Dependence and Data Flow Models

Why Data Flow Models?

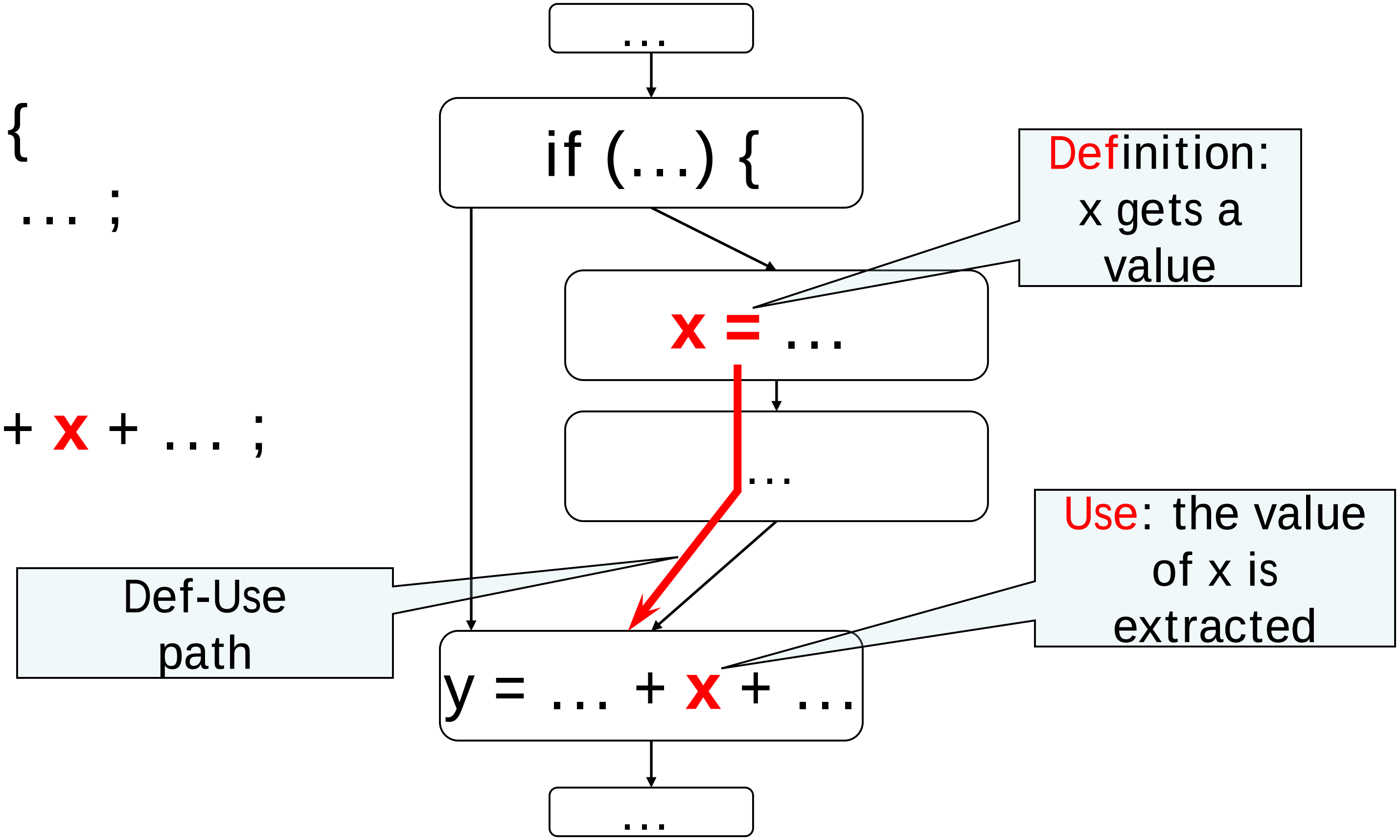
- Models discussed earlier emphasized control
  - Control flow graph, call graph, finite state machines
- We also need to reason about dependence
  - Where does this value of x come from?
  - What would be affected by changing this?
  - ...
- Many program analyses and test design techniques use data flow information
  - Often in combination with control flow
    - Example: "Taint" analysis to prevent SQL injection attacks
    - Example: Dataflow test criteria

Def-Use Pairs (1)

- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter
- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

