

SECURE SE – TAIN T ANALYSIS

CS3213 FSE

Prof. Abhik Roychoudhury

National University of Singapore



WHAT WE DID EARLIER

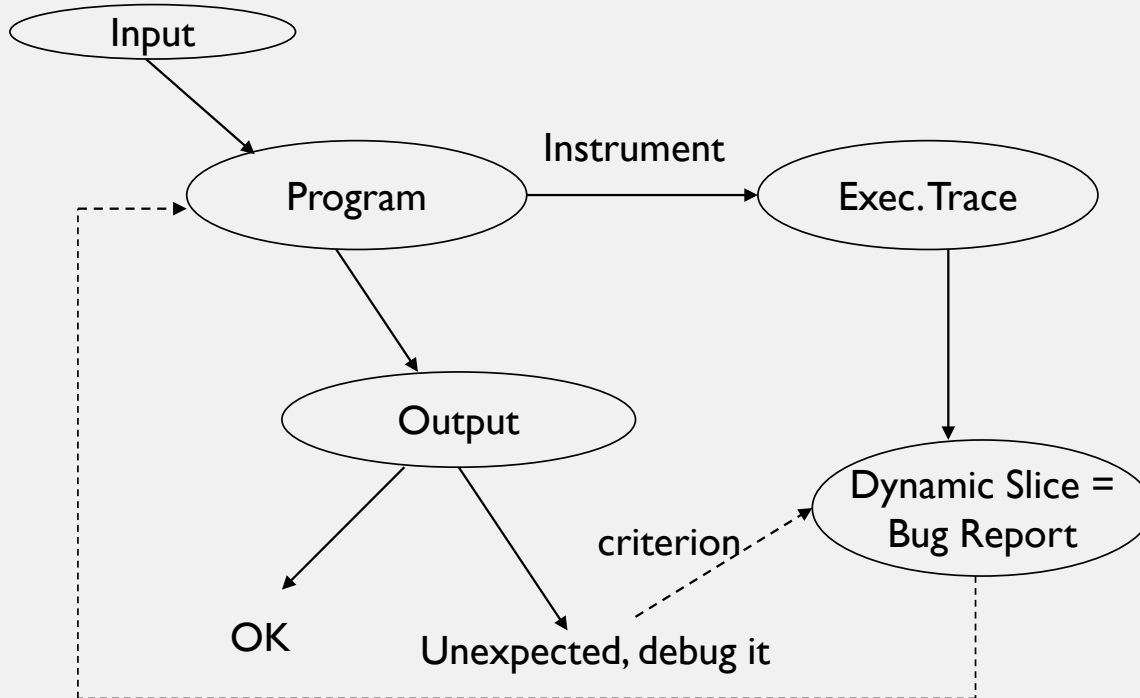
- System Requirements: Use-cases, Scenarios, Sequence Diagrams
 - System structure: Class diagrams
 - Discussion on semantics
 - System behavior: State diagrams
 - Discussion of the thinking behind your course project
 - Static analysis and vulnerability detection: Secure SE
 - Software Debugging
 -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

TAINT ANALYSIS (TODAY'S LECTURE)

proactively.

- Keeping track of **un-sanitized** input variables along program execution paths
 - Static taint analysis: flow analysis conducted on control flow graph, providing conservative results *conserve all possible results.*
 - Dynamic taint analysis: Taint status of variables is updated along a program execution, and execution is interrupted if a tainted variable is used in a security-critical statement
 - $X \rightarrow T$: variable X is tainted
 - $X \rightarrow F$: variable X is not tainted

DYNAMIC SLICING – A BASIC CONCEPT



DYNAMIC SLICING

Dynamic slicing:
backwards reasoning. start from
unexpected output / value.

Consider input $a == 2$

Taint:
forward reasoning. start from
input that attacker can control.

Control
Dependence

```
1 b=2;  
2 x=1;  
3 if (a>1){  
4     if (b>1){  
5         x=2;  
        }  
    }  
}
```

Data
Dependence

```
6 printf ("%d", x);
```

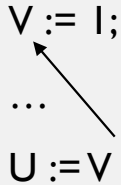
Slicing
Criterion

DYNAMIC SLICE

- Set slicing criterion
 - (Variable v at first instance of line 70)
 - The value of variable v at first instance of line 70 is unexpected.
- Dynamic slice
 - Closure of
 - Data dependencies &
 - Control dependencies
 - from the slicing criterion.

