Flagging Error Prone Segments in a C Program

By

Dibya Sarker

Roll: 1507020

&

Tuli Rani Das

Roll: 1507064

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

February, 2020

Flagging Error Prone Segments in a C Program

By

Dibya Sarker

Roll: 1507020

&

Tuli Rani Das

Roll: 1507064

A thesis submitted in partial fulfillment of the requirements for the degree of
"Bachelor of Science in Computer Science and Engineering"

Supervisor: **Dr. Muhammad Sheikh Sadi**
Professor
Department of Computer Science and Engineering
Khulna University of Engineering & Technology

_____
Signature

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh
February, 2020

Acknowledgement

With due to homage and honor, we wish to express our gratitude to the Almighty. Then we acknowledge, with gratitude, our debt of thanks to our honorable supervisor Dr. Muhammad Sheikh Sadi for his helpful contribution by giving necessary guidance, advice and encouragement. We thank him for his patience and the continuous support he has given us so far in completing this thesis.

We want to extend our thanks to all teachers of the CSE department in KUET who helped us by providing guidelines to complete this thesis.

Authors

Abstract

This thesis proposes an unorthodox approach to detect and highlight the error-prone segments of a c program. A mistake in a c program may result in unexpected behavior like wrong output, terminating prematurely, or not terminating at all. To avoid scenarios like that, adequate measures must be adopted to debug the program accurately. Many techniques are pretty efficient in dealing with different errors already in practice. Based on several heuristics, the proposed method would flag where the instructions are prone to error in a C program. Since memory level instruction interpretation can be intricate, we assumed a generic instruction size for each statement and calculated the erroneous probability of each segment. We embraced a generic instruction size for each statement, and our calculated results can convey a convenient idea of where the potential error may originate.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this chapter, the complex problem of debugging the error-prone segments and resolving them is discussed. The importance of doing this task has also been greatly emphasized. The objectives and scope of this thesis are explained in this chapter, alongside the contribution of this thesis.

1.2 Problem Statement

The need for testing and debugging a program has always been here. There are various debuggers and different methods of testing and debugging [1]. An error-prone software can create faults that may lead to unexpected behavior of the program. In order to avoid situations like that, the program must be debugged properly and efficiently. The main issue here is debugging a program is not always straightforward. It can be quite arduous and time-consuming [2]. The complexity of a program depends on the structure of the program it is based upon [3]. There can be a lot of variables and lots of operations in a program which can result in a complex program [3]. A complex program is hard to keep track of because of the number of variables and operations it has. Hence, it is a necessity to find out a way such that the whole process can be easily executed. A program can run into various types of errors like compile errors, runtime errors, and logical errors [4]. Hence, it's not just the programs that need to be tested before compiling, but also the checking must continue during the program execution [5]. And finally, the output of the program has to be checked because maybe the program has run fine, but that doesn't mean there is no error in the program, a logical error can exist in the program because of using the wrong logic, and as a result, the output will differ from the output that was expected [6]. Hence, we have to consider all of the above-mentioned cases and develop some model which will take all of the errors into consideration and yield a suitable and easy way to detect the segments of an error-prone source code.

1.3 Motivation

Debugging a complex program can be very time-consuming and tiresome. It takes a veteran programmer quite some time to completely debug a complex program [7]. The errors can stop the program from executing properly or may even stop the execution completely [8]. When an error-prone source code is compiled, and there are bugs present in the source code, it can potentially cause a fault in the program [9]. The fault can be a reason for the program's failure. A program failure means that the program is giving faulty output or terminating prematurely, or not terminating at all.

Debugging and testing a program is a fundamental step in every software development [10]. The general approach to debugging a code is to observe the outputs of each segment and check whether there is any difference between the observed and actual output [11]. If there is any deviation, then that segment is checked for bugs. This is the approach that has been in practice for quite some time now. Our work can greatly improve this process. We don't have to check the outputs of the segments to check for bugs anymore. We can directly sort out the segments which can cause potential errors according to their contribution to the actual output of the program [12].

Hence, our work will greatly aid them in debugging and help them to make a priority list of statements from which they can easily select the most important ones to debug first [13]. From our work, they can easily identify the priority statements or blocks that they should first take a look at for the bugs. We believe our work will benefit those who are in dire need of debugging and testing a complex c program quickly. We believe this will make a huge contribution to the whole process of debugging a program which is an integral part of system design itself.

1.4 Objectives of the Thesis

The target of our work is to achieve the following functionalities:

- Detecting the error-prone segments of a program efficiently.
- Highlighting the error-prone segments according to their significance in the source code.

Hence, the main objective of our work is to detect the error-prone segments and highlight them.

1.5 Scope of the Thesis

The proposed method is designed to easily detect the error-prone segment efficiently. Debugging and testing is a fundamental part of any C program. The scope of the proposed methodology includes detecting the most sensitive segment in a C program alongside ranking the segments according to their probability of being error-prone. It will also highlight the segments with different colors to represent different levels of probability which allows the user to easily visualize the potential error-prone segment.

1.6 Organization of Thesis

The remaining part of this thesis consists of the parts as follows:

Chapter 2 introduces the formal structures and terminologies used in this work. This section describes some of the existing methodologies that have already been proposed for the detection of sensitive segments in a program.

Chapter 3 describes the mechanism of detection of error-prone segments and highlights them according to their significance.

Chapter 4 represents experimental results and analysis of our proposed method for improving the testing and debugging process.

Chapter 5 draws the conclusion of the thesis and the limitations of our proposed methodology. It also states some of the future scopes of this work on the proposed model.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, the process of error analysis and debugging has been discussed at great length. For developing our method, we have undergone many previous works related to the debugging and testing of a program. They are also discussed in this chapter.

2.2 Error Analysis and Debugging

We have improved a lot in debugging and testing programs over the years. The method and tools of debugging and testing have seen a meteoric rise. The errors can be detected far more easily and convincingly.

Normally errors mean illegal operations performed by the user, which can cause wrongful executions of the programs. Programming errors can be hard to detect. Debugging a program can be quite tedious at times. Hence much research has been conducted to find the most efficient ways of debugging a program [13]. Programming errors can be quite notorious for detection. They can remain undetected until the program is fully compiled and executed. Some errors can stop programs from compiling [3]. This type of error should be removed before compiling and executing a program. Basically, there are three types of errors in c programming:

1. Runtime Errors
2. Compile Errors
3. Logical Errors

Runtime errors are the errors that happen when the program is being executed, and normally the reason behind the occurrence of these types of problems is some kind of illegal operation executed in the program [5]. Mainly the runtime errors raise some kind of exception that can potentially crash a program and stop execution. In order to handle this type of situation, a program should be

written in a way that, in spite of such unexpected behavior, the program should still operate normally.

Compile errors mainly happen when the compiler fails to compile a program because of some sort of wrong syntax or semantics. In other words, if some syntax of the code which is not defined by the specific language is written anywhere in the code, then a compiler error will occur.

Logical errors are those errors that cannot be detected by the compiler and are mainly caused by the usage of wrong logic to obtain the output[6]. Logical error leads to wrong answers; hence programmers have to test and debug the entire code for logical errors.

2.3 Related Works

Ma et al. coined a new assessment benchmark named Output Vulnerability (OVF) which can rank the variable importance in order to easily identify the more SDC-prone variable in the program [14]. Their proposed method increases the SDC detection rate significantly.

Thomas and Pattabiraman here suggest looking at the software side of the equation as a remedy to this problem [15]. Certain software applications can tolerate faults till reaching a threshold and are still able to put out near-accurate results. The authors emphasized the need to place the error detectors optimally to rank the data according to their susceptibility to cause EDC or Egregious Data Corruption.

Mukherjee et al. provide an overview of soft error issues from the point of view of an architect [16]. They describe some techniques used to reduce soft errors and discuss the possible steps which can be taken in the future.

