浙江大學
ZHEJIANG UNIVERSITY

Introduction, Control Flow Analysis

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: CMPSC 447 of PSU (Prof. Gang Tan), CS640 of GMU (Prof. Liz White), CS 252r of Harvard (Prof. Stephen Chong)

Overview

- Buffer overflow
- Code injection, code reuse
  - Return2libc, ROP, Blind ROP
  - Stack canary, ASLR
- UAF, Integer Overflow, Type Confusion
- Format string vulnerability

Introduction

Program Analysis

- Any *automated* analysis *at compile or dynamic time* to find potential *bugs*
- Broadly classified into
  - Dynamic analysis
  - Static analysis

Dynamic Analysis

- Analyze the code when it is running
    - Detection
        - E.g., dynamically detect whether there is an out-of-bound memory access, for a particular input
    - Response
        - E.g., stop the program when an out-of-bound memory access is detected

Dynamic Analysis Limits

- Major advantage
  - After detecting a bug, it is a real one
  - No false positives
- Major limitation
  - Detecting a bug for a particular input - coverage
  - Cannot find bugs for uncovered inputs
  - If (input == 0x134576) {bug()} else {normal(); }

Question

- Can we build a technique that identifies all bugs?
  - Turns out that we can: static analysis
  - Is this real? What's the potential issue?

Static Analysis

- Analyze the code before it is run (during compile time)
- Explore **all possible executions** of a program
    - **All possible inputs**
- **Approximate all possible states**
    - Build **abstractions** to “run in the aggregate”
    - Rather than executing on concrete states
    - Finite-sized abstractions representing a collection of states
- But, it has its own major limitation due to approximation
    - **Can identify many false positives (not actual bugs)**

Static Analysis

- Broad range of static-analysis techniques:
  - simple syntactic checks like **`grep`**

```
grep " gets(" *.cpp
```

- More advanced greps: ITS4, FlawFinder
  - A database of security-sensitive functions
    - gets, strcpy, strcat, …
    - For each one, suggest how to fix

Static Analysis

- More advanced analyses take into account semantics
  - dataflow analysis, abstract interpretation, **symbolic execution**, constraint solving, model checking
  - Commercial tools: Coverity, Fortify, Secure Software, GrammaTech

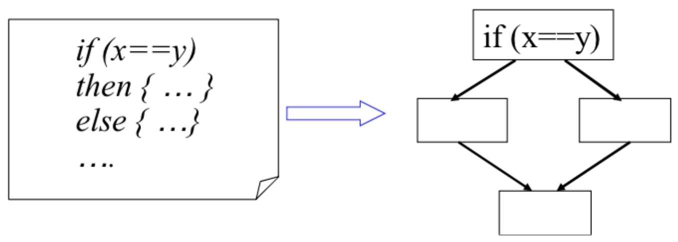
Control Flow Analysis

Program Control Flow

- Control flow
  - Sequence of operations
  - Representations
    - Control flow graph
    - Control dependency
    - Call graph
  - Control flow analysis
    - Analyzing program to discover its control structure

Control Flow Graph

- CFG models flow of control in the program (procedure)
- G = (N, E) as a directed graph
  - Node n ∈ N: basic blocks
    - A basic block is a maximal sequences of stmts with a single entry point, single exit point and no internal tranches
    - For simplicity, we assume a unique entry node n_0 and a unique exit node n_f in later discussions
  - Edge e=(ni, nj) ∈ E: possible transfer of control from block ni to block nj



Basic Blocks

- Definition
  - A basic block is a maximal sequence of consecutive statements with a single entry point, a single exit point, and no internal branches
- Basic unit in control flow analysis
- Local level of code optimizations
  - Redundancy elimination, register-allocation
- For security: reachability analysis, liveness analysis …

Basic Block Example

```
(1)   i := m - 1
(2)   j := n
(3)   t1 := 4 * n
(4)   v := a[t1]
(5)   i := i + 1
(6)   t2 := 4 * i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j - 1
(10)  t4 := 4 * j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
...
```

- How many basic blocks in this code fragment?
- What are they?

Basic Block Example

```
(1)  i := m – 1
(2)  j := n
(3)  t1 := 4 * n
(4)  v := a[t1]
(5)  i := i + 1
(6)  t2 := 4 * I
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
(9)  j := j – 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*I
(15) x := a[t6]
...
```

- How many basic blocks in this code fragment?
- What are they?

Identify Basic Blocks

- Input: A sequence of intermediate code statements
  - **Determine the leaders, the first statements of basic blocks**
    - The first statement in the sequence (entry point) is a leader
    - Any statement that is the target of a branch (conditional or unconditional) is a leader
    - Any statement immediately following a branch (conditional or unconditional) or a return is a leader
- For each leader, its basic block is the leader and all statements up to, but not including, the next leader or the end of the program

Example: Leaders

(1) i := m – 1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) If t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

Example: Leaders

(1) i := m – 1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) If t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

Example: Basic Blocks

```
(1)  i := m – 1
(2)  j := n
(3)  t1 := 4 * n
(4)  v := a[t1]
```

```
(5)  i := i + 1
(6)  t2 := 4 * i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
```

```
(9)  j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) If t5 > v goto (9)
```

```
(13) if i >= j goto (23)
```

```
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
```

```
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x
```

Generating CFGs

- Partition intermediate code into basic blocks
- Add edges corresponding to control flows between blocks
  - Unconditional goto
  - Conditional branch – multiple edges
  - Sequential flow – control passes to the next block (if no branch at the end)
- If no unique entry node n_0 or exit node n_f, add dummy nodes and insert necessary edges
  - Ideally no edges entering n_0; no edges exiting n_f
  - Simplify many analysis and transformation algorithms

Example: CFG

```
(1)   i := m – 1
(2)   j := n
(3)   t1 := 4 * n
(4)   v := a[t1]

(5)   i := i + 1
(6)   t2 := 4 * i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)

(9)   j := j - 1
(10)  t4 := 4 * j
(11)  t5 := a[t4]
(12)  If t5 > v goto (9)

(13)  if i >= j goto (23)

(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4 * i
(17)  t8 := 4 * j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4 * j
(21)  a[t10] := x
(22)  goto (5)

(23)  t11 := 4 * i
(24)  x := a[t11]
(25)  t12 := 4 * i
(26)  t13 := 4 * n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4 * n
(30)  a[t15] := x
```

Complications in CFG Construction

- Function calls
  - Instruction scheduling may prefer function calls as basic block boundaries
  - Special functions as setjmp() and longjmp()
- Exception handling
- Ambiguous jump
  - Jump r1 //target stored in register r1
  - Static analysis may generate edges that never occur at runtime

Nodes in CFG

- Given a CFG = <N, E>
  - If there is an edge $\mathbf{n_i} \rightarrow \mathbf{n_j} \in E$
  - $\mathbf{n_i}$ is a predecessorof $\mathbf{n_j}$
  - $\mathbf{n_j}$ is a successor of $\mathbf{n_i}$
- For any node $n \in N$
  - **Pred(n)**: the set of predecessors of n
  - **Succ(n)**: the set of successors of n
  - A **branch node** is a node that has more than one successor
  - A **join node** is a node that has more than one predecessor

