CS5218 Principal of Program Analysis

Assignment 1
Wang Jiadong (A0105703)

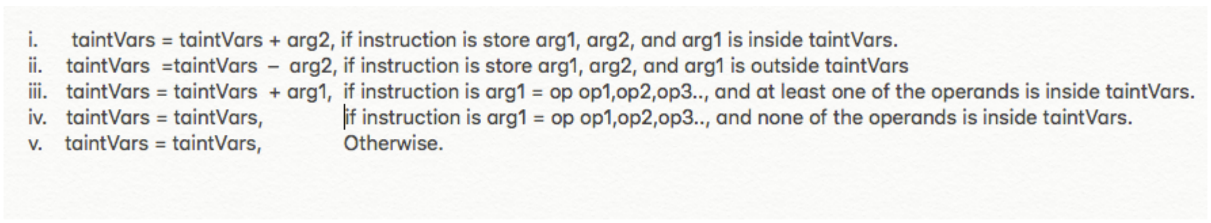
Task1: Designing Taint Analysis

Taint Analysis is to detect which of the variables are modified by user inputs. It is ubiquitously used to prevent security bleach. In order to design a taint detection analysis framework, we need to define some parameters and specifications.

- Flow (F).
  Intuitively, taint detection supposes to to flow from top down. Once a variable is contaminated by user input, the other variables may be contaminated through assignment of these contaminated variables. Hence, the analysis flow should be the same as the program execution flow.
- Starting Block(E).
  Since the analysis flow is top-down, then it is very clearly the starting block is init(S*).
- Initial Value(ι).
  Initially, there are no contaminated variables at the initial state, hence the initial value set is ø (empty set).
- May or Must.
  As long as there is a path causing one specific variable contaminated by user input, we consider it as contaminated. Hence we use the ∪ (Or Logic) to represent the "May" condition.
- Lattice and Partial Order
  Here, we use a lattice, with each node as an element the power set of all variables $\mathcal{P}(\mathbf{Var}_\star)$. Hence, each node represents a possible combination of contaminated variables. In this lattice, the bottom is ø (empty set) while the top is the complete set of all variables. Hence, the lattice is partially ordered by subset inclusion: $\sqsubseteq = \subseteq$
- Monotonicity
  As mentioned above, the top element is the lattice, is the complete set of all variables, while the bottom element is an empty set. Hence, for any pair of element "a" and element "b" in this lattice, if there is a path from a to b and a <= b, then a is a subset of b.
- Transfer Functions.
  Transfer function can be represented as
  $$f_\ell(l) = (l \backslash kill([B]^\ell)) \cup gen([B]^\ell) \textbf{ where } [B]^\ell \in blocks(S_\star)$$
  , where:

  Kill[a = exp] $^{\ell}$ = {a} when a is tainted at the entry while none of the variables inside the exp is tainted.

  Gen[a = exp] $^{\ell}$ = {a} when a is not tainted at the entry while one or more of the variables inside the exp is tainted.
  When inside the LLVM context:



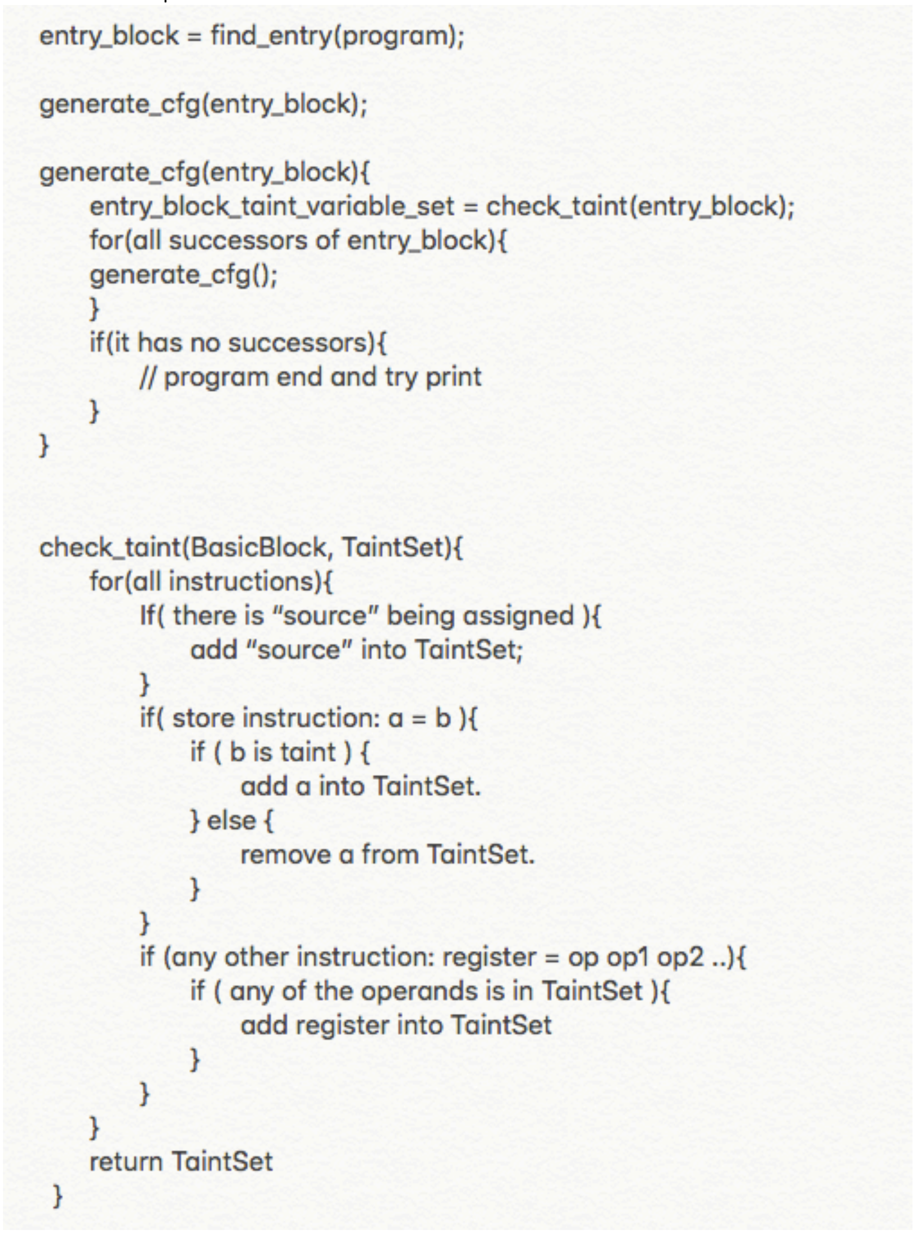
```
i.    taintVars = taintVars + arg2, if instruction is store arg1, arg2, and arg1 is inside taintVars.
ii.   taintVars =taintVars  –  arg2, if instruction is store arg1, arg2, and arg1 is outside taintVars
iii.  taintVars = taintVars  + arg1,  if instruction is arg1 = op op1,op2,op3.., and at least one of the operands is inside taintVars.
iv.   taintVars = taintVars,          if instruction is arg1 = op op1,op2,op3.., and none of the operands is inside taintVars.
v.    taintVars = taintVars,          Otherwise.
```