Parnin and Orso performed a study on the developers regarding how effective automatic debugging is by providing them with an automatic debugger and asking the developers to debug with and without the tool to get a comparative analysis [2]. A lot of the assumptions on which the automatic

debuggers are based on doesn't apply in real life, they said. They also share some insights about future works in this automated debugging sector.

Forman and Singpurwalla discussed a stopping rule to evaluate software reliability [1]. The approach can also be used for assumption under stop-stress testing, which is another benefit.

Kestor et al. present a comprehensive evaluation of five methods where the detectors performed admirably but far from perfect [17]. They also evaluate ML-based detector, which has a better success rate than traditional but it isn't perfect either.

Johnson and Pheanis present a new tool for error detection and error prevention when it comes down to assembly language [18]. It makes our life a lot easier by detecting and exposing the errors in assembly language automatically. The tool can detect memory access errors and stack errors which are sometimes missed by other detectors, and it lessens the testing time considerably.

Glukhikh et al**.** here propose a new method for software reliability assumption, which is based on using static code analysis [19]. It can also calculate error probabilities as well. The authors used the approach in the AEGIS tool and used it to evaluate many real-life software projects.

2.4 Summary

We have seen many previous works around the detection and debugging of soft errors in programs. The literature review is very important for our work because we have learned a lot from these studies. These ideas are fundamental for constructing a good understanding of the problem and help us to think of an approach to solve the problem. Without the previous works, we may have to start from scratch and then work our way forward, which can be a tough thing because we wouldn't have anything to lean on. Hence, from that aspect, the previous works helped us a lot in constructing a solid base from the beginning, which is needed for the detection of error-prone segments in a c program.

Chapter 3

Methodology

3.1 Introduction

In this chapter, we will discuss the methodology in detail. This will provide a guide to follow our method and understand it step by step. We will explain all the steps elaborately so that it provides a clear picture of our proposed methodology. Some examples will also be provided with the necessary screenshots of the sample code.

Our main goal is to find the error-prone segments of a C program and flag them according to their significance in the program. Detecting the error-prone segments is important because it can cause errors that can result in generating wrong outputs or stopping the program, or not stopping at all. To avoid facing this type of situation, we need to properly debug and test a program before executing it. Hence, we can basically divide the whole process into two steps. The first step is to detect the error-prone segments, and the second step is to highlight them according to their significance. Highlighting the segments according to their significance is important because it allows users to better read the program. The user can easily understand the significance level by just observing the highlight colors. This enables the user to easily rank the code segments and allows him to find out any possible bugs in code, if there are any. Hence, the whole process, from detecting to highlighting, will be discussed elaborately throughout this chapter.

3.2 Overall System Flow

The whole process can be sectioned into two portions:

- Error-prone segment detection
- Highlighting the code segments

Hence, first, we detect the error-prone code segments. Then after that, we highlight the error-prone segments according to their significance.

3.3 Proposed Heuristics for Detection

We have proposed four heuristics for detecting the error-prone segments in the input source code. We have applied these heuristics when calculating the most error-prone segment in the input source code. The heuristics are given below:

1. **H1:** The probability of an error-prone segment depends on the number of statements that are executed in the If-ElseIf-Else segment. As the number of statements in that segment increases, the probability also increases.
2. **H2:** The probability of an error-prone segment depends on the number of statements that are executed in the loop segment. If the number of iterations of the loop increases, then the probability of the segment being error-prone also increases.
3. **H3:** The probability of an error-prone segment depends on the number of statements that are executed in the function. As the number of statements increases in the function, the probability also increases alongside the executed statements.
4. **H4:** The probability of an error-prone segment depends on the execution time of statements in a segment that are executed in the program. If the execution time is longer, we can assume that there may be repeated iterations of the same or many statements, but it is not guaranteed.

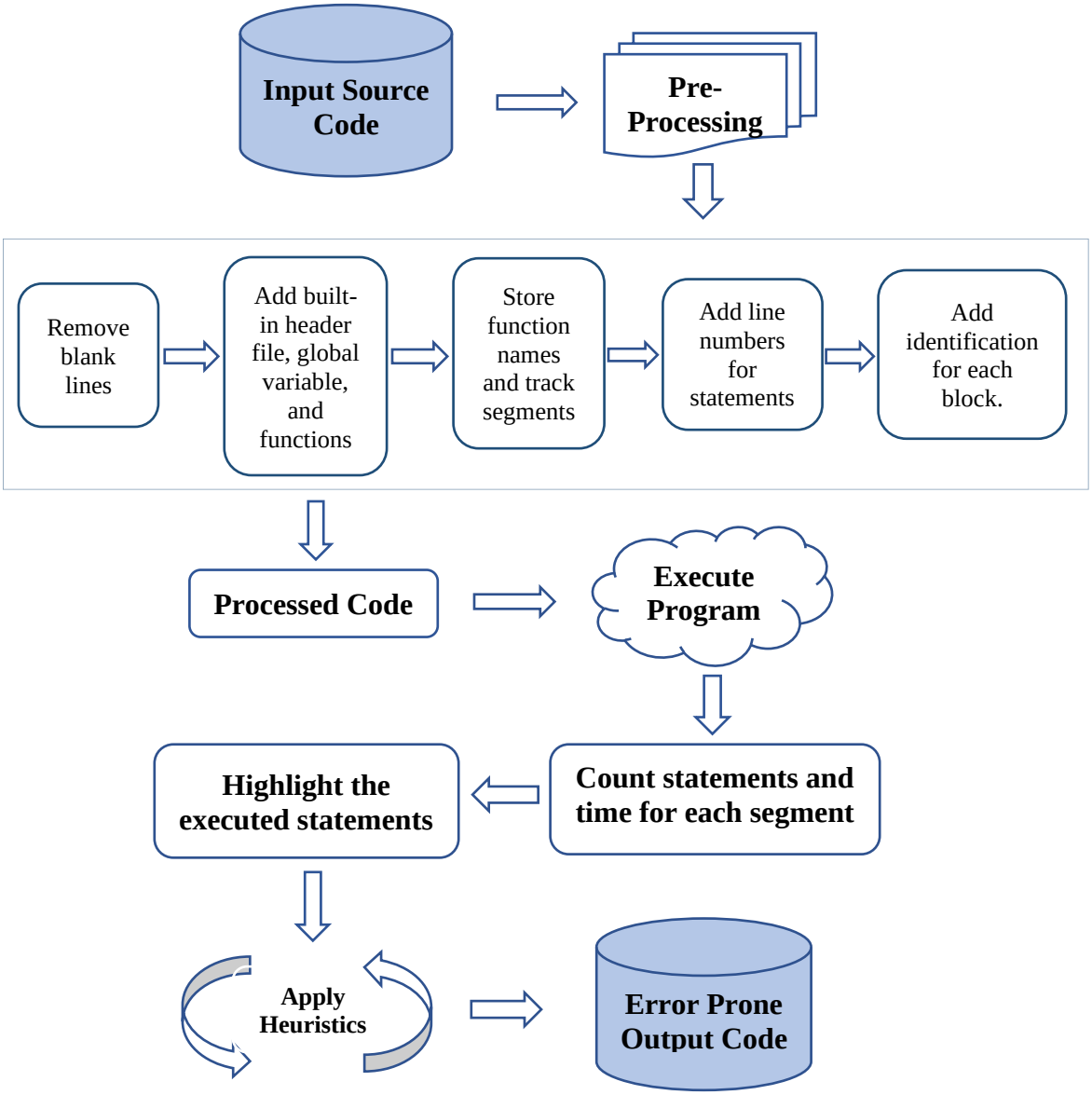

Figure 3.1: Flow Chart for the Proposed Method

3.4 Detailed Explanation of Proposed Algorithm

3.4.1 Input Source Code

The first step of the whole methodology is to obtain the source code. The source code is the code that we will use as the input program. We will analyze this program and detect the error-prone segments in the program. The input program can be a simple code with some basic operations like addition, subtraction, multiplication, and arithmetic operations. It can also be a complex program with a lot of operations. There can be long loops and functions in a single program. We have to analyze all these functions and find out the error-prone segments among them.

