SOFTWARE REPRESENTATIONS

PROF. BRENDAN SALTAFORMAGGIO

SCHOOL OF ECE

CREATING THE NEXT®

PLEASE CONSIDER THE ENVIRONMENT, AVOID PRINTING SLIDES!

MANY THANKS TO XIANGYU ZHANG FOR HIS CONTRIBUTIONS TO THESE SLIDES

WHY CREATE ABSTRACTIONS OF SOFTWARE?

- Original representations are hard for humans to analyze
  - Source code
    - Programs written in multiple languages
    - Millions of lines
    - External dependencies
  - Binaries
    - Across machines and platforms
    - Lack of semantic info. --- No symbols
  - Source code + binaries + test cases
    - Who thinks GDB is fun? I do… ☺

- These are even harder for a machine to analyze!
  - And wouldn't it be nice to make a machine do the reverse engineering for us??

SOFTWARE REPRESENTATIONS

- Software is translated into certain representations before analyses are applied

- Outline:
  - Basic blocks
  - Control flow graphs
  - Data flow graphs
  - Program dependence graphs
  - Super control flow graphs
  - Call graph

CREATING THE NEXT®

PROGRAM REPRESENTATION: BASIC BLOCKS

- A basic block is a sequence of consecutive statements with a single entry and a single exit
- Each block has a unique entry point and exit point

- Control always enters a basic block at its entry point and exits from its exit point
- There is no possibility of exit or halt at any point inside the basic block

- The entry and exit points of a basic block may coincide when the block contains only a single statement

BASIC BLOCKS: SOURCE CODE EXAMPLE

```c
float pow(int x, int y)
{
      int power;
      float z;
      if (y < 0)
            power = -y;
      else
            power = y;
      z = 1.0;
      while(power != 0) {
            z = z * x;
            power--;
      }
      if (y < 0)
            z = 1/z;
      return z;
}
```

- Basic blocks are a valid abstraction for software analysis at any level!
  - Both source code and binary analysis

CREATING THE NEXT®

BASIC BLOCKS: SOURCE CODE EXAMPLE

```
1.   float pow(int x, int y)
2.   {
3.         int power;
4.         float z;
5.         if (y < 0)
6.               power = -y;
7.         else
8.               power = y;
9.         z = 1.0;
10.        while(power != 0) {
11.              z = z * x;
12.              power--;
13.        }
14.        if (y < 0)
15.              z = 1/z;
16.        return z;
17.  }
```

- Basic blocks are a valid abstraction for software analysis at any level!
  - Both source code and binary analysis

Block	Lines	Entry point	Exit point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

BASIC BLOCKS: BINARY EXAMPLE

Compiled on 64-bit Linux

```
.text:0000000000000000 pow             proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal   = dword ptr -1Ch
.text:0000000000000000 var_Y           = dword ptr -18h
.text:0000000000000000 var_X           = dword ptr -14h
.text:0000000000000000 var_Z           = dword ptr -8
.text:0000000000000000 var_Power       = dword ptr -4
.text:0000000000000000
.text:0000000000000000                 push    rbp
.text:0000000000000001                 mov     rbp, rsp
.text:0000000000000004                 mov     [rbp+var_X], edi
.text:0000000000000007                 mov     [rbp+var_Y], esi
.text:000000000000000A                 cmp     [rbp+var_Y], 0
.text:000000000000000E                 jns     short loc_1A    ; if ( y < 0 )
.text:0000000000000010                 mov     eax, [rbp+var_Y]
.text:0000000000000013                 neg     eax             ; power = -y
.text:0000000000000015                 mov     [rbp+var_Power], eax
.text:0000000000000018                 jmp     short loc_20
.text:000000000000001A ; ---------------------------------------------------------------
.text:000000000000001A
.text:000000000000001A loc_1A:
.text:000000000000001A                 mov     eax, [rbp+var_Y] ; else power = y
.text:000000000000001D                 mov     [rbp+var_Power], eax
.text:0000000000000020
.text:0000000000000020 loc_20:
.text:0000000000000020                 mov     eax, cs:const_float_1_0
.text:0000000000000026                 mov     [rbp+var_Z], eax ; z = 1.0
.text:0000000000000029                 jmp     short loc_42
```

Compiled on 64-bit Linux

BASIC BLOCKS: BINARY EXAMPLE

```
.text:000000000000002B loc_2B:
.text:000000000000002B                 cvtsi2ss xmm0, [rbp+var_X]
.text:0000000000000030                 movss    xmm1, [rbp+var_Z]
.text:0000000000000035                 mulss    xmm0, xmm1       ; z = z * x
.text:0000000000000039                 movss    [rbp+var_Z], xmm0
.text:000000000000003E                 sub      [rbp+var_Power], 1 ; power = power - 1;
.text:0000000000000042
.text:0000000000000042 loc_42:
.text:0000000000000042                 cmp      [rbp+var_Power], 0 ; while ( power != 0 )
.text:0000000000000046                 jnz      short loc_2B
.text:0000000000000048                 cmp      [rbp+var_Y], 0  ; if ( y < 0 )
.text:000000000000004C                 jns      short loc_60
.text:000000000000004E                 movss    xmm0, cs:const_float_1_0
.text:0000000000000056                 divss    xmm0, [rbp+var_Z] ; z = 1 / z;
.text:000000000000005B                 movss    [rbp+var_Z], xmm0
.text:0000000000000060
.text:0000000000000060 loc_60:
.text:0000000000000060                 mov      eax, [rbp+var_Z]
.text:0000000000000063                 mov      [rbp+var_ReturnVal], eax ; return value = z
.text:0000000000000066                 movss    xmm0, [rbp+var_ReturnVal]
.text:000000000000006B                 pop      rbp
.text:000000000000006C                 retn
.text:000000000000006C pow             endp
```