Figure 1 Transfer Function in LLVM.

Task 2: Implementing Loop-Free Taint Analysis Based On LLVM

- Design.

  Task 2 requires us to design a LLVM pass to identify the taint variables for loop free programs. Since it is loop free, we can find the fix point with only one iteration. Hence the idea is that, we find the entry point of the program firstly, and then, block by block, we find the taint variables, feeding them to the successors. This process will terminate at the end point. Since it is an isolated entry & isolated exit program, there should be only one entry point and one exit point.



```
entry_block = find_entry(program);

generate_cfg(entry_block);

generate_cfg(entry_block){
    entry_block_taint_variable_set = check_taint(entry_block);
    for(all successors of entry_block){
    generate_cfg();
    }
    if(it has no successors){
        // program end and try print
    }
}


check_taint(BasicBlock, TaintSet){
    for(all instructions){
        If( there is "source" being assigned ){
            add "source" into TaintSet;
        }
        if( store instruction: a = b ){
            if ( b is taint ) {
                add a into TaintSet.
            } else {
                remove a from TaintSet.
            }
        }
        if (any other instruction: register = op op1 op2 ..){
            if ( any of the operands is in TaintSet ){
                add register into TaintSet
            }
        }
    }
    return TaintSet
 }
```

Figure 2 Pseudo-code for loop-free taint detection.

- Building & Run

  The code is attached, and building process is straight forward:

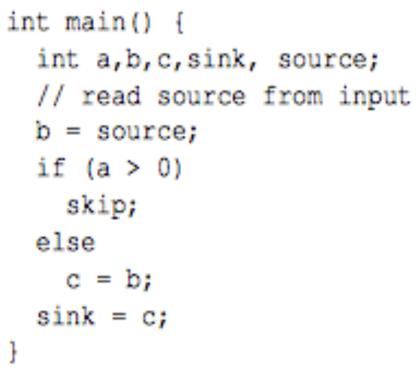
$$\begin{aligned}\mathrm{clang}++-3.5\ \ -\mathrm{o}\ \mathrm{TaintAnalysis}\ \mathrm{TaintAnalysis.cpp}\ \text{`}\mathrm{llvm}-\mathrm{config}-3.5\\ --\mathrm{cxxflags}\text{` `}\mathrm{llvm}-\mathrm{config}-3.5\ \ --\mathrm{ldflags}\text{` `}\mathrm{llvm}-\mathrm{config}-3.5\\ --\mathrm{libs}\text{`}\ \ -\mathrm{lpthread}\ \ -\mathrm{lncurses}-\mathrm{ldl}\end{aligned}$$

  This will give us the TaintAnalysis Pass. We can use this pass to conduct taint analysis on testing programs, as:

  ./TaintAnalysis test1.ll.

