Appendix C

Tutorials

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 1

NOTE You are expected to prepare for this tutorial by sketching answers to the tasks and questions before attending the tutorial.

What is Testing Really?

"*It works! Trust me, I've tested the code thoroughly!*"

. . . and with those bold words many software projects have been made bankrupt.

The aim of this tutorial is to start to get an understanding of our major themes; that is, of the factors that make up quality and how testing effects them. While we have not yet looked at testing formally yet you will be asked in this tutorial to test a small program fragment with the purpose of starting to see where the difficulties of testing lay in practice.

As a first step, let's discuss what testing is, and why we would want to perform testing on our software.

Important Terminology

Before beginning the exercises, consider the following four definitions, the first three of which are defined in Chapter 1 of the lecture notes:

Fault: An incorrect step, process, or data definition in a computer program. Faults are the source of failures – a fault in the program triggers a failure under the right circumstances.

Failure: A deviation between the observed behaviour of a program, or a system, from its specification.

Error: An incorrect *internal* state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

Failures and errors are the result of faults – a fault in a program can trigger a failure and/or an error under the right circumstances. In normal language, software faults are usually referred to as "bugs", but the term "bug" is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

The Programs

The following are taken from the exercises in Chapter 1 of *Introduction to Software Testing* by Offutt and Ammann. For each program, the test case below the program results in a failure that is caused by a fault in the program.

```
/**
 * If x == null throw NullPointerException.
 * Else return the index of the last
 * element in x that equals y.
 * If no such element exists, return -1
 */
public int findLast (int[] x, int y)
{
   for (int i = x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}

//Test
//Input: x = [2, 3, 5]; y = 2
//Expected output = 0
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the index of the LAST 0 in x.
 * Return -1 if 0 does not occur in x.
 *
 */
public static int lastZero (int[] x)
{
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}

//Test
//Input: x = [0, 1, 0]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of positive
 * elements in x.
 */
public int countPositive (int[] x)
{
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}

//Test
//Input: x = [-4, 2, 0, 2]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of elements in x
 * that are either odd or positive (or both)
 */
public static int oddOrPos(int[] x)
{
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i]%2 == 1 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}

//Test
//Input: x = [-3, -2, 0, 1, 4]
//Expected output = 3
```

Your tasks

For each of the programs, perform the following tasks:

1. Specify the input domain and valid input domain of the function.
2. Identify the fault.
3. If possible, identify a test case that does not execute the faulty statement.
4. If possible, identify a test case that executes the faulty statement, but does not result in an failure.
5. If possible identify a test case that forces the program into an error (that is, the first step that a program deviations from its intended behaviour), but does not produce a failure.
6. Fix the fault and verify that the given test now produces the expected output.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 2

Introduction

The aim of this tutorial is twofold. First, it aims to give you some practise at deriving test cases from specifications. Second, it aims for you to start exploring the limits of your test cases, and of the specifications.

The Program

Introduction: LWIG (Less Work Is Good) is a new marking program developed for the department of Information Programming.

The LWIG program takes a .in file as an argument and produces a sorted .out file. The program assumes that the .in file contains at least one student record.

Input File: The input file format is as follows. Each line will contain the data for a single student, and will contain several fields. Each field is separated by a colon.

Each line consists of the following fields, in order:

- A student number, which must be a 5 digit, 6 digit or 9 digit number.
- The student's month of birth, which must be a string of 3 alphabetic characters with the first character capitalised and the remaining as lower case. That is, it must be from the set:

$$\{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec\}$$

- The date of birth, which must be a number from 1–31 and must be the correct number of days for the month; that is, it cannot be 30 February, because February never has 30 days.
- The student's surname, which must be a string of alphabetic characters all in capitals.
- The first letter of the student's first name, which must be a single capitalised alphabetic character.
- The number of lectures that they slept through, which must be an integer between 0 and 24.

If any input row is invalid, the program should print a warning message and continue with the next record. If the program encounters a more serious problem (e.g. unable to open input file), it will print an error message and exit gracefully.

Example 1 As an example suppose we had the following data.

- Student number: 12345
- Month of Birth: May

- Date of Birth: 26
- Surname: CHAN
- First letter of first name: K
- Slept during one or more lectures: 0

The input line for this data would be `12345:May:26:CHAN:K:0`

Output File: The output file format is as follows. Each line will contain the data for a single student, and each line will contain several fields. Each field is separated by a colon.

Each line consists of the following fields, in order:

- Student number which must be a 5 digit, 6 digit, or 9 digit number.
- Assignment 1 mark: (0–10)
- Assignment 2 mark: (0–10)
- Project mark: (0–30)
- Exam mark: (0-50)
- Final grade: [H1, H2A, H2B, H3, P, HCP, N, N+]
- Comment: String of arbitrary length

Example 2 As an example, consider the following output:

- Student number: 12345
- Assignment 1 mark: 10
- Assignment 2 mark: 10
- Project mark: 25
- Exam mark: 38
- Final grade: H1
- Comment: "Excellent."