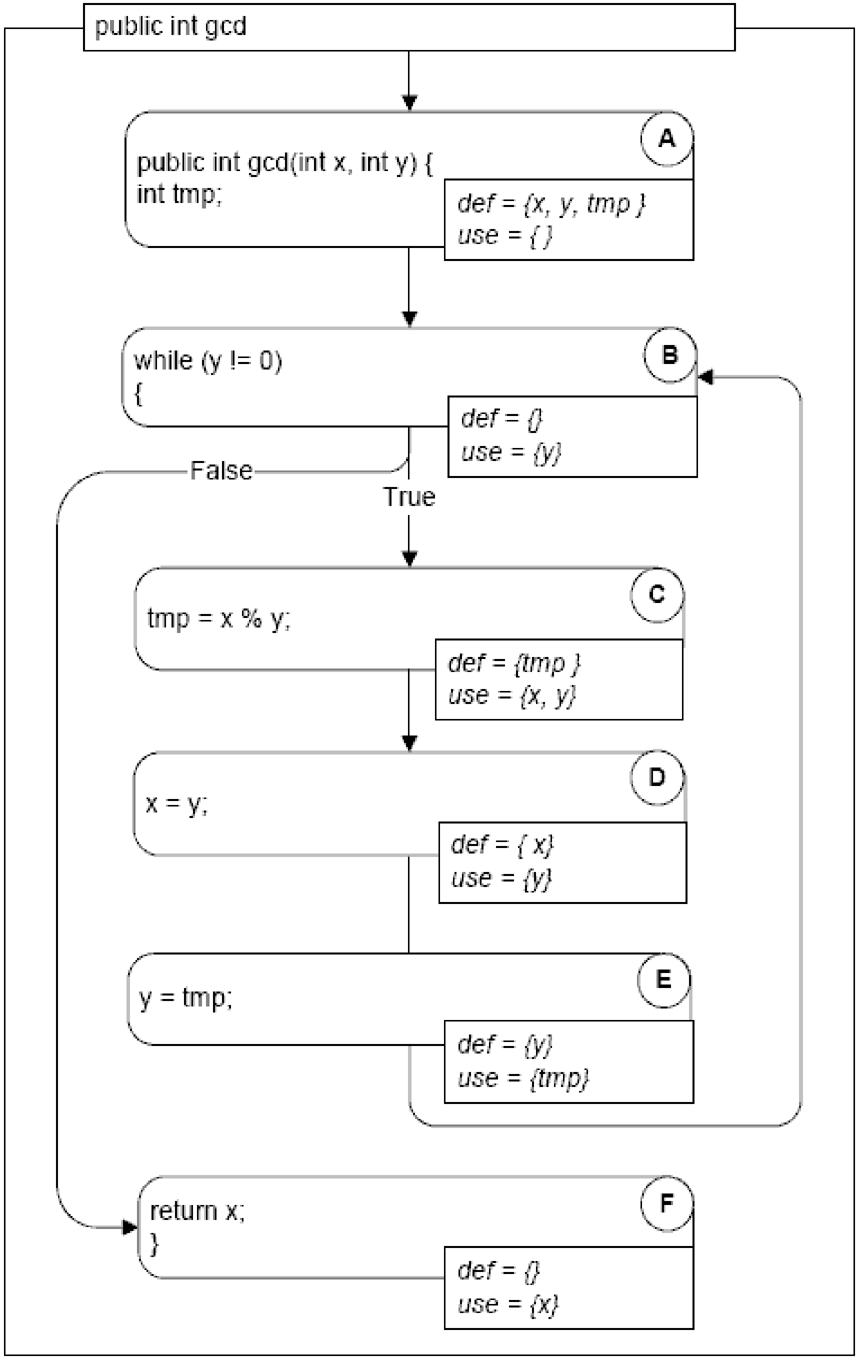
Def-Use Pairs

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



Def-Use Pairs (3)

```
/**  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
  }
```