Prev. Slide Jumps Here

CREATING THE NEXT®

BASIC BLOCKS: BINARY EXAMPLE

```
.text:0000000000000000 pow             proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal   = dword ptr -1Ch
.text:0000000000000000 var_Y           = dword ptr -18h
.text:0000000000000000 var_X           = dword ptr -14h
.text:0000000000000000 var_Z           = dword ptr -8
.text:0000000000000000 var_Power       = dword ptr -4
.text:0000000000000000
.text:0000000000000000                 push    rbp
.text:0000000000000001                 mov     rbp, rsp
.text:0000000000000004                 mov     [rbp+var_X], edi
.text:0000000000000007       Block 1   mov     [rbp+var_Y], esi
.text:000000000000000A                 cmp     [rbp+var_Y], 0
.text:000000000000000E                 jns     short loc_1A    ; if ( y < 0 )
.text:0000000000000010                 mov     eax, [rbp+var_Y]
.text:0000000000000013       Block 2   neg     eax             ; power = -y
.text:0000000000000015                 mov     [rbp+var_Power], eax
.text:0000000000000018                 jmp     short loc_20
.text:000000000000001A ; ----------------------------------------------------------------
.text:000000000000001A
.text:000000000000001A loc_1A:
.text:000000000000001A       Block 3   mov     eax, [rbp+var_Y] ; else power = y
.text:000000000000001D                 mov     [rbp+var_Power], eax
.text:0000000000000020
.text:0000000000000020 loc_20:
.text:0000000000000020                 mov     eax, cs:const_float_1_0
.text:0000000000000026       Block 4   mov     [rbp+var_Z], eax ; z = 1.0
.text:0000000000000029                 jmp     short loc_42
```

BASIC BLOCKS: BINARY EXAMPLE

```
.text:000000000000002B loc_2B:
.text:000000000000002B             cvtsi2ss xmm0, [rbp+var_X]
.text:0000000000000030             movss    xmm1, [rbp+var_Z]
.text:0000000000000035   Block 5   mulss    xmm0, xmm1       ; z = z * x
.text:0000000000000039             movss    [rbp+var_Z], xmm0
.text:000000000000003E             sub      [rbp+var_Power], 1 ; power = power - 1;
.text:0000000000000042
.text:0000000000000042 loc_42:
.text:0000000000000042   Block 6   cmp      [rbp+var_Power], 0 ; while ( power != 0 )
.text:0000000000000046             jnz      short loc_2B
.text:0000000000000048   Block 7   cmp      [rbp+var_Y], 0  ; if ( y < 0 )
.text:000000000000004C             jns      short loc_60
.text:000000000000004E             movss    xmm0, cs:const_float_1_0
.text:0000000000000056   Block 8   divss    xmm0, [rbp+var_Z] ; z = 1 / z;
.text:000000000000005B             movss    [rbp+var_Z], xmm0
.text:0000000000000060
.text:0000000000000060 loc_60:
.text:0000000000000060             mov      eax, [rbp+var_Z]
.text:0000000000000063             mov      [rbp+var_ReturnVal], eax ; return value = z
.text:0000000000000066   Block 9   movss    xmm0, [rbp+var_ReturnVal]
.text:000000000000006B             pop      rbp
.text:000000000000006C             retn
.text:000000000000006C pow         endp
```

CREATING THE NEXT®

SOURCE BLOCKS != BINARY BLOCKS

- Basic blocks may be a valid abstraction for software analysis at any level
- But they are not comparable across levels!
- As we have seen, compilation will significantly rearrange the logic of a program
  - Consider: Our example has no optimization!

```
   float pow(int x, int y)
   {
1      int power;
       float z;
       if (y < 0)
2          power = -y;
       else
3          power = y;
4      z = 1.0;
5      while(power != 0) {
6          z = z * x;
           power--;
       }
7      if (y < 0)
8          z = 1/z;
9      return z;
   }
```

```
loc_2B:
Block 5   cvtsi2ss xmm0, [rbp+var_X]
          movss    xmm1, [rbp+var_Z]
          mulss    xmm0, xmm1         ; z = z * x
          movss    [rbp+var_Z], xmm0
          sub      [rbp+var_Power], 1 ; power = power - 1;

loc_42:
Block 6   cmp      [rbp+var_Power], 0 ; while ( power != 0 )
          jnz      short loc_2B
Block 7   cmp      [rbp+var_Y], 0   ; if ( y < 0 )
          jns      short loc_60
Block 8   movss    xmm0, cs:const_float_1_0
          divss    xmm0, [rbp+var_Z] ; z = 1 / z;
          movss    [rbp+var_Z], xmm0

loc_60:
Block 9   mov      eax, [rbp+var_Z]
          mov      [rbp+var_ReturnVal], eax ; return value = z
          movss    xmm0, [rbp+var_ReturnVal]
          pop      rbp
          retn
pow       endp
```