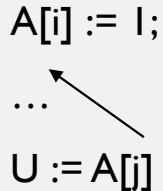
DYNAMIC DATA DEPENDENCIES

V := I;
...
U := V



An edge from a variable usage to the latest definition of the variable.

A[i] := I;
...
U := A[j]



→ Do we consider this data dependence edge ?

→ Remember that the slicing is for an input, so the addresses are resolved

→ We thus define data dependencies corresponding to memory locations rather than variable names.

*data dependency → memory location.
X variable names.*

INFORMAL DEFINITIONS

- Data dependence: t is dependent on s if *(t, s are stmts).*
 - 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
- There is a definition-clear path (where?)
 - In *some* execution (static)
 - In *this* execution (dynamic)


STATIC VS DYNAMIC DATA DEPENDENCE

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 VS DYNAMIC DATA DEPENDENCE

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 p.f = 1;  
2   x = q.f;  
3 printf ("%d", x);
```



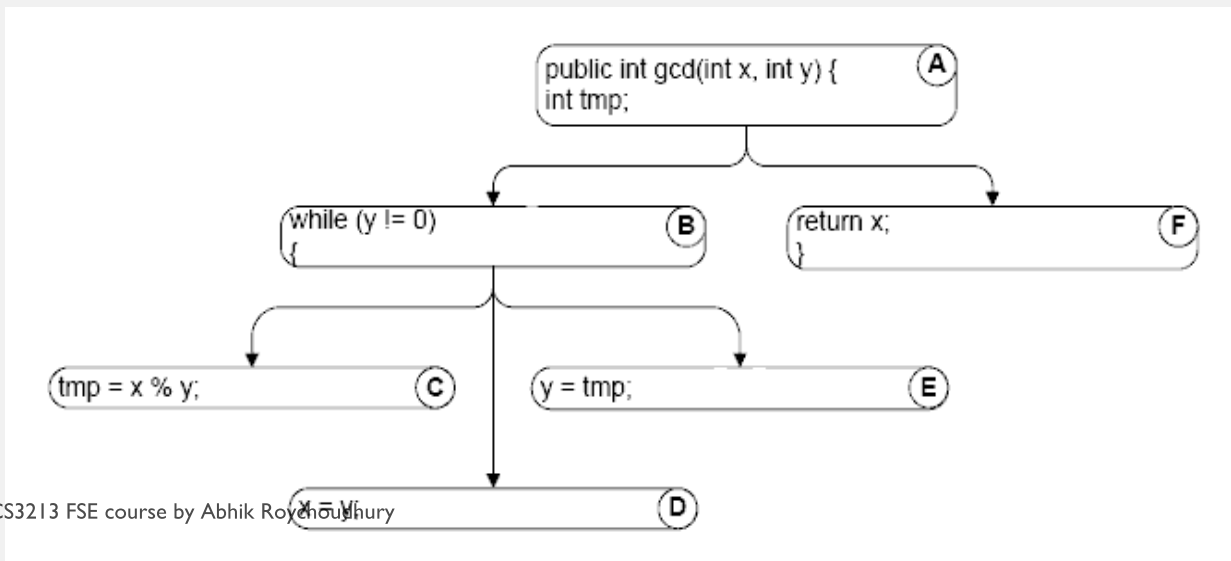
p and q point to the same object?

Slicing Criterion

- Static points-to analysis is always conservative

CONTROL DEPENDENCE

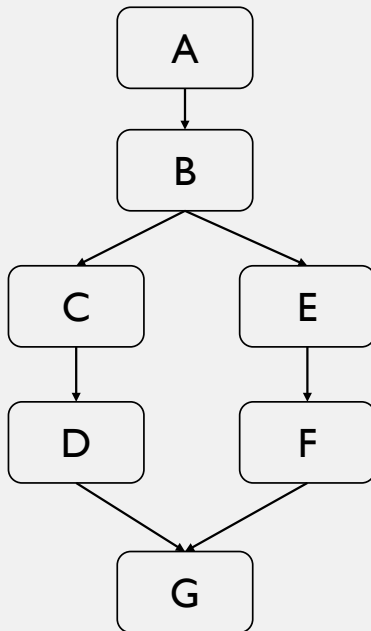
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
 - Nodes: as in the CFG
 - Edges: unlabelled, from entry/branching points to controlled blocks



