WEI YILEI A0276571W E1132304@u.nus.edu

Task 1: Designing Taint Analysis

Lattice:

Partially ordered by subset inclusion $\sqsubseteq = \subseteq$. The bottom is empty set and the top is set with all variables.

Type of the analysis:

Taint analysis is a may analysis with forward data flow. Because if any of the tainted variables is used to execute dangerous commands, the taint checker warns that the program is using a potentially dangerous tainted variable, but not every variable that is warned is necessarily contaminated by "source". As for the direction of the data flow analysis, since the program is running one line of code after another in the forward direction, for each block, we first get its entry set, then combine it with the code in the block to explore the set of variables that may be tainted, and finally generate the exit set. So it's forward and may.

Data structure used in the analysis:

std::map<std::string, std::set<std::string>> analysisMap: A collection of variables used to record the tainted values of each block.
std::stack<std::pair<BasicBlock*,std::set<std::string> > > traversalStack: During the program loop, any sub-block that meets the conditions will be pushed onto the stack. The program won't terminate unless the stack is empty.
std::set<std::string> findTaintedVars(BasicBlock*,std::set<std::string>): For a particular block, given its entry set, the function will get the exit set.

Gen and kill functions:

$$gen_I([z := a]^l) = \begin{cases} \{z\}, & \text{if } \exists a' \in FV(a),\ a' \in TV_{entry}(l) \cup TV_{before}(l) \\ \emptyset, & \text{otherwise} \end{cases}$$

$$gen_I([skip]^l) = \emptyset$$

$$gen_I([b]^l) = \emptyset$$

$$kill_I([z := a]^l) = \begin{cases} \{z\}, & \text{if } \forall a' \in FV(a),\ a' \notin TV_{entry}(l) \cup TV_{before}(l) \\ \emptyset, & \text{otherwise} \end{cases}$$

$$kill_I([skip]^l) = \emptyset$$

$$kill_I([b]^l) = \emptyset$$

Entry and exit equations:

$$TV_{entry}(l) = \begin{cases} \emptyset, & \text{if } l = init(S_F) \\ \cup\{TV_{exit}(l') \mid (l', l) \in flow(S_F)\}, & \text{otherwise} \end{cases}$$

$$TV_{exit}(l) = (TV_{entry}(l) \setminus kill_I(B^l)) \cup gen_I(B^l), \text{ where } B^l \in block(S_F)$$

Reach fixpoint:

Only add successor nodes to the stack if the the new union set of tainted variables for the successor node is different from the currently stored set of tainted variables for the successor node.

$$[z:=1]^1; \text{while } [x>0]^2 \text{ do } ([z:=z*y]^3; [x:=x-1]^4)$$

For example, only when the exit set of block 4 is different from the currently stored set of tainted variables of block 2, will we add pair<block2, tainted variable set> to the stack.

Task 2: Implementing the Taint Analysis in LLVM

After initialization, the entry block is pushed onto the stack, after which it keeps removing block from the stack until the stack is empty. Each time a block is taken out, it is analyzed to get the corresponding set containing tainted variables, and its sub-blocks are pushed onto the stack.

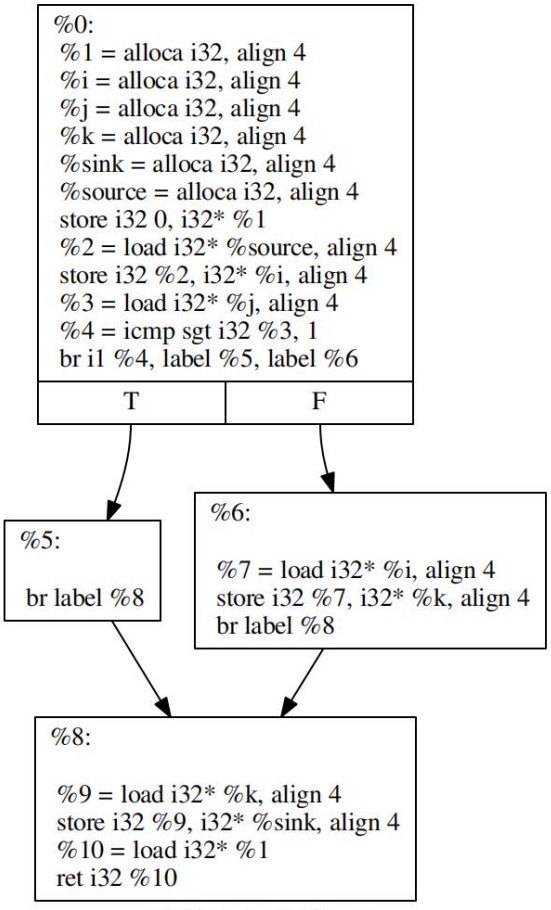
Algorithm used:

```
std::set<std::string> findTaintedVars(BasicBlock*,std::set<std::string>);
std::string getSimpleNodeLabel(const BasicBlock *Node);
std::string getSimpleVariableName(const Instruction *Ins);
std::string getSimpleValueName(const Value *Value);
std::set<std::string> union_sets(std::set<std::string> A, std::set<std::string> B);
void printAnalysisMap(std::map<std::string,std::set<std::string>> analysisMap);
```