CONTROL FLOW GRAPH (CFG)

- The most commonly used program representation
- A CFG abstracts the paths that might be traversed through a program during its execution guided solely by branching constructs
- A control flow graph (sometimes called flow graph) G is defined as a finite set N of nodes and a finite set E of edges
- An edge (i, j) in E connects two nodes n_i and n_j in N
- We often write G=(N, E) to denote a control flow graph G with nodes given by N and edges by E

CREATING THE NEXT®

- In a control flow graph of a program, each basic block becomes a node
- Edges are used to indicate the flow of execution (i.e., control) between blocks
- A CFG edge (i, j) connecting basic blocks b_i and b_j implies that control can be transferred from block b_i to block b_j
- We assume that there exists a node labeled Start in N that has **no** incoming edge
  - The Start node is assigned outgoing edges to all other nodes which have no incoming edge
- We also assume that there exists a node labeled End in N that has no outgoing edge
  - The End node is assigned incoming edges from all other nodes which have no outgoing edge
- The Start and End nodes are important to simplify automated analyses

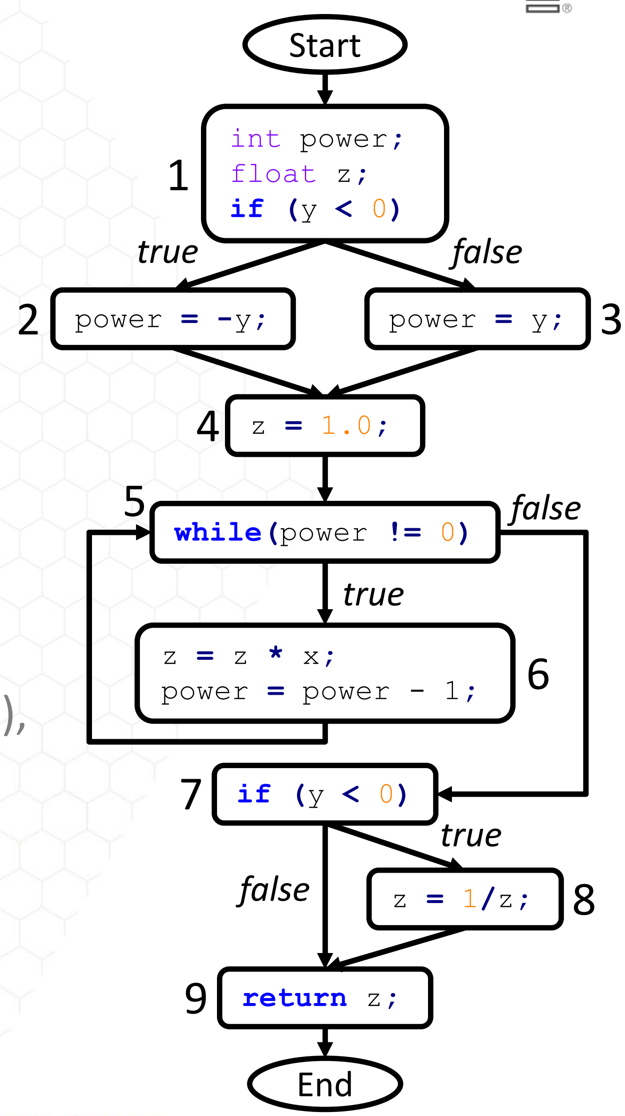
CFG EXAMPLE

```
    float pow(int x, int y)
    {
1     int power;
      float z;
      if (y < 0)
2         power = -y;
      else
3         power = y;
4     z = 1.0;
5     while(power != 0) {
6         z = z * x;
          power = power - 1;
      }
7     if (y < 0)
8         z = 1/z;
9     return z;
    }
```

CFG(pow) = (N,E)