Depth First Traversal

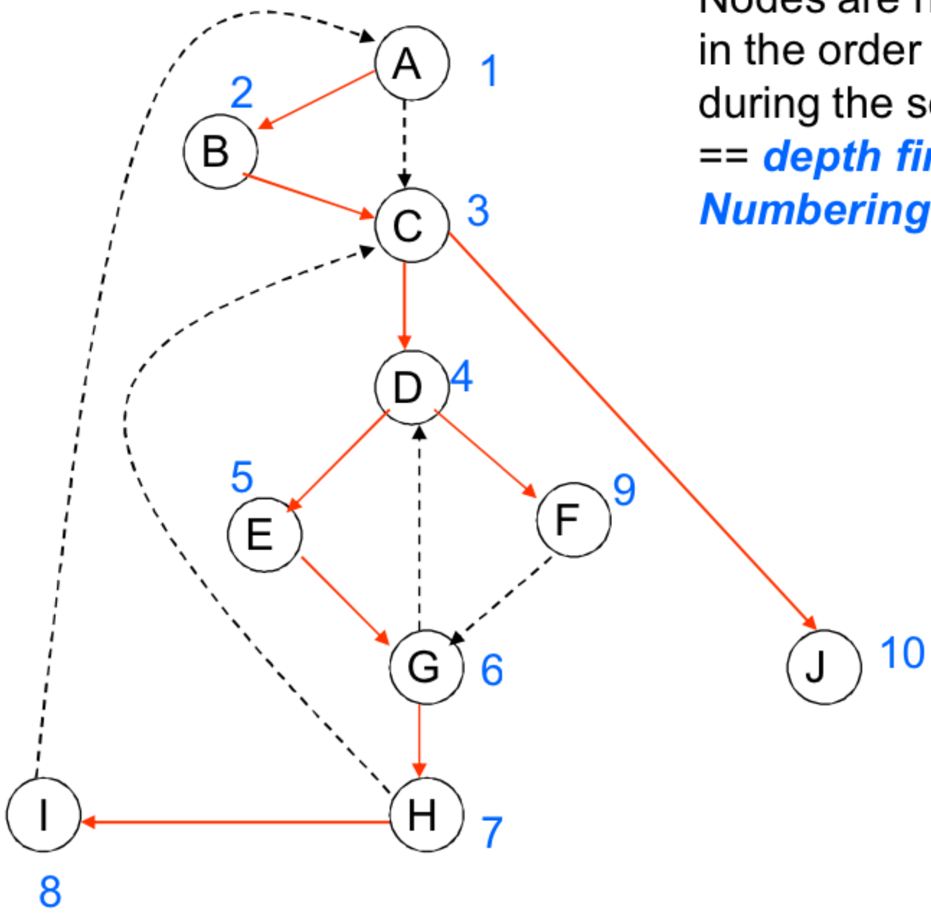
- CFG is a rooted, directed graph
  - Entry node as the root
- Depth-first traversal (depth-first searching)
  - Idea: start at the root and explore as far/deep as possible along each branch before backtracking
  - Can build a spanning tree for the graph
- Spanning tree of a directed graph G contains all nodes of G such that
  - There is a path from the root to any node reachable in the original graph and there are no cycles

DFS Spanning Tree

```
procedure span(v)  /* v is a node in the
  graph */
   InTree(v) = true
   For each w that is a successor of v do
        if (!InTree(w)) then
        Add edge v → w to spanning tree
        span(w)
end span
```

- Initial: **span(n_0)**

DFST Example

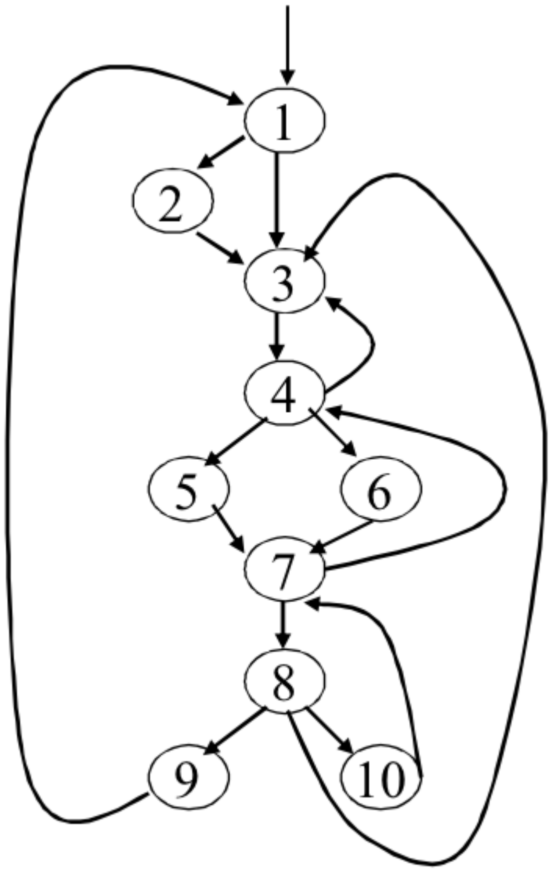


Nodes are numbered in the order visited during the search == ***depth first pre-order Numbering.***

Dominance

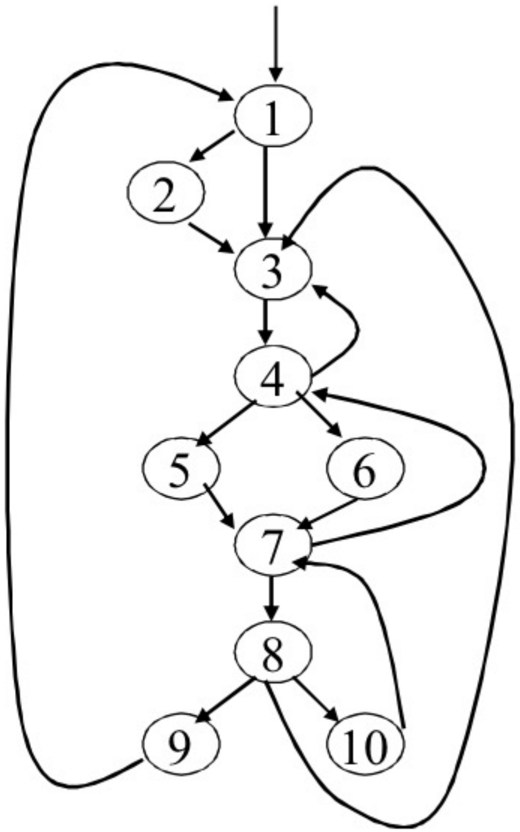
- Node d of a CFG dominates node n if every path from the entry node of the graph to n passes through d (d dom n)
  - Dom(n): the set of dominators of node n
  - Every node dominates itself: $n \in \text{Dom}(n)$
  - Node d strictly dominates n if $d \in \text{Dom}(n)$ and $d \neq n$
  - Dominance-based loop recognition: entry of a loop dominates all nodes in the loop
- Each node n has a unique immediate dominator m which is the last dominator of n on any path from the entry to n (m idom n), $m \neq n$
  - The immediate dominator m of n is the strict dominator of n that is closest to n