- Results

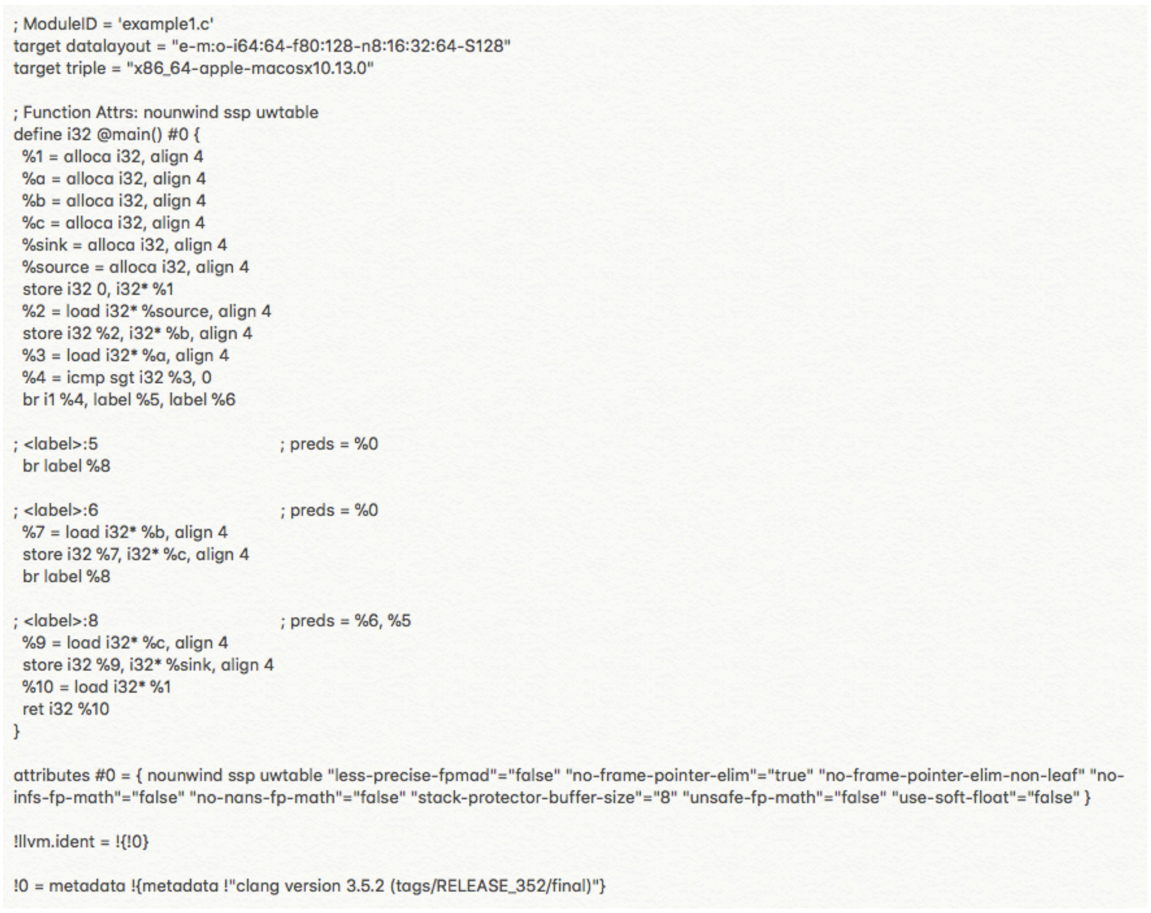
  Test Case 1 Source Code:



```
int main() {
  int a,b,c,sink, source;
  // read source from input
  b = source;
  if (a > 0)
    skip;
  else
    c = b;
  sink = c;
}
```

Figure 3 Test Example 1 source code.

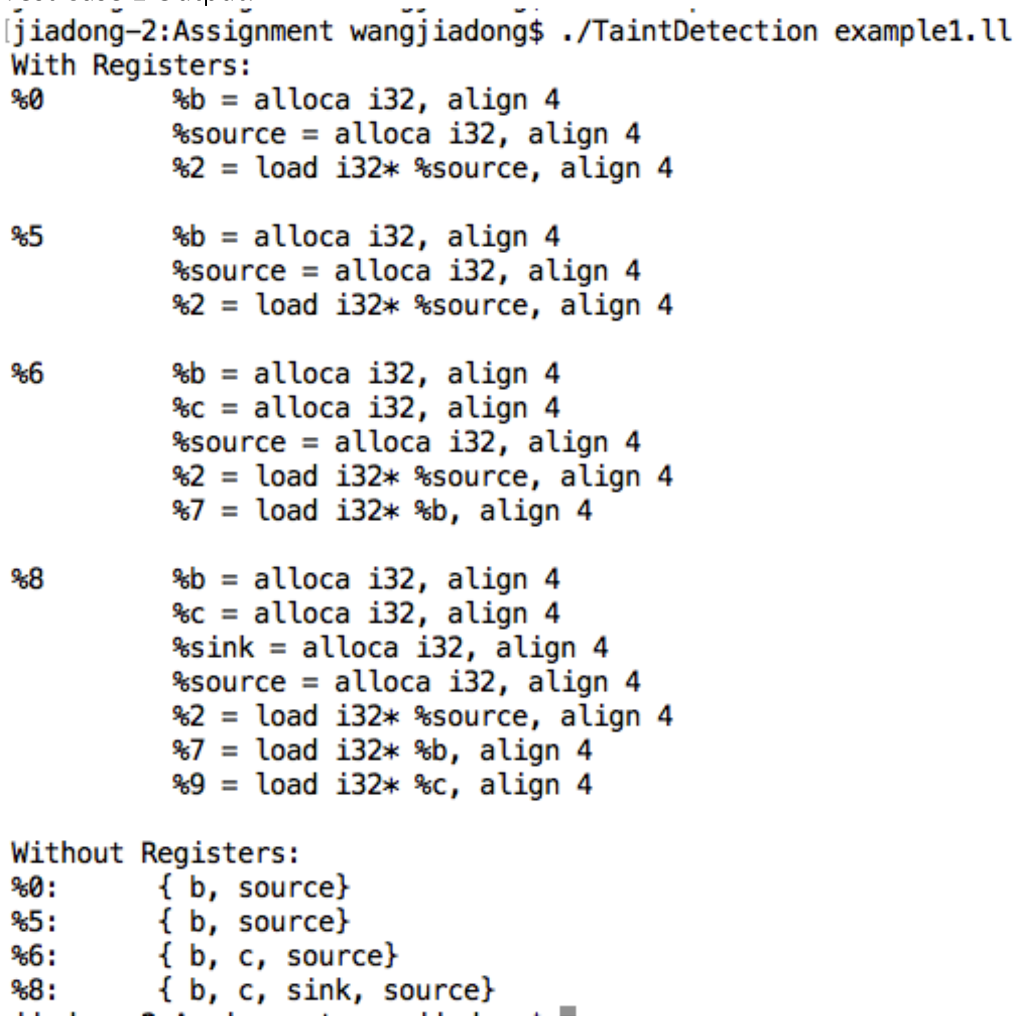
Test Case 1 IR Code:



```
; ModuleID = 'example1.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %sink = alloca i32, align 4
  %source = alloca i32, align 4
  store i32 0, i32* %1
  %2 = load i32* %source, align 4
  store i32 %2, i32* %b, align 4
  %3 = load i32* %a, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5                                       ; preds = %0
  br label %8

; <label>:6                                       ; preds = %0
  %7 = load i32* %b, align 4
  store i32 %7, i32* %c, align 4
  br label %8

; <label>:8                                       ; preds = %6, %5
  %9 = load i32* %c, align 4
  store i32 %9, i32* %sink, align 4
  %10 = load i32* %1
  ret i32 %10
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-
infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.5.2 (tags/RELEASE_352/final)"}
```