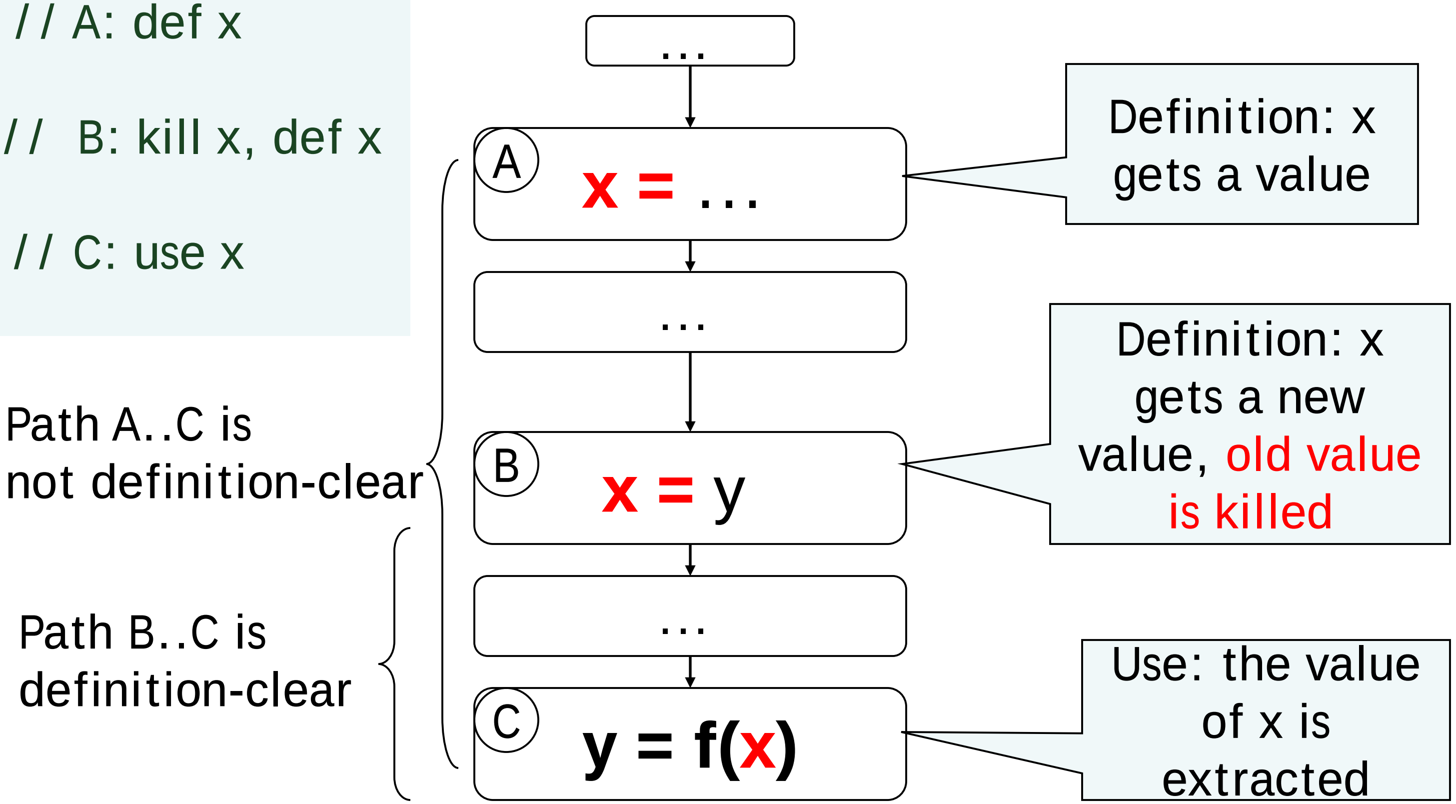


Def-Use Pairs (3)