The source code which we are going to use should follow the syntax of the C programming language. The program should be free from all sorts of compilation errors. We didn't design the whole methodology for the programs with a compilation error. This method assumes that the source code is free from any kind of compile error. The program should be well-indented and should be easily understandable.

3.4.2 Preprocessing

In order to reduce computational complexity and for achieving a better result, some preprocessing must be done to the input source code. Preprocessing can greatly decrease the computational time in our case. Preprocessing means preparing the input before the actual processing so that the actual process can speed up rapidly. In our work, the preprocessing is done in the following sections:

- Removing extra blank lines and adding necessary header files and functions for future operations.
- Adding global variables to keep track of loops, if-else, functions, etc.
- Storing the names of all the functions
- Adding a line number for each statement
- Adding block-wise id for each block

```c
#include<stdio.h>
int abc;
int ttt;

void FuctionTerminatingBranch(int s,int e)
{
   if(s<0 || e>=100)
   {
      return;
   }

  int sum = 0;
   int ara[100];
   int i;
   for(i=s ; i<=e ; i++)
   {
      ara[i] = s;
   }
   for(i=s ; i<=e ; i++)
   {
      printf("%d\n",ara[i]);
   }
}

int checkOddEven(int num)
{
   if(num%2==0)
   {
      return 0;
   }
   return 1;
}

int main()
{
   int a = 10;
   int testAra[1000];

   int temp = 0 ;
   int i,j,k;
   for(i=0;i<5;i++)
   {
      for(j=0;j<5;j++)
      {
         temp += checkOddEven(j);
         testAra[i] = temp ;
      }
   }
FuctionTerminatingBranch(2,20);

   int z = 13;
   while(z--)
   {
      int t = 10;
   }

   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         int a = checkOddEven(j);
         for(int k=0;k<3;k++)
         {

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

   if(a == 10)
   {
      int b = 10;
      int c = b + 1;
      int s = b + c;
   }

   else
   {
         for(int i=0;i<=1000;i++)
         {
            int xx = i;
         }
   }
   return 0;
}
```

Figure 3.2: Input Source Code (1)

```c
#include<stdio.h>

void fun()
{
   int i=0;
   i=1;
   i=2;
   return;
}

void fun2()
{
  int i ;
  for(i=0;i<=100000;i++)
  {
     int a = i * 2;
  }
}

int main()
{
   int i,j,k;
   for(i=0; i<5; i++)
   {
      for(j=0; j<10; j++)
      {
         int a = i+j;
      }
   }
   for(i=0; i<1000000; i++)
   {
      int a = i+i;
   }

  k = 10;
   if (k>0)
   {
      for(i=0;i<100;i++)
      {
         j = j + i;
      }
   }

 if(k==10)
   {
      for(i=0;i<500;i++)
      {
         j = j + i;
 }
   }

   fun();
   fun2();
   return 0;
}
```

Figure 3.3: Input Source Code (2)

i. Removing Extra Blank Line

The input program file can use a lot of blank spaces and unnecessary blank lines, which is bad for readability. Initially, when the source code is obtained, there can be a lot of extra spaces between each of the code snippets, which are completely unnecessary. Hence, we can remove the extra space to make the code look better. We can also remove the unnecessary comments in the code, which will be ignored anyway. As a part of the preprocessing, we will remove these sorts of blank

spaces and blank lines from the given input program. It will make our task a lot easier in the following steps after removing these blank lines and spaces.

ii. Adding Necessary Header File and Functions

We will need to calculate the execution time from the given input source file. In order to do that, we need to use the **"chrono"** header file. Hence, we added the header file in this step. To calculate the execution time for each segment, we use this **"getTicks()"** function, which returns the execution time in nanoseconds. Another important step in preprocessing is to add an extra line in the source code so that the output of the program is written onto a text file. We normally use the **"freopen"** function, which allows us to save the output of the program to a specific text file. This line is important because normally, programs execute in the command prompt, which doesn't store the output of the program, but in our case, we need the output of the program, and for that, we need to store the output somewhere. That is the reason why we added this extra line to the input source code. Another advantage of this function is it doesn't change any operation of the source code. It just writes the output to a text file. In this case, we write the output to a file named **"Output.txt"**. Hence, the output remains the same from the very beginning.

iii. Adding Global Variables

In this step, we add some global variables to the input source code. This is needed because we need to keep track of the loops, if…else, function blocks, etc. By adding these variables, we can name each of the blocks with their id. In the upcoming steps, we will use these ids to identify the blocks and then store their start line and end line of them.

```
#include<stdio.h>
#include <chrono>
static int global_loop_id = 0, global_ifelse_id = 0, global_function_id = 0;
```

Figure 3.4: Adding Necessary Header File and Global Variables

iv. Storing Names of the Functions

In order to speed up the process in the future, we are storing the names of all the functions present in the input source file. In this way, we can easily keep track of the number of functions that exist in the input source code. Each of the function's start point and end point can be managed easily in this way.

v. Storing Track of Program Segments

In order to keep track of the If-ElseIf-Else segments, Loops (for / while), and Functions, we have implemented a stack to store the track of all these segments. The stack is implemented in a way so that it can output the start and end of a segment properly. The stack initially holds the starting parenthesis of a segment, and if another starting parenthesis occurs, then it pushes that parenthesis into the stack. Similarly, if any ending parenthesis is found, it pops the top of the stack. After this operation, we can detect that a full segment has ended. This is how we track the starting and end of each segment.

vi. Adding Line Number for Each Statement

In order to find out the block number of each statement, we have to add another additional line in the input source code. This line will return the line number of that statement. This is needed because we will group the statements under specific blocks like a function, loop, or if-else branch. Hence, if we know the line number of each statement, we can easily determine the block it belongs to. We just have to check whether the line number is between the start point and the end point of a block. If it is then it belongs to that block, or else the statement is not in that block.

```
printf("line = %d\n",__LINE__);          int b = 10;
printf("line = %d\n",__LINE__);          int c = b + 1;
printf("line = %d\n",__LINE__);          int s = b + c;
```

Figure 3.5: Adding Line Number for Each Statement

vii. Adding Block Wise Identification Number for Each Block

We need to add a block id for each block in the input source code. In this way, we can manage each block easily using the id. We used this id multiple times in the next steps when we tried to get the statements that a particular block has. For this reason, adding a block is important in this preprocessing stage because it makes our task a lot easier.

```
int checkOddEven(int num)
{
    printf("function_start_id %d line = %d\n", ++global_function_id, __LINE__);
    if(num%2==0){
        printf("ifelse_start_id %d line = %d\n", ++global_ifelse_id, __LINE__);
           printf("line  =  %d\n",__LINE__);printf("ifelse_end_id  %d  line  =  %d\n",
global_ifelse_id--, __LINE__);
        printf("function_end_id %d line = %d\n", global_function_id--, __LINE__);
        return 0;
        printf("ifelse_end_id %d line = %d\n", global_ifelse_id--, __LINE__);
    }
      printf("line  =  %d\n",__LINE__);printf("function_end_id  %d  line  =  %d\n",
global_function_id--, __LINE__);   return 1;
    printf("function_end_id %d line = %d\n", global_function_id--, __LINE__);
}
                                        .
                                        .
                                        .
                                        .
printf("loop_start_id %d line = %d\n", ++global_loop_id, __LINE__);    for(i=0;i<5;i++)
    {
printf("loop_start_id       %d       line       =       %d\n",       ++global_loop_id,
__LINE__);        for(j=0;j<5;j++)
        {
printf("line = %d\n",__LINE__);            temp += checkOddEven(j);
printf("line = %d\n",__LINE__);            testAra[i] = temp ;
        }printf("loop_end_id %d line = %d\n", global_loop_id--, __LINE__);
    }printf("loop_end_id %d line = %d\n", global_loop_id--, __LINE__);
```

Figure 3.6: Adding Block Wise Identification Number for Each Block

3.4.3 Execute Processed Program

After we are done with the preprocessing, we can now go to our next step, which is executing the program. We have already done most of the hard work in the preprocessing step, which will greatly aid us in the next steps. Now we will execute the program. As we have already added the necessary statements to the input program. As this is a C++ program, we need to execute it with a **GCC Compiler**. As we are handling the input file as a string input, and we are using **Python** as our main language for all internal operations. We created a function in the python language named **"executeProcessedSourceCode()"** which executes the cpp file with the **GCC** compiler with the **C++11** version. We used this C++11 version because we used a header file named **"chrono"** which needs this version of C++ to execute. The output of this program will be stored in the **"Output.txt"** file.