Example1:

Program:

```c
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)
        ;
    else
        k = i;
    sink = k;
}
```



CFG for 'main' function

CFG:

- entry: {source, i}
- if.then: {source, i}
- if.else: {source, i, k}
- if.end: {source, i, k, sink}

Expected output:

Analysis output:

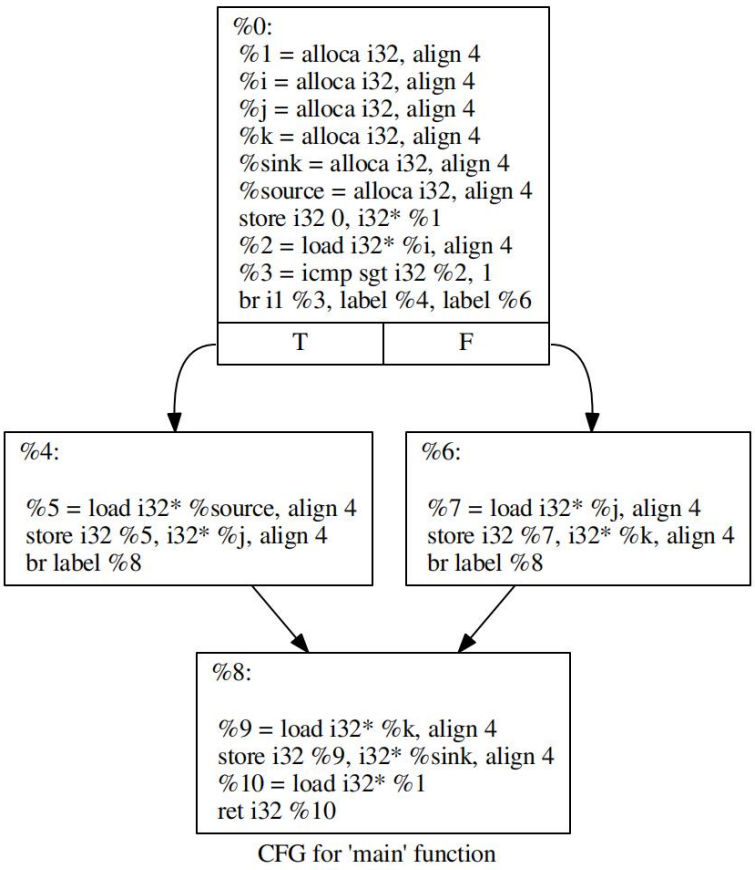
```
PRINTING ANALYSIS MAP:
%0:
        %2      i       source
%5:
        %2      i       source
%6:
        %2      %7      i       k       source
%8:
        %2      %7      %9      i       k       sink    source
```

Note: %0 -> “entry”; 5% -> “if.then”; 6% -> “if.else”; 8% -> “if.end”

Example2:

Program:

```
1   int main() {
2       int i,j,k,sink, source;
3   // read source from input
4       if (i > 1)
5           j = source;
6       else
7           k = j;
8       sink = k;
9   }
10
```



CFG for 'main' function

CFG:

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

Expected output:

Analysis output:

```
PRINTING ANALYSIS MAP:
%0:
        source
%4:
        %5      j       source
%6:
        source
%8:
        %5      j       source
```

Note: %0 -> “entry”; 4% -> “if.then”; 6% -> “if.else”; 8% -> “if.end”