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

Definition-Clear or Killing

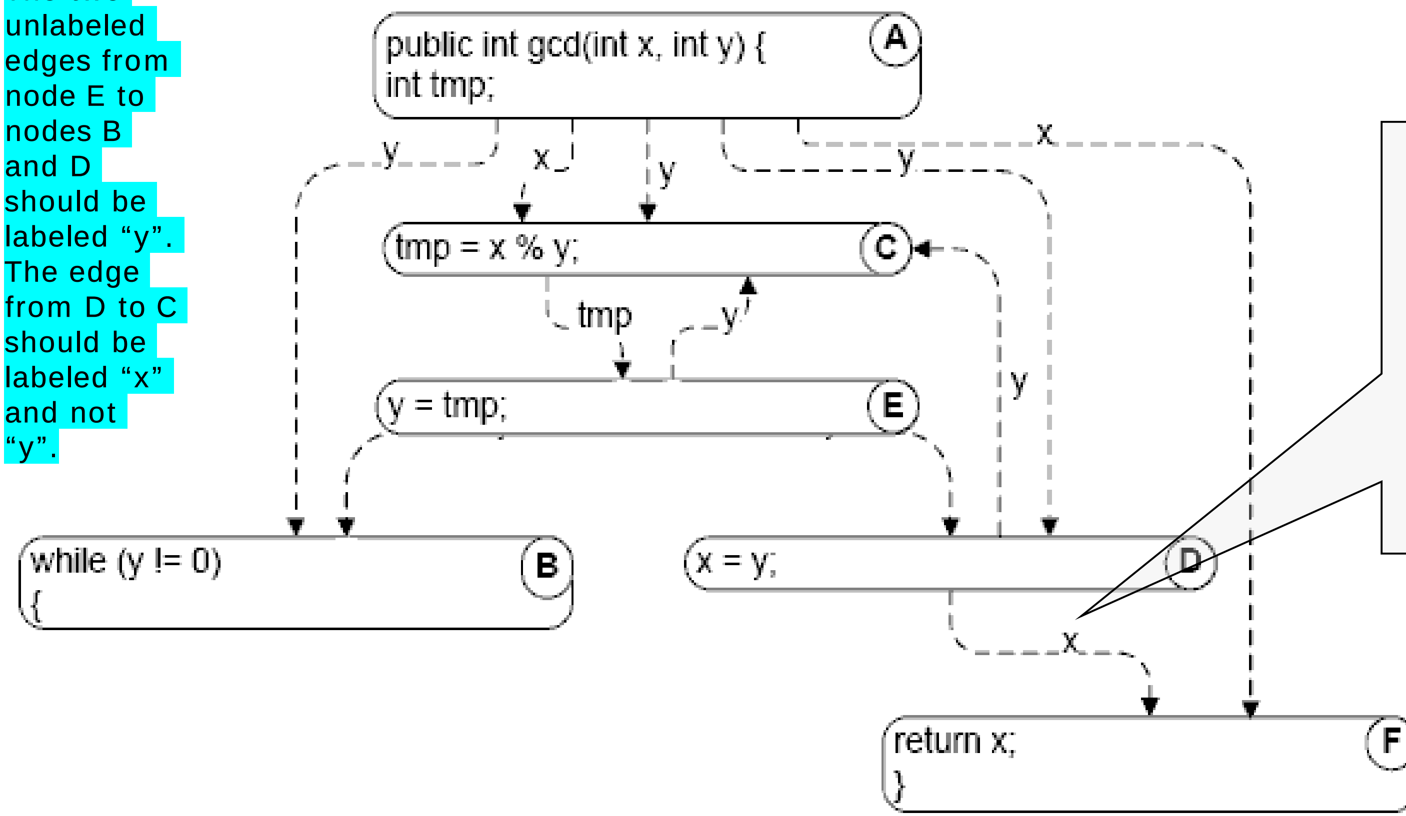
```
x = ...       // A: def x
q = ...
x = y;        //  B: kill x, def x
z = ...
y = f(x);   // C: use x
```