3.4.4 Count Instructions for Each Segment

In this step, we have to count the instructions for each segment. In other words, we have to find out the number of statements that each segment has. We will count this number using the **"Output.txt"** file. In this file, we have the line number of each statement which we will need to check whether the statement belongs to a segment or not. Here the output is given in **"line = n"** format, which means the statement in the nth line has been executed. The "output.txt" file will also have the output format like this **"loop_start_id i line = a"** or **"loop_end_id i line = b"** which means the loop which has the id "i" starts at the a^{th} line and ends at b^{th} line. The same format is applied for both if-else and functions. Hence, for each statement, we have to check if the line number is between the start line and end line of the loop, if-else, or any function or not. If we find that a line number is between the start and end points, then we can add that statement in that segment. We do it by using a dictionary in Python.

3.5 Highlight the Executed Statements

After counting the statements for each block, the statements which are executed are highlighted in an HTML file. In order to do that, every statement from the "Output.txt" is highlighted. To do that,

the input source code is written into an HTML file. Then if we find the statement in the "Output.txt" file, we change the color attribute of the HTML file to our preferred color. That is how the statements are highlighted.

```
#include< stdio.h>

int abc;
int ttt;
void  FuctionTerminatingBranch(int s,int e)
{
   if(s< 0 || e> =100)
   {
   return;
   }
   int sum = 0 ;
   int ara[100];
   int i;
   for(i=s;i< =e;i++)
   {
      ara[i] = s ;
   }
   for(i=s;i< =e;i++)
   {
      printf("%d\n",ara[i]);
   }

}
int checkOddEven(int num)
{
if(num%2==0)
{
return 0;
}
return 1;
}

int main()
int a = 10;
int testAra[1000];
int temp = 0 ;
int i,j,k;

for(i=0;i< 5;i++)
{
for(j=0;j< 5;j++)
{
temp += checkOddEven(j);
testAra[i] = temp ;
}
}
FuctionTerminatingBranch(2,20);
int z = 13;

while(z--)
{
int t = 10;
}
for(i=0;i< 3;i++)
{
for(j=0;j< 3;j++)
{
int a = checkOddEven(j);
for(int k=0;k< 3;k++)
{
printf("%d %d %d\n",i,j,k);
}
}
}
if(a == 10)
{
int b = 10;
int c = b + 1;
int s = b + c;

}
else
{
for(int i=0;i< =1000;i++)
{
int xx = i;
}
}
return 0;
}
```

Figure 3.7: Highlight the Executed Statements of Source Code (1)

In Figure 3.7, **Green** color refers to the statement executed only one time, **Red** refers to the statement executed most for this source code, **and Blue** and **Orange** are ranked between the minimum and the maximum number of executed statements.

3.6 Highlight the Segments after Applying Heuristics

After highlighting all the statements that have been executed, it is time to highlight the most error-prone segments. This is the part where we apply the heuristics we have established before.

1. **Heuristic (H1):** The If-ElseIf-Else block with the most executed instructions will be highlighted according to this heuristic. The font color of every statement in this segment will be **Violet,** and the line with a conditional logic will have the background color of **Blue.**
2. **Heuristic (H2):** The loop (while/for) block with the most executed instructions will be highlighted according to this heuristic. The font color of every statement in this segment will be **Red,** and the line with conditional logic will have a background color of **Turquoise Blue.**
3. **Heuristic (H3):** The function block with the most executed instructions will be highlighted according to this heuristic. The font color of every statement in this segment will be **Blue,** and the line with conditional logic will have a background color of **Yellow.**
4. **Heuristic (H4):** The segment with the longest execution time will be highlighted according to this heuristic. This heuristic solely depends on the execution time of the segments. If we can detect that the time for a particular segment is longer than others, then we can say that the segment will have a higher probability of being error-prone according to this heuristic. The segment can be a loop, an if-elseif-else branch, or can be a function. The main restriction of this heuristic is we cannot use this in every situation. Normally the execution time of a segment is very short. Hence, there needs to be a significant difference in executed statement number if we want to use this heuristic. Unless we cannot distinguish between two segments if the executed statement number difference is too small, for this reason, generally, this heuristic is only applied in loops because of the difference in the iteration count. The loop with the most iteration is highlighted. The loop (while/for) block with the most executed instructions will be highlighted according to this heuristic. The font color of every statement in this segment will be **Red,** and the line with conditional logic will have a background color of **Turquoise Blue**. The font color and highlighting color for if-elseif-else and function will be the same as the color used for H1 and H3.

Sample Input	Sample Output After Applying H1

#include <stdio.h>
 	
int main()
{
  int a = 10;
  if(a == 10)
  {
  int x = 1000;
  int y = x + a;
  }
  if(a == 10)
  {
  int x = 5;
  int y = 6;
  int z = x + y;
  x = z + x + y;
  }
 	
  return 0;
 	
}
	#include <stdio.h>
	
	int main()
	{
	  int a = 10;
	  if(a == 10)
	  {
	  int x = 1000;
	  int y = x + a;
	  }
	  if(a == 10)
	  {
	  int x = 5;
	  int y = 6;
	  int z = x + y;
	  x = z + x + y;
	  }
	
	  return 0;
	
	}

Figure 3.8: Sample Input and Output After Applying H1

Sample Input	Sample Output After Applying H2

#include <stdio.h>
 	
int main()
{
int i,j,k;
for(i = 0 ; i < 5 ; i++)
{
for(j = 0 ; j < 10 ; j++)
{
int a = i + j;
printf("summation is %d: \n",a);
}
}
 	
for(i = 0 ; i < 40 ; i++)
{
int a = i + 1;
printf("summation is %d: \n",a);
}
 	
return 0;
}
	#include <stdio.h>
	
	int main()
	{
	int i,j,k;
	for(i = 0 ; i < 5 ; i++)
	{
	for(j = 0 ; j < 10 ; j++)
	{
	int a = i + j;
	printf("summation is %d: \n",a);
	}
	}
	
	for(i = 0 ; i < 40 ; i++)
	{
	int a = i + 1;
	printf("summation is %d: \n",a);
	}
	
	return 0;
	}

Figure 3.9: Sample Input and Output After Applying H2

Sample Input	Sample Output After Applying H3

#include <stdio.h>
 	
void fun()
{
  int i = 0;
  int b = 10;
  int a = b + 1;
}
 	
void fun2()
{
  int i = 0;
  int b = 10;
  int a = b + 1;
  i = 0;
  b = 10;
  a = b + 1;
}
 	
int main()
{
  fun();
  fun2();
  return 0;
}
	#include <stdio.h>
	
	void fun()
	{
	  int i = 0;
	  int b = 10;
	  int a = b + 1;
	}
	
	void fun2()
	{
	  int i = 0;
	  int b = 10;
	  int a = b + 1;
	  i = 0;
	  b = 10;
	  a = b + 1;
	}
	
	int main()
	{
	  fun();
	  fun2();
	  return 0;
	}

Figure 3.10: Sample Input and Output After Applying H3

Sample Input	Sample Output After Applying H4

#include <stdio.h>
int main()
{
int i,j,k;
for(i = 0 ; i < 1000000 ; i++)
{
k = i + j;
}
for(i = 0 ; i < 100000 ; i++)
{
for(j = 0 ; j < 10 ; j++)
{
k = i + j;
}
}
 	
return 0;
 	