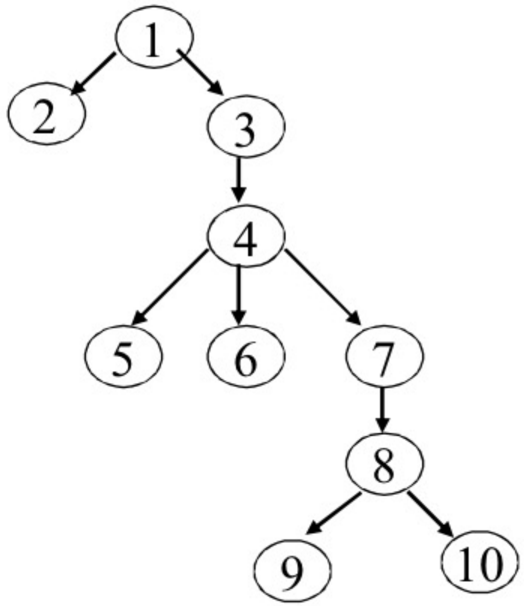
Dominator Example



Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8

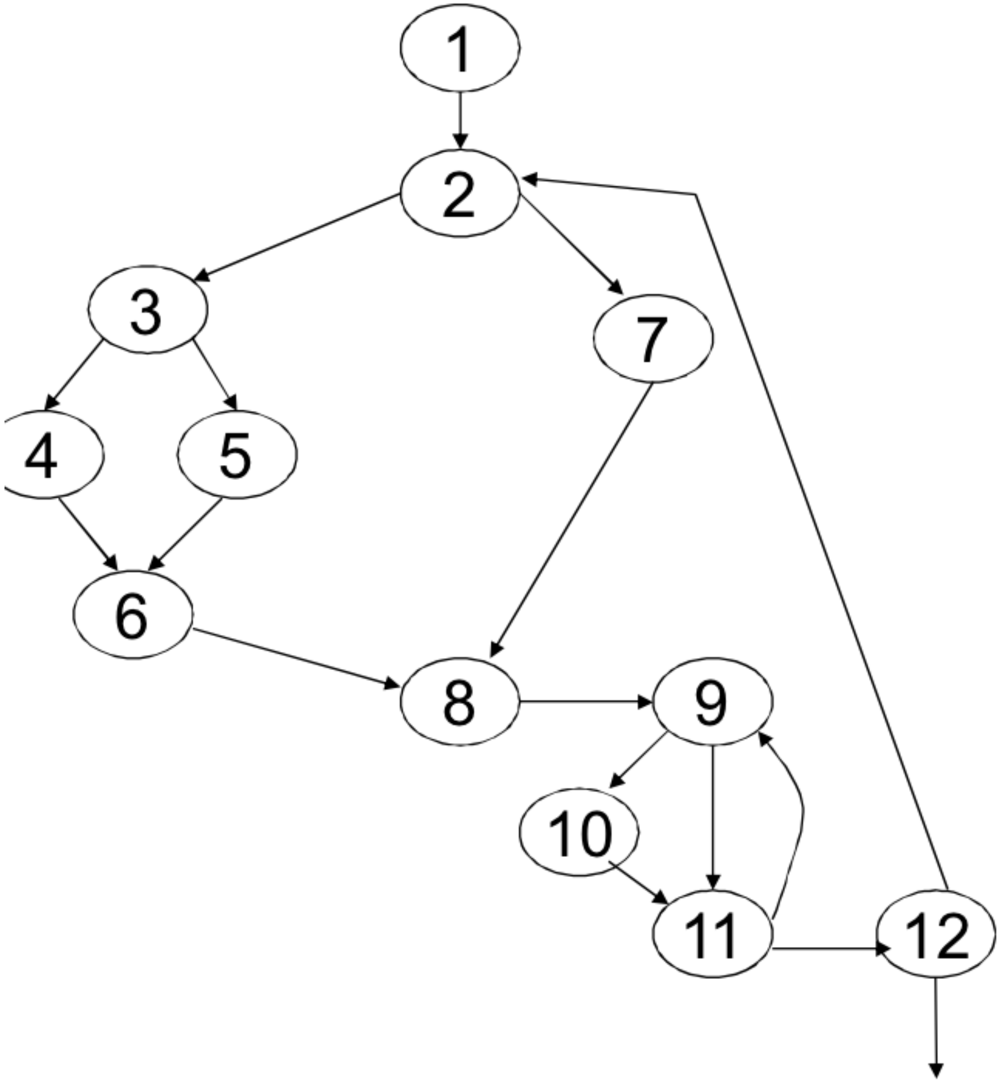
Dominator Tree





- In a dominator tree, a node's parent is its immediate dominator

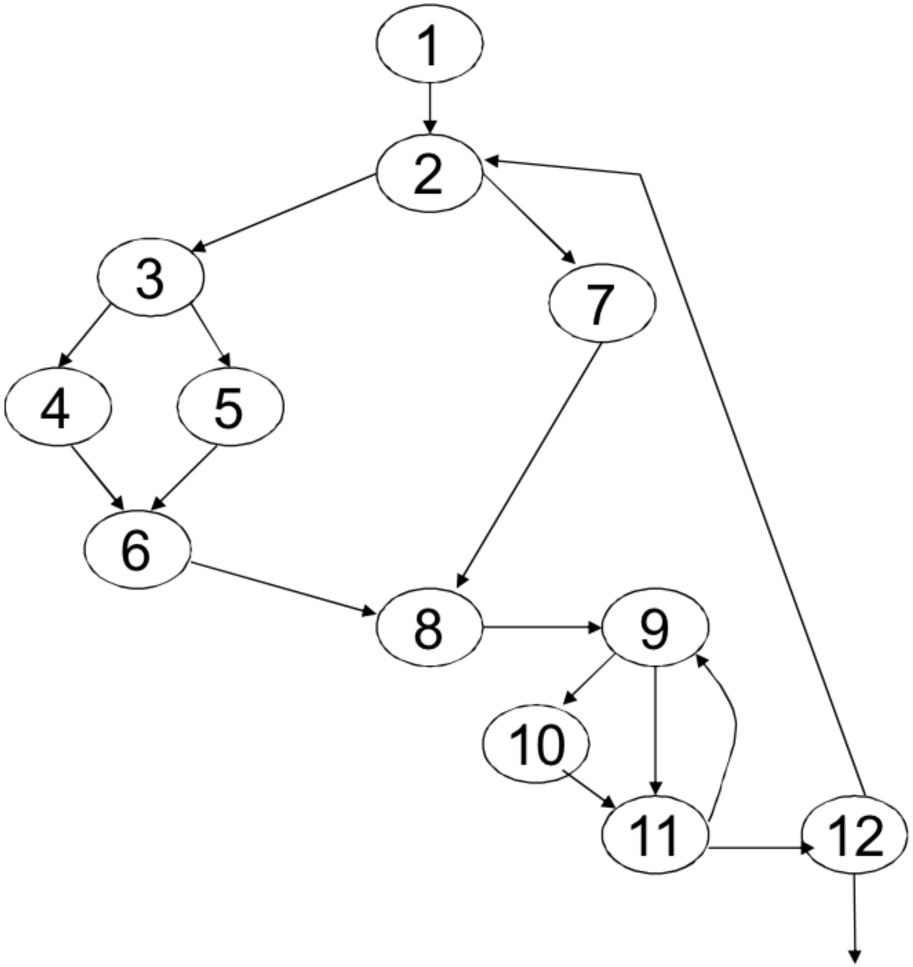
Example 2



Block	Dom	IDom
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

Example 2

Example 2



Block	Dom	IDom
1	1	-
2	1,2	1
3	1,2,3	2
4	1,2,3,4	3
5	1,2,3,5	3
6	1,2,3,6	3
7	1,2,7	2
8	1,2,8	2
9	1,2,8,9	8
10	1,2,8,9,10	9
11	1,2,8,9,11	9
12	1,2,8,9,11,12	11

