Assignment 1

Task 1

Taint analysis can also be designed with the overall Monotone Framework defined in [1] as follows:

$$Analysis_{\circ}(\ell) = \begin{cases} \iota & if\ \ell \in E \\ \bigsqcup\{Analysis_{\bullet}(\ell') \mid (\ell', \ell) \in F\} & otherwise \end{cases}$$

$$Analysis_{\bullet}(\ell) = f_{\ell}\big(Analysis_{\circ}(\ell)\big)$$

At some program point ℓ a variable could be tainted by the previous program input, whenever the tainted variable is used for assignments or expressions. However, the tainted variable may not influence the result of program because it may be reassigned by some other value at a program point, i.e. the output will not be affected by whatever input the point has.

Consider the following example program written in the While language.

$$[a := source]^1; [b := a]^2; if\ [a > b]^3\ then\ [b := b + 5]^4\ else\ [b := 1]^5; [sink := b]^6$$

Following the Monotone Framework above, the taint analysis has the complete lattice $(\mathbb{P}(\{source, a, b, sink\}), \subseteq)$, the instance of the framework, and the transfer function below.

$$F = flow(S_{\star})$$

$$E = \{init(S_{\star})\}$$

$$\iota = \emptyset$$

$$f_{\ell}(l) = \big(l \backslash kill(B^{\ell})\big) \cup gen\big(B^{\ell}\big)\ where\ B^{\ell} \in blocks(S_{\star})$$

$$gen\big(B^{\ell}\big): Blocks_{\star} \rightarrow \mathbb{P}(Var_{\star})$$

$$kill\big(B^{\ell}\big): Blocks_{\star} \rightarrow \mathbb{P}(Var_{\star})$$

We now are able to list all of the associated equations based on the instance for each ℓ as follows:

$$TV_{entry}(1) = \emptyset$$
$$TV_{entry}(2) = TV_{exit}(1)$$
$$TV_{entry}(3) = TV_{exit}(2)$$
$$TV_{entry}(4) = TV_{exit}(3)$$
$$TV_{entry}(5) = TV_{exit}(3)$$
$$TV_{entry}(6) = TV_{exit}(4) \cup TV_{exit}(5)$$

$$TV_{exit}(1) = TV_{entry}(1) \cup \{source, a\}$$
$$TV_{exit}(2) = TV_{entry}(2) \cup \{b\}$$
$$TV_{exit}(3) = TV_{entry}(3)$$
$$TV_{exit}(4) = TV_{entry}(4)$$
$$TV_{exit}(5) = TV_{entry}(5) \backslash \{b\}$$
$$TV_{exit}(6) = TV_{entry}(6) \cup \{sink\}$$

After solving the equations, it shows that the **source** would potentially reach the **sink** as affected by the tainted variables at the end.

Task 2

Based on the method designed for the taint analysis in Task 1, the implementation of it is shown as the following algorithm.

Algorithm

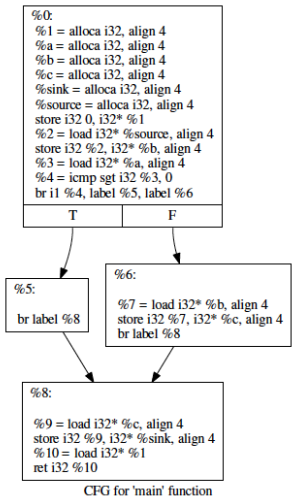
sourceVars: a tainted variable set for each path
finalVars: a tainted variable set for all the paths
x, **v**: some variable in an instruction
If **v** from **sourceVars** is used for an instruction where the expression or assignment includes another variable **x**
    insert **x** into **sourceVars**, continue updating **sourceVars** in the same block
    move on to the next basic block **BB**
If **# of successors** is **0**
    insert **sourceVars** into **finalVars**
    continue for the next path
Print out **finalVars** to check if there is the variable **sink**.

In the following example program

$$[b := source]^1;\ if\ [a > 0]^2\ then\ [skip]^3\ else\ [c := b]^4; [sink := c]^5$$

while being traversed, each block was labelled as a number in ascending order, not necessarily identical to the above superscripted numbers. The program analysis is explained with the CFG below. For brevity, it is possible to ignore the virtual registers generated by LLVM as we will do in the following examples.