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (8, 9), (9, End)}



CFG EXAMPLE

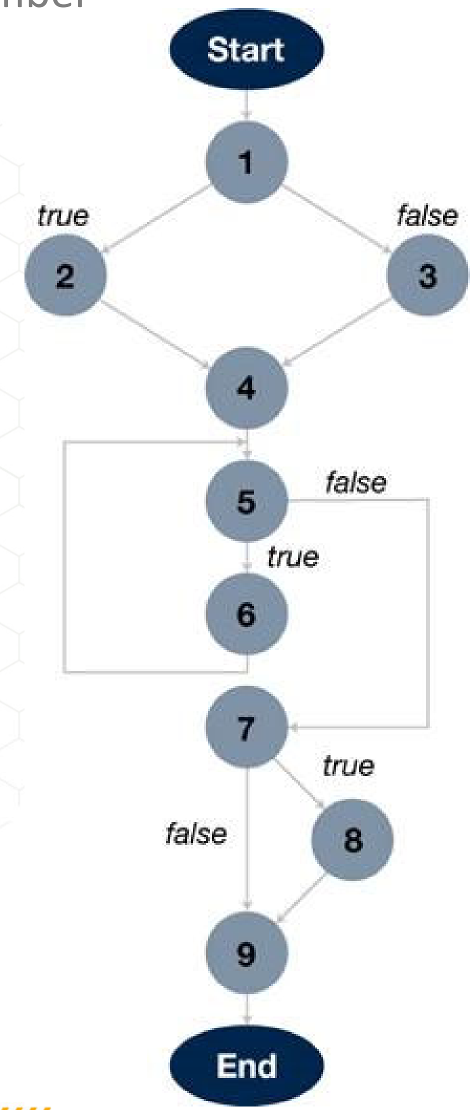
- CFG nodes are typically represented by only their basic block number

```
    float pow(int x, int y)
    {
 1    int power;
      float z;
      if (y < 0)
 2        power = -y;
      else
 3        power = y;
 4    z = 1.0;
 5    while(power != 0) {
 6        z = z * x;
          power = power - 1;
      }
 7    if (y < 0)
 8        z = 1/z;
 9    return z;
    }
```

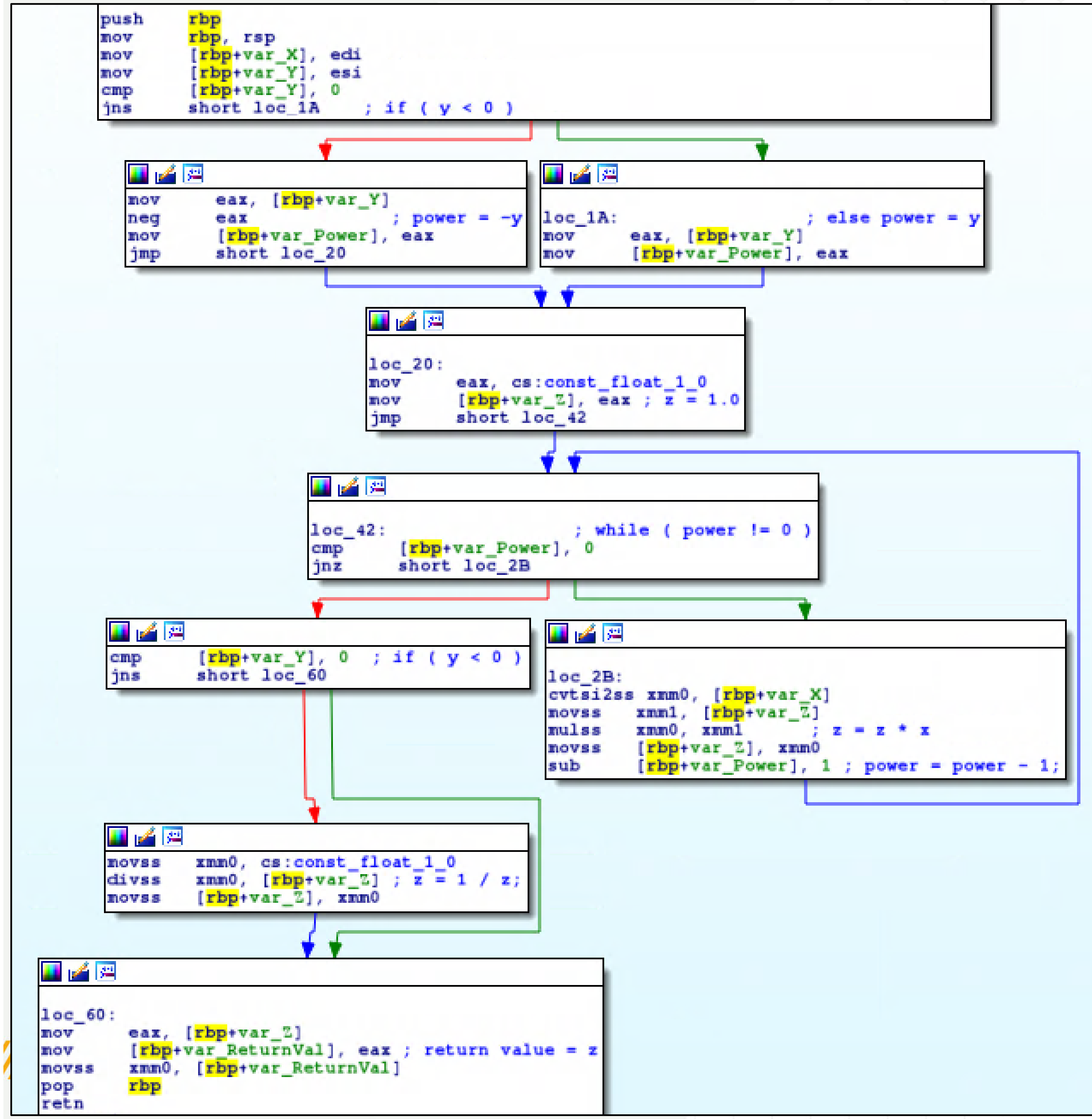
CFG(pow) = (N,E)

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (8, 9), (9, End)}



WHERE HAVE I SEEN THIS BEFORE?



