TAINT ANALYSIS

CS3213 FSE

Prof. Abhik Roychoudhury

National University of Singapore

WHAT WE DID EARLIER

  - System Requirements: Use-cases, Scenarios, Sequence Diagrams
  - System structure: Class diagrams
  - Discussion on semantics
  - System behavior: State diagrams
  - Discussion of the thinking behind your course project
  - Static analysis and vulnerability detection: Secure SE
  - Software Debugging
  - White-box Testing: estimation of a given test-suite
- Today
  - **Debugging and Fault Localization**
  - **Taint Analysis: effect of malicious inputs**

TOPICS

- Taint Analysis
  - Propagation of tainted inputs through the program
    - Through data flows – passing tainted value from one variable to another
    - Through implicit flows – a decision being made by a tainted value.
  - Can study data dependencies for this purpose
- Why do we need taint analysis
  - To understand the impact of malicious inputs.
  - To "harden" programs against malicious inputs.
  - …

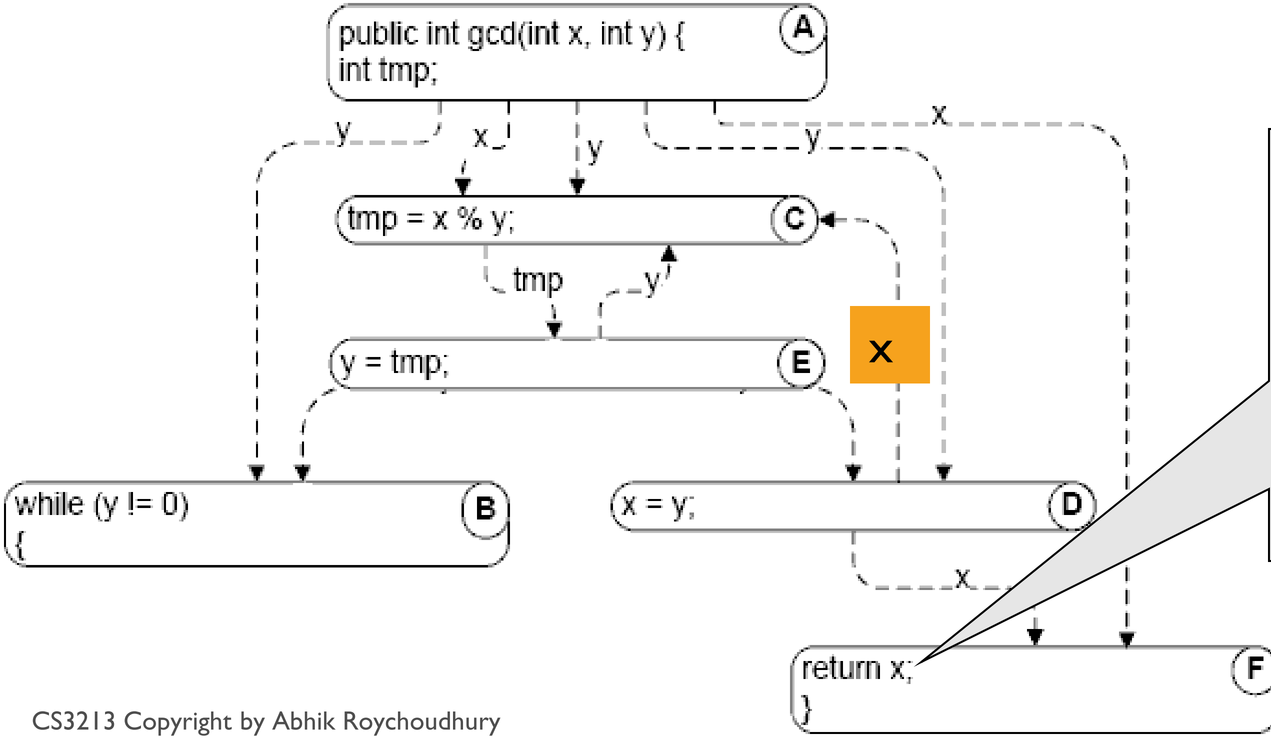
DATA DEPENDENCY

- Data dependence: t is dependent on s if
  - t uses a variable v which is defined in s
  - There is a definition-clear path w.r.t variable v (a path in which v is not set) between s and t
- Difference between static and dynamic data dependence is implicit here.
  - **Exercise in class:** show variants of the above definition for static and dynamic dependencies with suitable code examples

DATA DEPENDENCE GRAPH

```
/** Euclid's algorithm */
public class GCD
{
public int gcd(int x, int y) {
   int tmp;            // A: def x, y, tmp
   while (y != 0) {     // B: use y
     tmp = x % y;     // C: def tmp; use x, y
     x = y;            // D: def x; use y
     y = tmp;          // E: def y; use tmp
   }
   return x;           // F: use x
  }
```