}
	#include <stdio.h>
	int main()
	{
	int i,j,k;
	for(i = 0 ; i < 1000000 ; i++)
	{
	k = i + j;
	}
	for(i = 0 ; i < 100000 ; i++)
	{
	for(j = 0 ; j < 10 ; j++)
	{
	k = i + j;
	}
	}
	
	return 0;
	
	}

Figure 3.11: Sample Input and Output After Applying H4

By applying those heuristics, we can find out the most error-prone segments to the least error-prone segment. One thing to note is that we are not saying that the error must generate in the most error-prone segment. We are just saying that in case any error happens, it is most likely that the error will be in the error-prone segment according to the heuristics. As we have four heuristics, we are highlighting the most error-prone segment according to these heuristics. So, we have three highlighted segments that are the most error-prone according to each of the three heuristics.

```
#include< stdio.h>

int abc;
int ttt;
void FuctionTerminatingBranch(int s,int e)
{
   if(s< 0 || e> =100)
   {
      return;
   }
   int sum = 0 ;
   int ara[100];
   int i;
   for(i=s;i< =e;i++)
   {
      ara[i] = s ;
   }
   for(i=s;i< =e;i++)
   {
      printf("%d\n",ara[i]);
   }
}
```

```
int checkOddEven(int num)
{
   if(num%2==0)
   {
      return 0;
   }
   return 1;
}
int main()
{
   int a = 10;
   int testAra[1000];
   int temp = 0 ;
   int i,j,k;
   for(i=0;i< 5;i++)
   {
      for(j=0;j< 5;j++)
         {
         temp += checkOddEven(j);
         testAra[i] = temp ;
         }
      }

   FuctionTerminatingBranch(2,20);
```

Figure 3.12: Output After Applying H1, H2, H3 in Source Code (1)

```
#include <stdio.h>
void fun()
{
   int i = 0;
   i = 1;
   i = 2;
   return;
}

void fun2()
{
   int i ;
   for(i = 0 ; i <= 100000 ; i++)
   {
      int a = i * 2;
   }
}
int main()
{
   int i,j,k;
   for(i = 0 ; i < 5 ; i++)
   {
      for(j = 0 ; j< 10 ; j++)
      {
         int a = i + j;
      }
   }
   for(i = 0 ; i < 1000000 ; i++)
   {
      int a = i + i;
   }
   k = 10;
   if(k > 0)
   {
      for(i = 0 ; i < 100 ; i++)
      {
         j = j +1;
      }
   }
  if(k == 10)
   {
      for(i = 0 ; i < 500 ; i++)
      {
         j = j + 1;
      }
   }
   fun();
   fun2();
   return 0;
}
```

Figure 3.13: Output After Applying H4 in Source Code (2)

3.7 Summary

So far, we have developed our own methodology, which detects the most error-prone segment in a source code. In this methodology, we tried our best to implement and detect code segments that are most likely to cause an error. Our method is far from perfect, but we have studied other research papers and gradually improved it.

Chapter 4

Experimental Analysis and Results

4.1 Introduction

We have implemented our proposed method in the previous chapter. In this chapter, we will calculate the result for different source codes and discuss the result elaborately, and explain why we are getting such type of result. We will use our method to obtain our result, and finally, we will evaluate our result with other methods to see how our method performs compared to other methods.

4.2 Experimental Tools

1. CPU: A central processing unit (CPU), which is commonly known as a processor, is an electronic circuit that executes instructions that consists of a computer program. It can execute different operations like basic arithmetic, logic, controlling, and io operations written in computer programs though specialized processors like GPU or graphics processing units don't do these kinds of operations. In spite of evolving significantly over time, the core components of a CPU remain almost the same.

2. RAM: Random access memory or known as Ram, is a type of computer memory that can be read and written in any order. RAM is known as a volatile kind of ram because it cannot store data permanently. The main function of ram is to store concurrent data and applications. A special attribute of ram is that the data can be read or written from any position with little fluctuation in time.

3. Programming Language: The language which is used for composing programs is known as a programming language. The language can be a graphical language, or it can be a text language. Programming language can be divided into two main parts. One of them is syntax, and another one is semantics. Machine or assembly language can execute instructions directly and is often referred to as a low-level language, while the languages which need some translations are

referred to as high-level language. In spite of having a lot of programming languages, very few are actually used in practice.

4. Editor: A source-code editor is a program that is designed for conveniently editing a computer program. It can be a completely separate program on its own, or it can be built right into a web browser or an IDE (Integrated Development Environment). Source code editors are crucial for writing and editing source code. They offer a lot of features like syntax highlighting, indentation, quick fixes, etc., and at the same time, provide an easy pathway to run a compiler, interpreter, or debugger.

The specification of our experimental system is stated below.

- **CPU:** Intel Core I5-4200U @ 1.60 GHz
- **RAM:** 8GB 1600MHz
- **LANGUAGE:** Python 3.6.8, C, C++11, HTML
- **EDITOR:** VS Code, Code blocks

4.3 Experimental Analysis

4.3.1 Execution of a Program

Before analyzing our experiment result, we have to consider the execution of a simple program. Before the execution, the program goes through a number of steps that are integral for program execution. There are some tools needed to make a program run. These tools are:

- Preprocessor
- Compiler
- Linker
- Loader

Preprocessor: This is the first stage of the program execution. The preprocessor operates the programs before compilation. It also extends the macros and also prepares the header files included in a c program.

Compiler: This is the second step in program execution. In this stage, the compiler outputs an object file after preprocessing. Compilers mainly check for syntax errors. If there is no error found in the source code, then the compiler generates the object file.

Linker: This is the third stage of program execution. In this step, the linker links the other packages or libraries together and generates the executable file or .exe file.

Loader: This is the final stage of the program, where the loader loads the executable file into the ram temporarily and executes the program.

After loading the program into the RAM, the program execution cycle proceeds as follows:

- The CPU outputs the address of the memory location containing the required instruction from the program, but it is output in binary form on the address lines from the processor. The address decoder logic uses the address to select the RAM chip that has been allocated to this address. The address bus also connects directly to the RAM chip to select the individual location, giving a two-stage memory location selection process.
- The instruction code is returned to the CPU from the RAM chip via the data bus. The CPU reads the instruction from the data bus into an instruction register. The CPU then decodes and executes the instruction. The operands are fetched from the following locations in RAM via the data bus in the same way as the instruction.
- The instruction execution continues by feeding the operands to the data processing logic. Additional data can be fetched from memory. The result of the operation is stored in a data register and then, if necessary, in memory for later use. In the meantime, the program counter has been incremented to the address of the next instruction code. The address of the next instruction is then output, and the sequence repeats from the previous step.

4.3.2 Result Calculation

When a statement of a program is executed, an instruction code is generated, which is stored in the instruction register. The CPU then decodes the instruction and executes it. The operands are fetched from the different locations in the RAM via the data bus. The key point here is the instruction length depends on the architecture of the system. If it is a 32-bit system, then the length of the instruction will be 32-bit, and if the system is 64 bits, the instruction will be 64-bit long. As our test system has 64-bit architecture, the instruction will be 64-bit long.

