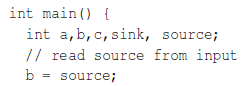
**Task 1: Designing Tainted Variables Analysis**

Similar to what was presented in Assignment 1, we are tasked with creating a similar variable analysis, this time being for Tainted Variables of a given program. We understand as tainted a variable that has been assigned to some input provided by the user. Any other variable that is modified by a tainted variable whether by declaration, Boolean operations or arithmetic operations is also considered to be tainted. Variables are only untainted when they are assigned a non-user input value from a non-tainted source.

|  |  |  |
| --- | --- | --- |
| Label |  |  |
| 1 |  | Source, b |
| 2 | Source, b | Source, b |
| 3 | Source, b | Source, b, c |
| 4 | Source, b, c | Source, b, c, sink |

 (1)



(2) (3)

 (4)

For this analysis represented by TV, we define the Monotone Framework as such:

Since the flow of the statements for this analysis can come either from the if.then as well as from the if.else, both are included in the Union of the final statement.

For the lattice, we are looking for a Powerset of all tainted variables in the program:

This powerset is partially ordered by subset inclusion given that the tainted variables will be either a sub-set or the set itself of variables but cannot be a superset. And it is an intersection rather than a Union because if at a given point a variable becomes untainted, then the analysis needs to take the intersection of entry and exit rather than the union. At the end of the program we are looking at the set of variables being tainted.

**Task 2: Implementing the Tainted Variables Analysis in LLVM**

For this task, for every .bc or .ll file that is introduced into the LLVM Pass, the code will delve deep into the main of the program and then analyse each of the Basic Blocks that are presented.

The example provided as the solution for Initialized Variables Analysis was utilized and re-factored to serve for Tainted Variables Analysis purposes. Mostly the usage of the traversal of the CFG in Depth First Order using a Stack and the method of union comparison to determine the least fixpoint during the traversal.

For purposes of this algorithm, we initialized a global set containing the string of the names of registries and variables that are known to be Tainted. In all these examples we work with ‘source’ as the initial tainted variable. Had this been used with methods such as getchar() or other input methods, the algorithm would have been adapted to such.

While analysing all the instructions for each BasicBlock, we focus on 3 different types of instructions which tell us where the tainted variables propagate: Alloca Instructions, Store Instructions and Load Instructions.

Alloca instructions are used to store the instructions where the initial tainted variables from the set are located at the moment of initialization. As explained above, had this analysis been made with basis on methods that read from user input, this check for Alloca Instructions would not be needed and instead replaced for checks for usage of these methods.

Since variables are not passed from one to another, but instead using a register as a middleman, we need to identify the registers to which the tainted variable propagates before it reaches the next variable. Because of this, once we detect any register that has been tainted through the Store instruction, we brand them and include them in the set of Tainted elements.

Lastly, Load instructions are used to determine whether a variable is being loaded with the value from a tainted register which is now part of the tainted elements set from the checks on Store Instructions. If this is the case, then we add the variable instruction to the list of Tainted Variables for the BasicBlock. These checks are repeated for every instruction across every basic block in the program until a fixed point is reached.

Output observed for test1.ll:

LLVM and Clang version 3.5 was used for this assignment to be at the same level as the one shown in the documentation from the Demo.

The following commands were run in the following orders to build and run the pass and output of the examples to be tested on the programs:

To build the Pass:

clang++-3.5 -o TaintedVars TaintedVars.cpp `llvm-config-3.5 --cxxflags` `llvm-config-3.5 --ldflags` `llvm-config-3.5 --libs` -lpthread -lncurses -ldl

To compile the .ll files:

clang-3.5 -emit-llvm -S -o <filename>.ll <filename>.c

To compile files as .bc instead:

clang-3.5 -emit-llvm -c -o <filename>.bc <filename>.c

To run the pass:

./InitializedVariables <filename>.ll