- IDA's Graph View displays a **CFG**
- IDA detects basic blocks based on control transfer instructions and its (very limited) knowledge of the control transfer targets
- IDA's basic blocks will often be wrong if the control transfer target is aliased or dynamically computed

PATHS

- A CFG represents all paths (that we know of) which **might** be traversed during execution
- To reason about **actual** executions we need to define the notion of a Path

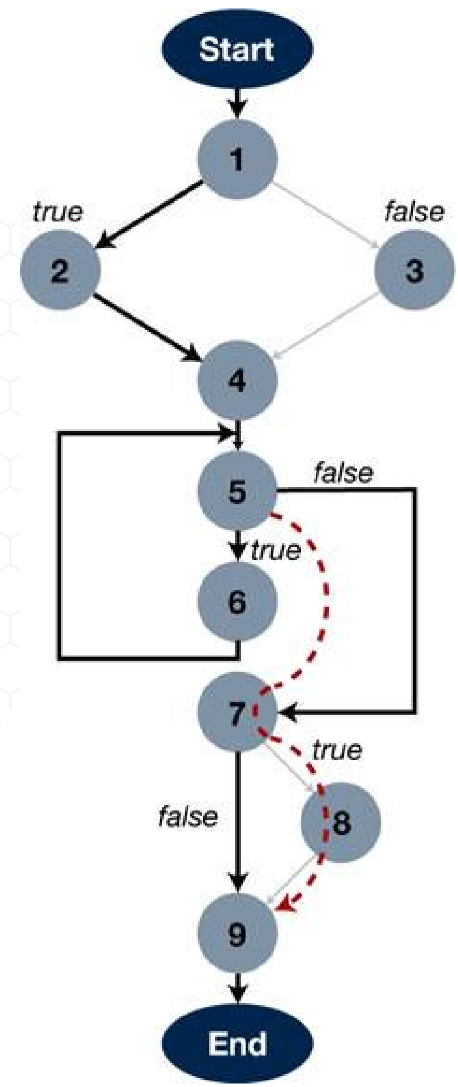
- Consider a control flow graph G= (N, E)
- A Path P consists of k edges from E, where k>0
  - That is: P = (e_1, e_2, … e_k)
- P denotes a path of length k through the control flow graph **if** the following sequence condition on the sequence of edges holds true
- Given that n_p, n_q, n_r, and n_s are nodes belonging to N and 0<i<k
- If $e_i = (n_p , n_q)$ and $e_{i+1} = (n_r , n_s)$ then n_q must be n_r

- Put simply: Every node in a Path must be reachable by a single traversal from the Path's first node to its last

CREATING THE NEXT®

COMPLETE PATHS VS. SUBPATHS

- Our definition of a Path allows for two types of valid Paths:
  - Complete Path: A valid Path (by our previous definition) which includes both the Start and End nodes from the CFG
  - Subpath: A valid Path (by our previous definition) which forms a subsequence of a Complete Path
- In the figure:
  - The set of bold edges forms a Complete Path:
  - P1 = (Start, 1, 2, 4, 5, 6, 5, 7, 9, End)
  - Specified unambiguously using edges:
  - P1= ((Start, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, End))
  - The set of dashed edges forms a Subpath:
  - P2 = (5, 7, 8, 9)
  - NOT a valid Path:
  - P0 = (Start, 1, **2, 3,** 4, 5, 6, 5, 7, 9, End)



CREATING THE NEXT®

FEASIBLE PATHS VS. INFEASIBLE PATHS

- One of the most important Path analyses is Path Feasibility
  - Path Feasibility is used in security (e.g., can the malware execute that payload?)
  - Software engineering (e.g., how can we optimize the sequence of these program components?)
  - Debugging (e.g., given our current state, which branch will the program take next?)
  - Compilers (e.g., is this dead code able to be removed?)
- A path P through a CFG is considered **feasible** if there exists at least one test case which when input to the program causes **every** node in P to be traversed
- Note that by this definition, Subpaths can also be considered feasible
- In the face of bugs or exploits, a general solution for Path feasibility is not possible
- Techniques which solve localized versions of Path feasibility do exist (compilers do it)

