Software Testing Project: Phase 2 – Fuzz Testing on LLVM IR

1 Introduction

There are several goals for this assignment:

- Designing a simple fuzz testing tool.
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation used by LLVM.
- Using LLVM to perform a sample testing.

This assignment is a group assignment. Each team should have **2 members**

1.1 Fuzz Testing

Fuzz testing or fuzzing is an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool injects these inputs into the system and then monitors for exceptions such as crashes or information leakage. Source: synopsys.com

2 Task 1: Implementing Fuzz Testing in LLVM

In this task, you will extend the random testing tool designed in part 1 of the project. using LLVM compiler infrastructure tools.

Example 1: For example, consider the c program below:

```
int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x; // bug: should be '-x'
    }
}
```

The program contains a bug. Now assume, the initial seed for the fuzzing of this function is $\{35\}$ which does not reach the bug point. The goal of the second part of the project is to use fuzzing technique to generate other possible tests to reach higher coverage on the program.

The generated LLVM IR will have four basic blocks with labels: entry, if.then, if.else and if.end. The first test case, picks the true path in the program. The block coverage is 75% since 3 out of 4 basic blocks are visited.

Assume the second test case that you generate is {x=36} which alters 1 digit compared to the first test and the third test is { x=-36} which negates the previous test case. Now, these two additional test cases increases the block coverage from 75% to 100% and also triggers the bug.

- value of x: 35{true path}
- value of x: 36{true path}
- value of x: -36 {false path}

Note 1: For simplicity you can consider the arguments are defined in the beginning of the program using *alloca* instruction. Moreover, input arguments are always marked with a_1, a_2, etc. For example, the above example program would be changed to the following and provided to your fuzz tester:

```
int main() {
    int a1;
    int x = a1;
     if (x > 0) {
        return x;
    }
    else {
        return x; // bug: should be '-x'
    }
}
```

Note 2: Your LLVM pass should provide the value for $a1$ and the sequence of traversed LLVM basic blocks as output for each test case and the block coverage:

- $a1 = 35$
- Sequence of Basic Blocks:
- entry
- if.then,
- if.end,
- block coverage: 75%

Note 2: The initial seed can be chosen randomly.

3 Task 2: Adding Support for Loops to the Fuzz Tester

In order to handle loops, the designed tester should be continued until it can jump out of the loop. Extend your analysis from task 1 to support loops. Your tester needs to track the loop counter to check if the loop has iterated enough. Assume variable (i) is the loop counter.

Example 2: In this example, consider the c program with a loop below:

```
int main() {
  int a1;
  int b,c;
  int i = 0;
  while (i < 3) {
    if (a1 > 0){
      b = 7;
      i++;
      a1 = a1 - 2;
    } else {
      c = 12;
      i=i+2;
    }
  }
}
```

Before the loop i is 0. Applying the fuzz tester, assuming the initial seed is $a1 = 1$, the true path inside the loop is traversed and i becomes 1. The path continues to the second iteration. This time $a1 = -1$, and i becomes 3. Now, as we finish the second iteration of the loop and move to the third iteration, this time the condition $i < 3$ is not true and the path continues to the exit point of the loop:

- $a1 = 1$
- Sequence of Basic Blocks:
- %0,
- %2,
- %5,
- %8,
- %10,
- %2,
- %5,
- %9,

- %10,
- %2,
- %13

4 Submission - Due Date: Wed 10th Khordad 1401, 11.59pm

Please submit the following in a single archive file (`zip` or `tgz`):

1. A report in PDF (`proj2.pdf`). It should describe your design for task 1 and task 2, the implementation details, the algorithm used, and the steps to build and run your code, and finally the output of examples programs tested.
2. Your source code.
3. Your test C files with corresponding `ll` files.

Make sure your reports and source files contain information about your name, matric number and email. Your zip files should have the format *Surname-Matric*-proj2.zip (or `tgz`).

Please email the submissions to (`sajjadrsm@gmail.com`).