- A data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name

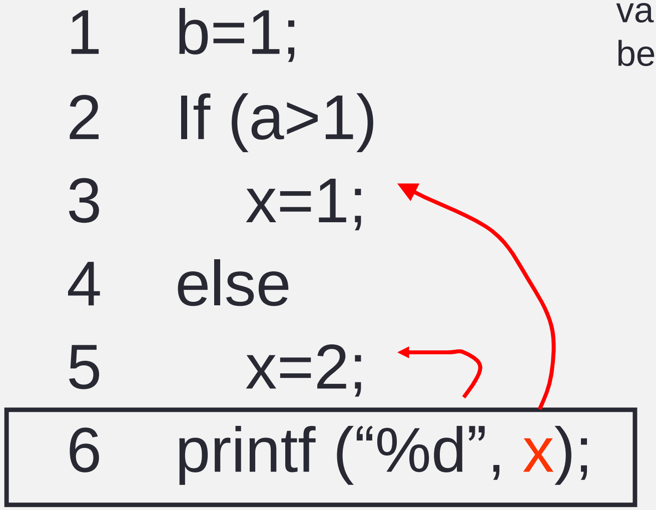


Dependence edges show this x value could be the unchanged parameter or could be set at line D

STATIC & DYNAMIC DATA DEPENDENCY

Data dependence:
t is dependent on s if t uses a variable v which is defined in s

There is a definition-clear path w.r.t variable v (a path in which v is not set) between s and t



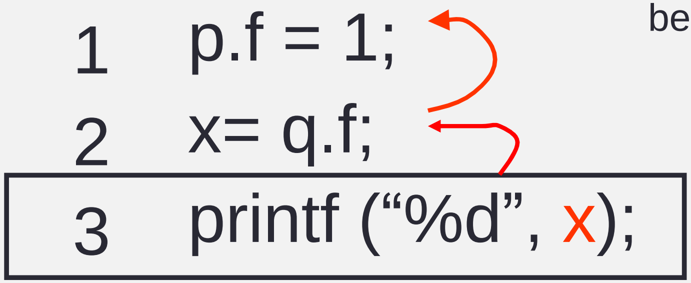
```
1    b=1;
2    If (a>1)
3        x=1;
4    else
5        x=2;
6    printf ("%d", x);
```

Slicing Criterion

STATIC & DYNAMIC DATA DEPENDENCY

Data dependence:
t is dependent on s if t uses a variable v which is defined in s

> There is a definition-clear path w.r.t variable v (a path in which v is not set) between s and t



```
p.f = 1;
x= q.f;
printf (“%d”, x);
```

p and q point to the same object?

Slicing Criterion

- Static points-to analysis is always conservative

TAINT POLICY

- Taint Introduction
  - All variables are, by default, untainted.
  - All inputs are tainted?
- Taint propagation
  - Specified as rules.
  - Taint is simply a bit.
- Taint Checking
  - When do you check?
  - For example, while going to an address, need to check whether the address is tainted.

TAINTED JUMP POLICY

- Protect from control flow hijacking

- Inputs are tainted.
- Propagate in a straightforward fashion
  - In a binary operation, taint the result if any operand is tainted
  - In assignment, taint the LHS if RHS is tainted.
  - What to do in the case of a branch?
    - Does not matter whether it is conditional or unconditional branch
    - Check that the jump target is not tainted.

EXAMPLE

```
1   x = 2 * get_input();
2   y = 5 * x;
3   go to y
```

Line 1: Taint source, and propagation

Line 2: Taint propagation

Line 3: Taint sink and check

EXAMPLE IN ACTION

```
1   x = 2 * get_input();
2   y = 5 * x;
3   go to y
```

- Taint policy might be tainted jump policy.
  - Taint source is at get_input()
  - Taint propagation
    - RHS of line 1 is tainted.
    - LHS of line 1 is tainted, so x is tainted.
    - RHS of line 2 is tainted
    - LHS of line 2 is tainted, so y is tainted.
  - Taint check at line 3 --- control transfer to tainted address.

ADDRESS AND VALUE

```
x = 2 * get_input();
y = 5 * x;
go to y
```

- When we say "x" is tainted
  - Do we mean the address of x is tainted?
  - Or the value in x is tainted?
- Taint policies
  - Track the status of addresses and memory values separately.
  - The taint status of a pointer p, and the data object *p, are **independent**.

UNDER-TAINTING

- Example

```
1   x = get_input();
2   y = load(z + x);
3   go to y
```

- Value of x is clearly tainted.
- The address (z + x) is therefore tainted.
- Value of y is NOT tainted, so jump in line 3 is allowed.

Untainted but attacker determined jump address!