Block 1: {b, source} // %0



CFG for 'main' function

First path
Block 2: {b, source} // %5
Block 3: {b, source} // %8

Second path
Block 4: {b, c, source} // %6
Block 5: {b, c, sink, source} // %8

Tainted variables: {b, c, sink, source}
// Join all the path outcomes, **Block 3** and **Block 4**

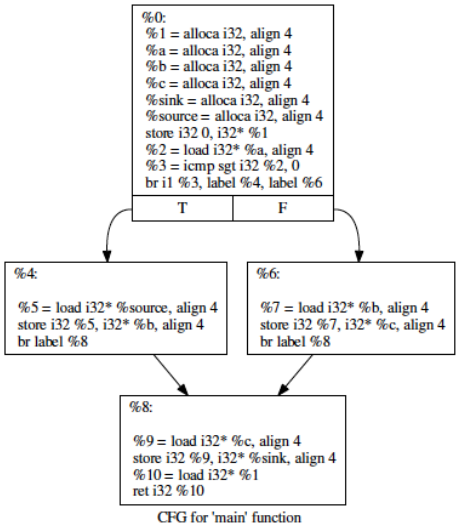
As a result, the variable **sink** is included in the tainted variable set.

In another example program

$$if\ [a > 0]^1\ then\ [b := source]^2\ else\ [c := b]^3; [sink := c]^4$$

it is slightly different from the previous program because the variable **source** was moved into the **if-else** statement. Its program analysis is described as follows.

Block 1: {source} // %0



CFG for 'main' function

First path
Block 2: {b, source} // %4
Block 3: {b, source} // %8

Second path
Block 4: {source} // %6
Block 5: {source} // %8

Tainted variables: {b, source}
// Join all the path outcomes, **Block 3** and **Block 5**

As a result, the variable **sink** does not appear in the final set this time.

Task 3

To support the taint analysis for a loop program, we shall adjust our previous algorithm with a fixed point to avoid an infinite loop of analysis.

Algorithm

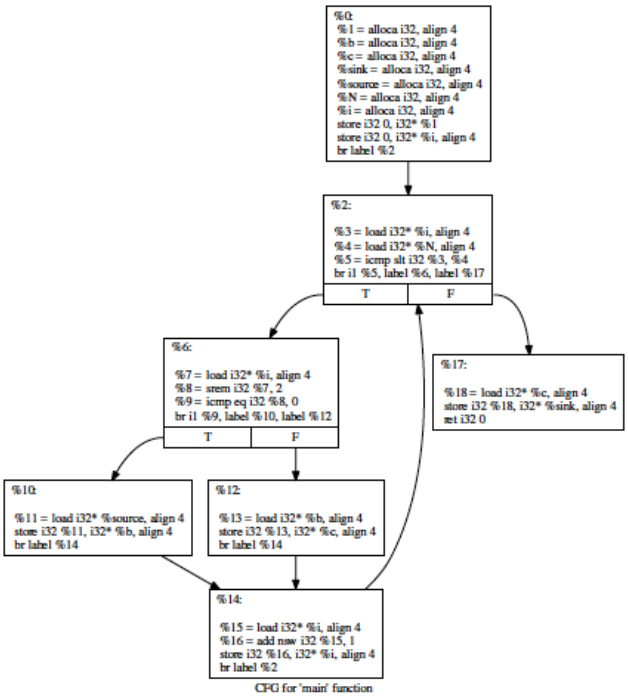
sourceVars: a tainted variable set for the entry point to a basic block **BB**
sinkVars: a tainted variable set for comparison of variable set change at the entry and exit of each block
traversalBlocks: a set that contains traversed blocks
x, **v**: some variable in an instruction
sinkVars := **sourceVars** before analyzing a basic block
If **v** from **sinkVars** is used for an instruction where the expression or assignment includes another variable **x**
    insert **x** into **sinkVars**, continue updating **sinkVars** in the same block
    move on to the next basic block **BB**
If **BB** already in **traversalBlocks** && (**sourceVars == sinkVars**)
    move on to the next path
    otherwise, update the **traversalBlocks** and **sourceVars**
If **# of successors** is **0**
    print out **sourceVars** to check if there is the variable **sink**.

In this example program

$$[i := 0]^1; while\ [i < N]^2\ do\ (if\ [i\ \%\ 2\ == 0]^3\ then\ [b := source]^4\ else\ [c := b]^5;\ \ [i := i + 1]^6); [sink := c]^7$$

there is an **if-else** statement inside the loop that form two possible paths on the analysis. Note that the algorithm does not print out a basic block which contains the same set of variables between the entry and exit.

Block 1: {source} // %0
Block 2: {source} // %2
Block 3: {source} // %6
Block 4: {b, source} // %10
Block 5: {b, source} // %14
Block 6: {b, c, source} // %12
Block 7: {b, c, sink, source} // %17
Tainted variables: {b, c, sink, source}
// same as Block 7



As a result, the variable **sink** appears in the final set.

Reference

[1] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. Data Flow Analysis. In *Principles of Program Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 35–139. DOI:https://doi.org/10.1007/978-3-662-03811-6_2