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



pre-dominators
里离得最近的

- 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

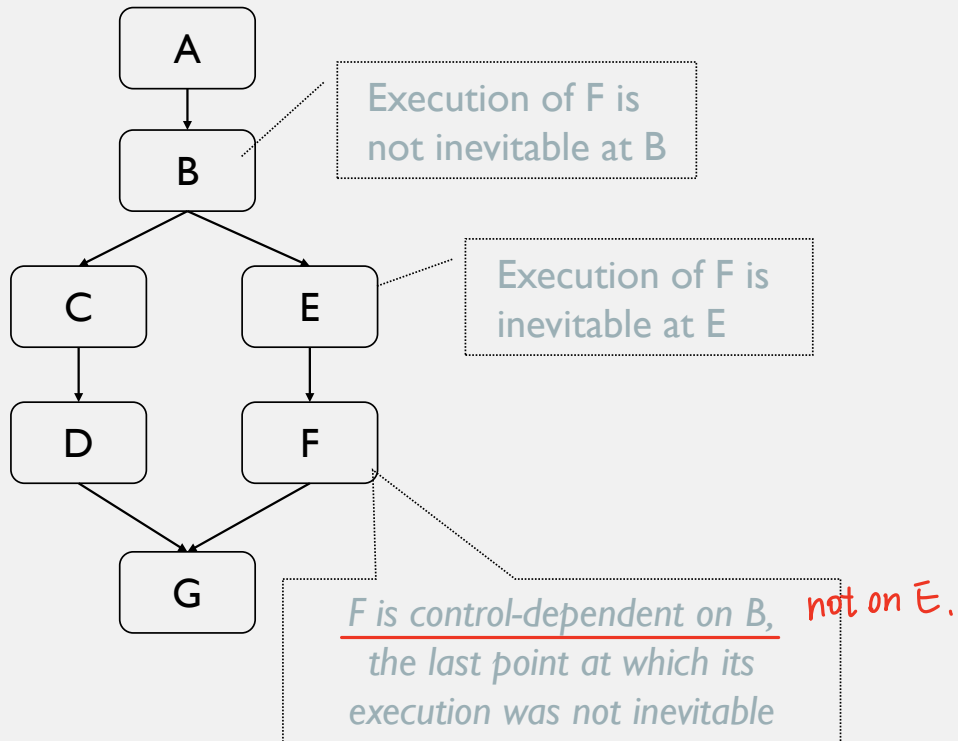
CONTROL DEPENDENCE

- 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



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.

INFORMAL DEFINITION

- A statement t is control dependent on s if
 - s contains a predicate
 - The outcome of s decides whether t will be executed.
- Difference between static and dynamic control dependence is implicit here.
 - **Exercise in class:** show variants of the above definition for static and dynamic dependencies with suitable code examples
- The outcome of s decides whether t will be executed (where?)
 - Decides in *some* execution (static)
 - Decides in *this* execution (dynamic)

STATIC VS. DYNAMIC CONTROL DEPENDENCE

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

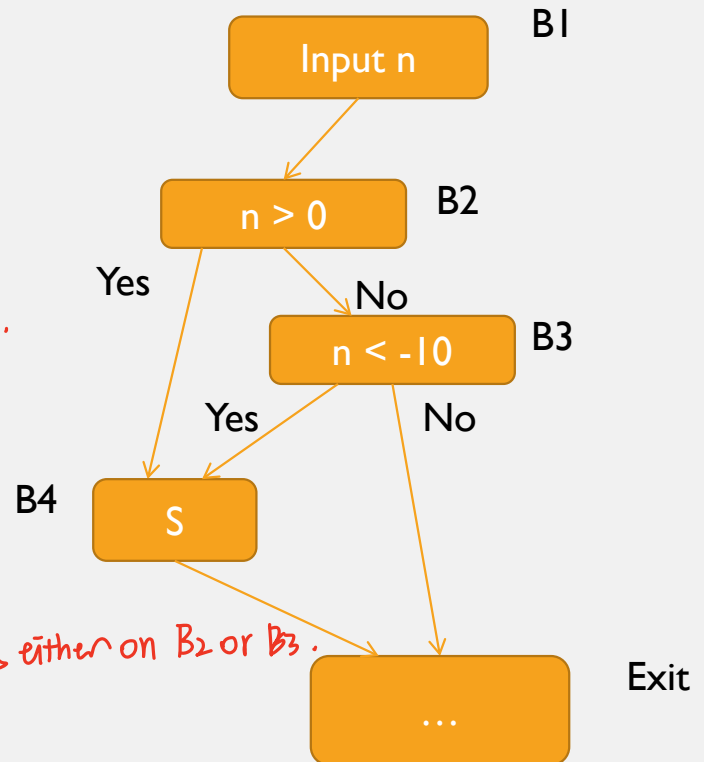
...

∴ in some execution.
Static control dependence *B2, B3 都有*

- B2 → B4
- B3 → B4

Dynamic control dependence

B4 decides either on B2 or B3.
One of these, depending on value of n



STATIC VS DYNAMIC SLICING

- Static Slicing
 - source code
 - statement
 - static dependence
- Dynamic Slicing
 - a particular execution
 - statement instance
 - dynamic dependence

DYNAMIC SLICE - ILLUSTRATION

