CS5218: Assignment 1 – DataFlow Analysis on LLVM IR

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 a sample analysis.

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.

Task 1: Designing Taint Analysis

In this task, you will design a taint analysis based on principles of Data Flow Analysis. In order to do so, the lattice, type of the analysis (May/Must, Forward/Backward) and why, description of the data structure used in the analysis, *gen* and *kill* functions for different instructions, block *entry* and *exit* equations and description on how to reach Fixpoint over a simple while language (slide 2 of CS5218-23-02-DFA.pdf) should be defined for the taint analysis.

Note: The next three tasks are programming based. To keep the things simpler you can limit your analysis to the following instructions: Alloca, Add, Sub, Div, Mul, Rem, and Load/Store.

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 **taint analysis** on loop-free programs.

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

```
int main() {
  int i,j,k,sink, source;
  // read source from  user input or it can be
  // initilized with a tainted value. e.g. source = 1234567
  i = source;
  if (j > 1)
    skip;
  else
    k = i;
  sink = k;
}
```

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 block, your llvm pass should print the list of tainted variables for that block and it would be in the format as follows (registers are skipped for brevity):

- entry: {source, i}
- if.then: {source, i}
- if.else: {source, i, k}
- if.end: {source, i, k, 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 i,j,k,sink, source;
  // read source from input
  if (i > 1)
    j = source;
  else
    k = j;
  sink = k;
}
```

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

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

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 `sink` from the list of tainted variables.

Note: A variable may be tainted at one point and become untainted later. This process is known as 'untainting'.

Task 3: Determine the Abstract Tainted Data Flow Paths

In this task, you have to design a graph traversal llvm pass. It will determine the abstract path (only the block label information) traversed by the tainted data from the source variable to the reach other variables present in the program. Also, assume that there are *no loops* in the input program for this llvm pass.

Example 3: Consider the C-program below with block labels:

```
entry: int main() {
         int i,j,k=0,sink=0, source=1234567;
         // read source from input or initilized with
         // a tainted value. e.g. source = 1234567
         i=0;
if.then: if (j > 1)
             i=source;
         else
if.else:     k = source;
if.end: sink = i+k;
}
```

In the above example program, the abstract path followed by tainted value in the variable `source` to reach variable `k` at `if.else` block is {`entry->if.else`} and for variable `sink` at `if.end` block is as follows {`entry->if.then->if.end`} or {`entry->if.else->if.end`}. We can see that there are two possible paths to taint the `sink` variable.

In this task, you are expected to determine all the possible paths to taint variables present in each basic block.

Task 4: Adding Support for Loops to the Analysis

In this task, you have to extend your llvm pass to support the program with loops. In order to handle loops, the designed analysis should be continued until a fixpoint is reached. Extend your analysis from task 1 and task 2 to support loops.

Example 4: In this example, consider the c program with a loop below and its respective CFG in Figure 1:

```
int main() {
  int i,j, sink, source, N;
  // read source from input
  int a = 0;
  while (a < N) {
    if (a % 2 == 1)
      i = source;
    else
      j = i;
    a++;
  }
  sink = j;
}
```

Applying the analysis from task 2 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, i}
- if.else: {source}
- if.end: {source, %3, i}
- while.end: {source}

Assuming each basic block is visited only once, the above results show `sink` is not tainted. However, these results are incomplete,sine we have not considered the backedge between "if.end" and "while.cond". We need to continue the analysis for a few more rounds. But, in this manner, the analysis may fall into infinite loops. Therefore, we need to add a CONDITION to reach Fixpoint for all the basic blocks. To add check condition for fixpoint, please look into those parts of the loop body affecting the entry and exit conditions of block. Note that when you reach a fixpoint you shoud actually be getting a least fixpoint[1].

[1] An analysis which covers all the tainted variables and reaches a fixpoint too.

After computing the fixpoint, the generated results would be:

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

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

Task 5: Beyond taint analysis: Implement Very Busy Expression Analysis

In this task, we are moving away from taint analysis to very busy expression analysis. Now that you are familiar with LLVM passes, it is time to implement one of the optimizations techniques that have been covered in class. As we know, taint analysis is a *may analysis* with *forward data flow*. In contrast of that, we have selected the very busy expression analysis for you to implement in this task. Your analysis pass must print all the busy expressions at the entry and exit point of each basic block. You will get the definition of *kill* and *gen* functions for this task in Slide 65 of CS5218-24-02-DFA.pdf. A complete example is also available in the slides.

Similar to the taint analysis, test this analysis pass as well on your example programs and share the output. To keep the things simpler, consider expressions represented using a single Binary Operator only. For example, a=b+c instead a=b+(c+d).

Note: For this assignment, your analysis can be limited to the following instructions: Alloca, Add, Sub, Div, Mul, Rem, and Load/Store.

Submission - Deadline: 23:59 Sunday 25 February 2024

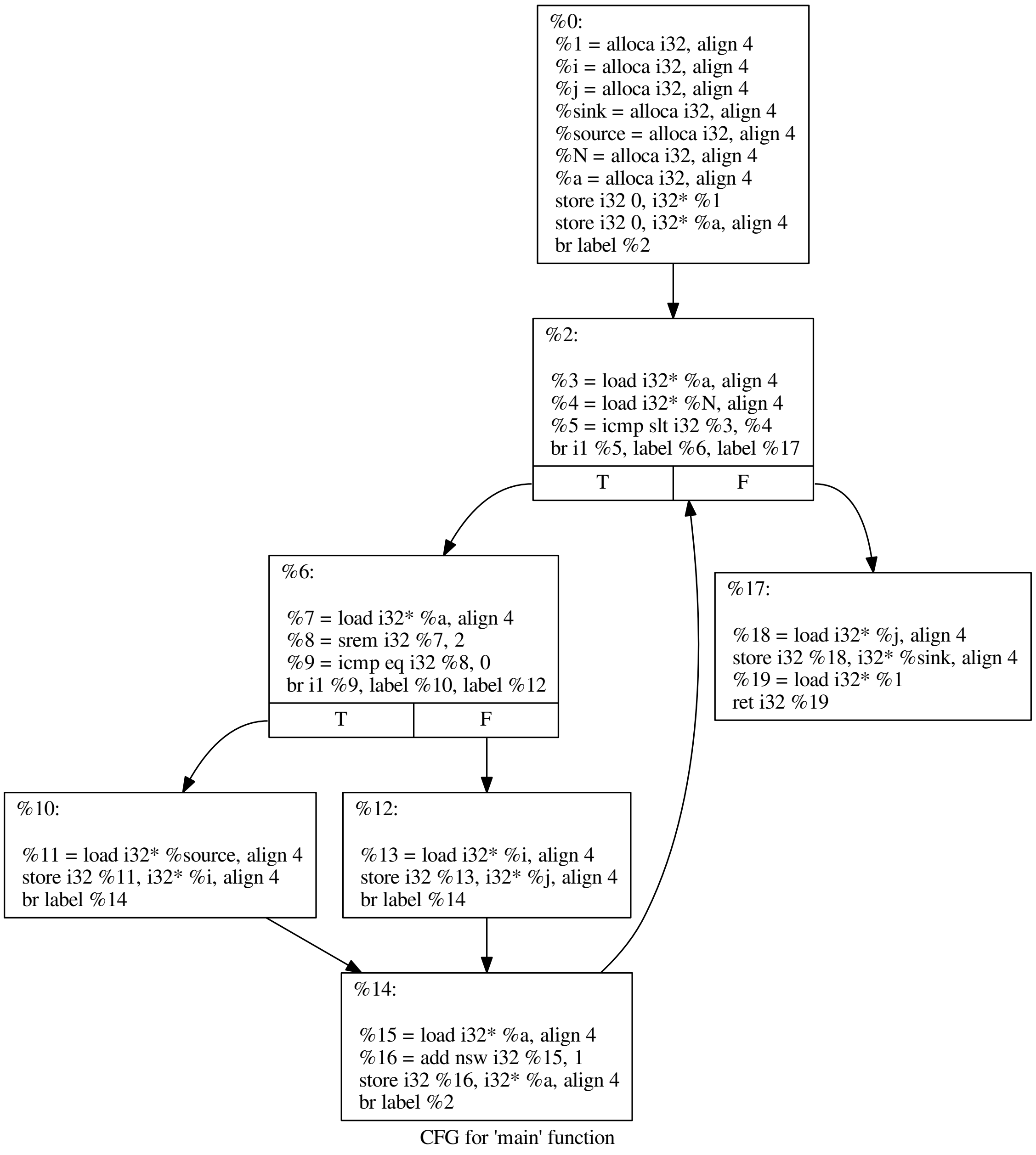
Please submit the following in a single archive file (`zip` or `tgz`):

1. A report in PDF (`asg1.pdf`). It should describe your design for task 1, the implementation of tasks 2, 3, 4 and 5, the algorithm used, 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.

For technical support on LLVM, you can contact Arpita Dutta (`arpita@comp.nus.edu.sg`) or Chew Wei Ze Alvin (`a.chew@nus.edu.sg`). Please have your subject line begin with “CS5218”.

Also please ensure 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 Canvas workbin.



```
%0:
 %1 = alloca i32, align 4
 %i = alloca i32, align 4
 %j = alloca i32, align 4
 %sink = alloca i32, align 4
 %source = alloca i32, align 4
 %N = alloca i32, align 4
 %a = alloca i32, align 4
 store i32 0, i32* %1
 store i32 0, i32* %a, align 4
 br label %2
```

```
%2:

 %3 = load i32* %a, align 4
 %4 = load i32* %N, align 4
 %5 = icmp slt i32 %3, %4
 br i1 %5, label %6, label %17
```

```
%6:

 %7 = load i32* %a, align 4
 %8 = srem i32 %7, 2
 %9 = icmp eq i32 %8, 0
 br i1 %9, label %10, label %12
```

```
%17:

 %18 = load i32* %j, align 4
 store i32 %18, i32* %sink, align 4
 %19 = load i32* %1
 ret i32 %19
```

```
%10:

 %11 = load i32* %source, align 4
 store i32 %11, i32* %i, align 4
 br label %14
```

```
%12:

 %13 = load i32* %i, align 4
 store i32 %13, i32* %j, align 4
 br label %14
```

```
%14:

 %15 = load i32* %a, align 4
 %16 = add nsw i32 %15, 1
 store i32 %16, i32* %a, align 4
 br label %2
```

Figure 1: CFG of C Program with Loop in Example 4