmark contains seven programs with lines of code (Loc) ranging from 7,243 to 362,769. The total number of mutants generated by CEER for each program is shown in the third column of this table. These mutants are generated by applying mutation operators listed in Table 2.2. Gzip and Make are GNU utility programs. Gzip performs file compression, and Make is used to build executable files from source code files. GSL is a numerical library providing a range of common mathematical functions. MSMTP is an SMTP client for sending and receiving messages. Git is a source code management system. Grep is a command line utility to search plain text data sets for lines matching regular expressions. Finally, Vim is a configurable text editor.

Table 5.1: Benchmark

Program	Loc	Mutants
Gzip-1.6	7,323	45,412
Gsl-1.16	228,863	566,853
Make-4.0	32,122	72,132
MSMTP-1.4	13,068	25,260
Git-2.1	106,012	679,525
Grep-2.24	7,243	64,289
Vim-7.2	362,769	981,599

This section answers the following questions:

- Q1: Do mutants with undefined behavior exist in real world programs?
- Q2: How long does it take to analyze real world projects to find invalid mutants?
- Q3: How effective are our techniques for finding invalid mutants in real world projects?

5.2.1 Do mutants have undefined behavior?

To explore the presence of undefined behavior in mutants and find the answer to question Q1, we have measured the number of detected invalid mutants by CEER using each of the three analyses (NGA, MPA, SVA) individually as well as all three together for the programs in the benchmark. We also measure the proportions of the detected invalid mutants per program, computed as the percentage of the invalid mutants to the number of related mutants (all mutants coming from a mutation operator that may cause a related undefined behaviour). For NGA, the related mutants are those created using OCNG, ROR, LCR, and UOI mutation operators. Related mutants for MPA are the ones created by UOI. Finally, ABS and CRCR create the related mutants for SVA. For calculating the proportion of the total number of invalid mutants, we have used the total number of mutants in the program.

Table 5.2: Effectiveness

Program	NGA			MPA			SVA			Total		
-	UB	OCNG+ROR+ LCR+UOI	%	UB	UOI	%	UB	ABS+ CRCR	%	UB	mutants	%
Gzip-1.6	62	25,848	0.23	2	19,884	0.01	837	14,387	5.81	901	45,412	1.98
Gsl-1.16	320	269,394	0.11	724	239,256	0.30	3,178	193,140	1.64	4,222	566,853	0.74
Make-4.0	283	53,753	0.52	28	43,420	0.06	157	12,804	1.22	468	72,132	0.64
MSMTP-1.4	52	12,709	0.04	2	10,284	0.02	385	8,455	4.55	439	25,260	1.73
Git-2.1	1,906	480,325	0.39	72	391,124	0.02	2,408	145,409	1.65	4,386	679,525	0.64
Grep-2.24	122	44,722	0.27	18	35,408	0.05	247	14,084	1.75	387	64,289	0.60
Vim-7.4	1,762	643,427	0.27	70	483,272	0.01	7,420	244,870	3.03	9,252	981,599	0.94
Total	4,507	1,530,178	0.29	916	1,222,648	0.07	14,632	633,149	2.31	20,055	2,435,070	0.87

Table 5.2 reports the results per program and per analysis. For each analysis (NGA, MPA, and SVA), the corresponding column shows the number of potential mutants with undefined behaviour (UB) found, the number of related mutants (as discussed above), and ratio of the former to the latter (%). The last column shows this information using all three analyses.

Overall, the results indicate that our analyses can detect a total of 20,055 invalid mutants among the programs, which is 0.87% of the total number of mutants. This number ranges from 387 (0.60%) in Grep to 9,252 (0.94%) in Vim. SVA found the most invalid mutants with 14,632 (2.31%). NGA detected the second most with 4,507 (0.29%), and lastly, MPA detected a total of 916 (0.07%).

The number of UB found by each analysis depends on the characteristics of each program. For example, if the program uses a large number of memory allocations, there is a better chance for MPA to find related UB. As another example, programs that use shift operators and array accesses are more likely to have potential UB found by SVA. Finally, the programs that use guards for pointers tend to have a higher number of UB using NGA. A simple search for the word “malloc” in the GSL project finds 427 matches, but the same search

for MSMTP and Gzip projects have less than 10 matches in total. This explains why MPA finds 724 UB in Gsl and only 4 UB in total for the other two projects.