(Direct) Data Dependence Graph

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

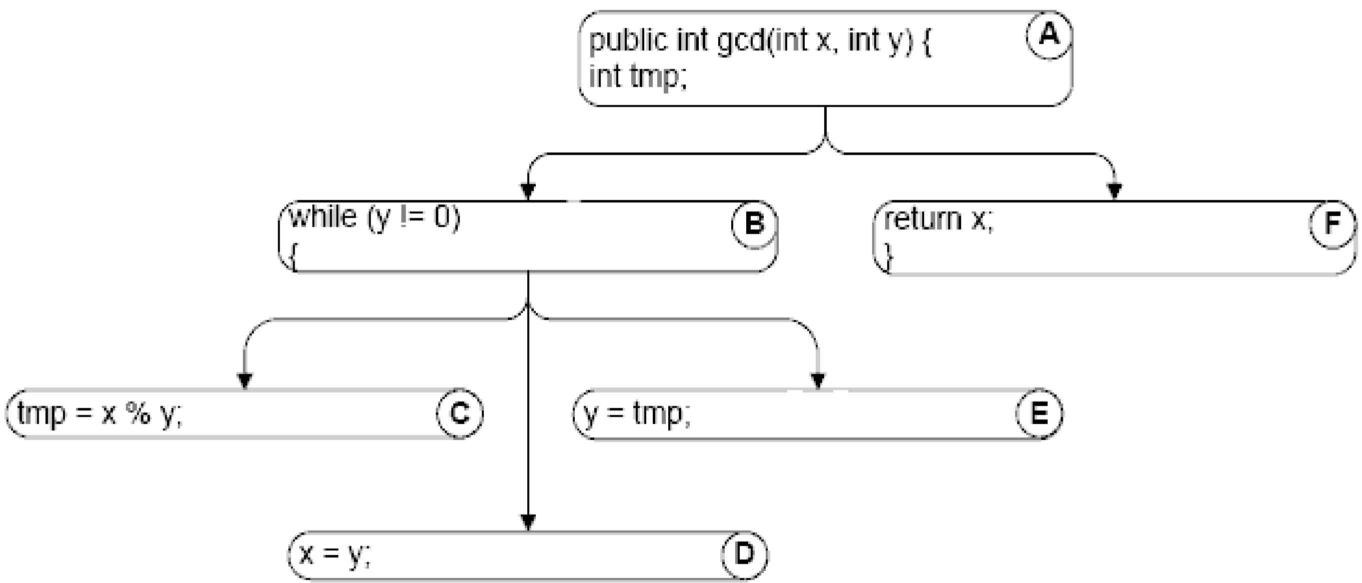
The two unlabeled edges from node E to nodes B and D should be labeled “y”. The edge from D to C should be labeled “x” and not “y”.



E.g., x is defined in D and used in F.

Control dependence (1)

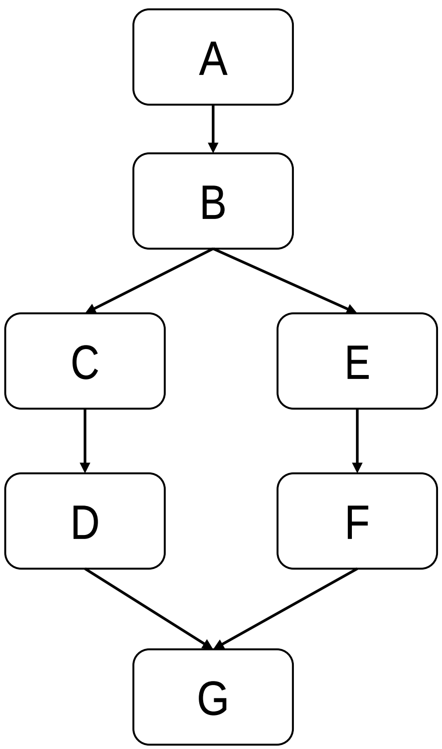
- **Data** dependence: Where did these values come from?
- **Control** dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



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.

Dominators (example)



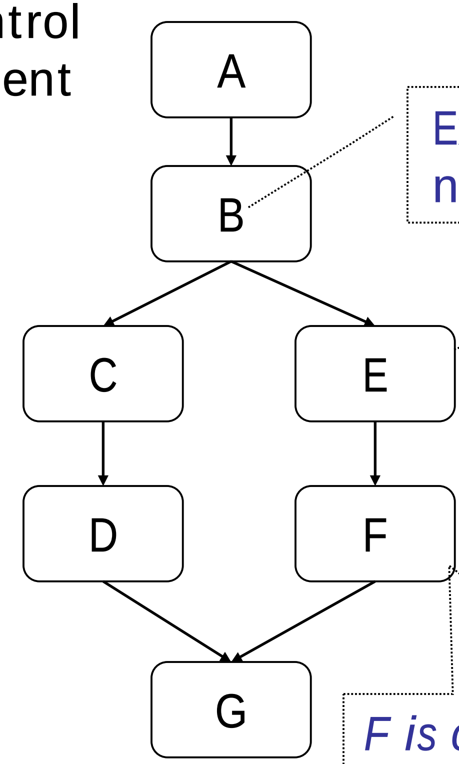
- A pre-dominates (i.e. is a pre-dominator of) all nodes; G post-dominates (i.e. is a post-dominator of) all nodes
- F and G post-dominate E
- G is the *immediate* post-dominator of B
  - C does *not* post-dominate B
- B is the *immediate* pre-dominator of G
  - F does *not* pre-dominate G
- B and all of its post-dominators form a tree

Control dependence (2)

- 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 conditions:
    - 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 NO successor of C in the control flow graph such that the above two conditions are true.
  - We say node N is ***control-dependent*** on node C.
    - <u>Intuitively: C was the last decision that controlled whether N executed</u>

Control Dependence

F is control dependent on B.