Task 3: Determine the Abstract Tainted Data Flow Paths

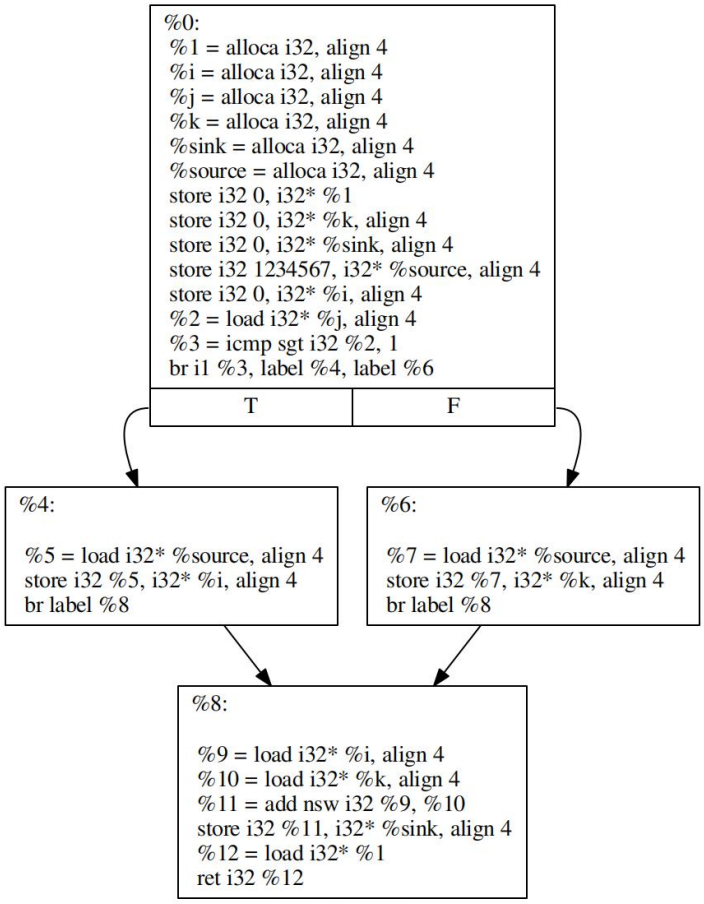
Based on task2, the path of each tainted variable is recorded in a roadMap in real time. In the analysis of each basic block, every time a variable is tainted, the path of its tainted cause is updated in time. Finally, we traverse the roadMap and output the analysis results.

Algorithm used:

```
std::set<std::string> findTaintedVars(BasicBlock*,std::set<std::string>);
std::string getSimpleNodeLabel(const BasicBlock *Node);
std::string getSimpleVariableName(const Instruction *Ins);
std::string getSimpleValueName(const Value *Value);
std::set<std::string> union_sets(std::set<std::string> A, std::set<std::string> B);
void printAnalysisMap(std::map<std::string,std::set<std::string>> analysisMap);
void printRoadMap(std::map<std::string, std::vector<std::string>> roadMap);
std::map<std::string, std::vector<std::string>> roadMap;
int roadNum = 0;
```

Program:

```
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 (j > 1)
        i=source;
    else
        k = source;
    sink = i+k;
}
```



CFG for 'main' function

CFG:

Expected output: something like this, spread to every variable.

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.

Analysis output:

```
PRINTING ROAD MAP:
%10:
%0 %6 %8
%11:
%0 %4 %8
%5:
%0 %4
%7:
%0 %6
%9:
%0 %4 %8
i:
%0 %4
k:
%0 %6
sink0:
%0 %6 %8
sink1:
%0 %4 %8
source:
%0
```

Note:

For block “entry”(%0):

Tainted variable: ***source***, path: {entry(%0)}

For block “if.then”(%4):

Tainted variable: ***i***, path: {entry(%0) -> if.then(%4)}

For block “if.else”(%6):

Tainted variable: ***k***, path: {entry(%0) -> if.else(%6)}

For block “if.end”(%8):

Tainted variable: ***sink***, path: {entry(%0) -> if.then(%4) -> if.end(%8)
entry(%0) -> if.else(%6) -> if.end(%8)}

Task 4: Adding Support for Loops to the Analysis

Implementation is the same as for task 2. The only difference is the addition of a new judgment when determining whether a sub-block of each block should be pushed onto the stack or not, so called fixpoint judgement.