Software testers may choose to only use specific mutation operators and omit the others for various reasons. For example, one way to reduce the cost of mutation testing is to reduce the number of mutants created. One approach to achieving this is using selective mutation approach in which mutants for a program are generated without using the operators that create the most mutants (the expensive operators) [15]. NGA has four different related mutation operators and one of them (UOI) is categorized as an expensive one, while the other three are not. Therefore, we have included a breakdown of UB found by NGA in Table 5.3. The columns show information about each of the related mutation operators (OCNG, ROR, LCR, and UOI) and each column itself shows the number of invalid mutants (UB) found by NGA, total number of mutants (mutants), and the proportion (%) of them (all in regards to the corresponding mutation operator of that column).

Table 5.3: NGA

Program	OCNG			ROR			LCR			UOI		
-	UB	mutants	%	UB	mutants	%	UB	mutants	%	UB	mutants	%
Gzip-1.6	22	860	2.49	0	3,740	0.00	0	1,342	0.00	40	19,844	0.20
Gsl-1.16	2	2,044	0.10	0	23,391	0.00	32	4,669	0.68	286	238,970	0.11
Make-4.0	87	2,229	3.75	0	5,314	0.00	20	2,683	0.73	176	4,3244	0.40
MSMTP-1.4	26	535	4.63	0	1,441	0.00	0	423	0.00	26	10,258	0.25
Git-2.1	1,063	33,806	3.04	0	31,090	0.00	279	22,963	1.20	564	390,560	0.14
Grep-2.24	60	2,184	2.67	0	4,478	0.00	0	2,592	0.00	62	35,346	0.17
Vim-7.4	56	17,200	0.32	0	90,616	0.00	344	51,939	0.65	1,362	481,910	0.28
Total	1,316	77,758	1.70	0	160,070	0.00	675	86,611	0.78	2,516	1,220,132	0.20

In total 2,516 (0.20%) UB were found in the UOI category. OCNG and LCR have 1,035 (1.7%) and 675 (0.78%) respectively. Finally, no mutants were found in the ROR category.

In our implementation of NGA, we only report mutation operators that are local to the undefined behaviour identified. More precisely, assume the condition `cond = (ctx != 0)` is defined and is used later to guard the use of the pointer `ctx`, e.g. `if(cond)`, then CEER filters out the OCNG operator resulting in `if(!cond)`, but cannot reason about the structure of `cond` itself and therefore does not filter out the mutant `cond = (ctx = 0)` where `!=` becomes `=`. Our implementation is currently limited in this regard and can only reason about UB locally. Future work can address this problem by analyzing these relationships more precisely. This further contributes to analyses finding no undefined mutants for the ROR operator in Table 5.3. In addition, our implementation only supports simple conditions as shown in Figure 5.1. More complicated cases (nested expressions as conditions) require more analysis to understand which mutants can be filtered out without producing false positives. This is another direction for future work.

```
1 /* cond is a simple expression guarding a
     pointer*/
2 if(cond) {...}
3 if(!cond){...}
4 if(cond && ...){...}
5 if(!cond || ...){...}
```

Figure 5.1: Supported cases for NGA

In conclusion, the results show that mutants with undefined behaviour do exist in real world programs and filtering them out ahead of time reduces the overall number of overall mutants. As discussed earlier, this can mitigate the potential bias in computing the mutation score. In addition, killed and surviving mutants may be used for purposes other than computing the mutation score, and in such cases counting invalid mutants can create similar biases. For example, MUSE [13] is a mutation based fault localization technique. It ranks statements of a buggy program based on passing and failing mutants in order to locate a fault in a program. The key idea of MUSE is to identify the buggy statement by utilizing different expected characteristics of two groups of mutants: one group that mutates a faulty statement and the other that mutates a correct statement [13]. Therefore, a mutant that is invalid should be filtered out for computing the ranking of statements.

5.2.2 Efficiency

To assess the efficiency of the approach, answering question Q2, we report the execution time. Table 5.4 summarizes the execution time of our analyses in total, on average, and per program in the benchmark. The results show that the execution time of the invalid mutant detection is reasonably small. It ranges from only 0.2 seconds for MSMTP to less than 9 minutes for Vim. On average it takes about 1.5 minutes to analyze each program of the benchmark and around 10 minutes in total for all the programs. This reveals that running this analysis has a small overhead of time for real world programs compared to other tools for mutation analysis. For example, TCE takes 3 seconds to analyze each mutant [17] which adds up to be days to analyze the benchmark programs.