OVER-TAINTING

- Tainted address policy: A memory cell is tainted if either address or value is tainted.

```
x = get_input();
y = load(z + x);
go to y
```

  - y is then always tainted and the jump is not allowed.
- Imagine the actual code in ***tcpdump*** program
  - Read network packet.
  - x = first byte of packet.
  - z = base address of function_pointer_table
  - y = function_pointer_table[z+x]
  - Go to function pointed by y

TAINT MARKERS

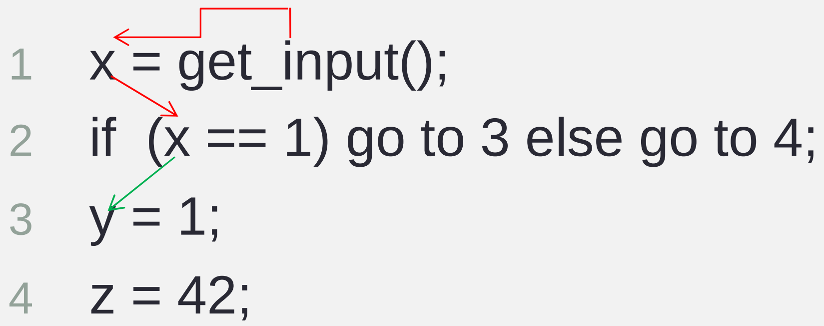
- Capturing tainted or non-tainted for each variable – one bit information.
- Instead can capture "taint markers" to explain the source of taint.
- Each variable gets associated with a set of taint markers, could be {}

```
input a, b;
w = 2 * a;
x = b + 1;
y = w + 1;
z = x + y;
output z;
```

Taint marker set for z = $\{t_a, t_b\}$

What is the taint marker set for y?

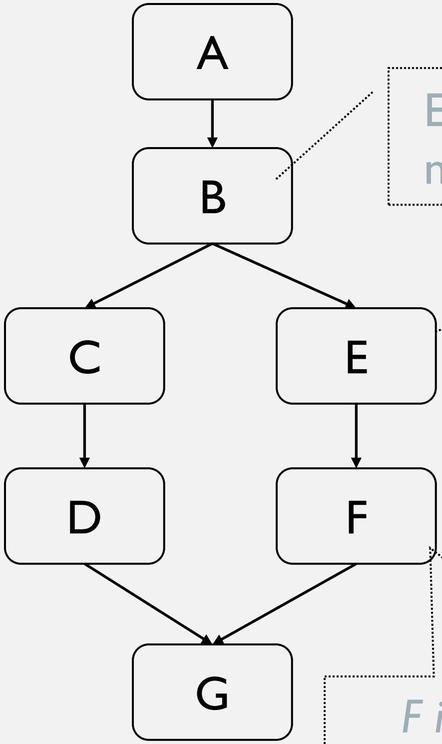
IMPLICIT FLOWS



```
1   x = get_input();
2   if  (x == 1) go to 3 else go to 4;
3   y = 1;
4   z = 42;
```

Line 4 is not affected by tainted input value. Whatever be the value, z is being set to 42.