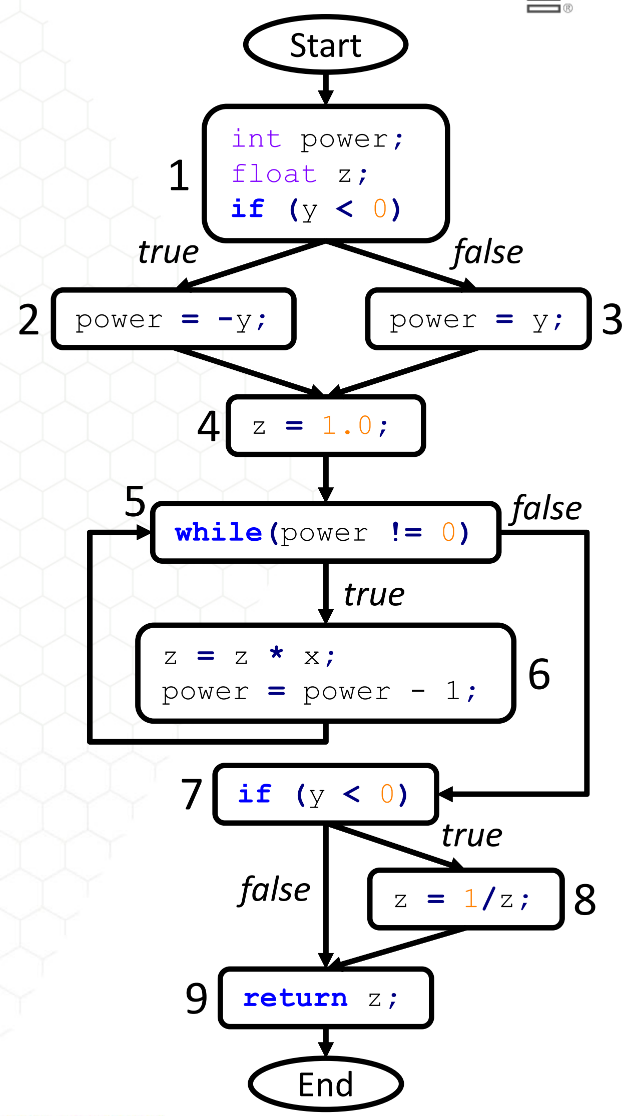
FEASIBLE PATHS VS. INFEASIBLE PATHS

- Two Feasible and Complete Paths:
- P1= (Start, 1, 2, 4, 5, 6, 5, 7, 8, 9, End)
- P2= (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)

- Two Feasible Subpaths:
- P3= (Start, 1, 2, 4)
- P4= (5, 7, 8, 9, End)

- Two Infeasible Paths:
- P1= (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)
- P2= (Start, 1, 2, 4, 5, 7, 9, End)

- Notice that Paths can be Complete but Infeasible



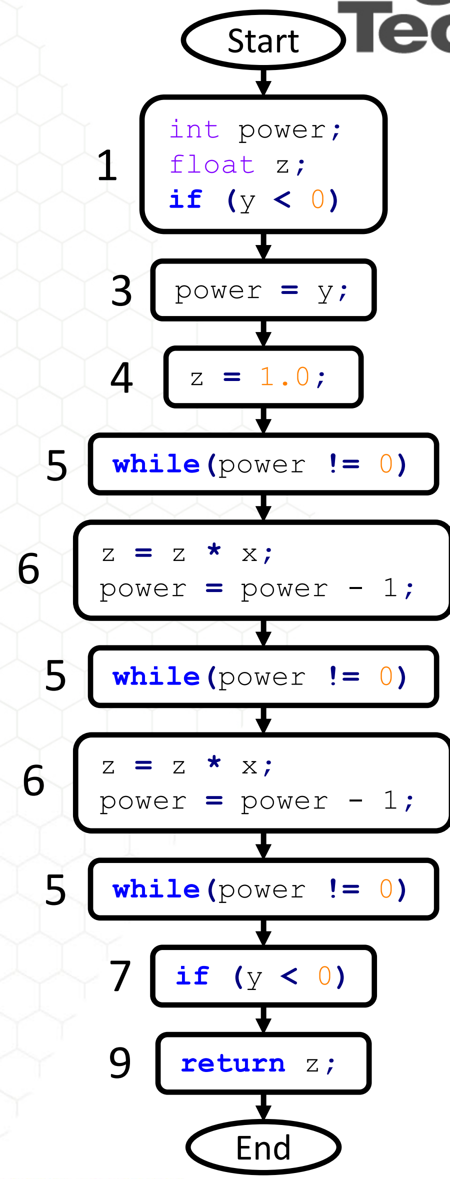
CREATING THE NEXT®

NUMBER OF PATHS

- There can be many distinct paths through a program
- A program with no conditional statements contains exactly one path
  - It begins at the Start node, traverses every node, and terminates at the End node
- Every additional condition in the program can increase the number of distinct paths by **at least** one
- Depending on their location in the CFG, conditional statements can have a **multiplicative** effect on the number of paths
- This leads to a problem that nearly ALL static analysis techniques suffer from: Path Explosion!
- Research tools are always struggling to scale to real world programs because exploring all their paths becomes impossible!

REASONING ALONG PATHS

- Just like basic blocks make analysis easier by giving structure to sequences of statements…
- Many problems which are globally intractable (i.e., cannot be solved for entire programs) can be solved locally (i.e., on a single Path)
- This is because a single Path allows for direct inference of execution behaviors
- Example: What was the value of Y that produced this path?
- Now how can we teach an algorithm to figure that out??