The output for this record would be `12345:10:10:25:38:H1:‘‘Excellent’’`

Tasks

Question 1

What is the input domain for the LWIG program? What are the input conditions for the LWIG program?

Question 2

Derive input test-cases for the program using equivalence partitioning and boundary-value analysis.

Question 3

Of course, the client has not completely specified the program (but, that is to be expected). There is an additional requirement that the records can be sorted by different output fields.

What are the implications of sorting on the various fields and what test cases would you choose to ensure that sorting has been correctly implemented?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 3

Introduction

The purpose of this tutorial is for you to gain a deeper understanding of control- and data-flow techniques, and to compare these to input partitioning techniques.

Talking About Sets

There are still questions about how to characterise, or describe, input/output domains and what level of description and how many words are necessary.

You should aim to get used to characterising sets of values using proper set notation - we are after all, dealing with sets. The set notation does not need to be completely formal for this subject, but we will encourage and expect a mathematical discourse as the subjects goes on and in the exam.

To get the feeling for the level of discourse required read the tutorial and assignment answers on the web-page. Here are some examples of what we expect.

Example 1 Start by giving meaning to all of the (mathematical) variables that you intend to use and then use them to describe the input domain (i.e the set of all inputs that you will use for testing).

(i) Let string be the set of alpha-numeric strings, $\mathsf{str} \in \mathsf{string}$ be any individual alpha-numeric string, and length be a function that returns the length of a string. The input domain is then the set string and the input condition is that

$$\text{for all } \mathsf{str} \in \mathsf{string} \ \bullet 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Equivalently we could have written the input condition as:

$$\forall\, \mathsf{str} \bullet \mathsf{str} \in \mathsf{string} \implies 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Both are acceptable.

(ii) Let int be the set of all 32-bit integers and string be the set of all strings of characters. The input domain is then the set of all pairs

$$\mathsf{Input} = \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{x} \in \mathsf{string} \text{ and } \mathsf{y} \in \mathsf{int}\}.$$

Working With the Program

In this tutorial we will focus on the procedure `bubble`, found in Figure C.1. The `main` program serves to show you how we intend to use `bubble`. Of course, `bubble` implements a bubble sort. The source code can be downloaded from the tutorials page of the LMS.

```
#include <stdio.h>
#define SIZE 10

void bubble(int data[SIZE], int size, int (*compare)(int, int));

main()
{
  int data[SIZE] = {11, 4, 8, 22, 15, 7, 8, 19, 20, 1};
  int counter, order;

  int up(int, int);
  int down(int, int);

  printf("Enter 0 to sort in ascending order and 1 to sort in descending order: ");

  scanf("%d", &order);

  if (order == 0) {
    bubble(data, SIZE, up);
  }
  else {
    bubble(data, SIZE, down);
  }

  for (counter = 0; counter < SIZE; counter++) {
    printf("%4d", data[counter]);
  }

  printf("\n");
}

void bubble(int *data, int size, int (*compare)(int, int))
{
  int pass, count;
  void swap(int *, int *);

  for (pass = 0; pass < SIZE - 1; pass++) {
    for (count = 0; count < SIZE - 1; count++) {
      if ((*compare)(data[count], data[count + 1])) {
        swap(&data[count], &data[count + 1]);
      }
    }
  }
}

void swap(int *A, int *B)
{
  int temp;

  temp = *A;
  *A = *B;
  *B = temp;
}
```

Figure C.1: An implementation of bubble sort

Make special note of the third parameter to `bubble`, which is a function parameter. This parameter allows us to create a flexible sorting algorithm, which can be used to sort in either ascending or descending order. Both functions take two integers and return an integer (which is intended to be a boolean value). For this tutorial consider just

```
int up(int A, int B) {
  return A < B;
}
```

and

```
int down(int A, int B) {
  return B < A;
}
```

as the possible comparison functions. Note that the functions `up` and `down` appear in the `main` function.

Another interesting aspect of both `bubble` and `main` below is that they declare *nested functions*. The functions `up` and `down` are declared within `main` and so can refer to any of the variables in `main` prior to their declaration, for example, they can refer to the variable `count`.

Your Tasks

Task 1

First, make sure you understand the program (expected to be done **before** the tutorial).

Task 2

Determine the input domains and output domains for the functions `bubble`. What is the set of possible functions that can be passed into the `bubble` function? Is this what you expected?

Next, what are the input conditions, and (perhaps a little more challenging) what are the output conditions?

Task 3

Create a control-flow graph for the `bubble` function. Ideally, should you include the functions `up` and `down` in your control flow graph?