CONTROL DEPENDENCE - EXAMPLE



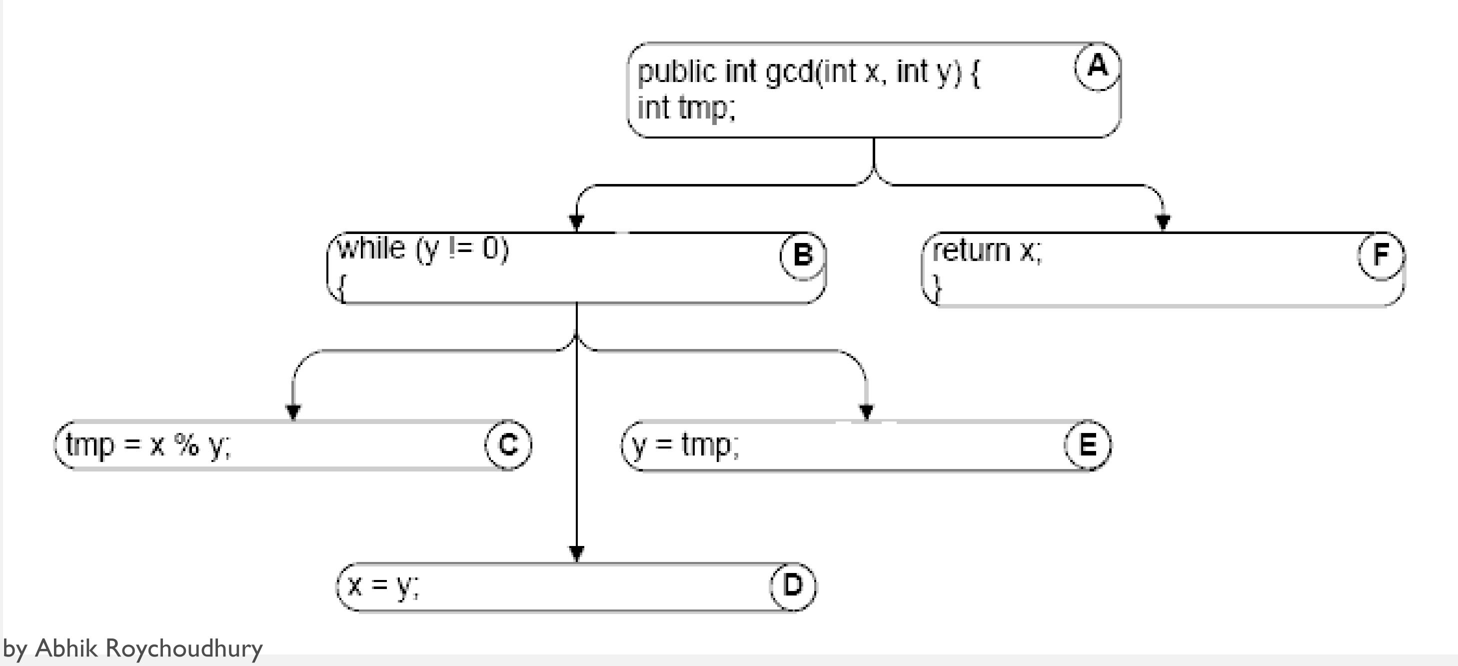
Execution of F is not inevitable at B

Execution of F is inevitable at E

F is control-dependent on B, the last point at which its execution was not inevitable

CONTROL DEPENDENCE

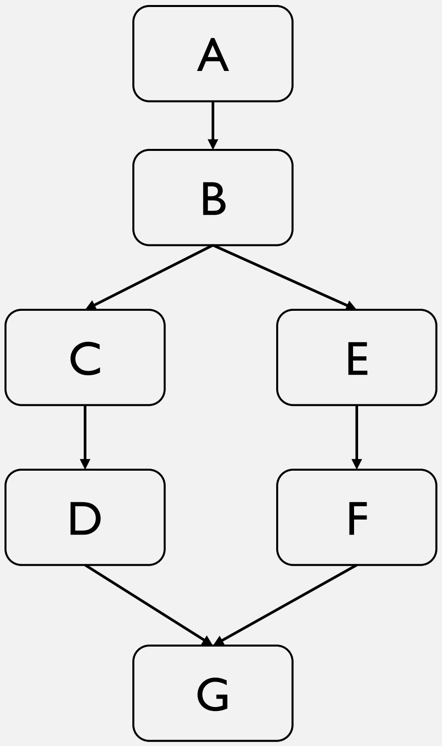
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



DOMINATORS

- **Pre-dominators** in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise.
- Node M dominates node N if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique **immediate dominator** of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.
- **Post-dominators**: Calculated in the reverse of the control flow graph, using a special "exit" node as the root.

EXAMPLE OF DOMINATOR

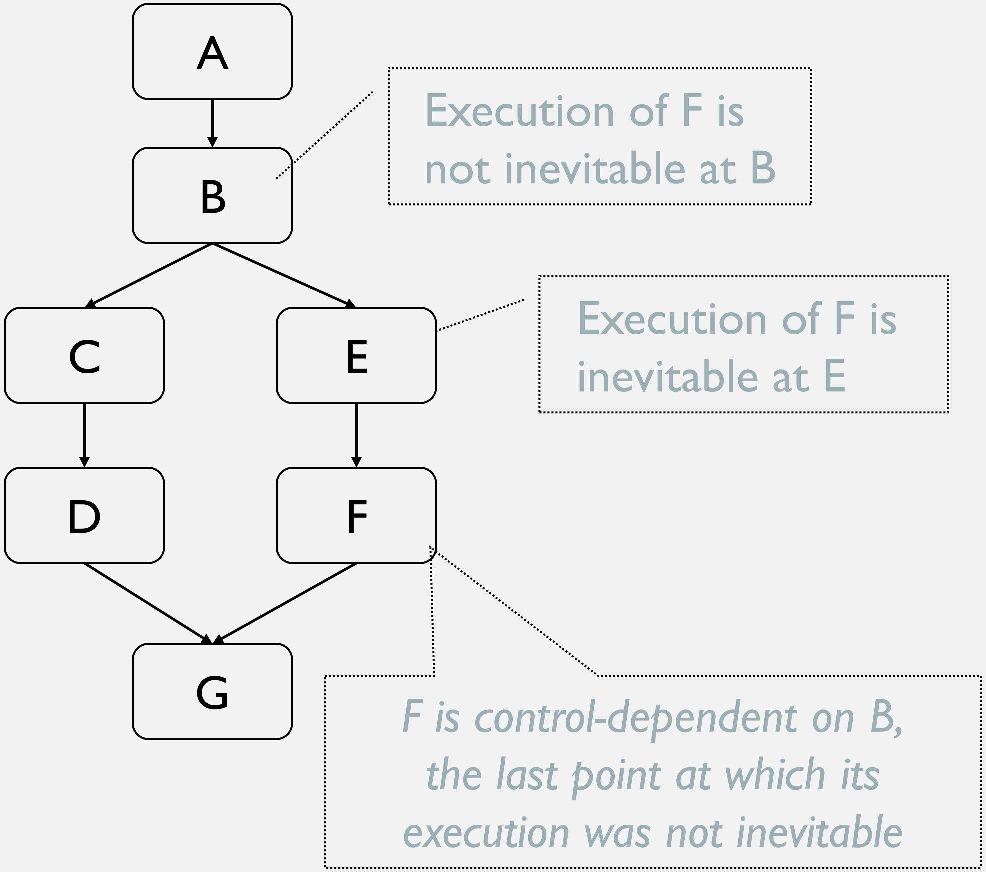


- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

CONTROL DEPENDENCE

- We can use post-dominators to give a more precise definition of control dependence:
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision);
    - C is not post-dominated by N
    - there is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.
    - Intuitively: C was the last decision that controlled whether N executed