```
0  scanf("%d", &A);  
1.  if (A == 0){  
2      W = X;  
3      U = A;  
4  }  
5  printf("%d\n", U);
```

data dependency is always between the same variable. there is a data dependency between the definition of the variable and the usage of the variable. 5 -> 3, 1 -> 0. There is no data dependency between 3 and 0.

Criterion: 5,U with input A == 0

Trace: <0,1,2,3,4,5> Slice = {0,1,3,5}

Data dependences encountered 5 -> 3, 1 -> 0 → def-use pair of variables.

Control dependence encountered 3 -> 1

TOPICS

Taint tracking

-> Goes beyond program dependencies.

-> We can define taint policies – and the propagation simply enforces these policies to find tainted statements / variables

So, what constitutes a taint policy?

TAINT POLICY

- Taint Introduction
 - All variables are, by default, untainted.
 - All inputs are tainted? *Yes.*
- Taint propagation
 - Specified as rules.
 - Taint is simply a bit. *bit: tainted or not.*
- Taint Checking
 - When do you check?
 - For example, while going to an address, need to check whether the address is tainted.

EXAMPLE: 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. $\longrightarrow a = b + c$, a is tainted if b or c 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();      → Taint source
2  y = 5 * x;
3  go to y    jump target is tainted.
```

Line 1: Taint source, and propagation

Line 2: Taint propagation

Line 3: Taint sink and check

since branch is potentially under control of attacker.
not that branch is not allowed.
but there will be a check before it.

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

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

- Example
- 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

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

- Example

- Value of x is clearly tainted.

Address is tainted

- The address (z + x) is therefore tainted. *value (z+x) is untainted.*

but value at that add. is not.

- Value of y is NOT tainted, so jump in line 3 is allowed.

- Attacker may know/guess z, and adjust the value of x, to hijack the control flow to any location he/she wants.

OVER-TAINTING

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

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

- 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]

this function_pointer_table is explicitly confirmed, defined. no matter what value of x is (x is tainted, it is depended by attacker), as long as z + x is a valid index of the table, y is not tainted.