Question 4

Do the branches in your control flow graph partition the input domain? Do the branches in your control flow graph partition the output domain?

Task 5

Perform a static data-flow analysis on `bubble` for the variables `data` and `size`.

Task 6

Design a set of black box test cases using whatever techniques you feel appropriate.

Question 7

What extra information does your data-flow analysis give you that your black-box test cases do not?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 4

Introduction

The aim of this tutorial is for you to familiarise yourself with the various coverage criteria and analysis of the program for the various coverage criteria. When you get back to your revision you should try comparing the test cases that you derive for a program using different techniques.

The different type of program that we encounter this week is a numerical program. One of the challenges of numerical programs is that we can never be certain that we will get an *exact* answer to our computation. Instead what we typically require is an answer to within some *error* value[1]. Numerical programs are tricky to debug, because they are often used to *find* the answer to some problem in the first place. For example, solving some integration or differentiation problems is too hard to do by hand and so we use a *numerical* method to approximate the answer.

Working With the Program

The program implements the standard bisection method for root finding. The root-finding problem is expressed as follows:

> We are given a function $f(x)$ taking a real number and returning a real number[2]. The function is negative at some point x_0 and positive at some point x_1. Find the value x for which $f(x) = 0$ on an interval $[Lower, Upper]$. The point x is a root of f on the interval $[Lower, Upper]$.

As an example, consider the natural logarithm function, ln. The graph in Figure C.2 shows the values of $ln(x)$ for various values of x. We can see that the value of $ln(x)$ is equal to 0 when $x = 1$. The bisection algorithm finds this value of x.

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $f(Lower)$ is negative, and $f(Upper)$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time). Does this sound familiar?

[1]Recall from the lecture notes that an error is the difference between a computed value and the exact value.
[2]That is a function $f : \mathbb{R} \to \mathbb{R}$.

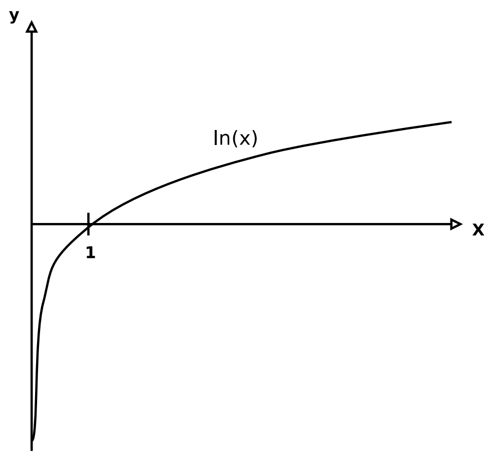

Figure C.2: Root finding problems.

Your Tasks

Question 1

What is the input domain for the `Bisection` program below?

Question 2

Draw the control-flow graph for the `Bisection` function. You may break the function up into basic blocks to simplify your CFG.

Recall that a *basic block* is a continuous sequence of statements where control flows from one statement to the next, a single point of entry, a single point of exit and no branches or loops.

Question 3

Suppose that we concentrated on the (nice and linear) function $f(x) = x - 2$. Derive a set of test cases that achieve:

(i) Statement coverage; and

(ii) Condition coverage.

Note, that you will have to determine what it means for the `Bisection` function to return the *correct* or *expected* output first.

The Program

```
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535

double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */

    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */

    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as f(Lower) or f(Upper) is easy:
       we know that f(Lower) is negative and we know that f(Upper) is positive.
       Multiple fr by f(Lower) and if the result is positive then fr must be
       negative. If the result is negative then fr must be positive. */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

    printf("iteration %d = (%f, %f, %f, %f, %f)\n", iteration,
            Lower, Upper, xr, ea, Sign);

  } /* end while */

  return xr;
}

int main()
{
  double f(double);

  double fx = Bisection(-1.0, 7.0, 1, 10, &f);
  printf("value = %f\n", fx);
}

double f(double x)
{
  return x - 2;
}
```

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 5

Introduction

Encapsulation is an abstraction mechanism that aids in programming, but that adds complexity to testing. We often need to break the information hiding utilised by classes in order to examine the class state for the testing purposes.

The aim of this tutorial is for you to explore some of the issues in object oriented testing through the simple Graph given below. The graph uses an adjacency matrix that records which vertices are "*adjacent*" in the graph.

Task 1

Consider the `addEdge` and `deleteEdge` methods in the `Graph` class below. Derive test cases for path coverage and condition coverage for these two methods. **Note** that it may be necessary to examine the state variables in your test cases. Sketch how you would achieve this.

Task 2

Draw a finite state automaton for `Graph` class. **Note** that it is not always possible to add an edge and it is not always possible to delete a vertex. You will need to consider the states and the guards on transitions to ensure all of the conditions.