Figure 4 Test Example 1 IR.

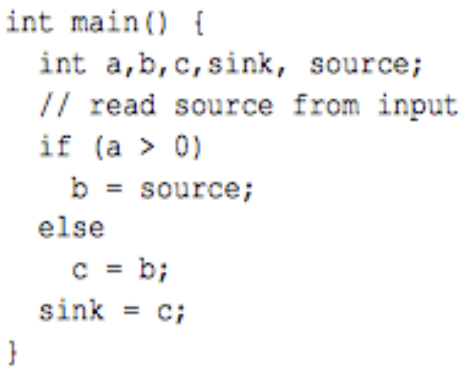
Test Case 1 Output:



```
jiadong-2:Assignment wangjiadong$ ./TaintDetection example1.ll
With Registers:
%0         %b = alloca i32, align 4
           %source = alloca i32, align 4
           %2 = load i32* %source, align 4

%5         %b = alloca i32, align 4
           %source = alloca i32, align 4
           %2 = load i32* %source, align 4

%6         %b = alloca i32, align 4
           %c = alloca i32, align 4
           %source = alloca i32, align 4
           %2 = load i32* %source, align 4
           %7 = load i32* %b, align 4

%8         %b = alloca i32, align 4
           %c = alloca i32, align 4
           %sink = alloca i32, align 4
           %source = alloca i32, align 4
           %2 = load i32* %source, align 4
           %7 = load i32* %b, align 4
           %9 = load i32* %c, align 4

Without Registers:
%0:      { b, source}
%5:      { b, source}
%6:      { b, c, source}
%8:      { b, c, sink, source}
```

Figure 5: Test Example 1 Analysis Output.

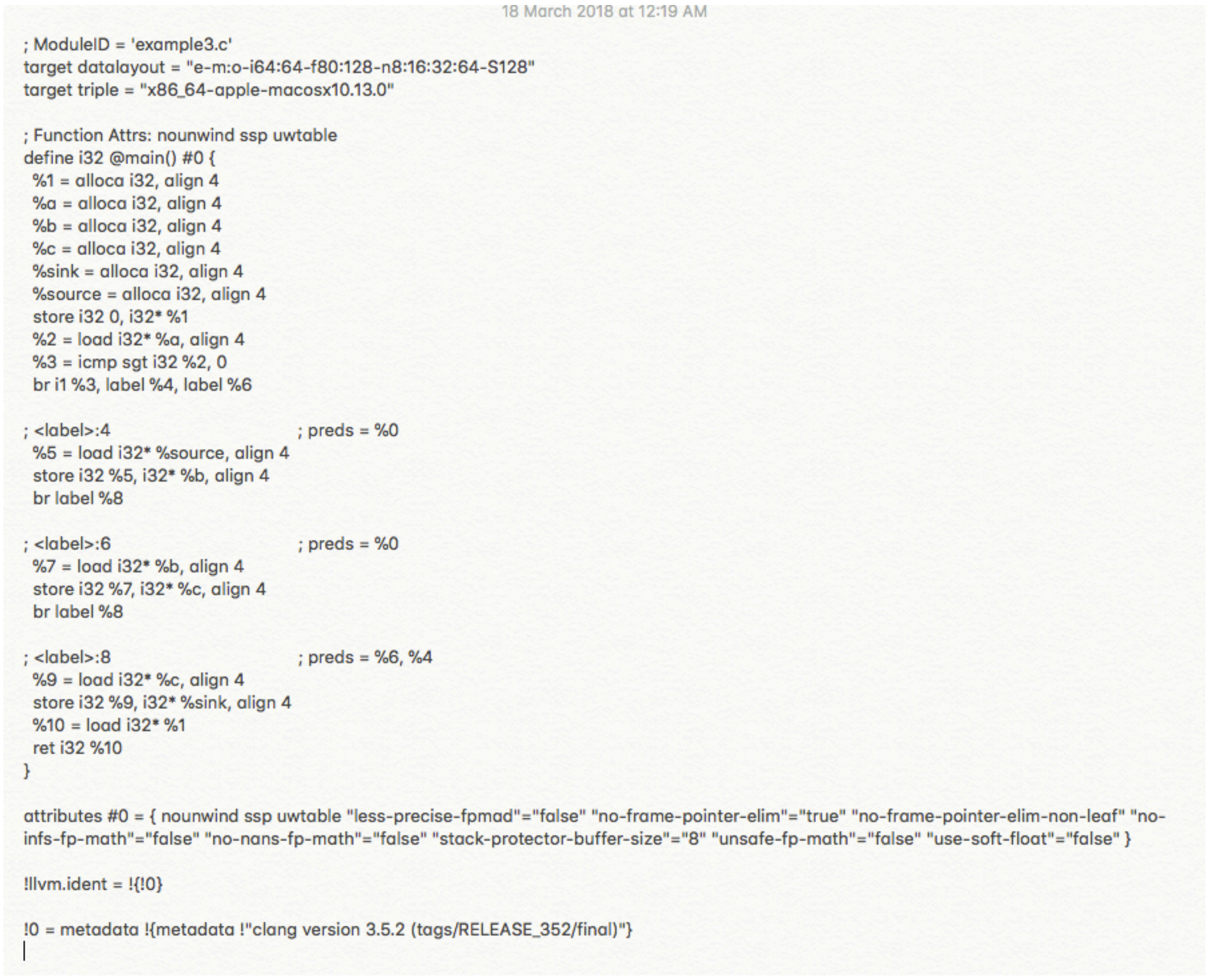
Test Case 2 Source Code:



```
int main() {
  int a,b,c,sink, source;
  // read source from input
  if (a > 0)
    b = source;
  else
    c = b;
  sink = c;
}
```