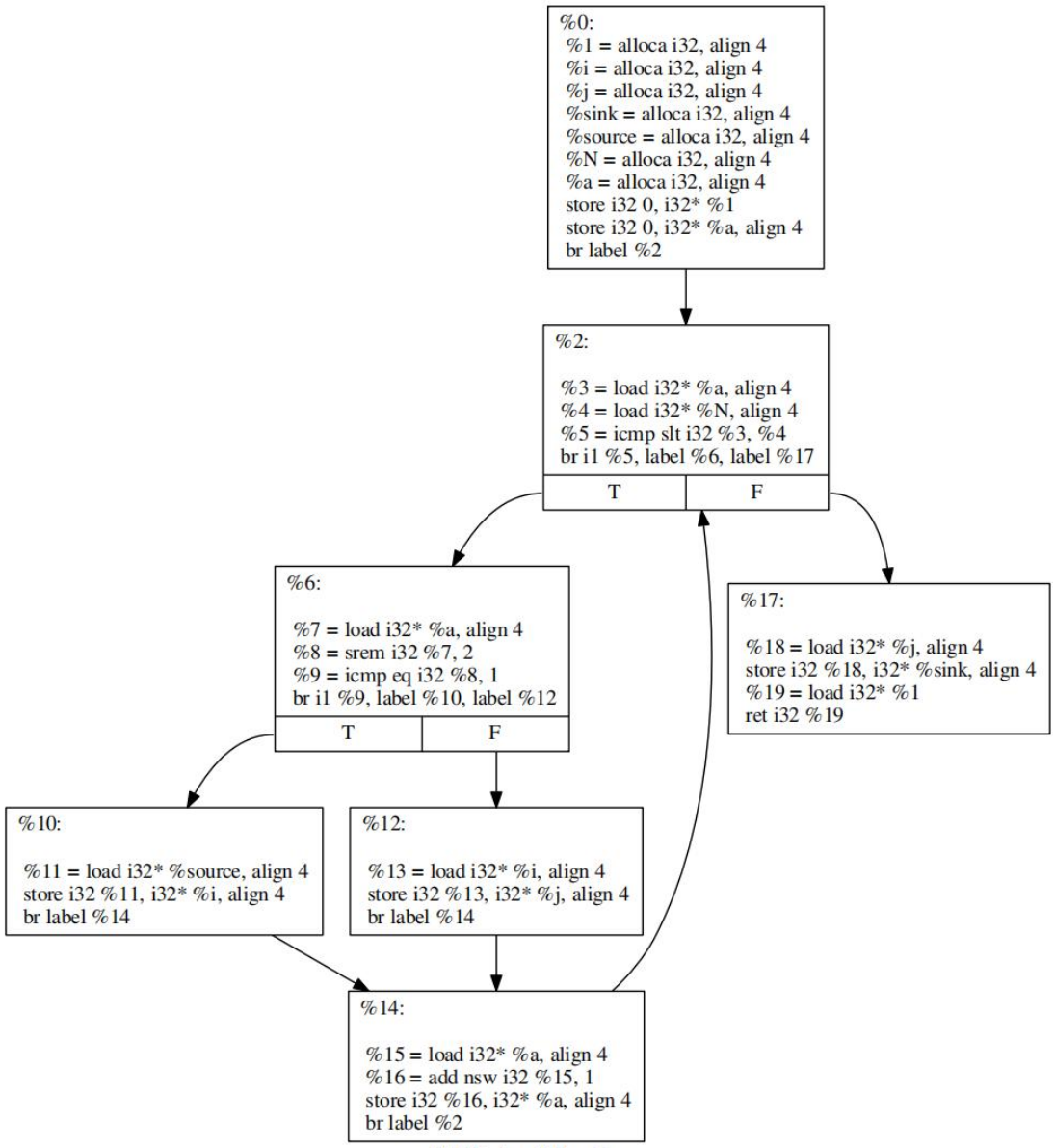
```
const TerminatorInst *TInst = BB->getTerminator();
int NSucc = TInst->getNumSuccessors();
for (int i = 0;  i < NSucc; ++i) {
    BasicBlock *Succ = TInst->getSuccessor(i);
    std::set<std::string> succTaintedVars = analysisMap[getSimpleNodeLabel(Succ)];
    if(succTaintedVars != exitTaintedVars){
        std::pair<BasicBlock*,std::set<std::string> > succAnalysisNode = std::make_pair(Succ, updatedTaintedVars);
        traversalStack.push(succAnalysisNode);
        std::string BBLabel = getSimpleNodeLabel(Succ);
        errs() << "Adding to the stack:\n\t" << BBLabel << ": \n";
        for (const std::string& str : updatedTaintedVars){
            errs() << "\t" << str;
        }
    errs() << "\n";
    }
}
```

Algorithm used: same as task2.