Task 3

Derive a set of test cases to test every transition in your graph.

Question 4

Using 5 different mutant operators from Chapter 6 of the lecture notes, derive 5 mutants for the `addEdge` operation. Do the test cases derived from your finite state automaton kill all of the mutants? If not, are the live mutants equivalent, or are the test cases not yet adequate?

The Graph Abstract Data Type

The Graph Interface

```
class Graph
{
    public Graph(int n);
      // The constructor for graphs. It initialises
      // the graph state and sets up the graph
      // representation for graphs of n or less nodes.

    public void addVertex(int v);
      // Add a vertex to the vertex graph.

    public void addEdge(int m, int n)
      // Add an edge to the graph. Edges are specified by pairs
      // of vertices. To add the edge correctly it is necessary
      // that m and n have already been added to graph as vertices.

    public void deleteVertex(int v);
      // We can only delete a node if it is not part of some edge
      // in the graph and it exists as an actual vertex in the
      // graph.

    public void deleteEdge(int m, int n);
      // We can only delete an edge if the two specified
      // vertices, m and n, are in the graph.
}
```

The Graph State and Hidden Functions

```
//------------ Private Attributes ----------------------------
//    The representation of a graph consists of an array
//    of vertices that map nodes (just integers) to array
//    indexes. The adjacency matrix _matrix sets matrix[i][j]
//    to true if there is an edge between _vertices[i] and
//    _vertices[j].
//
//    The operations must maintain the following invariant:
//        _vertices[i] is defined iff i < _allocated
//        _allocated <= _order

private int     _order;      // The number of vertices allowed
private int     _allocated;  // The next free space in the vertex array
private int     _vertices[]; // A list of the actual vertices
private boolean _matrix[][]; // The adjacency matrix

//------------- Private Methods ------------------------------

private static boolean[][] _allocate(int n)
{
  return new boolean[n][n];
}

private static int[] _vertices(int n)
{
  return new int[n];
}

private int _lookup(int m)
{
  int index = 0;
  while (index < _allocated && !(_vertices[index] == m))
    index = index + 1;

  if (index == _allocated) {
    return _order + 1;
  }
  else {
    return index;
  }
}
```

Adding and Deleting Edges

```
public void addEdge(int m, int n)
{
  // Add an edge to the graph. Edges are specified by pairs
  // of vertices. To add the edge correctly it is necessary
  // that m and n have already been added to graph as vertices.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order)
  {
    _matrix[mIndex][nIndex] = true;
    _matrix[nIndex][mIndex] = true;
  }
}

public void deleteEdge(int m, int n)
{
  // We can only delete an edge if the two specified
  // vertices are in the graph.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order) {
    _matrix[mIndex][nIndex] = false;
  }
}
```

The University of Melbourne
SWEN90006: Software & Security Testing
Tutorial 6

Introduction

The aim of this tutorial is for you to familiarise yourself with the test oracles and random testing. This tutorial will help to gain an understanding of how to specify a test oracle to decide whether an arbitrary test case is correct, incorrect or undecidable, as well as how to randomly generate test cases that fit an operational profile.

Your Tasks

Repeat these tasks for both programs.

Task 1

Standard analysis: Determine the input/output domains and the input/output conditions.

Task 2

Use the specification, domains and conditions to determine if an arbitrary test input is correct, incorrect or otherwise undecidable.

Task 3

Determine an automated means for generating random test cases by selection points from the input set.

Working With the Programs

We shall start this tutorial by using a rather simple example to illustrate the concepts, techniques and issues faced when working with oracles. This is followed by applying these techniques to the root finding program from tutorial 4 to gain some practice with practical examples.

First Program

This *toLower* is an application which takes a string on standard input and puts a corresponding string on standard output, with all upper case letters changed to the matching lower case letters (i.e. ’A’ to ’a’, ’B’ to ’b’, etc.).

Strings may consist of all ASCII characters between 32 and 126 inclusive; characters outside this range are control characters and will cause an error message.

toLower Program

```
#include <stdio.h>
#include <string.h>

#define MAX_STRING 80

int main(int argc, char* argv[])
{
  char string[MAX_STRING];
  int i;
  while (fgets(string, MAX_STRING, stdin)) {
    for (i = 0; i < strlen(string) - 1; i++) {
      if (string[i] < 32 || string[i] == 127) {
        printf("Illegal character found.\n");
        return 1;
      }
      /* The upper case letters have ASCII codes
       * between 65 and 90 (90 to 65 = 26 letters)*/
      if (string[i] >= 65 && string[i] <= 90) {
        string[i] = string[i] + 'a' - 'A';
      }
    }
    fputs(string, stdout);
  }
  return 0;
}
```

Second Program

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $Lower$ is negative, and $Upper$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time).