Figure 6 Test Example 2 Source Code.

Test Case 2 IR Source Code:

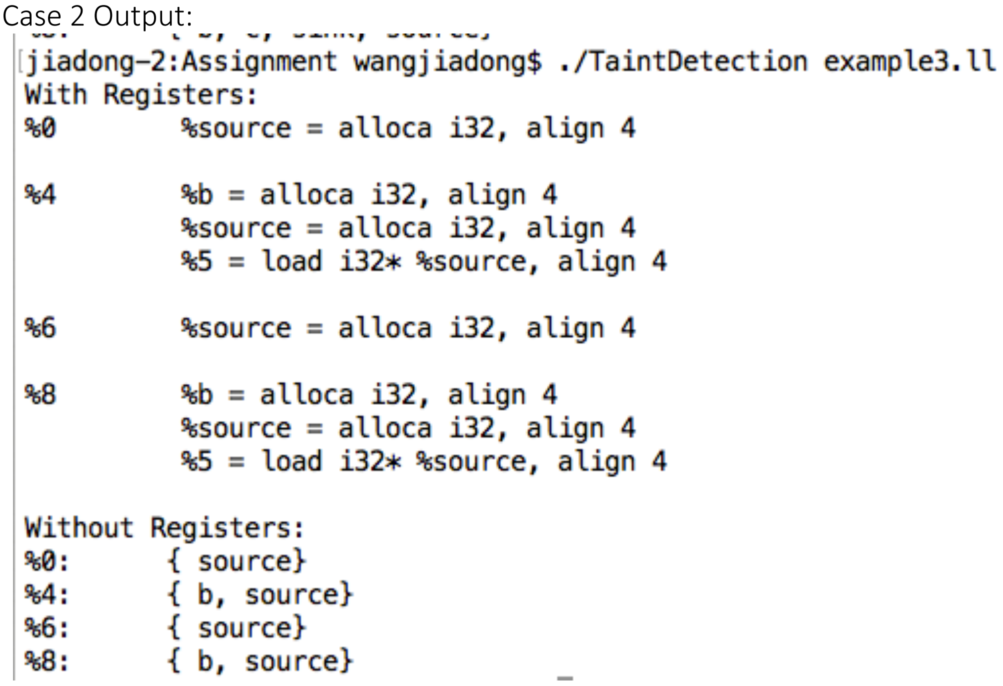


18 March 2018 at 12:19 AM

```
; ModuleID = 'example3.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
 %1 = alloca i32, align 4
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %sink = alloca i32, align 4
 %source = alloca i32, align 4
 store i32 0, i32* %1
 %2 = load i32* %a, align 4
 %3 = icmp sgt i32 %2, 0
 br i1 %3, label %4, label %6

; <label>:4                                       ; preds = %0
 %5 = load i32* %source, align 4
 store i32 %5, i32* %b, align 4
 br label %8

; <label>:6                                       ; preds = %0
 %7 = load i32* %b, align 4
 store i32 %7, i32* %c, align 4
 br label %8

; <label>:8                                       ; preds = %6, %4
 %9 = load i32* %c, align 4
 store i32 %9, i32* %sink, align 4
 %10 = load i32* %1
 ret i32 %10
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.5.2 (tags/RELEASE_352/final)"}
```

Figure 7 Test Example 2 IR File.