CONTROL DEPENDENCE - EXAMPLE



Execution of F is not inevitable at B

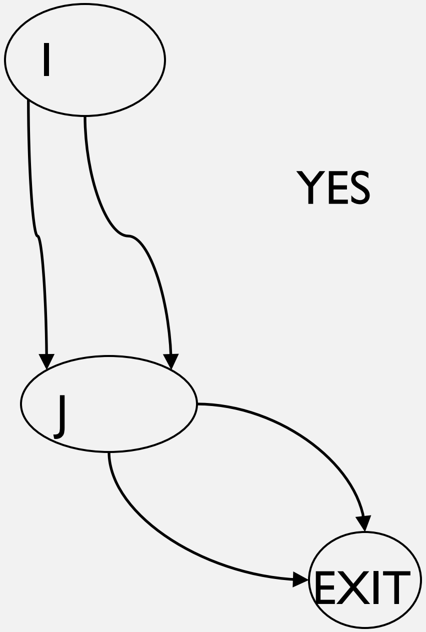
Execution of F is inevitable at E

F is control-dependent on B, the last point at which its execution was not inevitable

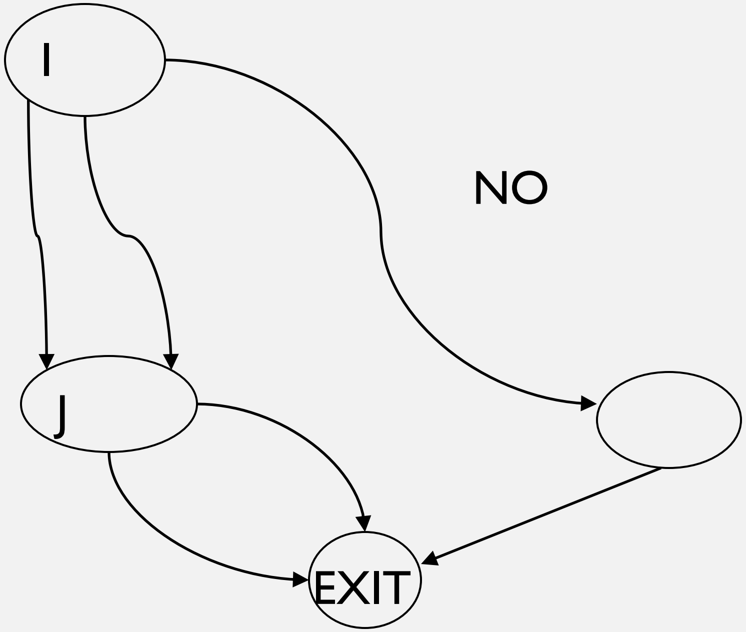
STATIC CONTROL DEPENDENCIES

Post-dominated: I,J – nodes in Control Flow Graph

I is post-dominated by J iff all paths from I to EXIT pass through J

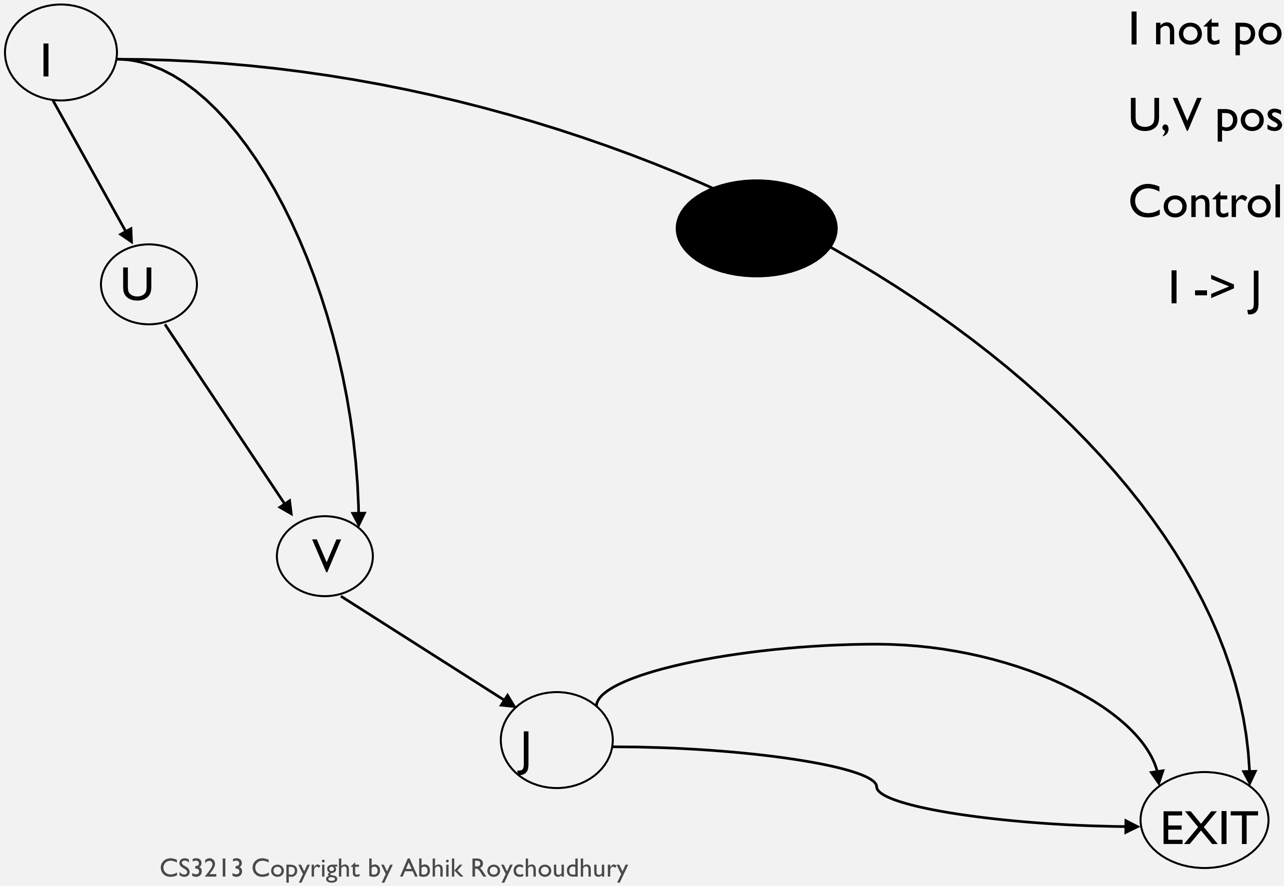


YES



NO

STATIC CONTROL DEPENDENCIES



I not post-dom by J

U,V post-dom by J

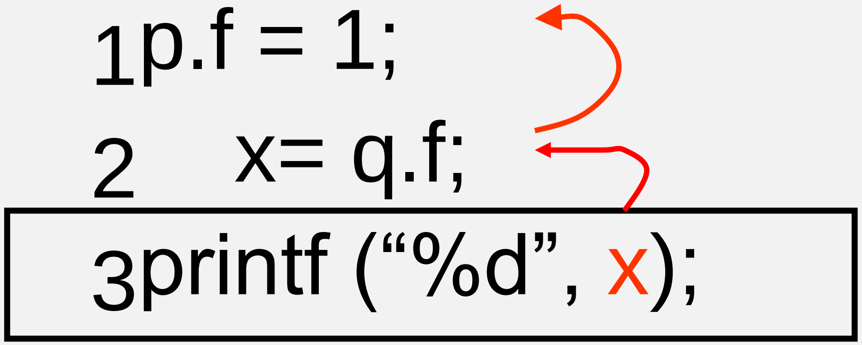
Control dependence

I -> J

DYNAMIC CONTROL DEPENDENCIES

- X is dynamically control dependent on Y if
  - Y occurs before X in the execution trace
  - X's stmt. is statically control dependent on Y's stmt.
  - No statement Z between Y and X is such that X's stmt. is statically control dependent on Z's stmt.
- Captures the intuition:
  - What is the nearest conditional branch statement that allows X to be executed, in the execution trace under consideration.

STATIC VS. DYNAMIC DATA DEPENDENCE



```
p.f = 1;
    x= q.f;
printf (“%d”, x);
```

p and q point to the same object?