As this is binary code, then all possible combination for this instruction is 2^{64} for each of the instructions. Hence, if we have **N** instructions, then all possible combinations for the N instructions will be,

$$Possible\ Combinations = 2^{N*64}$$

Which indicates the maximum ways we can inject fault in the registers. Hence, if we can count the bits in a specific segment, we can calculate the bit percentage of that segment relative to the whole c program. Then the bit percentage for a specific segment that has **R** instructions will be,

$$Bit\ Percentage, P = \frac{2^{64*R}}{2^{64*N}}$$

We wanted to use this equation for calculating the bit percentage, which represents the probability of being the error-prone segment, but we couldn't because when the value of the N » R then the formula returns zero; that is why we used this formula instead,

$$Bit\ Percentage, P = \frac{R}{N}$$

By using this formula, we get the outputs which are shown in the following:

1. Application of 1st Heuristic (H1)

When we execute our proposed method on the "Input Source Code (1)", we get the following output. This shows that the 1st conditional block has a bit percentage of about 11% and the 2nd conditional block has a bit percentage of 1.63%. The result is calculated from the formula we have introduced in our methodology, which calculates the bit percentage of each segment relative to the whole program. The first conditional block is executed far more than the second conditional block. Hence, the bit percentage for the first block is larger than the second one.

Table 4.1: Result after Applying 1st heuristic

Block	Bit Percentage

`if(num%2==0)`
`{`
`  return 0;`
`}`
	11.4130%

`if(a == 10)`
`{`
`  int b = 10;`
`  int c = b + 1;`
`  int s = b + c;`
`}`
	1.6304%

Table 4.2: Bit Percentage Usage of If-Else-If Operations

Line Segments Ranges Of Source Code(1)	Bit Percentage
28 ~ 31	11.41304347826087 %
71 ~ 76	1.6304347826086956 %

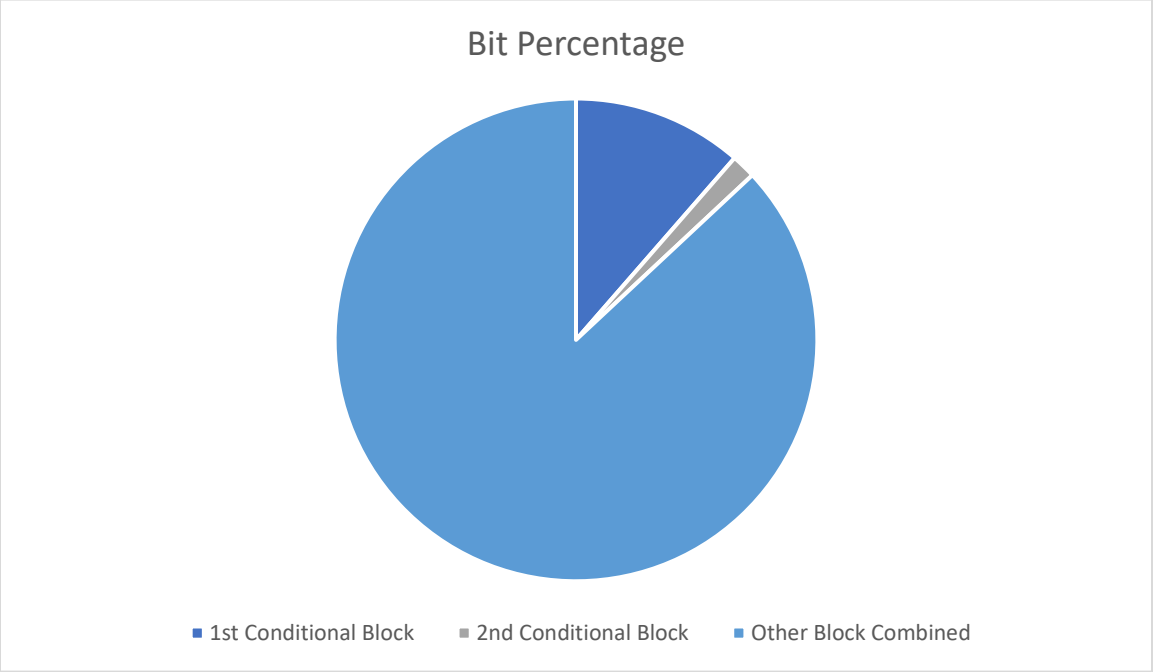

Figure 4.1: Bit Percentage Distribution for If-ElseIf-Else Segments

2. Application of 2nd Heuristic (H2)

When we execute our proposed method on the "Input Source Code (1)", we get the following output. This shows that the 1st loop block has a bit percentage of about 41%, 2nd loop block has a bit percentage of 10%, 3rd loop block has a bit percentage of 10%, the 4th loop block has a bit percentage of 7%, 5th loop block has the bit percentage of 24%. The result is calculated from the formula we have introduced in our methodology, which calculates the bit percentage of each segment relative to the whole program. The first loop segment has the greatest number of statements executed in the program.

Table 4.3: Result after Applying 2nd heuristic

Block	Bit Percentage

`for(i=0;i<5;i++)`
`{`
`  for(j=0;j<5;j++)`
`  {`
`  temp += checkOddEven(j);`
`  testAra[i] = temp ;`
`  }`
 	
`}`
	40.7608%

`for(i=s;i<=e;i++)`
`{`
`  ara[i] = s ;`
`}`
	10.3260 %

`for(i=s;i<=e;i++)`
`{`
`  printf("%d\n",ara[i]);`
`}`
	10.3260 %

`while(z--)`
`{`
`  int t = 10;`
`}`
	7.0652%

`for(i=0;i<3;i++)`
`{`
`  for(j=0;j<3;j++)`
`  {`
`  int a = checkOddEven(j);`
`  for(int k=0;k<3;k++)`
`  {`
`  printf("%d %d %d\n",i,j,k);`
`  }`
`  }`
`}`
	24.4565%

Table 4.4: Bit Percentage Usage of Loop Operations

Line Segments Ranges Of Source Code(1)	Bit Percentage
42 ~ 49	40.76086956521739 %
16 ~ 19	10.326086956521738 %
20 ~ 23	10.326086956521738 %
53 ~ 56	7.065217391304348 %
59 ~ 69	24.456521739130434 %