Test Case 2 Output:



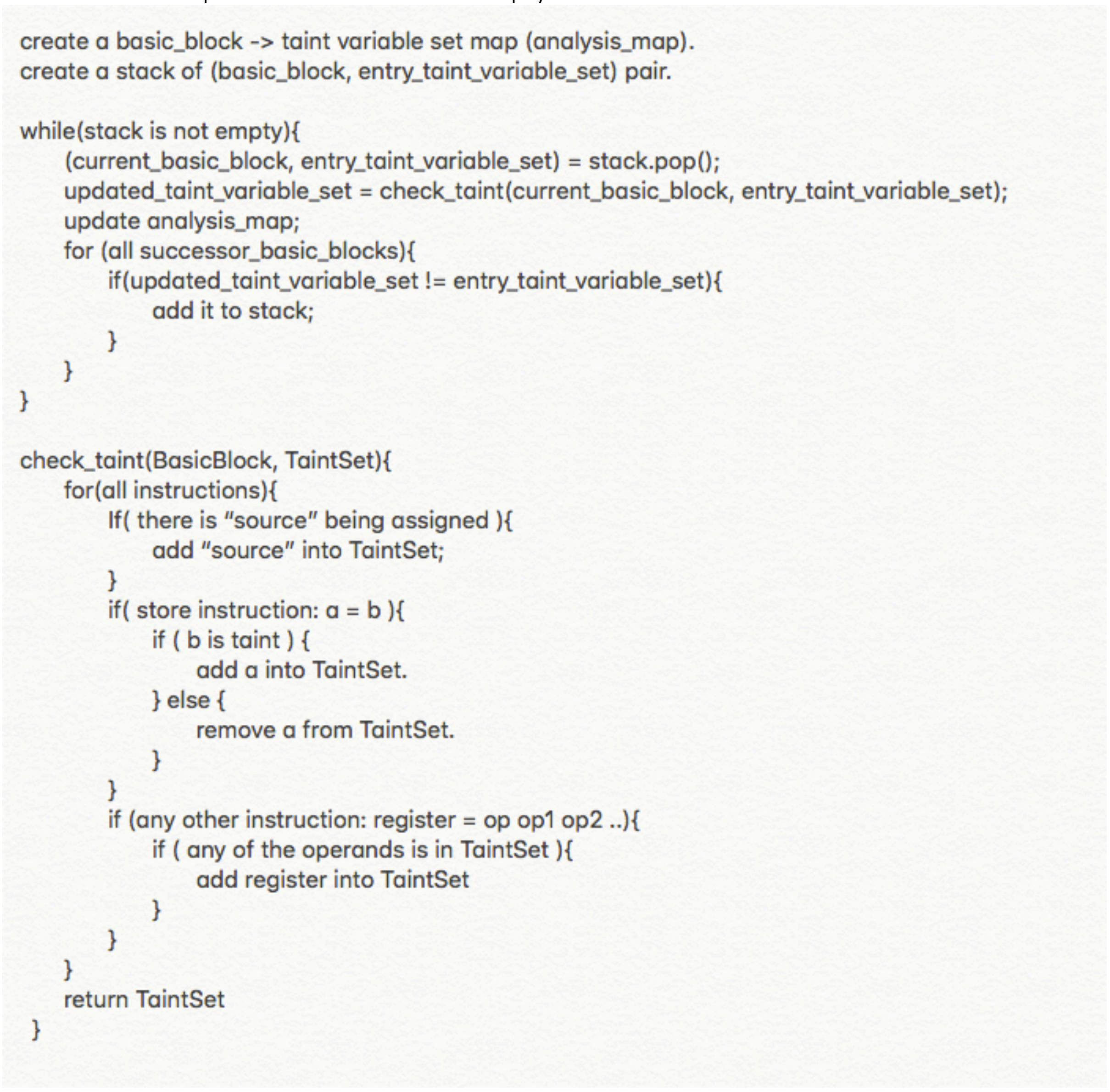
```
jiadong-2:Assignment wangjiadong$ ./TaintDetection example3.ll
With Registers:
%0          %source = alloca i32, align 4

%4          %b = alloca i32, align 4
            %source = alloca i32, align 4
            %5 = load i32* %source, align 4

%6          %source = alloca i32, align 4

%8          %b = alloca i32, align 4
            %source = alloca i32, align 4
            %5 = load i32* %source, align 4

Without Registers:
%0:         { source}
%4:         { b, source}
%6:         { source}
%8:         { b, source}
```

Figure 8 Test Example 2 Output.

Task 3. Adding Support for Loop in Taint Analysis

- Design

  After considering the while loops, the taint analysis becomes more complicated. In this program, we are adopting the depth-first strategy. In order to do so, we use a stack to hold the list of analysis outputs for basic blocks. Every time, we take the top basic block from the stack, and conduct taint analysis. Once completed, we will compare the new outcome with that from last iteration. If the output is updated, which means we have not reached the fix point yet, we will remove this basic block from the stack, and place its successors into it, with the updated entry set, and conduct the analysis consequently. We will reach the fix point once the stack is empty.



```
create a basic_block -> taint variable set map (analysis_map).
create a stack of (basic_block, entry_taint_variable_set) pair.

while(stack is not empty){
    (current_basic_block, entry_taint_variable_set) = stack.pop();
    updated_taint_variable_set = check_taint(current_basic_block, entry_taint_variable_set);
    update analysis_map;
    for (all successor_basic_blocks){
        if(updated_taint_variable_set != entry_taint_variable_set){
            add it to stack;
        }
    }
}

check_taint(BasicBlock, TaintSet){
    for(all instructions){
        If( there is "source" being assigned ){
            add "source" into TaintSet;
        }
        if( store instruction: a = b ){
            if ( b is taint ) {
                add a into TaintSet.
            } else {
                remove a from TaintSet.
            }
        }
        if (any other instruction: register = op op1 op2 ..){
            if ( any of the operands is in TaintSet ){
                add register into TaintSet
            }
        }
    }
    return TaintSet
 }
```