Root Finding Program: A Simple Interval Bisection Method for Finding Roots

```
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535

double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */
    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */
    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as Lower or Upper is easy:
       we know that Lower is negative and we know that Upper is positive.
       Multiple fr by Lower and if the result is positive then fr must be
       negative! If the result is negative then fr must be positive! */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

  } /* end while */

  return xr;
}
```

THE UNIVERSITY OF MELBOURNE

SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 7

NOTE You are expected to prepare for this tutorial by sketching answers to the tasks and questions before attending the tutorial.

Introduction

The aim of this tutorial is to get you thinking about software security and vulnerabilities, and the applicability of different kinds of security testing.

As a first step, think about what security testing is, and why we would want to perform security testing on our software.

The Bitmap File Format

BMP is an historical image file format that we will use in this tutorial. We will consider a simple class of BMP files whose format is as follows. (Specifically, we consider here BMP files with no compression, and in which each pixel is 32-bits wide in order to avoid issues of padding; see `http://www.fastgraph.com/help/bmp_header_format.html` and `https://en.wikipedia.org/wiki/BMP_file_format` for more details.)

Offset	Size (in bytes)	Description
0	1	first byte of signature, must be 0x42 (the ASCII character ‘B’)
1	1	second byte of signature, must be 0x4D (the ASCII character ‘M’)
2	4	size of the BMP file in bytes (unreliable, ignored)
6	2	Must be zero
8	2	Must be zero
10	4	Must be the value 54 (i.e. 0x00000036)
14	4	Must be the value 40 (i.e. 0x00000028)
18	4	*Width* (image width in pixels, as signed integer)
22	4	*Height* (image height in pixels, as signed integer)
26	2	Must be one
28	2	Number of bits per pixel (must be 32)
30	4	Compression type (must be 0 = no compression)
34	4	Size of image data in bytes (must be 4**Width***Height*)
38	4	unreliable (ignored)
42	4	unreliable (ignored)
46	4	Must be zero
50	4	Must be zero
54	4**Width***Height*	Pixel data, laid out in rows

The first byte (offset 0) of a valid BMP file is the character ’B’; the second byte (offset 1) is the character ’M’. The 3rd to 6th bytes (offsets 2 to 5 inclusive) indicate the total length of the BMP file but are unreliable in practice and so let us assume that they are ignored by all BMP parsing code. The 7th and 8th bytes (offsets 6 and 7) are interpreted as a 2-byte integer that must be zero, i.e. each of these bytes must be zero. The same is true for the 9th and 10th bytes (offsets 8 and 9), and so on.

Your Tasks

Question 1

Imagine you are choosing a value for each of the fields in the table above *in order*, i.e. you first choose a value for the first byte of the file, then choose a value for the second byte of the file, then for following 4-bytes, and so on. For each field, identify the total number of valid values there are to choose from, assuming you have already chosen values for all fields that have come before.

Question 2

The BMP header (i.e. everything excluding the pixel data) as described above has a fixed length of 54 bytes. Using the answer from the previous question or otherwise, what is the probability that a (uniformly) randomly generated string of 54 bytes is a valid BMP header?

Question 3

Suppose you have a valid 54-byte header and you mutate an arbitrary (uniformly randomly chosen) byte in the header to a new value (different from its original value). What is the probability of producing a valid header?

Question 4

Imagine you had to write a fuzzer to fuzz some BMP processing code that processed BMP files of the format described above. If you had to choose between generating completely random inputs vs. using random mutation on existing BMP files, which strategy would you choose?

Question 5

Suppose you are fuzzing some C code that reads BMP files and you observe that for some large set of inputs generated by your fuzzer that the code crashes some number N of times. What would happen if you ran the code on the same inputs but now under a memory debugger? For example, imagine you recompile the BMP parsing code using the ASan (Address Sanitizer) tool demonstrated in lectures and ran it again on the same fuzzer-generated inputs as before. Would you expect the number of crashes observed now to exceed N, be the same, or be less than N? Explain why.

The University of Melbourne
SWEN90006: Software & Security Testing
Tutorial 8

Introduction

The aim of this tutorial is to give you more experience with symbolic execution and dynamic symbolic execution, and how they can be used to verify software.

(Dynamic) Symbolic Execution

Recall from the lecture that the aim of symbolic execution is to, in effect, execute *multiple* inputs at the same time – possibly an infinite number. It does this by uses symbolic values to represent inputs, instead of single concrete values.

A complete symbolic execution of a program produces a *symbolic execution tree*. Each path through the tree represents a single path through the program, including all inputs on that path. Each node along a path represents the symbolic values of variables at that point in the execution.

Dynamic symbolic execution, on the other hand, follows the path taken by a concrete test input, and collects the relevant constraints along the way. It uses these constraints to generate tests that take new paths in the program.