Slicing Criterion

- Static points-to analysis is always conservative

STATIC VS. DYNAMIC CONTROL DEPENDENCE

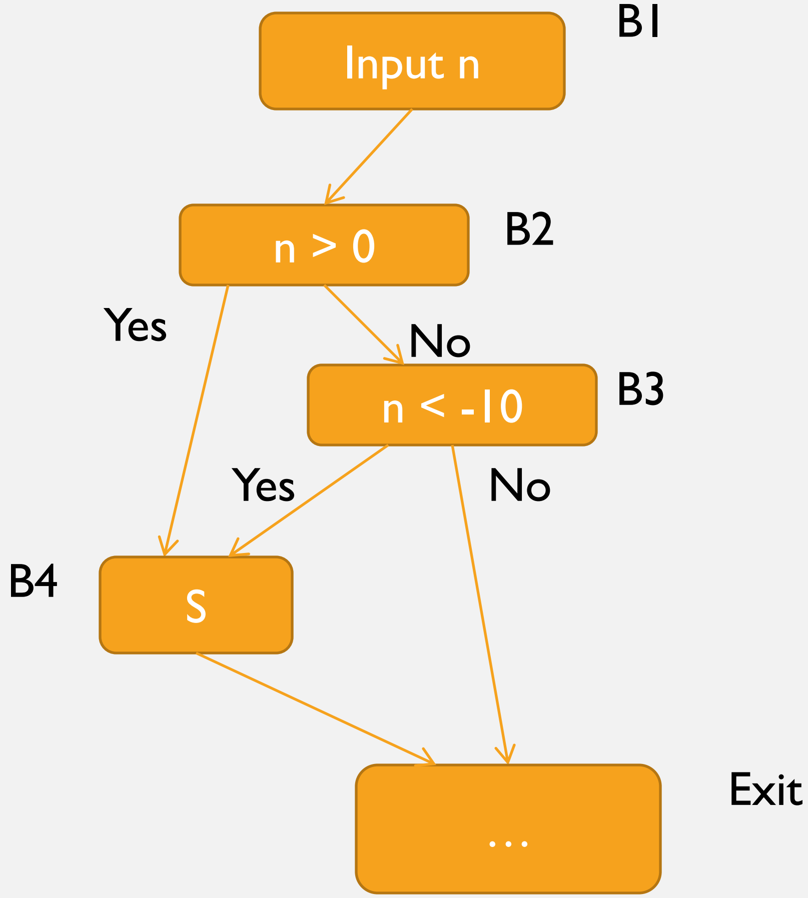
```
input n;
if (n > 0 || n < -10){
        S
}
…
```