Execution of F is not inevitable at B, i.e., F not post-dominated by B

Execution of F is inevitable at E

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

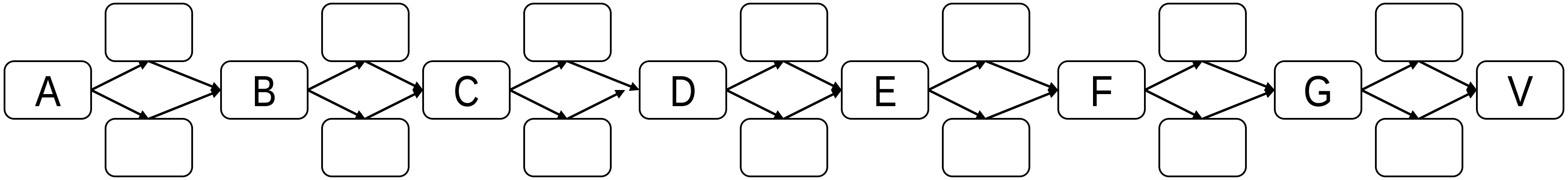
Data Flow Analysis

Computing data flow information

Calculating def-use pairs

- **Definition-use pairs** can be defined in terms of paths in the program control flow graph:
  - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
    - there is at least one control flow path from d to u which is also a definition-clear path.
  - Definition of v at line d (i.e., v_d) **reaches** u (v_d is a **reaching definition** at u).
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be *exponentially* larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

Exponential paths (even without loops)



2 paths from A to B

4 from A to C

8 from A to D

16 from A to E

…

128 paths from A to V

Tracing each path is not efficient, and we can do much better.

DF Algorithm

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way that reaching definitions at one node are related to reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
  - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p, i.e. **gen(p) = {v_p}**
  - If a definition v_q of variable v (where q denotes any node) reaches a predecessor node p, and if v is not redefined at p, then v_q is propagated on from p to n.

Equations of node E (y = tmp)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
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
  }
```

$$Reach(E) = ReachOut(D)$$

$$ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$$

Equations of node B (while (y != 0))

This line has two predecessors: Before the loop, end of the loop

```
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
  }
```

- $Reach(B) = ReachOut(A) \cup ReachOut(E)$
- $ReachOut(A) = gen(A) = \{x_A, y_A, tmp_A\}$
- $ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$

General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in pred(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$

$$\text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x,\ x \neq n \}$$

Avail equations*

$$\text{Avail}(n) = \bigcap_{m \in pred(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ \text{exp} \mid \text{exp is computed at } n \}$$

$$\text{kill}(n) = \{ \text{exp} \mid \text{exp has variables assigned at } n \}$$

Live variable equations*

$$\text{Live}(n) = \bigcup_{m \in succ(n)} \text{LiveOut}(m)$$

$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$$

$$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$$

Classification of analyses*

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path ($\cup$)	All-paths ($\cap$)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
  - For “any path” problems, first guess is “nothing” (empty set) at each node
  - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a “fixed point” solution where every new calculation produces the same value as the previous guess.

Worklist Algorithm for Data Flow

An iterative work-list algorithm to compute reaching definitions by applying each flow equation until the solution stabilizes.

Algorithm *Reaching definitions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$
$\text{pred}(n) = \{m \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{succ}(m) = \{n \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{gen}(n) = \{v_n\}$ if variable v is defined at n, otherwise $\{\}$
$\text{kill}(n) =$ all other definitions of v if v is defined at n, otherwise $\{\}$

Output: $\text{Reach}(n) =$ the reaching definitions at node n

```
for n ∈ nodes loop
    ReachOut(n) = {} ;
end loop;

workList = nodes ;
while (workList ≠ {}) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    n = any node in workList ;
    workList = workList \ {n} ;

    oldVal = ReachOut(n) ;