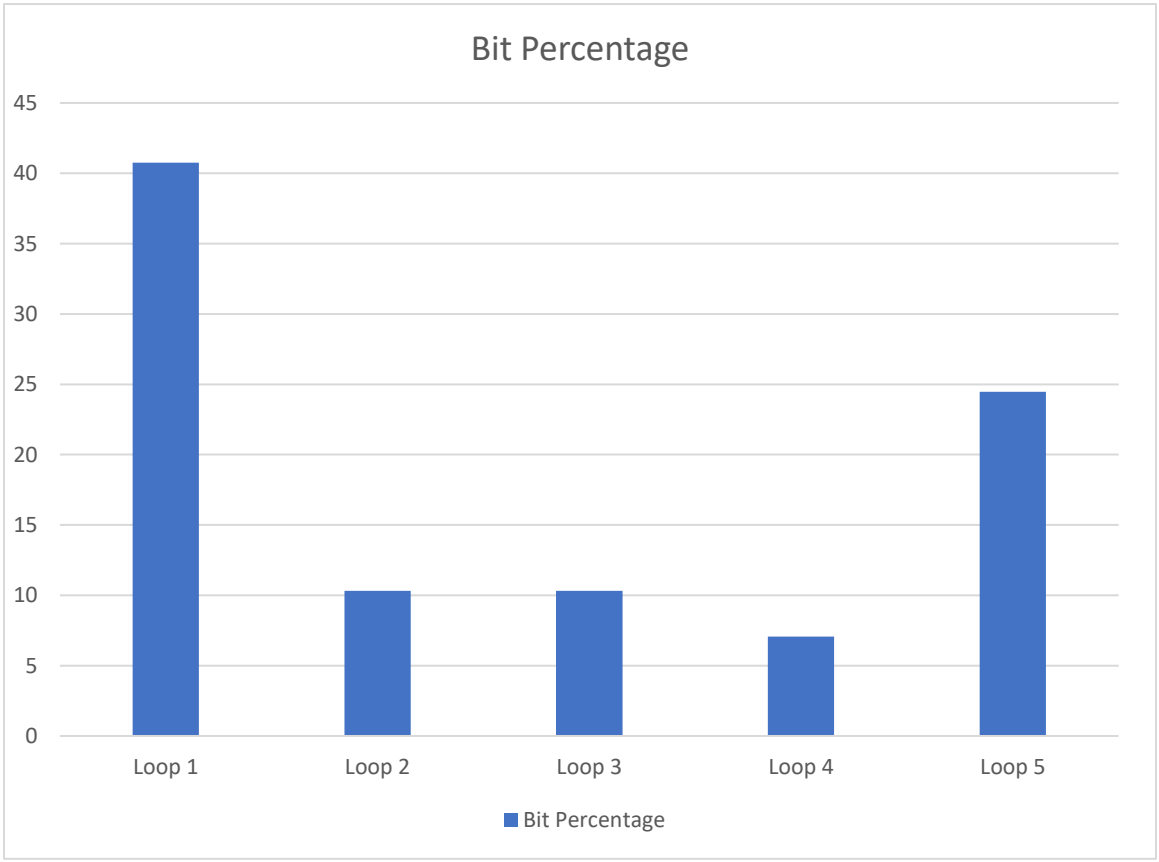

Figure 4.2: Bit Percentage Distribution for Loop Segments

3. Application of 3rd Heuristic (H3)

When we execute our proposed method on the "Input Source Code (1)", we get the following output. This shows that the 1st functional segment has a bit percentage of about 18%, and the 2nd functional segment has a bit percentage of about 22%. The result is calculated from the formula we have introduced in our methodology, which calculates the bit percentage of each segment relative to the whole program. The second functional segment has the greatest number of statements executed in the program.

Table 4.5: Result after Applying 3rd Heuristic (Function)

Block	Bit Percentage

`int checkOddEven(int num)`
`{`
`  if(num%2==0)`
`  {`
`  return 0;`
`  }`
`  return 1;`
`}`
	18.4782%

`void FuctionTerminatingBranch(int s,int e)`
`{`
`  if(s<0 || e>=100)`
`  {`
`  return;`
`  }`
`  int sum = 0 ;`
`  int ara[100];`
`  int i;`
`  for(i=s;i<=e;i++)`
`  {`
`  ara[i] = s;`
`  }`
`  for(i=s;i<=e;i++)`
`  {`
`  printf("%d\n",ara[i]);`
`  }`
`}`
	22.2826%

Table 4.6: Bit Percentage Usage of Functional Operations

Line Segments Ranges Of Source Code(1)	Bit Percentage
26 ~ 33	18.478260869565215 %
6 ~ 24	22.282608695652172 %

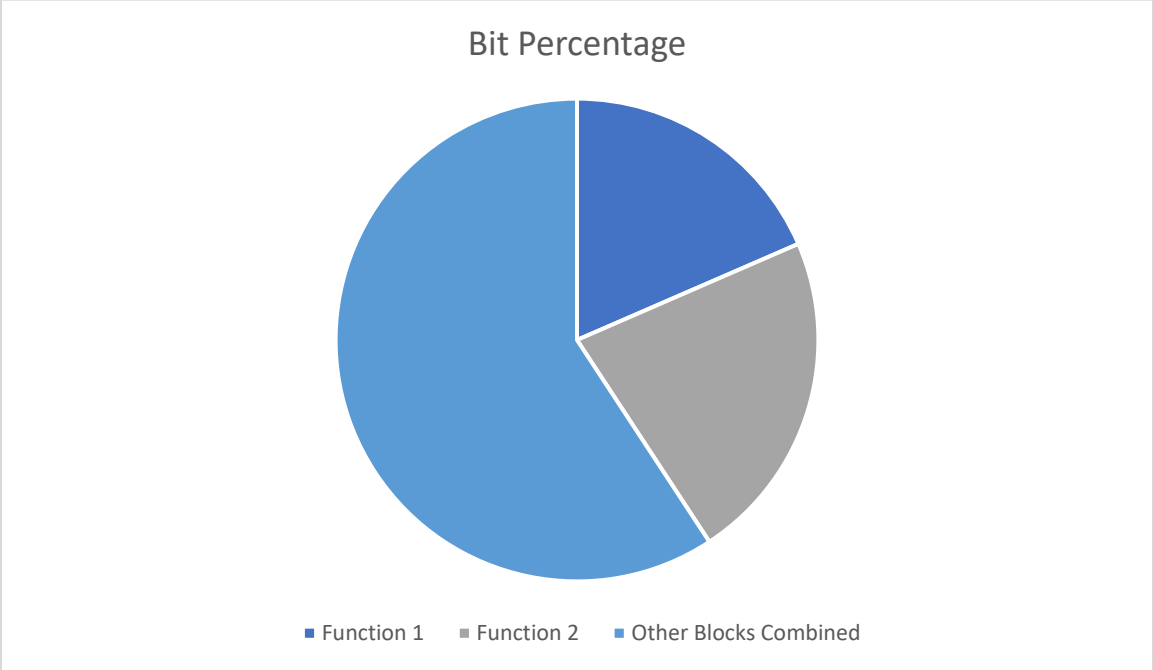

Figure 4.3: Bit Percentage Distribution for Functional Segments

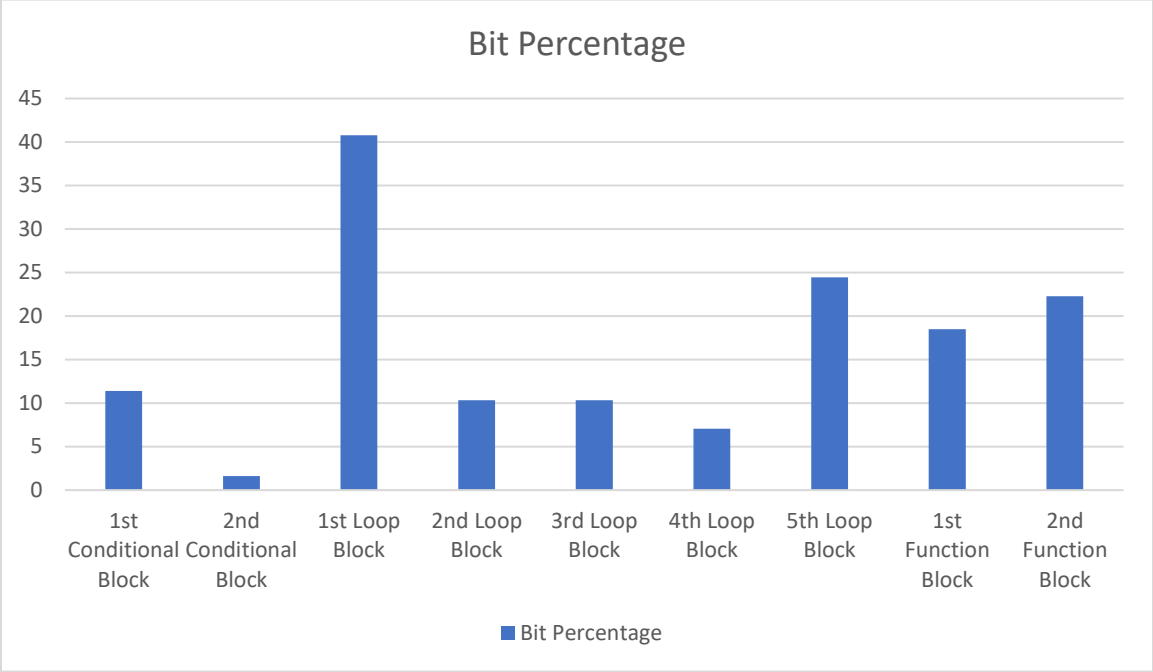

Figure 4.4: Bit Percentage Distribution for H1, H2, H3

4. Application of 4th Heuristic(H4)

When we execute our proposed method on the "Input Source Code (2)", we get the following output. This heuristic has a special use case. We can only use it when there is a large number of executed statements difference ($d > 10^6$) in the input code among the segments. Hence, since this input source code doesn't have that big of a margin between the numbers of executed statements. That is why the output is zero because it takes a very short time to execute the statements.