Tasks and Questions

Question 1

Figure C.3 shows an implementation of a program that returns the minimum of two integers. Draw the complete symbolic execution tree for this program. In your execution tree, use the variables `X_0` and `Y_0` to represent the initial symbolic values of `x` and `y` respectively.

At the return statement at line 10, assume the existence of a symbolic variable `RET`, to which the return value is assigned when a return statement is executed.

Question 2

Does the program in Figure C.3 ensure the precondition that the number returned is always the minimum; i.e. that `RET` is the minimum number of `X_0` and `Y_0`?

Question 3

Consider the program in Figure C.4 for counting the number of negative integers in an array. Assume that we run a *dynamic symbolic execution* (DSE) tool on this program, with the initial test being `x = [0,0]`.

```
 1.  int min(int x, int y)
 2.  {
 3.      int minimum = 0;
 4.      if (x > y) {
 5.          minimum = y;
 6.      }
 7.      else if (y > x) {
 8.          minimum = x;
 9.      }
10.      return minimum;
11.   }
```

Figure C.3: An implementation of the Min function

```
 1.  public int countPositive (int[] x)
 2.  {
 3.      int count = 0;
 4.      for (int i = 0; i < x.length; i++) {
 5.          if (x[i] < 0) {
 6.              count++;
 7.          }
 8.       }
 9.       return count;
10.  }
```

Figure C.4: The `countPositive` function

Specify the path constraint that this will generate.

Then, using the path constraint from this, flip one of the constraints corresponding the branch at line 5, specify a test that satisfies this, and its path constraint.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 9

Introduction

Reliability block diagrams are a simple and powerful method for making reliability assessments for systems. There are a couple of things to be wary about however:

- We need to be careful that all failure probabilities are for the same time period T;
- We should not confuse failure logic with reliability logic.

The aim of this tutorial is to give you practice at taking system descriptions, determining where failures can occur and creating parallel-serial reliability block models for those systems. Along the way we also get a brief look at Triple-mode Redundant systems – an instance of NMR, or N-mode Redundant, systems.

Understanding Reliability Blocks

Reliability block diagrams usually need to be created to model the reliability of a system. Typically we are given a system diagram that we need to analyse for reliability. To do this we create the reliability block diagram for the system and then perform the analysis on it. There are several steps involved in creating reliability block diagrams from system architectures:

Step 1 Determine what constitutes a *system* failure. This in turn determines which component failures cause a system to fail.

Step 2 Determine which components need to fail in order to cause a system failure. Some guidelines for constructing reliability block models is to consider how messages, signals or data flow through the system.

Step 3 Calculate system reliability from your reliability block diagram using either reliability logic or failure logic.

Some guidelines for creating reliability block models.

- Try to ensure that each block in the reliability block model captures one function of the system.
- Try to ensure that you capture the parallel/serial connections from the system accurately in your reliability block model.
- There may be more than one mode for the system - you will need to create a reliability block diagram for each mode of the system.

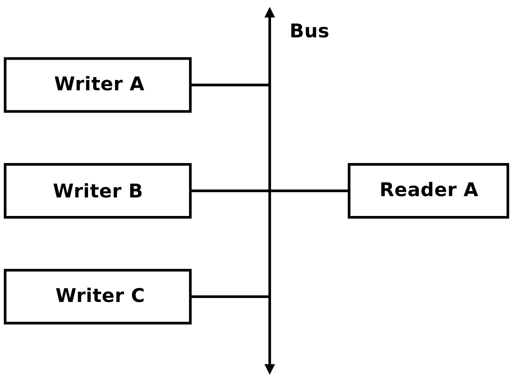

Figure C.5: Readers and writers communicating over a shared bus.

Working with the Sample System

Question 1

Consider the readers and writers system in Figure C.5. Three redundant writers Writer A, Writer B, and Writer C each read the same source to obtain a value. They then send that value via a Bus to Reader A. Each writer acts by placing a message on the bus when it has a message to send (assume that there is no contention between readers and writers for the bus). The reader then takes the message and processes it.

The system fails if no value is processed within a given time period T.

(i) List the ways in which the system can fail.

(ii) Next create a reliability block model for the writers, bus and reader.

(iii) After some measurement, it is estimated that the components have the following failure probabilities.

Component	**Failure Probability in time T**
Writer A	0.01
Writer B	0.05
Writer C	0.01
Bus	0.02
Reader	0.01

Calculate the *failure probability* and the *reliability* for the system with respect to time T.

Question 2

Consider the system of servers, routers and databases shown in Figure C.6.

The system consists of servers, routers and databases.

Each of the servers performs the same function. It makes external connections and serves pages. The server selects a database, retrieves the pages and other data and sends them back along the connection.

As long as one server is still working connections can be made at the cost of some degradation in performance.