Call Graph

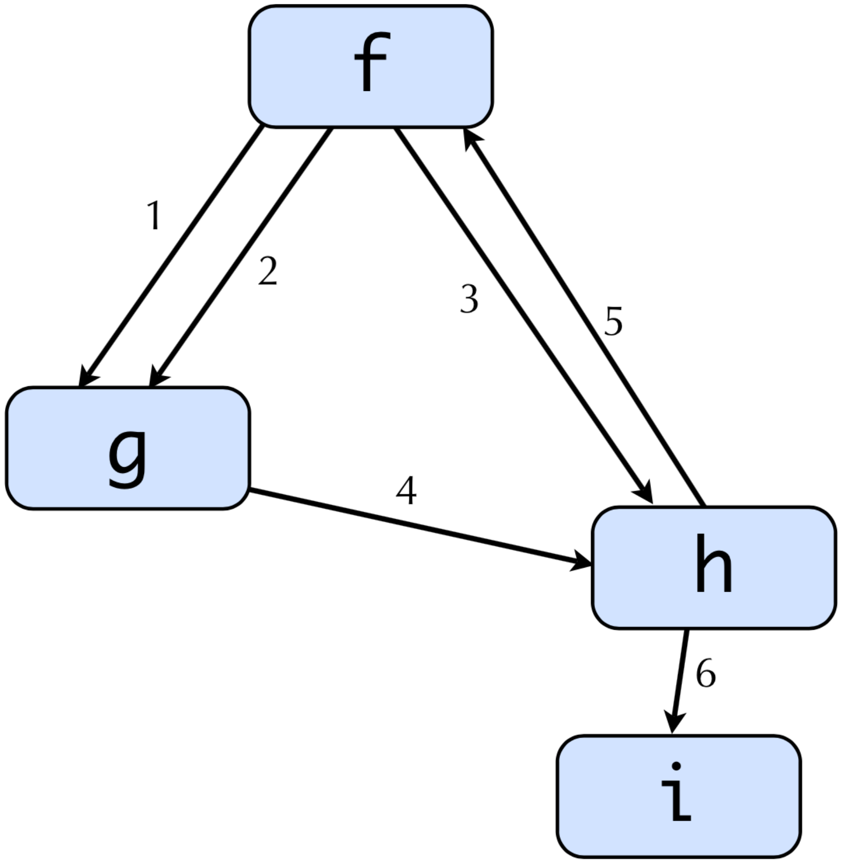
- So far looked at intraprocedural analysis: a single function
- **Inter-procedural analysis** uses calling relationships **among** procedures
  - Enables more precise analysis information

Call Graph

- First problem: **how do we know what procedures are called from where?**
  - Especially difficult in higher-order languages, languages where functions are values **(function pointer)**
  - We'll ignore this for now, and return to it later in course...
- Let's assume we have a (static) **call graph**
  - Indicates which procedures can call which other procedures, and from which program points.

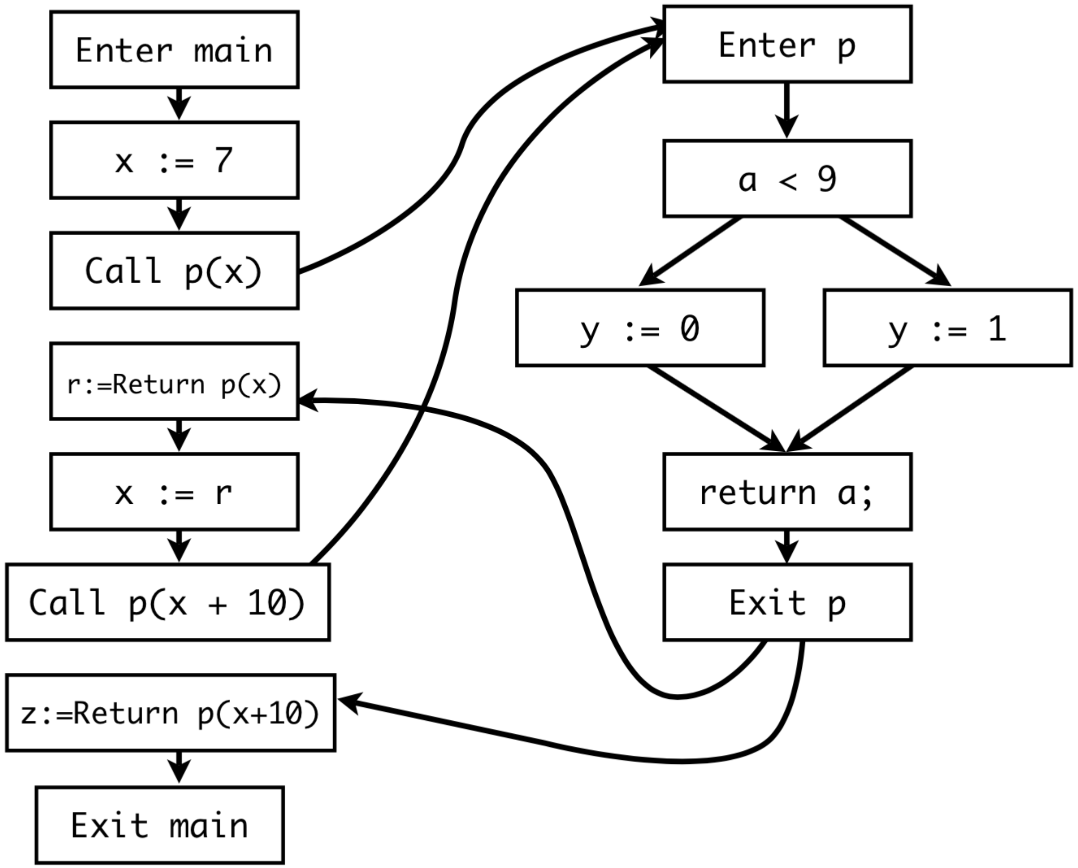
Call Graph Example

```
f() {
    1:  g();
    2:  g();
    3:  h();
}

g() {
    4: h();
}

h() {
    5: f();
    6: i();
}

i() { … }
```