the jump is allowed • 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;
```

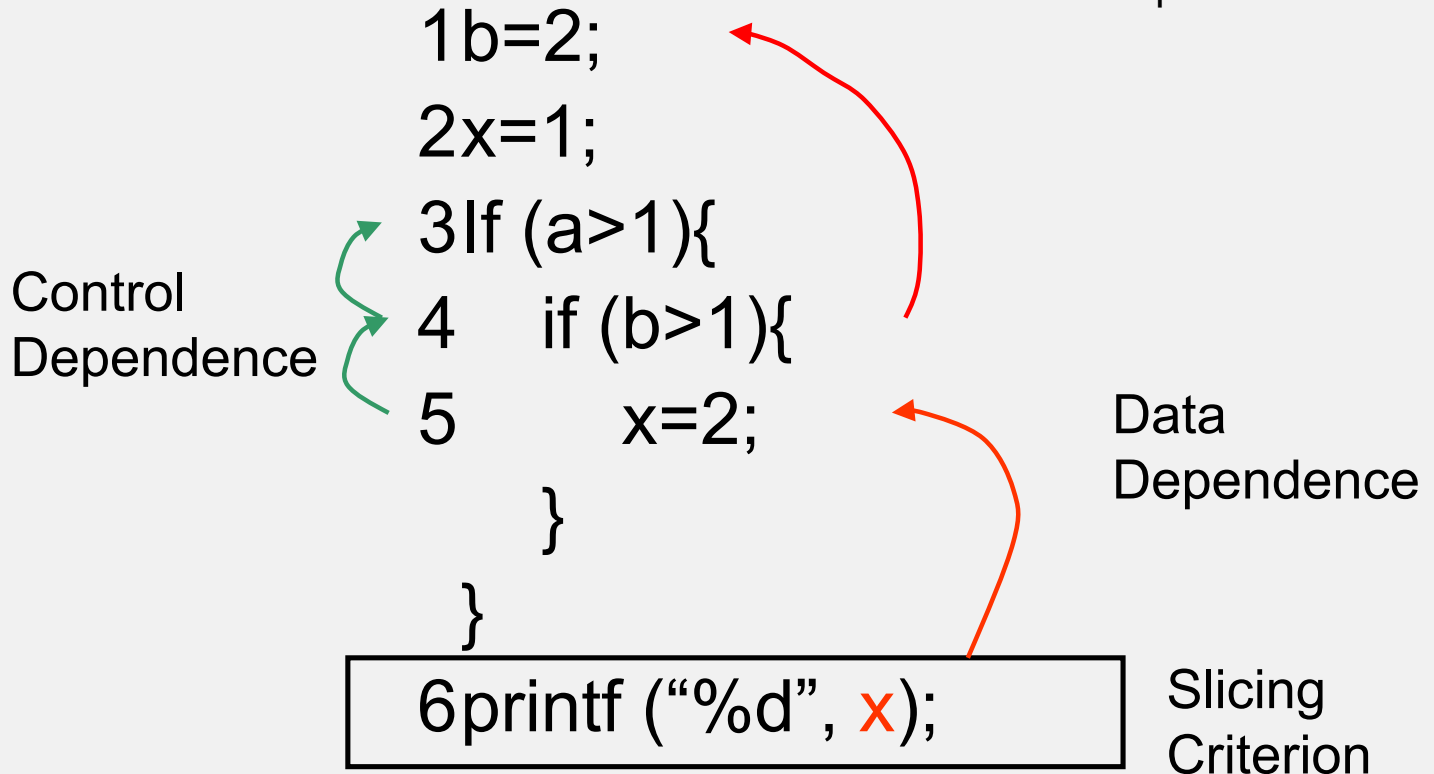
→ $y = \{t_a\}$.

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

What is the taint marker set for y ?

RECAP ON DYNAMIC SLICING (BACKWARD)

Consider input $a == 2$



DYNAMIC SLICING (FORWARD)

```
1  input a;  
2  x = a;  
3  if (x > 1){  
4      x=2;  
5  }  
6  printf ("%d", x);
```

Taint propagation

Control dependence

The diagram illustrates forward dynamic slicing. Red arrows indicate taint propagation: one from the variable 'a' in line 1 to 'x' in line 2, and another from 'x' in line 2 to 'x' in line 6. A green arrow indicates control dependence: from the 'if' statement in line 3 to the assignment 'x=2;' in line 4.

COMPARISON

from def to use.

- Dynamic taint analysis = (forward dyn. data dependence)*
- Forward dynamic slice
= (forward dyn. data dep. \cup forward dyn. control dep.)*

Classical taint analysis computes the closure of the forward dynamic data dependencies.

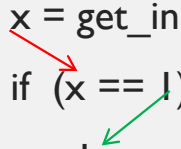
Extensions of taint analysis (also called taint analysis these days) computes the closure of forward dynamic data and control dependencies – bringing it closer to dynamic slice.

COMPARISON

- Is extended taint analysis
- = Forward dynamic slice computation?
- NO
 - Dynamic slice = set of statements
 - Taint analysis returns set of tainted variables
- **Exercise in class:**
 - Work out examples in class to show the difference.

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.

EXAMPLE

