CS5218: Assignment 2 – Taint Analysis on LLVM IR

1 Introduction

There are several goals for this assignment:

- Designing a sample analysis using the principles of Data Flow Analysis.
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation used by LLVM.
- Using LLVM to perform the sample analysis.

This assignment is not a group assignment.

1.1 Taint Analysis

Taint Checking is a popular method that checks which variables can be modified by the user input. All user inputs can be dangerous if they are not properly checked.

The concept behind taint checking is that any variable that can be modified by an outside user (for example a variable set by a field in a web form) poses a potential security risk. If that variable is used in an expression that sets a second variable, that second variable is now also suspicious. The taint checking tool proceeds variable by variable until it has a complete list of all variables which are potentially influenced by outside input. If any of these variables is used to execute dangerous commands (such as direct commands to a SQL database or the host computer operating system), the taint checker warns that the program is using a potentially dangerous tainted variable. The computer programmer can then redesign the program to erect a safe wall around the dangerous input (source: Wikipedia).

Taint analysis tracks information flows from an object x(source) to another object y(sink), whenever information stored in x is transferred to object y. Taint analysis enables us to perform Taint Checking over all input variables one at a time.

2 Task 1: Designing Taint Analysis

In this task, we first will design a taint analysis based on principles of Data Flow Analysis. In order to do that the lattice, transfer functions and the monotone framework over a simple while language (slide 2 of CS5218-02-DFA.pdf) should be defined for the taint analysis.

3 Task 2: Implementing the Taint Analysis in LLVM

LLVM is set of compiler infrastructure tools. You have seen `clang` and `clang++`, the LLVM C and C++ compilers in the demo. In this task, you will write an LLVM pass to perform the analysis on loop-free programs.

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

```
int main() {
  int a,b,c,sink, source;
  // read source from input
  b = source;
  if (a > 0)
    skip;
  else
    c = b;
  sink = c;
}
```

The generated LLVM IR will have four basic blocks with labels: entry, if.then, if.else and if.end. In the end of each of the basic blocks the list of tainted variables would be as follows (registers are skipped for brevity):

- entry: {source, b}
- if.then: {source, b}
- if.else: {source, b, c}
- if.end: {source, b, c, sink}

Since at the last basic block, "if.end", sink is in the list of tainted variables, we can infer sink might be a potentially dangerous tainted variable.

Example 2: Now, consider the c program below which is slightly different from the previous program:

```
int main() {
  int a,b,c,sink, source;
  // read source from input
  if (a > 0)
    b = source;
  else
    c = b;
  sink = c;
}
```

The generated LLVM IR will have four basic blocks with labels: entry, if.then, if.else and if.end. In the end of each of the basic blocks the list of tainted variables would be as follows (registers are skipped for brevity):

- entry: {source}
- if.then: {source, b}
- if.else: {source}
- if.end: {source, b}

This time, sink is not in the list of tainted variables at “if.end” and we can infer that no flow from source reaches sink. So, our slight change has removed source from the list of tainted variables.

4 Task 3: Adding Support for Loops to the Analysis

In order to handle loops, the designed analysis should be continued until a fixpoint is reached. Change your analysis from task 1 to support loops.

Example 3: In this example, consider the c program with a loop below and its respective CFG in Figure 1:

```
int main() {
  int b,c,sink, source, N;
  // read source from input
  int i = 0;
  while (i < N) {
    if (i % 2 == 0)
      b = source;
    else
      c = b;
    i++;
  }
  sink = c;
}
```

Applying the analysis from task 2 (assuming each basic block is visited only once) will generate the following list of tainted variables in the end of each of the basic blocks:

- entry: {source}
- while.cond: {source}

- while.body: {source}
- if.then: {source, %3, b}
- if.else: {source}
- if.end: {source, %3, b}
- while.end: {source}

Based on these results sink is not tainted. However, these results are incomplete and have not considered the backedge between "if.end" and "while.cond". We continue the analysis for a few more rounds until the least fixpoint is reached for all the basic blocks. Note that the reached fixpoint should be the least fixpoint[1]. The generated results would be:

- entry: {source}
- while.cond: {source, %3, b, %4, c}
- while.body: {source, %3, b, %4, c}
- if.then: {source, %3, b, %4, c}
- if.else: {source, %3, b, %4, c}
- if.end: {source, %3, b, %4, c}
- while.end: {source, %3, b, %4, c, %6, sink}

This time, sink is in the list of tainted variables at "while.end" and we can infer that sink is tainted.