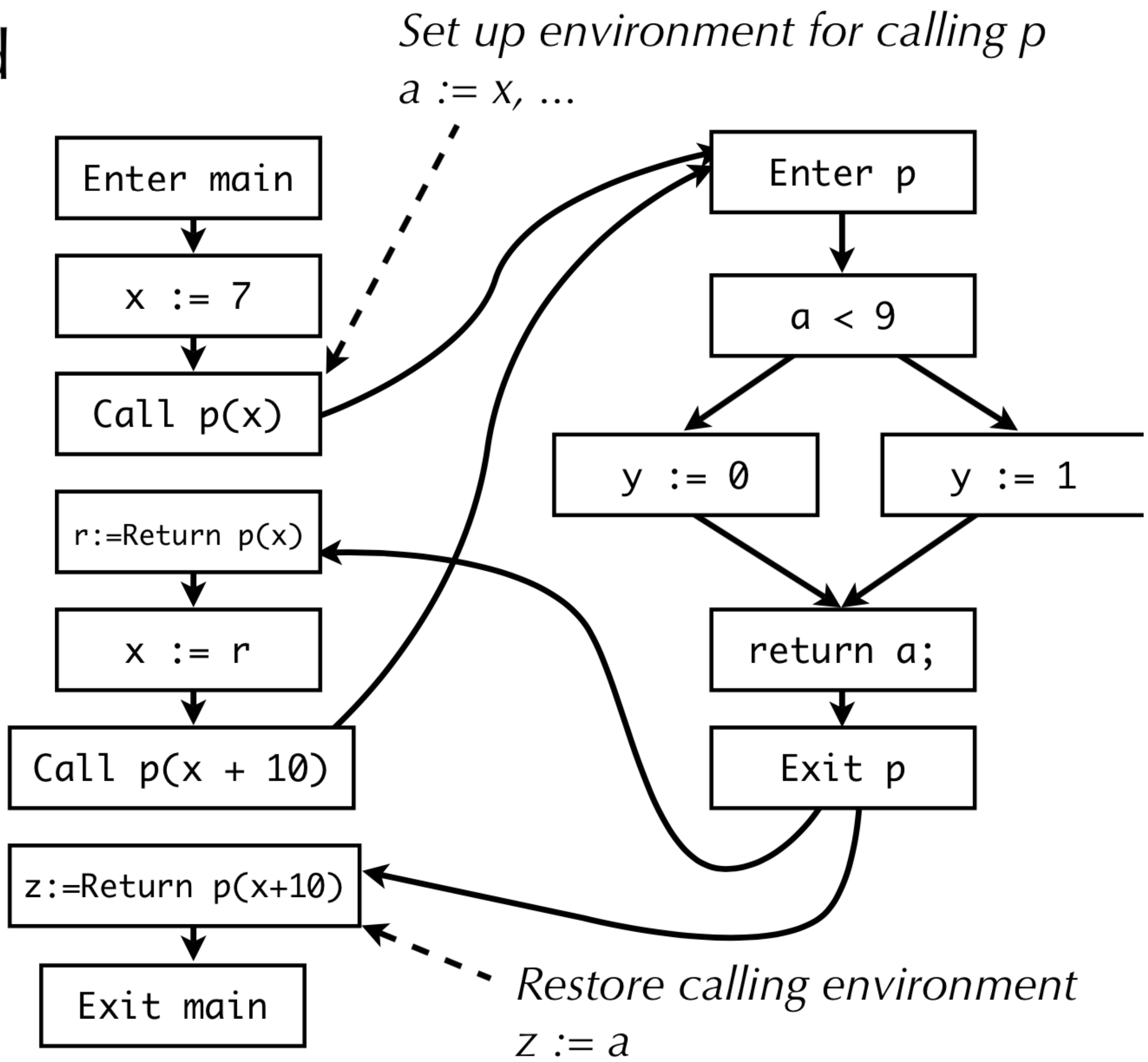
- How do we deal with procedure calls?
- Obvious idea: make one big CFG

```
main() {
  x := 7;
  r := p(x);
  x := r;
  z := p(x + 10);
}

p(int a) {
  if (a < 9)
    y := 0;
  else
    y := 1;
  return a;
}
```

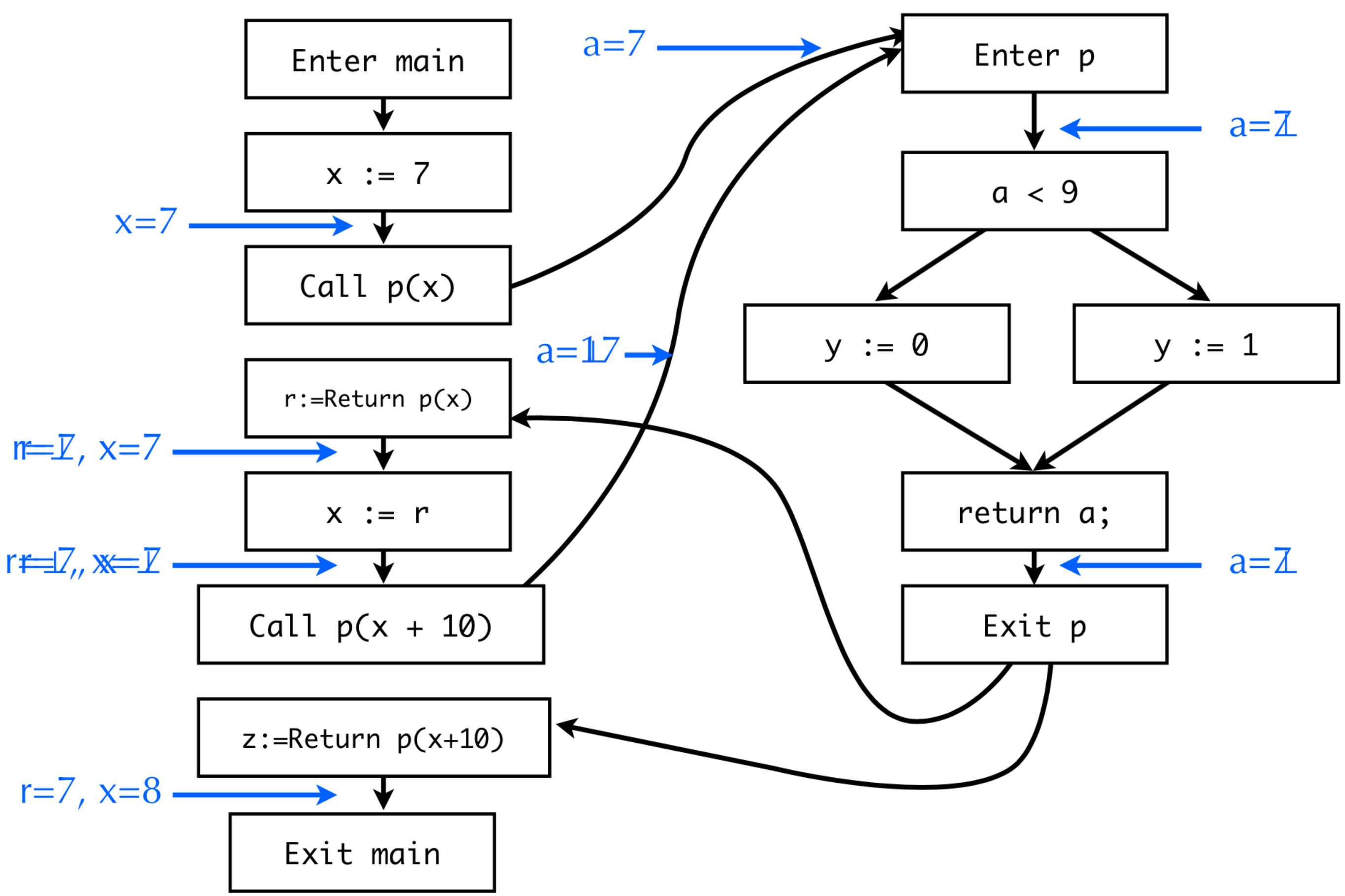


Interprocedural CFG (ICFG)

- CFG may have additional nodes to handle call and returns
  - Treat arguments, return values as assignments
- Note: a local program variable represents multiple locations