The routers perform the task of connecting a server with a databases. Routers also handle faults in the system. If they detect a faulty databases machine or a faulty server they can then route the other traffic around these.

The three databases are redundant data stores and each database contains an exact duplicate of the data. Only one of the databases needs to be functioning at any one time.

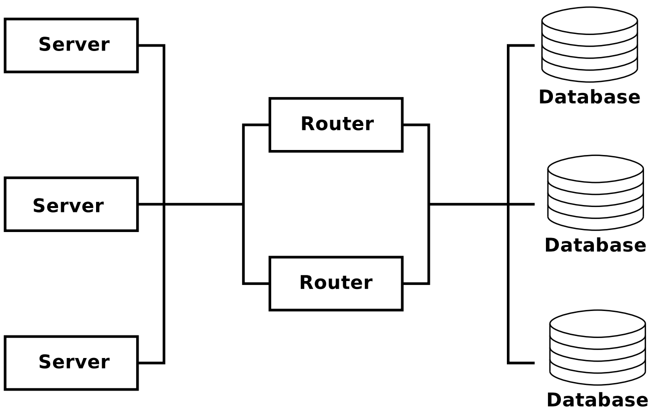

Figure C.6: A system of servers and databases communicating through duplicated routers.

(i) Define what it means for the system to fail and then determine which sets of components need to fail for the system to fail.

(ii) Create a reliability block model for the system.

(iii) Assume that the failure rate over 1 day of the three servers is 0.01, 0.02, and 0.03 respectively, the failure rate of each router is 0.02, and the failure rate of each database is 0.02. What is the reliability of the system over 2 days?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 10

Introduction

The aim of this tutorial is to give you a better understanding of Markov models, and how they can be used to help estimate system reliability.

Recall that a Markov model is a finite state machine in which the transitions have a probability attached to them representing the probability of that particular transition being taken. By mapping a Markov model to a matrix, calculating the probability of being in particular states can be achieved using a straightforward matrix multiplication (see Tutorial 1 for a matrix multiplication algorithm).

Tasks and Questions

Question 1

Consider a simple sensing device. The manufacturer gives you the following failure data for the device based on their measurements.

State	**Per-hour Failure Probability**	**Per-hour Repair Probability**
Functional → Degraded	10^{-5}	10^{-3}
Degraded → Non-Operational	10^{-2}	0
Functional → Non-Operational	10^{-6}	0

Table C.1: States and failure rates for the simple sensing device.

Table C.1 shows the probabilities for making transition between functional, degraded, and non-operational states (in the **Per-hour Failure Probability** column) in one hour of operation. It also gives the probability of repair (in the **Per-hour Repair Probability** column) in an hour; that is, the probability that the system transitions back to a functional state.

Draw the Markov model (in the form of a state transition diagram) with *all* of the transition probabilities clearly labelled. Determine the transition matrix for this model.

Question 2

Consider a simple system involving a node linked to a noisy channel. After some measurement the data in the following table is obtained. Assume that failures and repairs are independent.

(i) Determine the Markov states for the system.

Device	**Per-hour failure probability**	**Per-hour repair probability**
Node	10^{-4}	10^{-3}
Channel	10^{-3}	10^{-2}

Table C.2: Failure data for a simple noisy channel.

(ii) Draw the state transition diagram showing all of the transition probabilities.

(iii) Calculate the 1-step transition matrix for your model.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 11

Introduction

The aim of this tutorial is to start you thinking about reliability measurement and in manipulating and interpreting the different reliability formulae. Before beginning, there are a couple of things to note about reliability measurement, or any form of software measurement, for that matter.

In reliability measurement, we need to consider the following points:

1. **Deciding What "Failure" Means for an Application** — In reliability measurement you will need to decide what is means for a module to *fail*. To understand what constitutes a failure you will typically need to understand the module specifications in some detail.
2. **Deciding What to Measure** — Remember that we do not test every input with equal probability. Reliability estimates are based on an operational profile of the the system. If the estimated operational profile is not close to the actual operational profile then reliability estimates will be less accurate.
3. **Detecting Failures** — Specifying invariants for classes and specifying conditions for failures to occur are part of designing a test oracle for finding failures. Of course, the problem with writing test oracles is that we may not be able to detect every single failure. Thus, we may get a lower bound on the actual number of failures.
4. **Recording Failure Times** — Recording the failure times is also problematic as the time at which failures are experienced and the times between failures can NOT be measured accurately. However, the actual error in time measurement is often small compared to execution time scales. For example, an error of 10 milliseconds is negligible if the time between failures is measured in minutes.

Given that reliability measurement is inexact we need to understand: (1) *exactly* what we can say about reliability measurement; (2) how to interpret the data; and (3) how to make the best use from the information that we have.

