Software Testing Project: Phase 3 – Dynamic Symbolic Execution on LLVM IR

1 Introduction

There are several goals for this assignment:

- Designing a simple dynamic symbolic execution 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 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is a hybrid approach to software testing that attempts to strike a balance between the costs and benefits of dynamic and static analysis. As you saw, it generates concrete inputs one-by-one such that each input takes a different path through the program's computation tree. (Source [1])

2 Task 1: Implementing Fuzz Testing in LLVM

In this task, you will extend the fuzz testing tool designed in part 2 of the project using LLVM compiler infrastructure tools. Unlike fuzz testing in DSE the coverage is increase systematically. Here, each test case takes a new path.

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

```
int myAbs(int x) {
    if (x != 123456) {
        return x;
    }
    else {
        return -x; // bug: should be '-x'
    }
}
```

[1] https://www.cis.upenn.edu/m̃hnaik/edu/cis700/lessons/symbolic_execution.pdf

The program contains two paths. Now assume, the initial seed of this function is {35} which does not reach the bug point. As discussed in the classroom, both random testing and fuzz testing have no luck in reaching the bug point. However, a DSE tool is able to generate other possible tests and reach the bug point.

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. Now, the the second test case is generated by negation of the condition in the if statement {x!=123456} which generates { x=123456}. Now, this new test case increases the block coverage from 75% to 100% and also triggers the bug.

- value of x: 35{true path}
- value of x: 123456 {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 != 123456) {
        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 DSE Tool

Loops can be handled similar to phase 2.

4 Submission - Due Date: Wed 31st Khordad 1401, 11.59pm

Deadline is fixed and will not be changed.

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

1. A report in PDF (`proj3.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*-proj3.zip (or `tgz`).

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