5 Submission - Due Date: Sun 29th Mar., 11.59pm

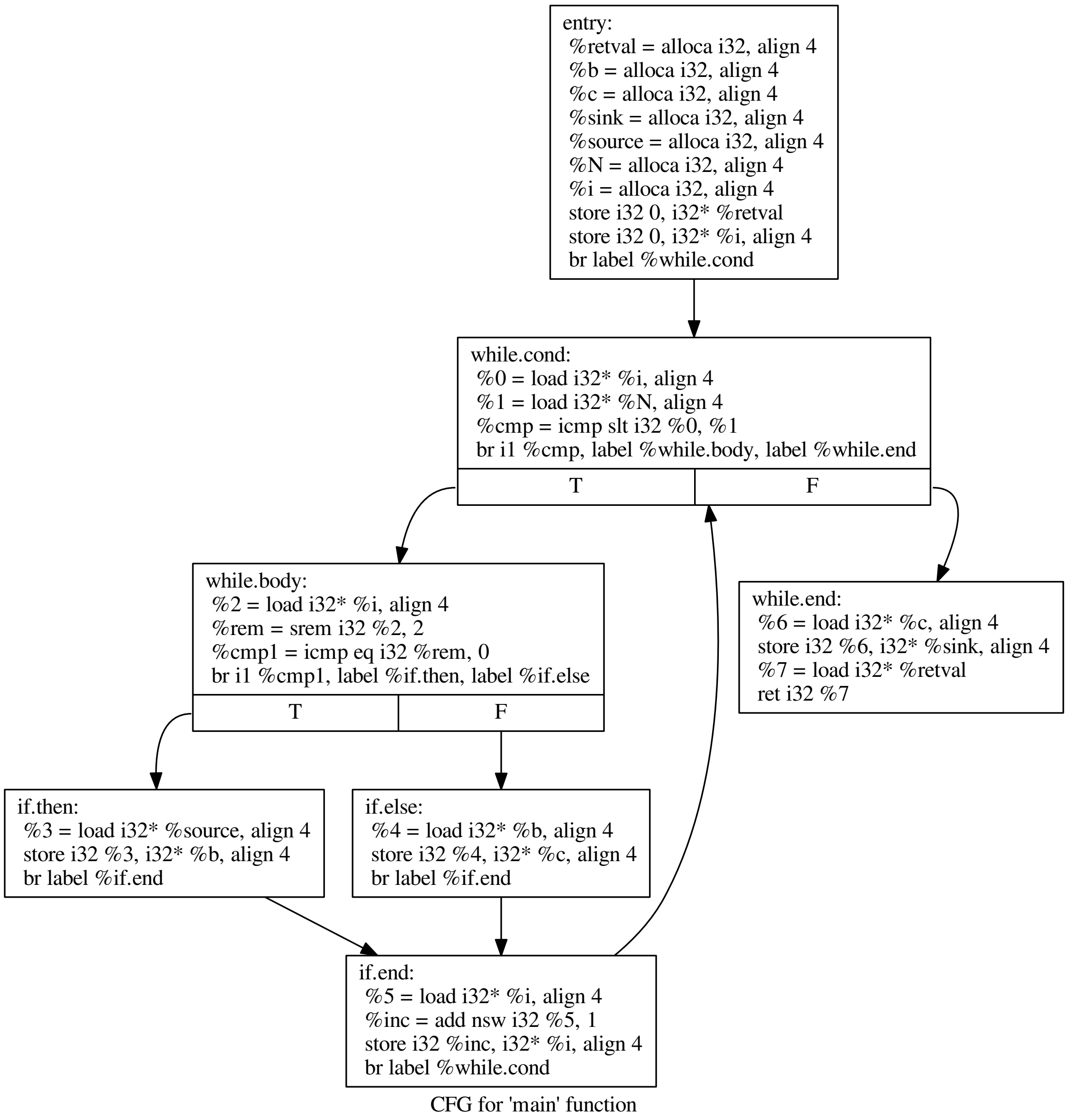
Please submit the following in a single archive file (`zip` or `tgz`):

1. A report in PDF (`asg2.pdf`). It should describe your design for task 1 and the implementation of tasks 2 and 3 and the algorithm used. How to build and run. The output of examples programs tested.
2. Your source code.
3. Your test C files with corresponding `ll` files.

For technical support on LLVM, you can contact Rasool or Sanghu (`{rasool,sanghara}@comp.nus.edu.sg`). Make sure your subject line begins with "CS5218".

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*-asg1.zip (or `tgz`). Submit all files above to the appropriate IVLE workbin.

[1]An analysis which taints all variables reaches a fixpoint too.



```
entry:
 %retval = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %sink = alloca i32, align 4
 %source = alloca i32, align 4
 %N = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %retval
 store i32 0, i32* %i, align 4
 br label %while.cond

while.cond:
 %0 = load i32* %i, align 4
 %1 = load i32* %N, align 4
 %cmp = icmp slt i32 %0, %1
 br i1 %cmp, label %while.body, label %while.end

while.body:
 %2 = load i32* %i, align 4
 %rem = srem i32 %2, 2
 %cmp1 = icmp eq i32 %rem, 0
 br i1 %cmp1, label %if.then, label %if.else

while.end:
 %6 = load i32* %c, align 4
 store i32 %6, i32* %sink, align 4
 %7 = load i32* %retval
 ret i32 %7

if.then:
 %3 = load i32* %source, align 4
 store i32 %3, i32* %b, align 4
 br label %if.end

if.else:
 %4 = load i32* %b, align 4
 store i32 %4, i32* %c, align 4
 br label %if.end

if.end:
 %5 = load i32* %i, align 4
 %inc = add nsw i32 %5, 1
 store i32 %inc, i32* %i, align 4
 br label %while.cond
```

CFG for 'main' function

Figure 1: CFG of C Program with Loop