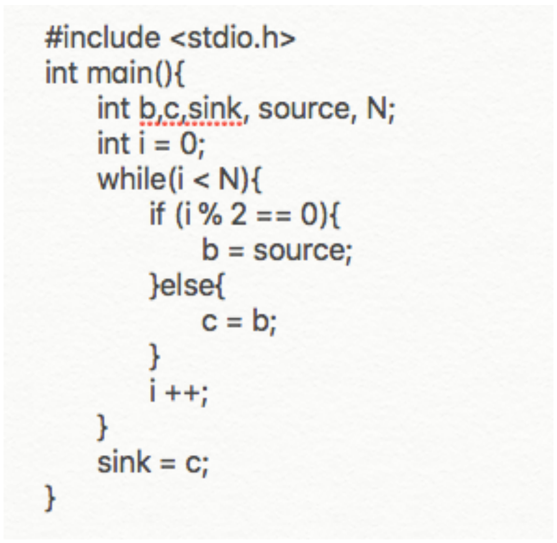
Figure 9 With-Loop Taint Analysis.

- Building & Run

  The building process is similar with task 2.

- Results

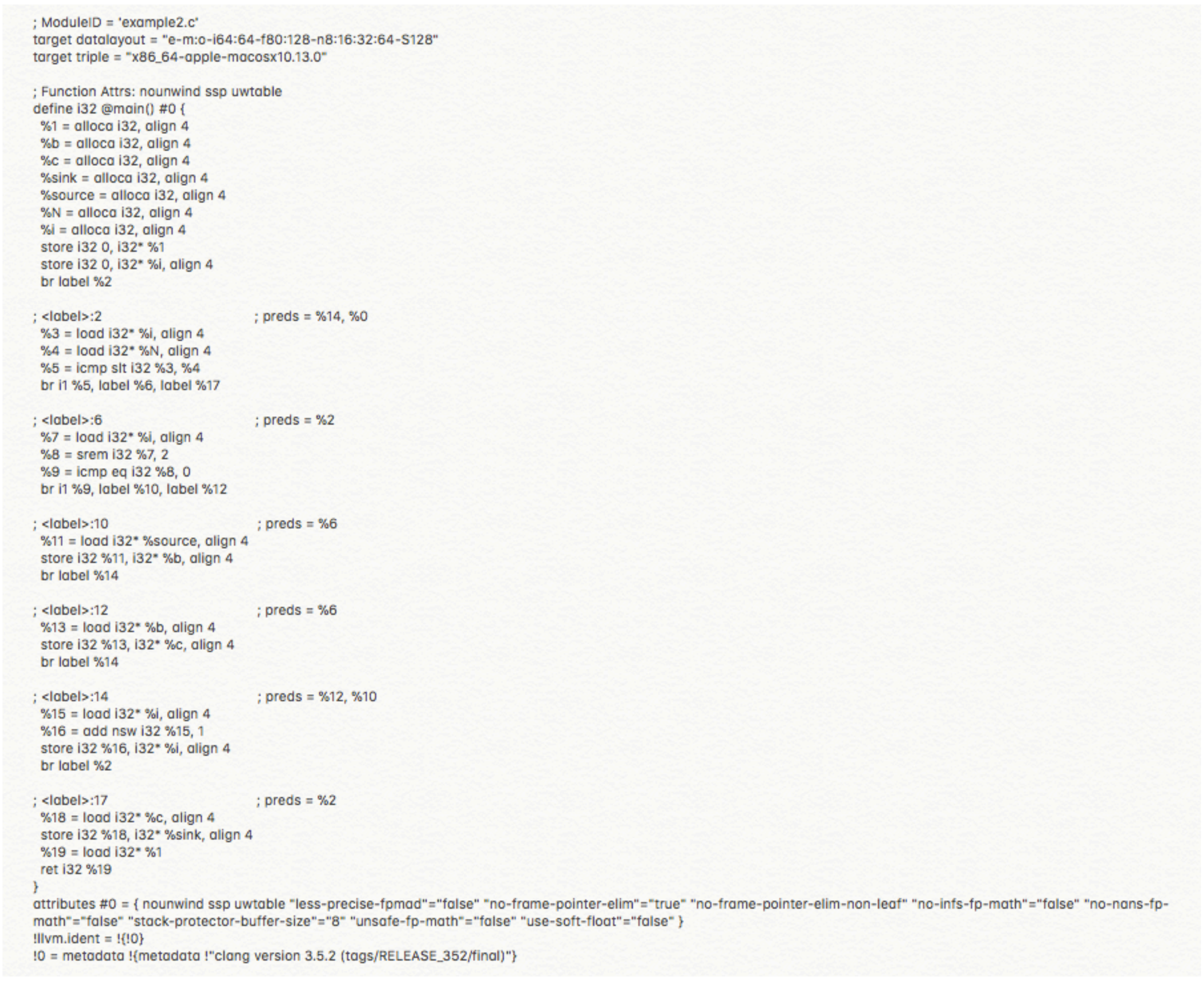
  Test Case 3 Source Code:



```c
#include <stdio.h>
int main(){
      int b,c,sink, source, N;
      int i = 0;
      while(i < N){
            if (i % 2 == 0){
                  b = source;
            }else{
                  c = b;
            }
            i ++;
      }
      sink = c;
}
```

Figure 10: Test Example 3 Source Code.