Static control dependence

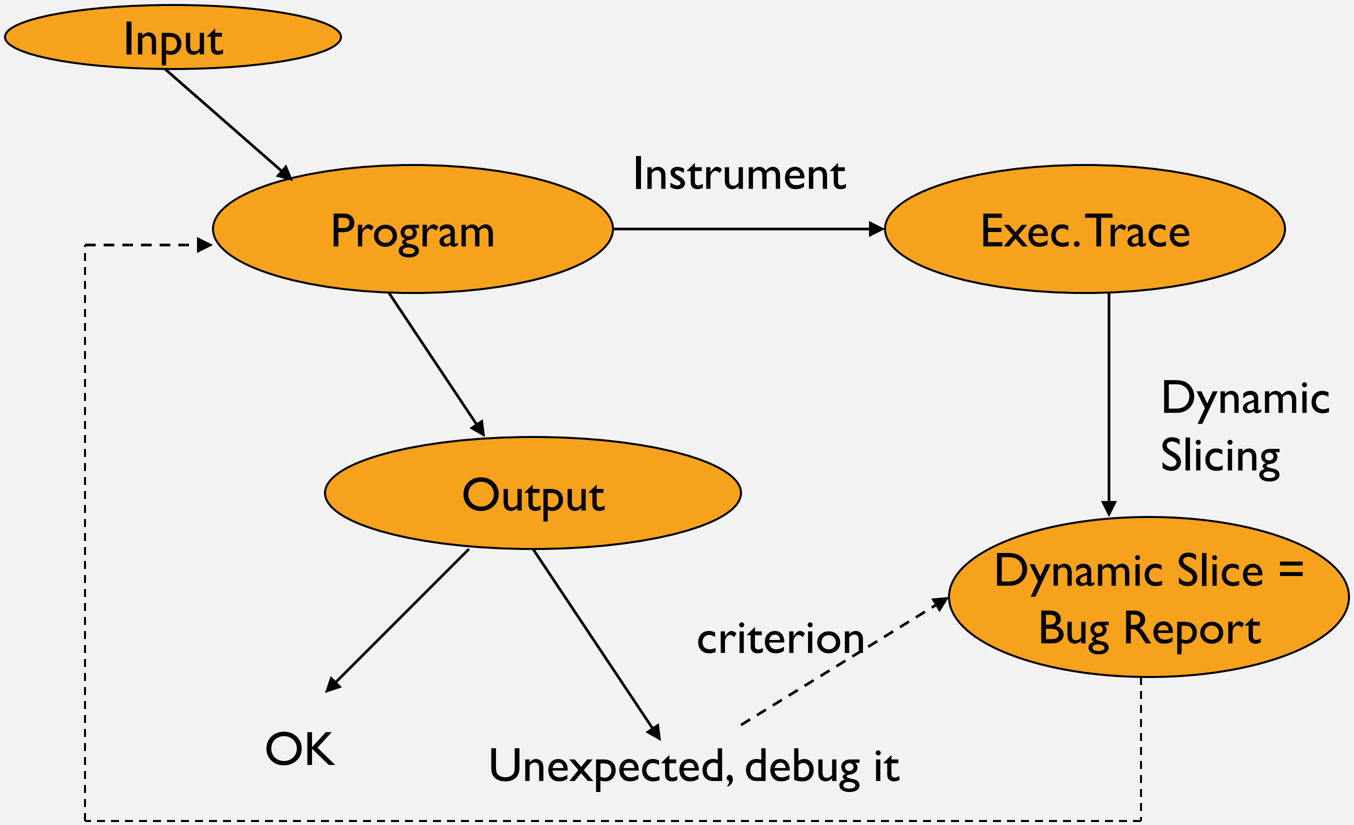
- B2 -> B4
- B3 -> B4

Dynamic control dependence

One of these, depending on value of n



DYNAMIC SLICING FOR DEBUGGING : RECAP



Input

Program

Instrument

Exec. Trace

Output

Dynamic Slicing

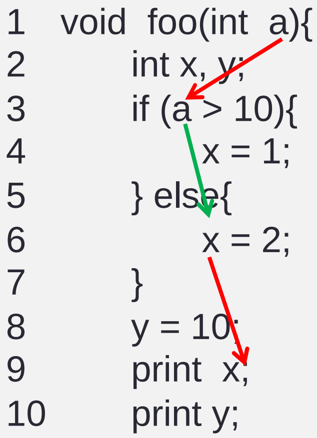
Dynamic Slice = Bug Report

criterion

OK

Unexpected, debug it

BACK TO AN EXAMPLE



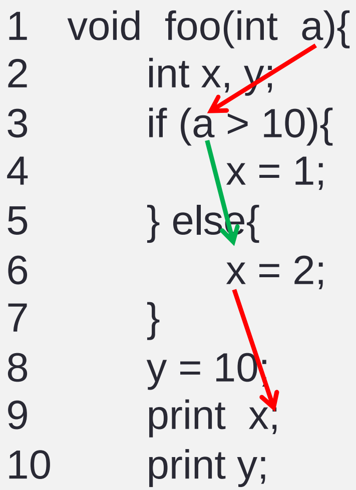
```
void  foo(int  a){
        int x, y;
        if (a > 10){
                x = 1;
        } else{
                x = 2;
        }
        y = 10;
        print  x;
        print y;
```

Input a == 1

a is the input value (tainted)
Value of a affects which assignment of x is executed.
The output for x is thus tainted with $\{t_a\}$

Dynamic tainting with implicit flows

EXAMPLE (HARD)



```
void  foo(int  a){
        int x, y;
        if (a > 10){
                x = 1;
        } else{
                x = 2;
        }
        y = 10;
        print  x;
        print y;
```

Input a == 1

<u>Source</u>:
Dytan: A Generic Dynamic Taint Analysis Framework, by Clause, Li and Orso, ISSTA 2007, see LumiNUS for web-link.

a is the input value (tainted)