Example



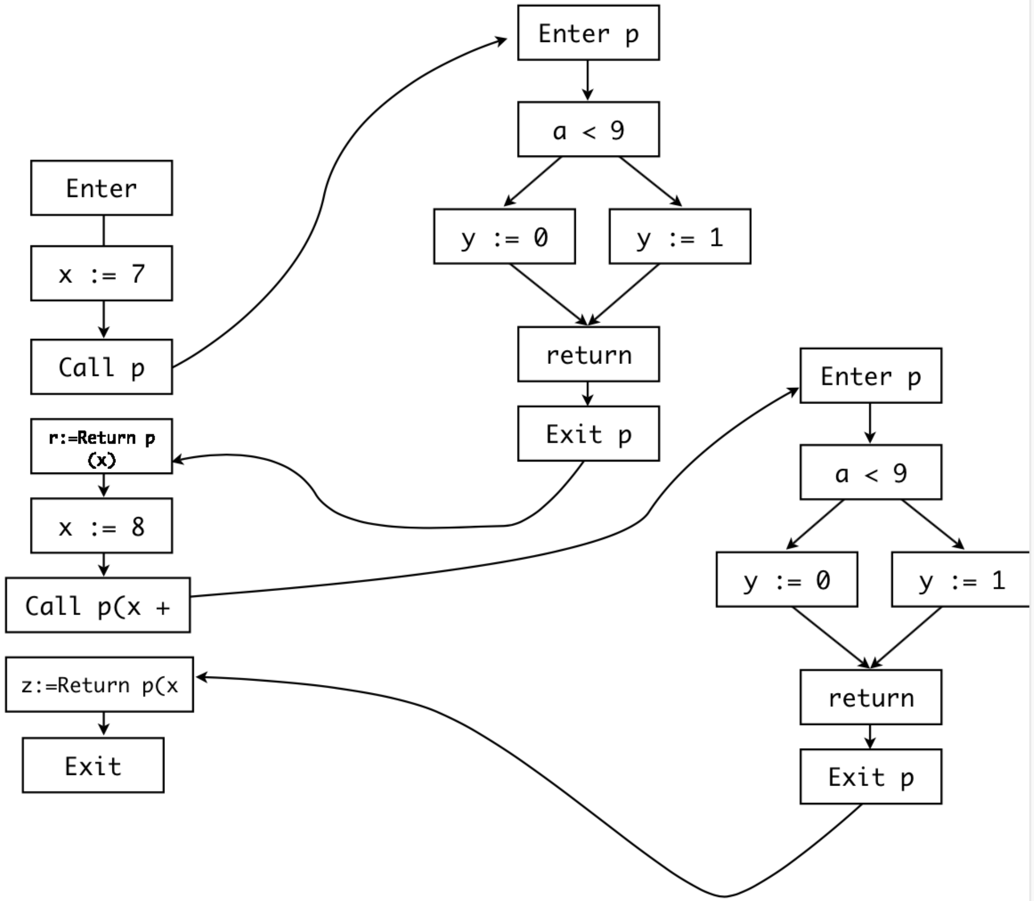
Enter main
x := 7
x=7
Call p(x)
r:=Return p(x)
x := r
Call p(x + 10)
z:=Return p(x+10)
r=7, x=8
Exit main
a=7
Enter p
a < 9
y := 0
y := 1
return a;
Exit p

Invalid Paths

- Problem: dataflow facts from one call site “tainting” results at other call site
  - p analyzed with merge of dataflow facts from all call sites
- How to address?

Inlining

- Inlining
  - Use a new copy of a procedure's CFG at each call site
- Problems? Concerns?
  - May be expensive! Exponential increase in size of CFG
    - p() { q(); q(); }   q() { r(); r() }   r() { … }
  - What about recursive procedures?
    - p(int n) { … p(n-1); … }
    - More generally, cycles in the call graph

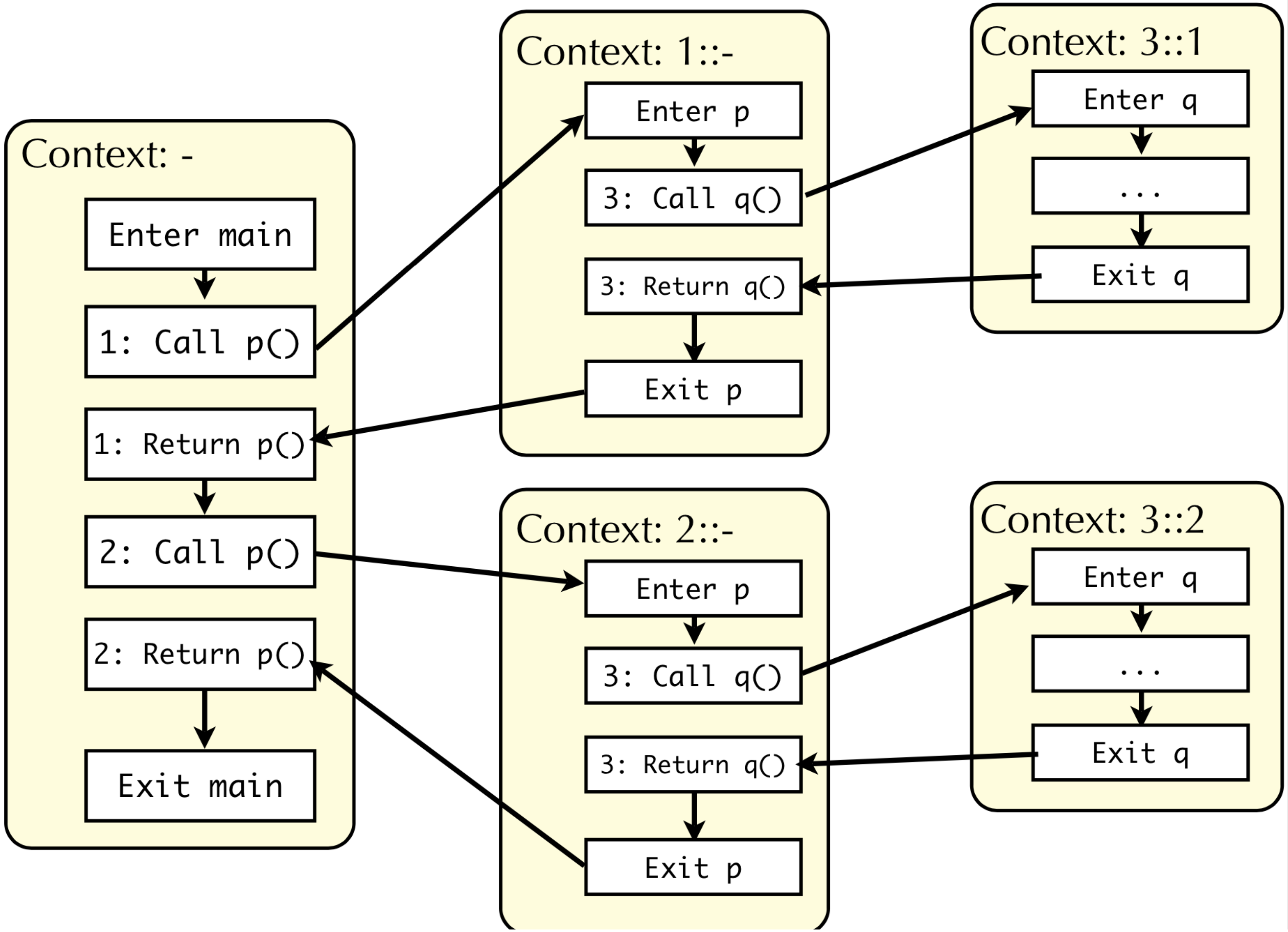


Context Sensitivity

- Solution: make a **finite** number of copies
- **Use context information to determine when to share a copy**
  - **Results in a context-sensitive analysis**
- Choice of what to use for context will produce different tradeoffs between precision and scalability
- Common choice: approximation of call stack