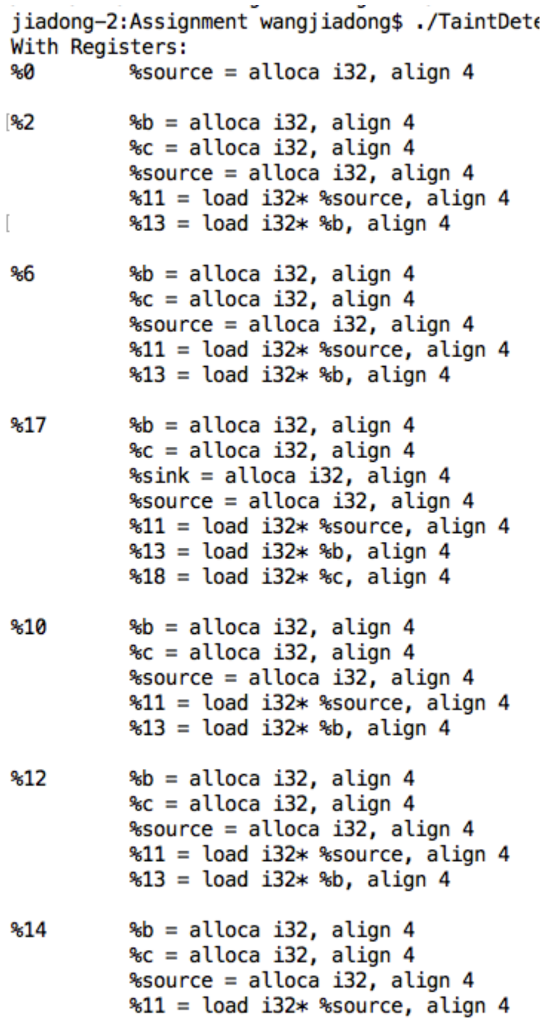
Test Case 3 IR Source Code:



```
; ModuleID = 'example2.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %sink = alloca i32, align 4
  %source = alloca i32, align 4
  %N = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2                                       ; preds = %14, %0
  %3 = load i32* %i, align 4
  %4 = load i32* %N, align 4
  %5 = icmp slt i32 %3, %4
  br i1 %5, label %6, label %17

; <label>:6                                       ; preds = %2
  %7 = load i32* %i, align 4
  %8 = srem i32 %7, 2
  %9 = icmp eq i32 %8, 0
  br i1 %9, label %10, label %12

; <label>:10                                      ; preds = %6
  %11 = load i32* %source, align 4
  store i32 %11, i32* %b, align 4
  br label %14

; <label>:12                                      ; preds = %6
  %13 = load i32* %b, align 4
  store i32 %13, i32* %c, align 4
  br label %14

; <label>:14                                      ; preds = %12, %10
  %15 = load i32* %i, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %i, align 4
  br label %2

; <label>:17                                      ; preds = %2
  %18 = load i32* %c, align 4
  store i32 %18, i32* %sink, align 4
  %19 = load i32* %1
  ret i32 %19
}
attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-
math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.ident = !{!0}
!0 = metadata !{metadata !"clang version 3.5.2 (tags/RELEASE_352/final)"}
```

Figure 11 Test Example 3 IR File.

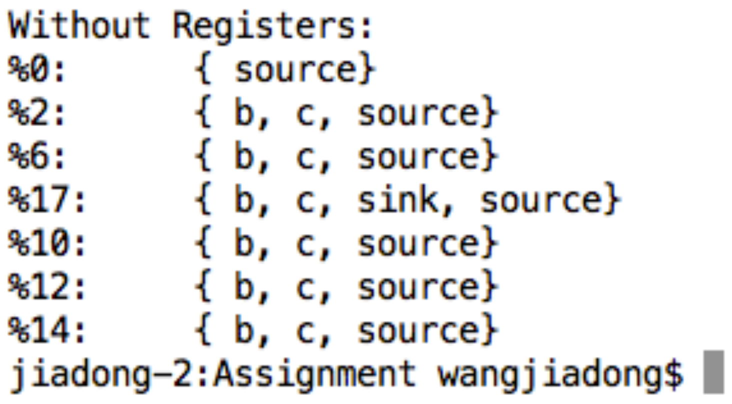
Test Case 3 Output I:



```
jiadong-2:Assignment wangjiadong$ ./TaintDete
With Registers:
%0        %source = alloca i32, align 4

%2        %b = alloca i32, align 4
          %c = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
          %13 = load i32* %b, align 4

%6        %b = alloca i32, align 4
          %c = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
          %13 = load i32* %b, align 4

%17       %b = alloca i32, align 4
          %c = alloca i32, align 4
          %sink = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
          %13 = load i32* %b, align 4
          %18 = load i32* %c, align 4

%10       %b = alloca i32, align 4
          %c = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
          %13 = load i32* %b, align 4

%12       %b = alloca i32, align 4
          %c = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
          %13 = load i32* %b, align 4

%14       %b = alloca i32, align 4
          %c = alloca i32, align 4
          %source = alloca i32, align 4
          %11 = load i32* %source, align 4
```

Figure 12 Test Example 3 Output (With Registers).

Test Case 3 Output II:



```
Without Registers:
%0:      { source}
%2:      { b, c, source}
%6:      { b, c, source}
%17:     { b, c, sink, source}
%10:     { b, c, source}
%12:     { b, c, source}
%14:     { b, c, source}
jiadong-2:Assignment wangjiadong$
```

Figure 13 Test Example 3 Output (Without Registers).

Some other test cases are also attached inside the zip file.