CREATING THE NEXT®

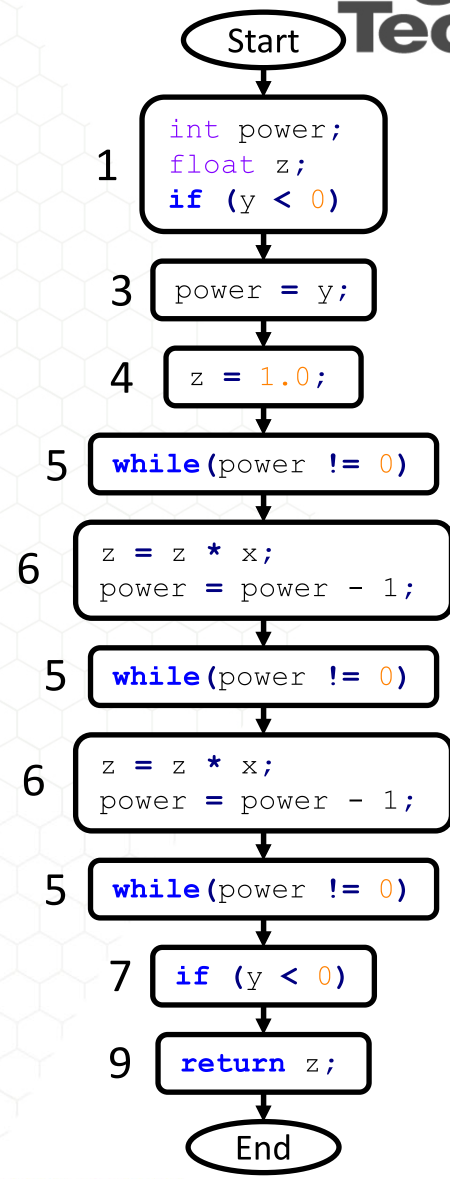
DEPENDENCY ANALYSIS

MAKE ALGORITHMS THINK LIKE WE DO

CREATING THE NEXT®

DEPENDENCY ANALYSIS

- We can look at a path and observe dependencies
  - “The value of power depends on the value of y”
  - “The execution of block 3 depends on the execution of block 1”
  - “The loop iteration depends on the value of power”
- These dependencies can be modeled so that an algorithm can analyze them
- Control Dependencies
  - Dominator
  - Post-dominator
  - Immediate Dominator/Post-dominator
- Data Dependencies



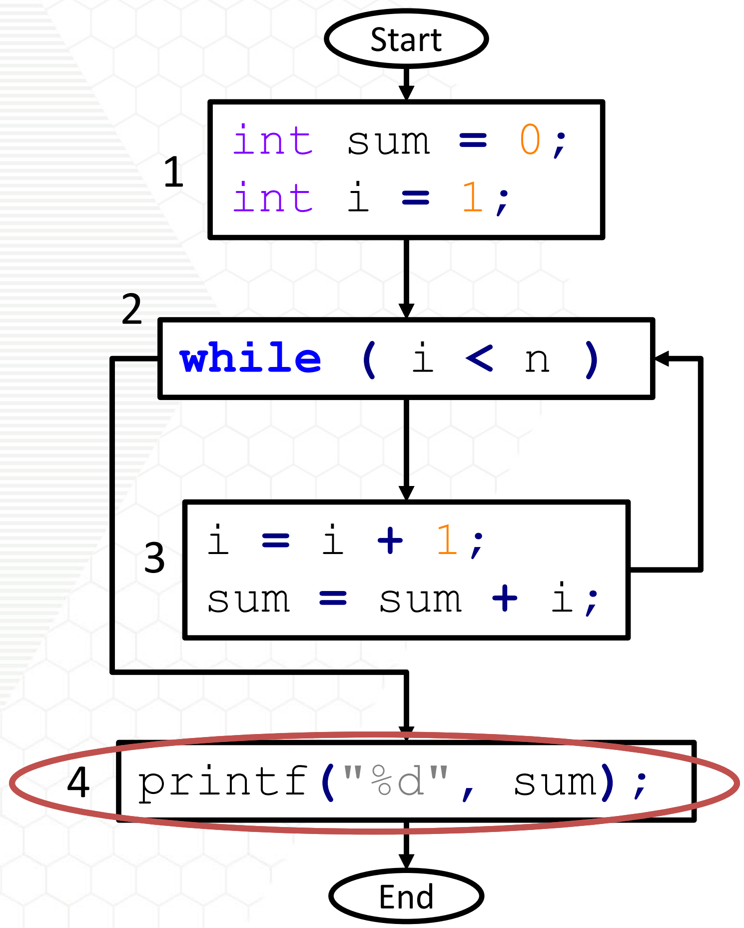
CREATING THE NEXT®

DOMINATOR

- X **dominates** Y if **all** possible program paths from START to Y have to pass through X