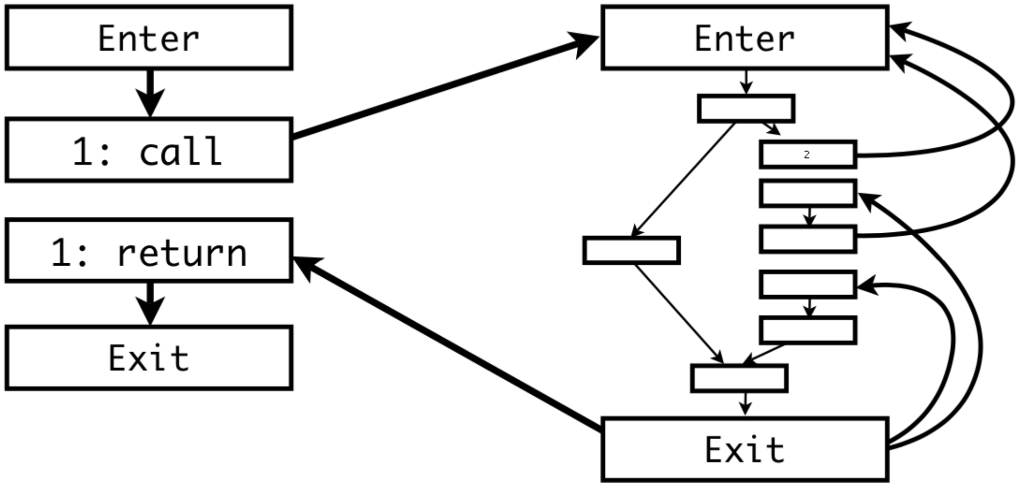
Context Sensitivity Example

```
main() {
   1: p();
   2: p();
}

p() {
   3: q();
}
q() {
   …
}
```



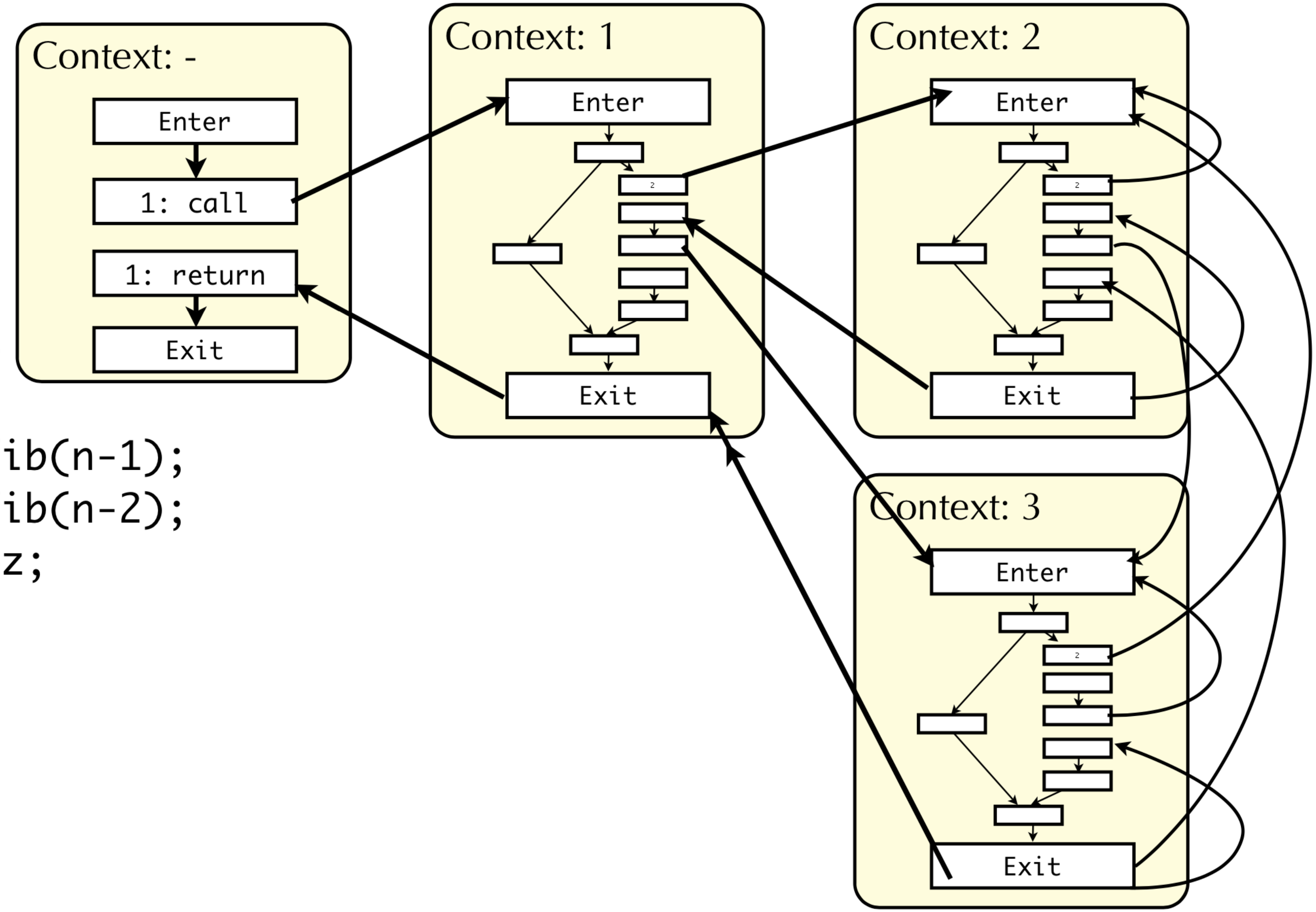
Fibonacci: context insensitive

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
     2: y := fib(n-1);
     3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



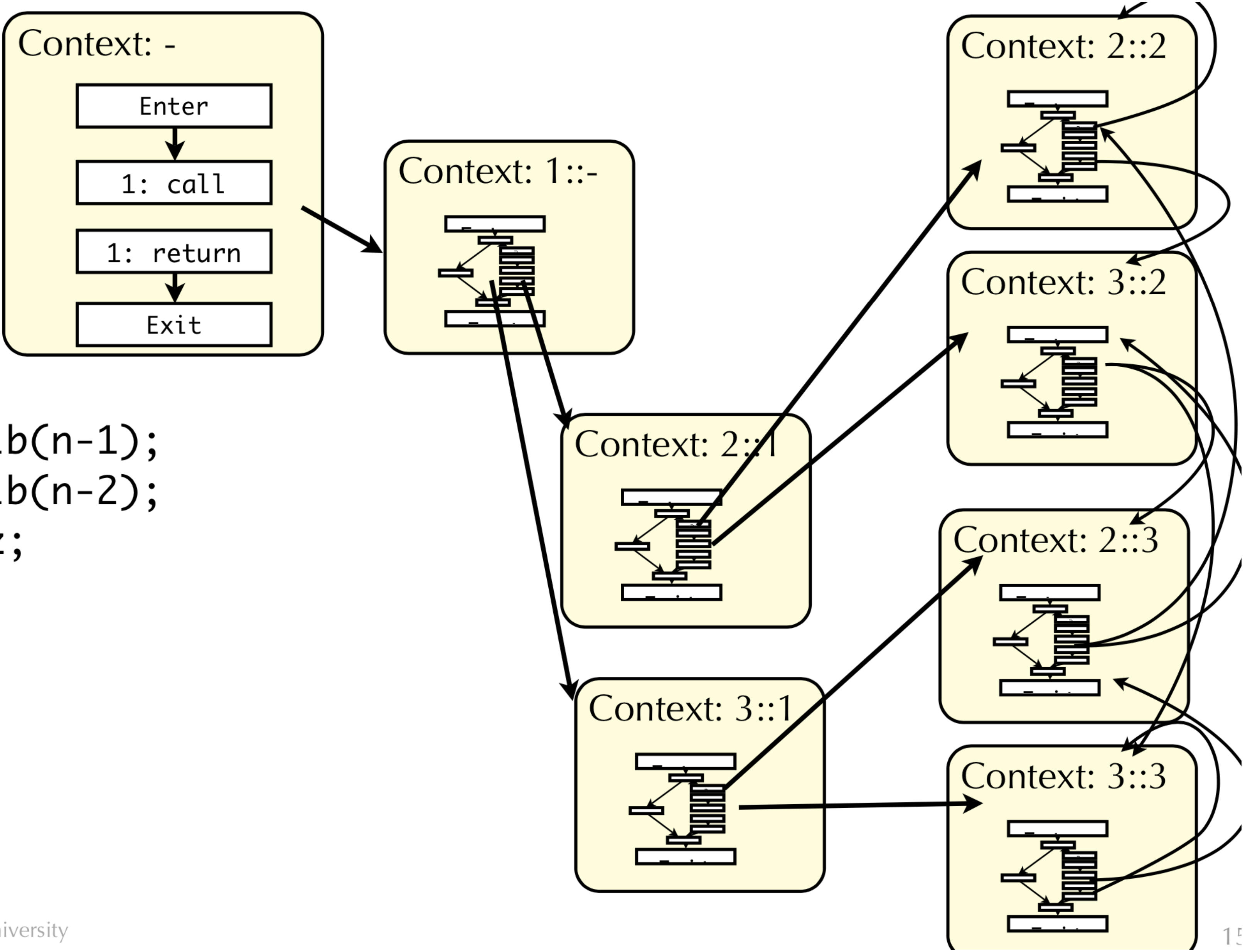
Fibonacci: context sensitive, stack depth 1