Value of a affects which assignment of x is executed.
The output for x is thus tainted with {t_a}

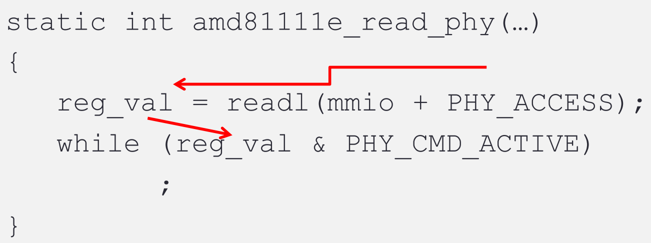
Dynamic tainting with implicit flows

REMOVING TAINT

- More and more variables get tainted as
  - Execution trace is analyzed – dynamic taint analysis
  - Program is analyzed – static taint analysis

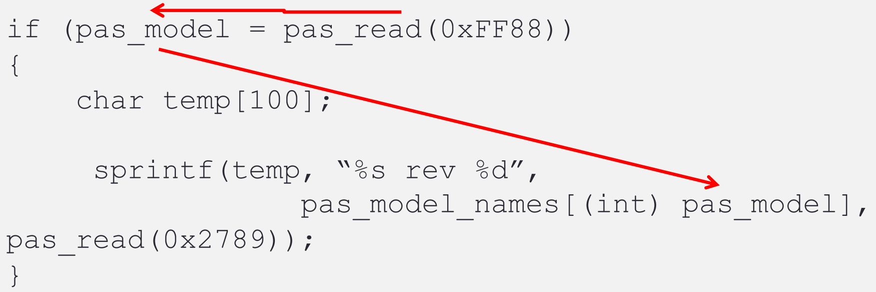
- Taint markers are simply added, never removed?
  - Consider b = a - a ;
  - If a is tainted, b should also be tainted?
  - But if a has no overflows etc, b is always zero
  - In general, operations which return constant results should not be tainted.

REAL EXAMPLES



```
static int amd81111e_read_phy(…)
{
   reg_val = readl(mmio + PHY_ACCESS);
   while (reg_val & PHY_CMD_ACTIVE)
         ;
}
```

AMD 8111e Network Driver



```
if (pas_model = pas_read(0xFF88))
{
    char temp[100];

     sprintf(temp, "%s rev %d",
                 pas_model_names[(int) pas_model],
pas_read(0x2789));
}
```

Pro Audio Sound Driver

READINGS

- All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)
  - Schwartz, Avgerinos, Brumley
  - Oakland 2010

- Supplementary reading
  - Dytan: A generic dynamic taint analysis framework
  - Clause, Li, Orso,
  - ISSTA 2007.