```
1 void foo(int a){
2     int x, y;
3     if (a > 10){
4         x = 1;
5     } else{
6         x = 2;
7     }
8     y = 10;
9     print x;
10    print y;
```

Input a == 1

implicit flows forward control dependency 3 → 6.

a is the input value (tainted)

Value of a affects which assignment of x is executed.

The output for x is thus tainted with $\{t_a\}$

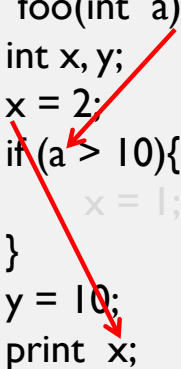
Dynamic tainting with implicit flows

WHY IMPLICIT FLOWS ARE HARD

- Involves computing dynamic control dependencies
 - This involves static control dependency computation
 - This involves static control flow graph construction and static analysis (hard)
 - Control flow graph construction is hard due to register indirect jumps.

EXAMPLE (HARD)

```
1 void foo(int a){  
2     int x, y;  
3     x = 2;  
4     if (a > 10){  
5         x = 1;  
6     }  
7     y = 10;  
8     print x;  
9     print y;
```



Consider the execution for $a == 1$

Source:

Dytan: A Generic Dynamic Taint Analysis Framework, by Clause, Li and Orso, ISSTA 2007, see IVLE for web-link.

Taint marker set for $a = \{t_a\}$

Taint marker set for $x = \{t_a\}$

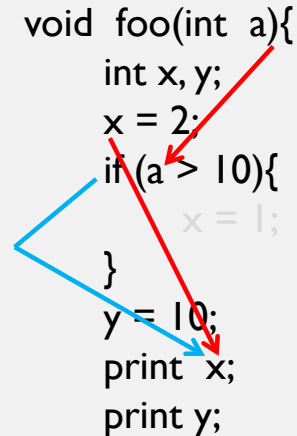
Taint marker set is a set containing all variables that might affect the value of x ; here the value of x depends on value of a (depends on the attacker).

[Truly implicit taint propagation – no execution of assignment to x]

Taint marker set for output value at line 8 is $\{t_a\}$

EXAMPLE (HARD)

```
1 void foo(int a){  
2     int x, y;  
3     x = 2;  
4     if (a > 10){  
5         x = 1;  
6     }  
7     y = 10;  
8     print x;  
9     print y;
```



Consider the execution for $a == 1$

Red Arrows == Dynamic data dependence

No control dependence between line 4 and line 8

- Line 4 contains a predicate
- Irrespective of the outcome of line 4, line 8 is executed.

Taint marker set for $a = \{t_a\}$

Taint marker set for $x = \{t_a\}$

[Truly implicit taint propagation – no execution of assignment to x]

Taint marker set for output value at line 8 is $\{t_a\}$

EXAMPLE (HARD)

```
1 void foo(int a){
2     int x, y;
3     x = 2;
4     if (a == 10){
5         x = 1;
6     }
7     y = 10;
8     print x;
9     print y;
```

Consider the execution for $a == 1$

No control dependence between line 4 and line 8

- Line 4 contains a predicate
- Irrespective of the outcome of line 4, line 8 is executed.

Static control dependence between line 4 and line 5

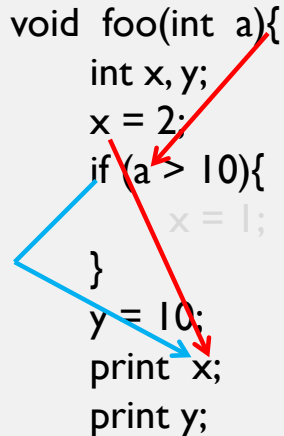
Line 4 contains a predicate

Outcome of line 4 decides whether line 5 is executed.

No dynamic control dependence between line 4 and line 5.

EXAMPLE (HARD)