Table 4.7: Results after Applying 4th Heuristic (If-ElseIf-Else)

Block	Time Percentage

`if(k > 0)`
`{`
`  for (i = 0 ; i < 100 ; i++){`
`  j = j + k;`
`  }`
`}`
	0.0%

`if(k == 10)`
`{`
`  for(i = 0 ; i< 500 ; i++)`
`  {`
`  j = j + k;`
`  }`
`}`
	0.0%

Table 4.8: Time Percentage Usage of If-ElseIf-Else Operations

Line Segments Ranges Of Source Code(2)	Bit Percentage
33 ~ 39	0.0 %
40 ~ 46	0.0 %

Table 4.9: Results after Applying 4th Heuristic (Loop)

Block	Time Percentage

`for(j = 0 ; j < 10 ; j++)`
`{`
`  int a = i + j;`
`}`
	0.0%

`for(i = 0 ; i < 5 ; i++)`
`{`
`  for (j = 0 ; j < 10 ; j++)`
`  {`
`  int a = i + j;`
`  }`
`}`
	0.0%

`for(i = 0 ; i < 1000000 ; i++)`
`{`
`  int a = i + i ;`
`}`
	75.1245%

`for(i = 0 ; i < 100 ; i++)`
`{`
`  j = j + i;`
`}`
	0.0%

`for(i = 0 ; i < 500 ; i++)`
`{`
`  j = j + i;`
`}`
	0.0%

`for(i = 0 ; i <= 100000 ; i++)`
`{`
`  int a = i * 2;`
`}`
	24.8754%

Table 4.10: Time Percentage Usage of Loop Operations

Line Segments Ranges Of Source Code(2)	Bit Percentage
23 ~ 26	0.0 %
20 ~ 27	0.0 %
28 ~ 31	100.0 %
35 ~ 38	0.0 %
42 ~ 45	0.0 %
13 ~ 16	0.0%

Table 4.11: Results after Applying 4th Heuristic (Function)

Block	Time Percentage

`void fun()`
`{`
`  int i = 0 ;`
`  i = 1;`
`  i = 2;`
`  return;`
`}`
	0.0%

`void fun2()`
`{`
`  int i;`
`  for(i = 0 ; i<= 100000 ; i++)`
`  {`
`  int a = i *2;`
`  }`
`}`
	24.8755%

Table 4.12: Time Percentage Usage of Functional Operations

Line Segments Ranges Of Source Code(2)	Bit Percentage
3 ~ 9	0.0 %
10 ~ 17	0.0 %

4.4 Discussion

After calculating the result and analysis, we can see that our proposed method outputs the convenient probability of a segment being erroneous. Our method isn't perfect, but it outputs a close enough result, from which we can assume the potential error may arise. This method can be improved further to calculate a better result.

Chapter 5

Conclusions

5.1 Concluding Remarks

This thesis focuses on the detection of the most error-prone segment detection in a c program. The methodology we proposed here can detect the most error-prone segments and highlight them according to the heuristics we have established in the methodology. Our methodology is far from perfect and there are a lot of situations when our methodology cannot be implemented or the output is not accurate. For that, we need to improve our methodology to make it a better way of detection and overall to make it a better debugging tool in real-life use cases.

In this research, we have tested our methodology with some sample input codes. We intended to make a methodology that can automatically detect and highlight the segment which is the most susceptible to causing an error in a program written in C language. We can improve upon this by spending more time and labor on this.

5.2 Future Work

We have implemented this methodology to detect and highlight sensitive segments in a c program. It involves checking whether the statement they are in a certain block or not. This requires quite some time if the input sample program is large and complex. We like to study more and research to improve this aspect of the methodology in the future.

References

[1] E. H. Forman and N. D. Singpurwalla, “An empirical stopping rule for debugging and testing computer software,” *J Am Stat Assoc*, 1977, doi: 10.1080/01621459.1977.10479951.

[2] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?,” in *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings*, 2011. doi: 10.1145/2001420.2001445.

[3] V. R. Basili and B. T. Perricone, “Software errors and complexity: an empirical investigation0,” *Commun ACM*, vol. 27, no. 1, pp. 42–52, Jan. 1984, doi: 10.1145/69605.2085.

[4] Q. Zhang, M. Cao, T. Wang, and J. Chang, “Programming Error Repair Guidance Based on Historical Learning Behavior,” in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, Oct. 2019, pp. 1–7. doi: 10.1109/ICSESS47205.2019.9040686.

[5] D. Steiner and P. Puschner, “Error detection based on execution-time monitoring,” in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, Jun. 2017, pp. 1–5. doi: 10.1109/MECO.2017.7977166.

[6] G. Samara, “A Practical Approach for Detecting Logical Error in Object Oriented Environment,” Dec. 2017.

[7] R. McCauley *et al.*, “Debugging: a review of the literature from an educational perspective,” *Computer Science Education*, vol. 18, no. 2, pp. 67–92, Jun. 2008, doi: 10.1080/08993400802114581.

[8] R. W. Selby and V. R. Basili, “Analyzing error-prone system structure,” *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 141–152, 1991, doi: 10.1109/32.67595.

[9] J. Verner, J. Sampson, and N. Cerpa, “What factors lead to software project failure?,” in *2008 Second International Conference on Research Challenges in Information Science*, Jun. 2008, pp. 71–80. doi: 10.1109/RCIS.2008.4632095.

[10] I. H. Sarker, F. Faruque, U. Hossen, and A. Rahman, “A Survey of Software Development Process Models in Software Engineering,” *International Journal of Software Engineering and Its Applications*, vol. 9, no. 11, pp. 55–70, Nov. 2015, doi: 10.14257/ijseia.2015.9.11.05.

[11] D. S. Coutant, S. Meloy, and M. Ruscetta, “DOC: a practical approach to source-level debugging of globally optimized code,” in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI ’88*, 1988, pp. 125–134. doi: 10.1145/53990.54003.

[12] X. Wang, L. Pollock, and K. Vijay-Shanker, “Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability,” in *2011 18th Working Conference on Reverse Engineering*, Oct. 2011, pp. 35–44. doi: 10.1109/WCRE.2011.15.

[13] Xiaojie Zhang and P. H. Siegel, “Efficient Algorithms to Find All Small Error-Prone Substructures in LDPC Codes,” in *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, Dec. 2011, pp. 1–6. doi: 10.1109/GLOCOM.2011.6133832.

[14] J. Ma, Z. Duan, and L. Tang, “A Methodology to Assess Output Vulnerability Factors for Detecting Silent Data Corruption,” *IEEE Access*, vol. 7, pp. 118135–118145, 2019, doi: 10.1109/access.2019.2936893.

[15] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2013. doi: 10.1109/DSN.2013.6575353.

[16] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2005. doi: 10.1109/HPCA.2005.37.

[17] G. Kestor, B. O. Mutlu, J. Manzano, O. Subasi, O. Unsal, and S. Krishnamoorthy, “Comparative analysis of soft-error detection strategies: A case study with iterative methods,” in *2018 ACM International Conference on Computing Frontiers, CF 2018 - Proceedings*, 2018. doi: 10.1145/3203217.3203240.

[18] L. Johnson and D. C. Pheanis, “Automated error-prevention and error-detection tools for assembly language in the educational environment,” in *Proceedings - Frontiers in Education Conference, FIE*, 2006. doi: 10.1109/FIE.2006.322560.

[19] M. Glukhikh, M. Moiseev, A. Karpenko, and H. Richter, “Software reliability estimation based on static error detection,” in *2011 7th Central and Eastern European Software Engineering Conference, CEE-SECR 2011*, 2011. doi: 10.1109/CEE-SECR.2011.6188470.