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



Fibonacci: context sensitive, stack depth 2

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



Other contexts

- Context sensitivity distinguishes between different calls of the same procedure
  - Choice of contexts determines which calls are differentiated
- Other choices of context are possible
  - Caller stack
    - Less precise than call-site stack
    - E.g., context “2::2” and “2::3” would both be “fib::fib”
  - Object sensitivity: which object is the target of the method call?
    - For OO languages.
    - Maintains precision for some common OO patterns
    - Requires pointer analysis to determine which objects are possible targets
    - Can use a stack (i.e., target of methods on call stack)

Common Concepts

Concepts in Static Analysis

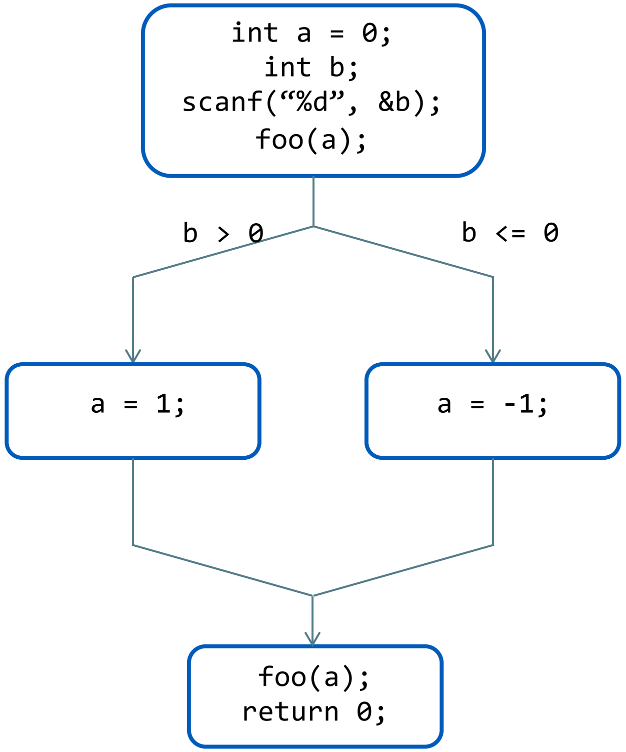
- Analysis scope
  - intra- and inter-procedural
  - flow sensitive, context sensitive and path sensitive

Analysis Scope

- Question: what is printed during execution of main()?
  - How many times is printf called in main?
  - What is the value of a?
  - What is the value of param?

```c
#include <stdio.h>
int foo(int param) {
    printf("%d", param);
    return 0;
}

int main() {
    int a = 0;
    int b;
    scanf("%d", &b);
    foo(a);
    if (b > 0) {
        a = 1;
    } else {
        a = -1;
    }
    foo(a);
    return 0;
}
```

Building CFG




```c
#include <stdio.h>
int foo(int param) {
    printf("%d", param);
    return 0;
}

int main() {
    int a = 0;
    int b;
    scanf("%d", &b);
    foo(a);
    if (b > 0) {
        a = 1;
    } else {
        a = -1;
    }
    foo(a);
    return 0;
}
```

Intra- and Inter-procedural

- Problem: how many times is printf called in main?
- Intra-procedural: printf is not called
- Inter-procedural: printf is called twice

```c
#include <stdio.h>
int foo(int param) {
    printf("%d", param);
    return 0;
}

int main() {
    int a = 0;
    int b;
    scanf("%d", &b);
    foo(a);
    if (b > 0) {
        a = 1;
    } else {
        a = -1;
    }
    foo(a);
    return 0;
}
```

Unknown

Unknown

Flow Sensitivity

- Problem: what is the value of a?
- Flow-insensitive: value of a may be -1, 0, or 1
- Flow-sensitive:

  $a_8 = a_{11} = 0$

  $a_{13} = 1$

  $a_{15} = -1$

  a_{17} can be -1 or 1

- SSA form computation in LLVM IR are already flow sensitive

```c
1  #include <stdio.h>
2  int foo(int param) {
3      printf("%d", param);
4      return 0;
5  }
6
7  int main() {
8      int a = 0;
9      int b;
10     scanf("%d", &b);
11     foo(a);
12     if (b > 0) {
13         a = 1;
14     } else {
15         a = -1;
16     }
17     foo(a);
18     return 0;
19 }
```

Context Sensitivity

- Problem: what is the value of param?
- Context insensitive: value of param may be

  -1, 0, or 1

- Context sensitive:
  - $_{(main,11)}$foo: param = 0
  - $_{(main,17)}$foo: param = -1 or 1

```c
1  #include <stdio.h>
2  int foo(int param) {
3      printf("%d", param);
4      return 0;
5  }
6
7  int main() {
8      int a = 0;
9      int b;
10     scanf("%d", &b);
11     foo(a);
12     if (b > 0) {
13         a = 1;
14     } else {
15         a = -1;
16     }
17     foo(a);
18     return 0;
19 }
```

Path Sensitivity

- Path insensitive: value of a_{17} may be -1 or 1

```c
1  #include <stdio.h>
2  int foo(int param) {
3      printf("%d", param);
4      return 0;
5  }
6
7  int main() {
8      int a = 0;
9      int b;
10     scanf("%d", &b);
11     foo(a);
12     if (b > 0) {
13         a = 1;
14     } else {
15         a = -1;
16     }
17     foo(a);
18     return 0;
19 }
```

Path Sensitivity

- Path sensitive:
  - *path*$_1$: b > 0; a_{17} = 1, printed = (0, 1)
  - *path*$_2$: b <=0; a_{17} = -1, printed = (0, -1)

- Precise but cost a lot