```
1 void foo(int a){  
2     int x, y;  
3     x = 2;  
4     if (a > 10){  
5         x = 1;  
6     }  
7     y = 10;  
8     print x;  
9     print y;
```



Consider the execution for $a == 1$

No control dependence between line 4 and line 8

- Line 4 contains a predicate
- Irrespective of the outcome of line 4, line 8 is executed.

Blue Arrow == Potential dependence between line 4 and line 8

Line 4 contains a predicate.

The outcome of line 4, decides the execution of an assignment on some variable v , and this variable v is used in line 8.

EXAMPLE (HARD)

```
1 void foo(int a){
2     int x, y;
3     x = 2;
4     if (a > 10){
5         x = 1;
6     }
7     y = 10;
8     print x;
9     print y;
```

Tainted value of `a` results in non-execution of line 5.
This affects the value printed in line 8.

If line 5 was executed

- Control dep. between line 4, 5
- Data dep. between line 5, 8

When line 5 is not executed

- Potential dep. between line 4, 8
(involves computing static data dependency)

Blue Arrow == Potential dependence between line 4 and line 8

Line 4 contains a predicate.

The outcome of line 4, decides the execution of an assignment on some variable `v`, and this variable `v` is used in line 8.

EXAMPLE (STATIC TAINTING)

Dynamic tainting

Input a == 1

```
1 void foo(int a){
2   int x, y;
3   if (a > 10){
4     x = 1;
5   } else{
6     x = 2;
7   }
8   y = 10;
9   print x;
10  print y;
```

Static tainting

```
1 void foo(int a){
2   int x, y;
3   if (a > 10){
4     x = 1;
5   } else{
6     x = 2;
7   }
8   y = 10;
9   print x;
10  print y;
```

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.

APPLICATIONS

Sample applications of taint analysis

- Static Taint Analysis
 - Applied in Device Driver hardening – changing the code to reduce chance of attacks when tainted variables are being manipulated
- Dynamic Taint Analysis
 - Applied in Targeted Grey-box Fuzzing – generate tests which fuzz tainted bytes in input files.
 - TaintScope paper – discussed last week, we will recapitulate today – time permitting.

APPLICATION OF (STATIC) TAINT ANALYSIS

Application in Hardening

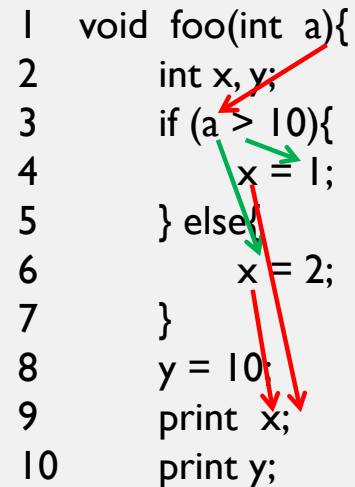
- Tolerating Hardware Device Failures in Software, Asim Kadav, Matthew J. Renzelmann, Michael M. Swift, SOSP 2009.

- **Static taint analysis**

- Static taint analysis: flow analysis conducted on control flow graph, providing conservative results

Static tainting

```
1 void foo(int a){  
2     int x, y;  
3     if (a > 10){  
4         x = 1;  
5     } else{  
6         x = 2;  
7     }  
8     y = 10;  
9     print x;  
10    print y;
```



APPLICATION OF (DYNAMIC) TAINT ANALYSIS

Application in Grey-Box Fuzzing

- Taint-based Directed WhiteBox Fuzzing
- Vijay Ganesh, Tim Leek, Martin Rinard
- International Conference on Software Engineering (ICSE 2009)

Dynamic taint analysis:

Taint status of variables is updated along a program execution, and execution is interrupted if a tainted variable is used in a security-critical statement

- $X \rightarrow T$: variable X is tainted
- $X \rightarrow F$: variable X is not tainted

Dynamic tainting

```
1 void foo(int a){  
2     int x, y;  
3     if (a > 10){  
4         x = 1;  
5     } else{  
6         x = 2;  
7     }  
8     y = 10;  
9     print x;  
10    print y;
```

Input a == 1