Program:

```
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;
}
```

CFG:



CFG for 'main' function

- 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}

Expected output:

Analysis output:

```
PRINTING ANALYSIS MAP:
%0:
        source
%10:
        %11     %13     i       j       source
%12:
        %11     %13     i       j       source
%14:
        %11     %13     i       j       source
%17:
        %11     %13     %18     i       j       sink    source
%2:
        %11     %13     i       j       source
%6:
        %11     %13     i       j       source
```

Note:

Entry(%0): {source}

While.cond(%2): {source, i, j}

While.body(%6): {source, i, j}

If.then(%10): {source, i, j}

If.else(%12): {source, i, j}

If.end(%14): {source, i, j}

While.end(%17): {source, i, j, sink}

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

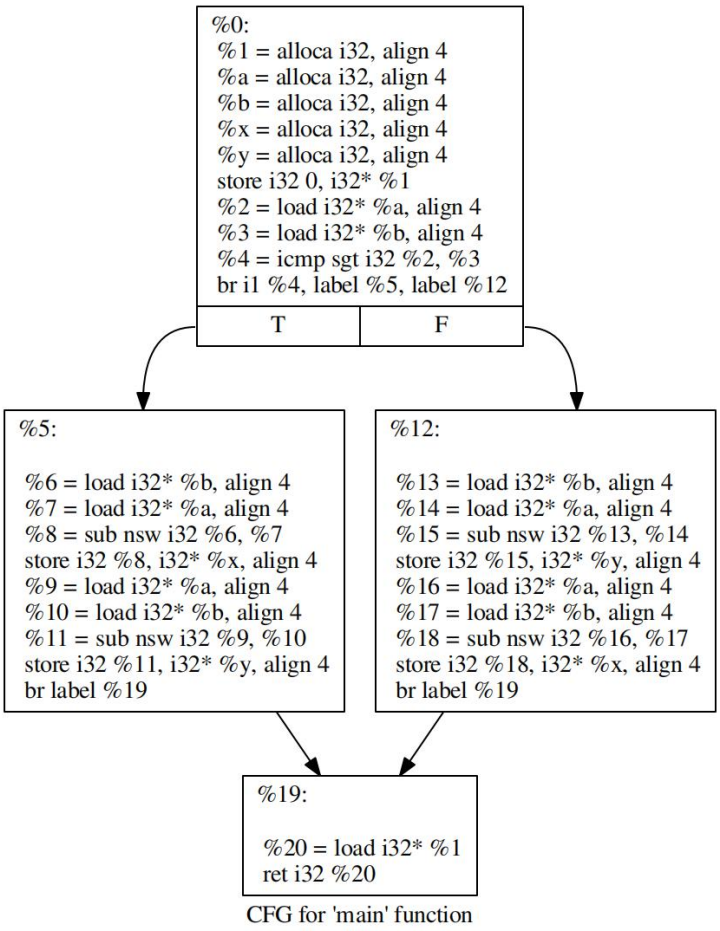
In contrast to tainted variable analysis, we first push the last exit block onto the stack, and then we take out elements of the stack each time we loop until the stack is empty. For each block we take out, we update the exit set and push all the instructions in it into a new instruction stack, so that we can analyze the instructions in the block backwards from the bottom to the top. For each instruction analyzed, we update the busy expressions until the last instruction is analyzed, record the result in the block's entry set, and push its parent block onto the stack.

Algorithm used:

```
std::set<std::string> findBusyExpressions(BasicBlock*,std::set<std::string>);
std::string getSimpleNodeLabel(const BasicBlock *Node);
std::string getSimpleVariableName(const Instruction *Ins);
std::string getSimpleValueName(const Value *Value);
std::set<std::string> union_sets(std::set<std::string> A, std::set<std::string> B);
std::set<std::string> intersect_sets(std::set<std::string>, std::set<std::string>);
void printAnalysisMap(std::map<std::string,std::set<std::string>> analysisMap);
std::map<std::string, std::set<std::string>> analysisMap;
std::map<std::string, std::set<std::string>> printMap;
```

Program:

```
int main(){
    int a,b, x,y;
    if(a > b){
        x = b - a;
        y = a - b;
    }else{
        y = b - a;
    x = a - b;
    }
}
```



CFG for 'main' function

CFG:

Analysis output: if we take every basic block as a unit, then the output should be like this:

```
PRINTING BUSY MAP:
%0entry:
        a-b     b-a
%0exit:
        a-b     b-a
%12entry:
        a-b     b-a
%12exit:

%19entry:

%19exit:

%5entry:
        a-b     b-a
%5exit:
```

Note:

%0 -> “entry”; 5% -> “if.then”; 12% -> “if.else”; 19% -> “if.end”