Tasks and Questions

Question 1

Using the regression equations and the data from the appendix, calculate the following:

(i) the initial failure intensity λ_0; and
(ii) the total number of failures ν_0;

Question 2

Using the regression equations and the data from the appendix, calculate the following:

(i) the expected number of failures after 10 000 CPU seconds; and

(ii) the expected failure intensity after 10 000 CPU seconds.

Question 3

Consider now two possible target failure intensities:

(i) one failure every 12 hours; and

(ii) one failure every 24 hours.

How many more failures would need to experienced before *each* of these two target failure intensities is reached?

Question 4

How much more execution time for testing would be required for *each* of these two target failure intensities to be reached?

Question 5

Consider the two failure intensity against failures experienced graphs in Figures C.7 and C.8 respectively. Suppose that the equation for the regression line in Figure C.7 is:

$$\lambda_A(\mu_A) = \lambda_{A_0}\left(1 - \frac{\mu_A}{\nu_{A_0}}\right) \tag{C.1}$$

and the equation for the regression line in Figure C.8 is:

$$\lambda_B(\mu_B) = \lambda_{B_0}\left(1 - \frac{\mu_B}{\nu_{B_0}}\right) \tag{C.2}$$

Notice that the slope of the line $\lambda_B(\mu_B)$ is steeper than the slope of $\lambda_A(\mu_A)$; that is,

$$-\frac{\lambda_{A_0}}{\nu_{A_0}} \geq -\frac{\lambda_{B_0}}{\nu_{B_0}}. \tag{C.3}$$

Discuss how each of the model parameters obtained from the lines in Figure C.7 and C.8 affect:

(i) The expected number of failures $\mu(t)$; and

(ii) The failure intensity $\lambda(t)$.

Appendix

The data in Table C.3 records the set of failure times for an actual system, which we call T_1. That is, the first failure occurred after 3 CPU seconds, the second failure after a further 30 seconds (33 seconds), the third after a further 111 seconds (146), etc. The test phase ended after 6600 seconds of execution time. Table C.4 summarises the data from Table C.3.

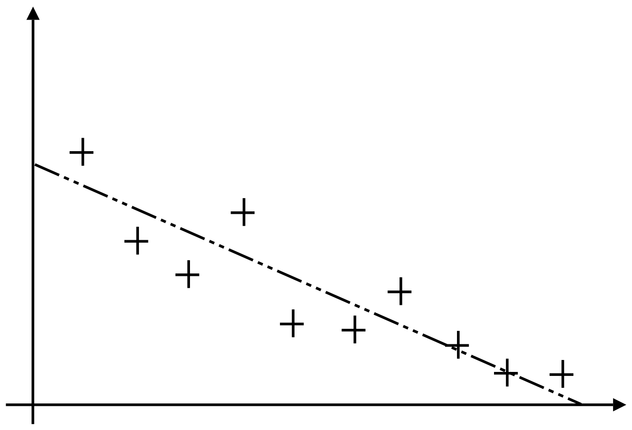

Figure C.7: Failure intensity vs failures experienced - Graph A

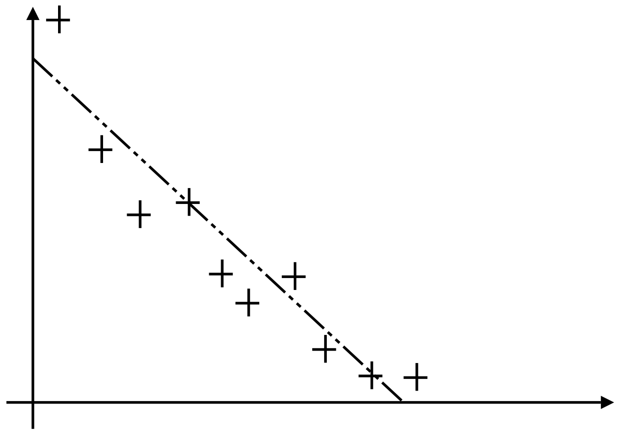

Figure C.8: Failure intensity vs failures experienced - Graph B

3	709	2311	5085
33	759	2366	5089
146	836	2608	5097
227	860	2676	5324
342	968	3098	5389
351	1056	3278	5565
353	1726	3288	5623
444	1846	4434	6080
556	1872	5034	6380
571	1986	5049	6477

Table C.3: Time of Failure for System T_1 (CPU seconds)

Failures	Time (CPU Sec.)	Failure Intensity (Failures/Second)
5	342	0.01462
10	571	0.02183
15	986	0.01205
20	1986	0.005
25	3098	0.004496
30	5049	0.002563
35	5389	0.014706
40	6477	0.004596

Table C.4: Failure Time Data for System T_1

Regression Equation

The regression equation (mean no. of failures experienced against Failure intensity) is

$$\lambda = -0.00032\mu + 0.017203$$