CS5218 Assignment 2 - Taint Analysis on LLVM IR
Tan Lai Chian Alan
A0174404L
e0210437@u.nus.edu

Design for Task 1

- Lattice is defined to be P(Var) with $\sqsubseteq = \subseteq$, $\sqcup = \cup$ and $\bot = \varnothing$, where Var is the set of all variables in the program.
- The forward flow, flow(S), is used.
- The monotone framework and transfer functions are as follow:

$$Tainted_{entry}(l) = \cup \{Tainted_{exit}(l') \mid (l', l) \in flow(S)\}$$

$$Tainted_{exit}(l) = (Tainted_{entry}(l) \cup gen_{Tainted}(B^l)) \setminus kill_{Tainted}(B^l) \text{ where } B^l \in blocks(S)$$

$$gen_{Tainted}([z := a]^l, a \in Tainted) = \{z\}$$
$$gen_{Tainted}([\text{All Other Instructions}]^l) = \varnothing$$

$$kill_{Tainted}([z := a]^l, a \notin Tainted) = \{z\}$$
$$kill_{Tainted}([\text{All Other Instructions}]^l) = \varnothing$$

LLVM and Clang Versions Used
llvm-3.5-dev
clang-3.5

Implementation of Task 2
Step 1: Read the IR file to extract Module M from IR, extract Function main from Module M.
Step 2: Initialize Analysis Map, Tainted Set, and Traversal Stack, traverse the CFG in Depth First Order (Similar to StackSet.cpp). Here, our Tainted Set will contain addresses of variables and registers that have been tainted by source. Intuitively, the Tainted will contain address of our source variable at the start of the program every time.
Step 3: Upon entry (and popping) of each Basic Block from stack, iterate through the Basic Block's set of instructions

Step 3.1: If instruction opcode is 27, means it is a load instruction and we look for registers that have been tainted, i.e., when the instruction is to load value from an address found in our Tainted set into the register. Hence, we add the address of this register to our Tainted set.

Step 3.2: Else if instruction opcode is 28, means it is a store instruction and we look for our two operands, operand 0 and operand 1. If address of operand 0 is found in our Tainted set, it means that operand 1 will be tainted because the store instruction basically copies value of operand 0 into operand 1. Hence, we add the address of operand 1 to our Tainted set.
Step 4: Extract last instruction of Basic Block to get successor blocks

Step 5: For each successor block, increase value of depth by 1 and push to traversal stack, and then push all tainted variables of parent Basic Block to the successor block.
Step 6: Print result of Analysis map

Implementation of Task 3
Algorithm used is MOP. The implementation for tasks 2 and 3 are pretty much the same, with just a small tweak for Step 5 in Task 3
Step 5: For each successor block, iterate through parent Basic Block's set of Tainted addresses. For each address, check if the address exists in successor block's list of addresses in Analysis map. If does not exist, set *hasChanged* to true and push address to successor's Analysis map. After iterating through all addresses, if *hasChanged* is true, then push successor block to traversal stack.

Example 1 Output:

```
Print Analysis Map
%0: { b, source, }
%5: { b, source, }
%6: { b, c, source, }
%8: { b, c, sink, source, }
```

Example 2 Output:

```
Print Analysis Map
%0: { source, }
%4: { b, source, }
%6: { source, }
%8: { b, source, }
```

Example 3 Output:

```
Print Analysis Map
%0: { source, }
%10: { b, c, source, }
%12: { b, c, source, }
%14: { b, c, source, }
%17: { b, c, sink, source, }
%2: { b, c, source, }
%6: { b, c, source, }

PLEASE NOTE THAT BUILDING BLOCKS ARE NOT PRINTED IN ORDER
%17 CORRESPONDS TO THE 'WHILE END' BLOCK
```

Build and Run Pass
Check out README.md in folder