**Homework 3**

**Due date: 18th March, 2018**

By doing HW2, you already knew how to traverse an AST with libraries provided by Clang and LLVM, and to do simple control flow analysis. In this homework, you will learn how to do simple data flow analysis. Specifically, you will finish 3 tasks in this homework.

**T1. (5 points) Dump call graph**

A call graph is a control flow graph that represents caller and callee relationship in a program. In this task, you are going to write a program that accepts a test program and dumps its call graph.

Clang’s command line tool can dump call graphs (See [this page](https://gist.github.com/quellish/9fb5506d9beda77a5cdf) for debug.DumpCallGraph). First try it and see what a call graph looks like. Your program should simulate this function by generating the call graph in a .dot file, as you did in HW2. You should put function names in the nodes of the dot graph.

**T2. (25 points) Check function pairs**

Some functions need to be called in pairs in C/C++, such as malloc & free, and file open and close. In this task, you need to write a program to check whether two functions are called in pairs.

Your program should accept three inputs: 1) a test program; 2) a function name A that needs to be called first; 3) another function name B that has to be called later if the first function is called. If A is called, B must be called somewhere in a path later. If A is not called, B cannot be called neither. You only need to check the function names.

Take the test program attached (***testlock.c***) as an example. lock() and unlock() form such a function pair. In h1(), unlock() is guaranteed to be called after lock(). However, in g(), unlock() might not be called if ***j <= 10***. In h3(), only unlock() is called. In these cases, your output should look like as the following lines. The output order is not important.

h1: True

g: False

h3: False

There are many different checkers in this folder: *$LLVM/tools/clang/lib/StaticAnalyzer/Checkers*.

I strongly recommend you read the first attached PDF file and understand it. This task is just a simple version of it. The second attached material contains how to register your checker in Clang.

You can write your own checker from scratch. But I recommend you change an existing one, like *$LLVM/tools/clang/lib/StaticAnalyzer/Checkers/TraversalChecker.cpp*. Program state is very important in symbolic execution. When implementing a Clang checker, you need to understand and leverage it, such as REGISTER\_MAP\_WITH\_PROGRAMSTATE. These are explained in the attached material number 2, which provided a lot of help in developing a clang checker.

Here is a list of pairs you need to handle: 1) lock() and unlock(); 2) mymalloc() and myfree(). Clang checkers do not accept arguments, so you need to hard-code these pairs.

**T3. (20 points) Dump all possible paths**

The number of paths of a program grows in exponential. In this task, you are going to print out all possible paths of the given program. Don’t worry, we will only test your program with small programs. For example, there are two paths in attached ***example3.c***. f2() is a recursive function and it will be called twice. Your program needs to print out all calls.

P1: main, f1, exit

P2: main, f2, f2, exit

**Some useful things:**

1. <http://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf>

This is super helpful. If you can understand the contents in this slides. Task 2 will be super easy.

1. <https://clang-analyzer.llvm.org/checker_dev_manual.html>
2. I recommend you read the manual and slides and try to run some checkers before you work on T2 and T3. Once you understand how Clang checkers work, this homework will be super easy.
3. To solve T2 and T3, your checker need to interact with the program state to add, update and get path sensitive data. Clang checker provide the following four macros. You can choose one of them for this kind of functionality.
   1. REGISTER\_TRAIT\_WITH\_PROGRAMSTATE
   2. REGISTER\_LIST\_WITH\_PROGRAMSTATE
   3. REGISTER\_SET\_WITH\_PROGRAMSTATE
   4. REGISTER\_MAP\_WITH\_PROGRAMSTATE