    // Apply flow equations, propagating values from predecessars
    Reach(n) = ∪_{m∈pred(n)} ReachOut(m);
    ReachOut(n) = (Reach(n) \ kill(n)) ∪ gen(n) ;
    if ( ReachOut(n) ≠ oldVal ) then
        // Propagate changed value to successor nodes
        workList = workList ∪ succ(n)
    end if;
end loop;
```

Worklist Algorithm for Data Flow (cont.)*

An iterative work-list algorithm for computing available expressions.

Algorithm *Available expressions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$, with a distinguished root node $start$.
pred$(n) = \{m \in \text{nodes} \mid (m,n) \in \text{edges}\}$
succ$(m) = \{n \in \text{nodes} \mid (m,n) \in \text{edges}\}$
gen(n) = all expressions e computed at node n
kill(n) = expressions e computed anywhere, whose value is changed at n;
kill$(start)$ is the set of all e.

Output: Avail(n) = the available expressions at node n

for $n \in$ nodes **loop**
    AvailOut(n) = set of all e defined anywhere ;
end loop;

workList = nodes ;
while (workList $\neq \{\}$) **loop**
    // Take a node from worklist (e.g., pop from stack or queue)
    n = any node in workList ;
    workList = workList $\setminus \{n\}$;
    oldVal = AvailOut(n) ;
    // Apply flow equations, propagating values from predecessors
    Avail$(n) = \bigcap_{m \in \text{pred}(n)}$ AvailOut(m);
    AvailOut(n) = (Avail$(n) \setminus$ kill(n)) $\cup$ gen(n) ;
    if (AvailOut$(n) \neq$ oldVal) **then**
        // Propagate changes to successors
        workList = workList $\cup$ succ(n)
    end if;
end loop;

Worklist Algorithm for Data Flow (cont.)*

Refer to the Figures in the previous two slides.

One way to iterate to a fixed point solution.

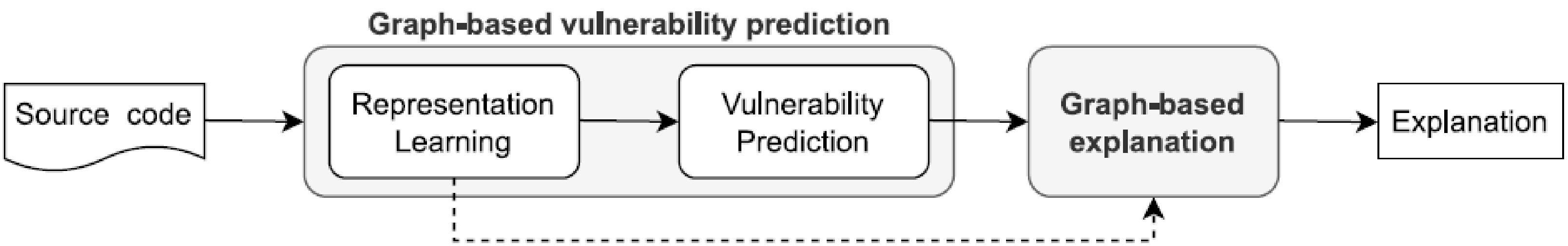
General idea:

- Initially all nodes are on the work list, and have default values
  - Default for "any-path" problem is the empty set, default for "all-path" problem is the set of all possibilities (union of all gen sets)