Table 5.4: Efficiency

Program	Loc	time (sec)
Gzip-1.6	7,323	19.32
Gsl-1.16	228,863	18.35
Make-4.0	32,122	16.24
MSMTP-1.4	13,068	0.22
Git-2.1	106,012	13.75
Grep-2.24	7,243	5.10
Vim-7.2	362,769	516.13
Total	757,400	589.11
Average	108,200	84.15

5.2.3 Evaluation Metric

STACK [19] is an open source static analysis tool that precisely identifies unstable code. Optimization-unstable code or unstable code, in short, is a class of software bugs in which code is unexpectedly discarded by compiler optimizations due to undefined behaviour in the program. The consequences of such bugs range from incorrect functionality to missing security checks [19].

To evaluate our approach and to answer question Q3, we have run STACK on the invalid mutants detected by CEER for each program. Table 5.5 shows the results for NGA and MPA. SVA is very simple and only filters out very straightforward cases (like negating array indices and shift values), and for this reason, we did not externally validate its results. Each column shows the number of UB identified by our analyses (CEER), the number of those that were also marked as UB by STACK (STACK), and the ratio of the latter to the former (%).

Table 5.5: Evaluation

Program	NGA			MPA		
-	CEER	STACK	%	CEER	STACK	%
Gzip-1.6	62	43	69.35	2	1	50.00
Gsl-1.16	320	249	77.81	724	62	8.56
Make-4.0	283	135	47.70	28	0	00.00
MSMTP-1.4	52	26	50.00	2	0	0.00
Git-2.1	1,906	1,021	53.56	72	0	0.00
Grep-2.24	122	61	50.00	18	2	11.11
Vim-7.4	1,762	964	54.71	70	30	42.85
Total	4,507	2,499	55.44	916	95	10.37

For NGA 2,499 invalid mutants were validated by STACK (55.44%). The maximum proportion of validated UB is Gsl in which 77.81% of identified UB is validated. In all programs, a minimum of 47.70% cases are validated. The missing cases (the ones not marked by STACK) can be either false positives by CEER (due to implementation faults) or false negatives by STACK. A manual inspection of 20 randomly selected files showed that the second case was true for all of them and CEER had correctly marked those as invalid while STACK did not. For instance, in the following example, `(--p)` is always evaluated to `true`. Therefore, dereferencing pointer `p` in line 3 is NPD. However, STACK does not mark this program as UB. Similarly it misses the case `if(1)` too. But it is able to validate `if((++p))` as UB. We concluded that this is the reason for STACK only validating around 50% of the identified UB by CEER in most cases.

```
1 void foo(int a, int *p) {
2     if ((--(p))) {     /* if(1) */
3         a = *p;
4     } else {
5     ...
6     }
7 }
```

Figure 5.2: Undefined Behaviour Missed by STACK (NGA)

This provides evidence that the results from NGA are accurate and conservative. However, since we have not externally validated all the files, we cannot claim that there are no false positives.

For MPA only 95 out of 916 (%10.37) invalid mutants were marked by STACK with a minimum of 0% and a max of 50% for the programs. We found this is due to different

approaches that CEER and STACK use to mark programs undefined. CEER reports UB if an invalid memory address is used. However, STACK reports such cases as UB only when the undefined behavior causes the compiler to remove code. For instance, the following code fragment is identified as UB by CEER but not by STACK. In this example, applying the UOI operator (predecrement) on pointer `c`, will point to an invalid address and therefore is undefined behaviour. However, since this does not cause the compiler to remove any code, STACK does not report it. These differences fundamentally come from the fact that STACK is designed to find unstable code, but CEER is designed to find UB in general even if it does not result in code being removed.

```
1 struct dep {
2     int num;
3     int name;
4 };
5
6 void foo(struct dep **c, int a) {
7     c = malloc(2 * sizeof(struct dep));
8     a = ((--(c))[0]->num);
9 }
```

Figure 5.3: Undefined Behaviour Missed by STACK (MPA)

We have run STACK on all the mutants generated by CEER for Gzip and Make projects to find the number of mutants with undefined behaviour detected by STACK. We only did this experiment for these two programs because of the time overhead involved in using STACK which takes days to analyze mutants of a real world project. The results show that 3.9% and 4.1% of the mutants were detected as invalid by STACK for Gzip and Make respectively. As a result we can conclude that there is potential to filter out more mutants with undefined behaviour by using automated analyses.