- While the work list is not empty
  - Pick any node n on work list; remove it from the list
  - Apply the data flow equations for that node to get new values
  - If the new value is changed (from the old value at that node), then
    - Add successors (for forward analysis) or predecessors (for backward analysis) on the work list
- Eventually the work list will be empty (because new computed values = old values for each node) and the algorithm stops.

Cooking your own: From Execution to Conservative Flow Analysis

- We can use the same data flow algorithms to approximate other dynamic properties
  - Gen set will be "facts that become true here"
  - Kill set will be "facts that are no longer true here"
  - Flow equations will describe propagation
- Example: Taintedness (in web form processing)
  - "Taint": a user-supplied value (e.g., from web form) that has not been validated
  - Gen: we get this value from an untrusted source here
  - Kill: we validated to make sure the value is proper

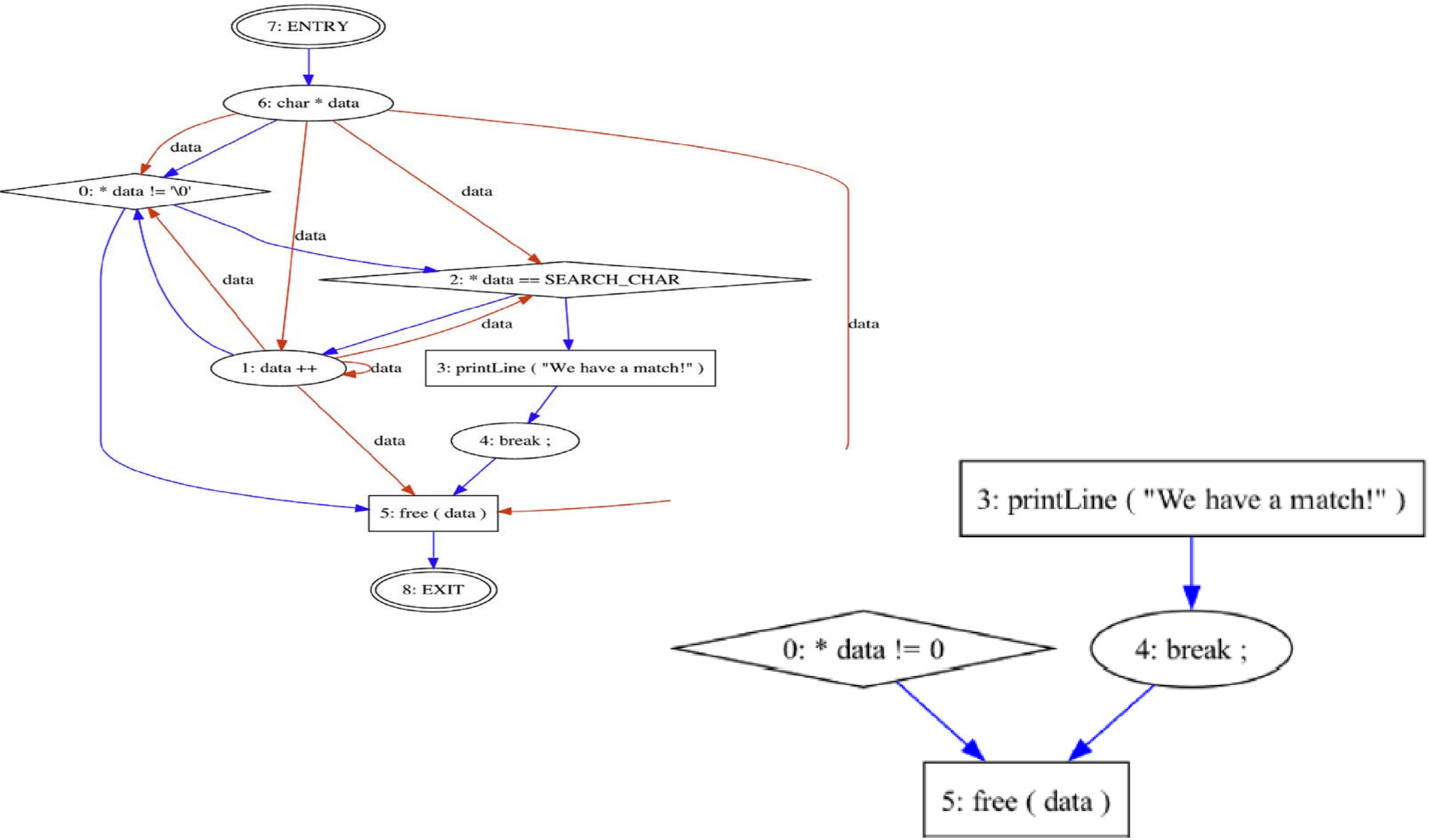
Program dependency in vulnerability prediction




```
void action(char * data) const {
    /* FLAW: We are incrementing the pointer in the loop - this will
     * cause us to free the memory block not at the start of the buffer */
     for (; *data != '\0'; data++) {
            if (*data == SEARCH_CHAR) {
            printLine("We have a match!");
            break;
      }
   }
   free(data);
}
```

Hong Quy Nguyen, Thong Hoang, Hoa Khanh Dam, and Aditya Ghose. 2025. ***Graph-based explainable vulnerability prediction****. Information and Software Technology 177, C (Jan 2025). https://doi.org/10.1016/j.infsof.2024.107566*

Program dependency in vulnerability prediction (cont.)



Hong Quy Nguyen, Thong Hoang, Hoa Khanh Dam, and Aditya Ghose. 2025. ***Graph-based explainable vulnerability prediction****. Information and Software Technology 177, C (Jan 2025). https://doi.org/10.1016/j.infsof.2024.107566*