5.3 Threats to Validity

To cope with possible threats to external validity, we have selected seven well-known and open source projects as the benchmarks. These programs differ in size and purpose. The results show that depending on the characteristics of the subject program, the number of invalid mutants may differ for each undefined behaviour type. However, the overall presence of invalid mutations is consistent and this shows that the problem of undefined behaviour in mutation testing does exist.

Threats to internal validity include our implementation of the analysis that might contain errors that affect the results. To cope with this threat, we have performed tests on the tool;

however, testing alone cannot prove the absence of errors. In addition, we have used STACK to validate our results, but since the design and purpose of STACK is different from CEER, we could not validate all cases.

Our implementation is dependent on the proposed communication channel between the Clang AST and LLVM IR. We have observed that in some cases changing the debugging information in Clang expressions does not propagate to LLVM instructions and they maintain their old debugging information. In such cases, the channel fails to correctly find the corresponding expression of an instruction. This results in false negatives since CEER cannot identify and filter out such cases even though they have been found in the IR level.

Chapter 6

Conclusions and Future Work

This thesis presents the first study of undefined behaviour in mutation testing, a problem that results in increased number of mutants that needs to be considered and also creates a potential for bias in the mutation score. Our results show that traditional mutation testing, in the C programming language, does create meaningless mutants in real world programs. We have identified cases that directly lead to undefined behaviour in mutation testing based on each mutation operator type as discussed in chapter 3. Then in chapter 4 we proposed an automated analysis including three techniques: null guard analysis, malloc pointer analysis, and sign value analysis in order to identify mutants with undefined behaviour. Each of these three techniques aims to address a particular type of undefined behaviour that is related to a set of mutation operators. Furthermore, our implementation proves that these techniques are able to filter out invalid mutants ahead of time and reduce the overall number of mutants with undefined behaviour.

6.1 Future Work

In this work, we have proposed analysis techniques for identifying and preventing undefined behaviour in mutation testing. However, the proposed methods do not cover all the possible cases of undefined behaviour types. Future work includes adding other techniques to cover such cases. For instance SVA can be extended to determine lower and upper bounds on values of expressions instead of just the signs of simple expressions. Adding alias analysis to NGA and MPA enables them to reason about pointers that point to the same address and potentially filter out more mutants. As discussed earlier, our current implementation only reports invalid mutants locally. Future work can address this problem by analyzing the flow of information and relationships between the variables. For NGA, supporting more complicated cases like conditions inside nested expressions can be another direction for future work.

In addition, our current approach is intraprocedural and can be extended to interprocedural to cover more cases. Also, a better communication channel between LLVM IR and the Clang AST is needed to reduce false negatives. Currently in some cases, the channel fails to make a mapping between an instruction and its corresponding expression, and invalid mutants will be missed in such cases.

Bibliography

[1] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J Martin, Aditya Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[2] SS Ahamed. Studying the feasibility and importance of software testing: an analysis. *arXiv preprint arXiv:1001.4193*, 2010.

[3] Paul Ammann and Jeff Offutt. In *Introduction to Software Testing*, pages 182–185. Cambridge University Press, 2008.

[4] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411. IEEE, 2005.

[5] Eli Bendersky. Abstract vs. Concrete Syntax Trees . `http://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/`. Accessed: 2009-16-02.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[7] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[8] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[9] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior. `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`. Accessed: 2011-11-05.

[10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[11] Rudolf Eigenmann Zhiyuan Li and Samuel P Midkiff. Languages and compilers for high performance computing, 2005.

[12] Thomas J Marlowe and Barbara G Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.

[13] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[14] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[15] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[16] Jie Pan. Using constraints to detect equivalent mutants. Master's thesis, George Mason University Master's thesis, 1994.

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

[18] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[19] Kirsten Winter, Chenyi Zhang, Ian J Hayes, Nathan Keynes, Cristina Cifuentes, and Lian Li. Path-sensitive data flow analysis simplified. In *International Conference on Formal Engineering Methods*, pages 415–430. Springer, 2013.

[20] Michal Young and Mauro Pezze. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.