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```

DOM(4) = {1, 2, 4}



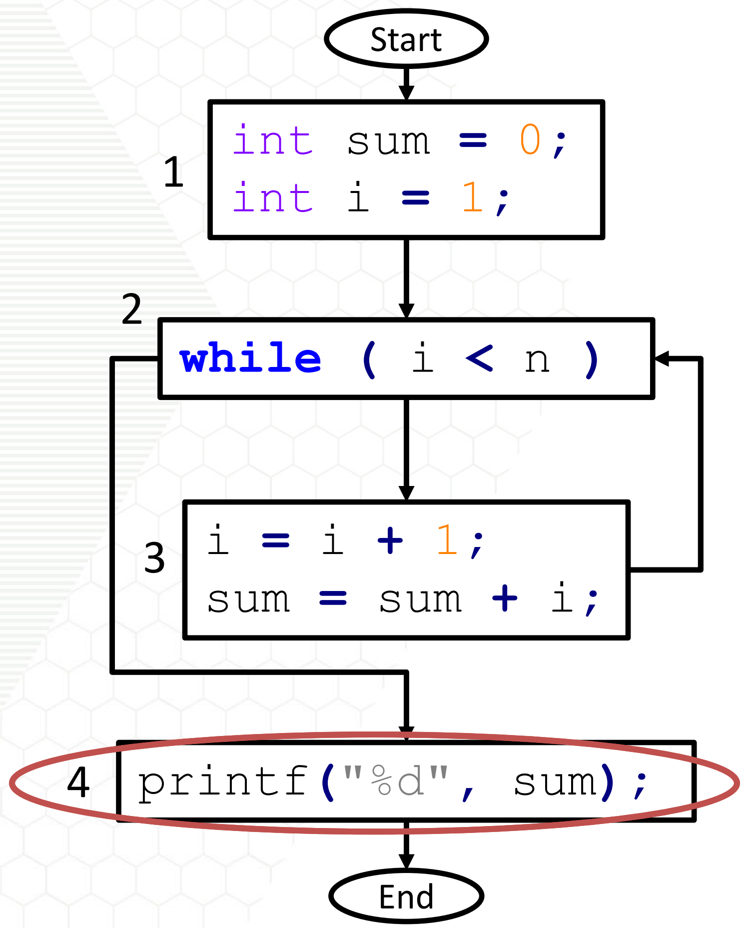
CREATING THE NEXT®

STRICT DOMINATOR

- X strictly dominates Y if X dominates Y and X!=Y

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```

SDOM(4) = {1, 2}



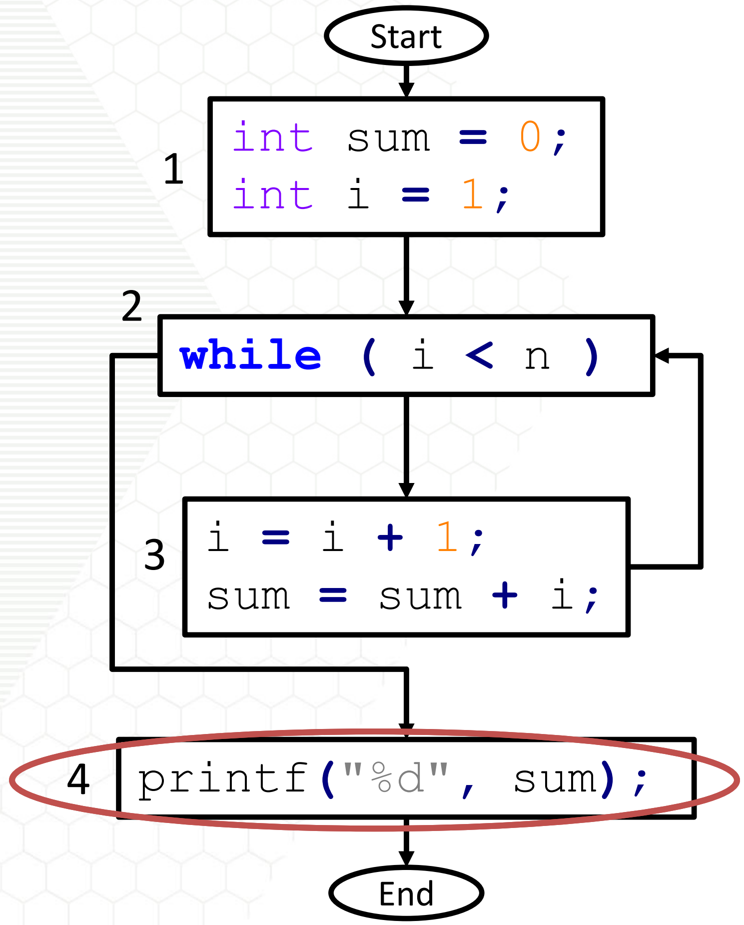
CREATING THE NEXT®

IMMEDIATE DOMINATOR

- X is the immediate dominator of Y if X is the **last dominator** of Y along a path from Start to Y

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```

IDOM(4) = {2}



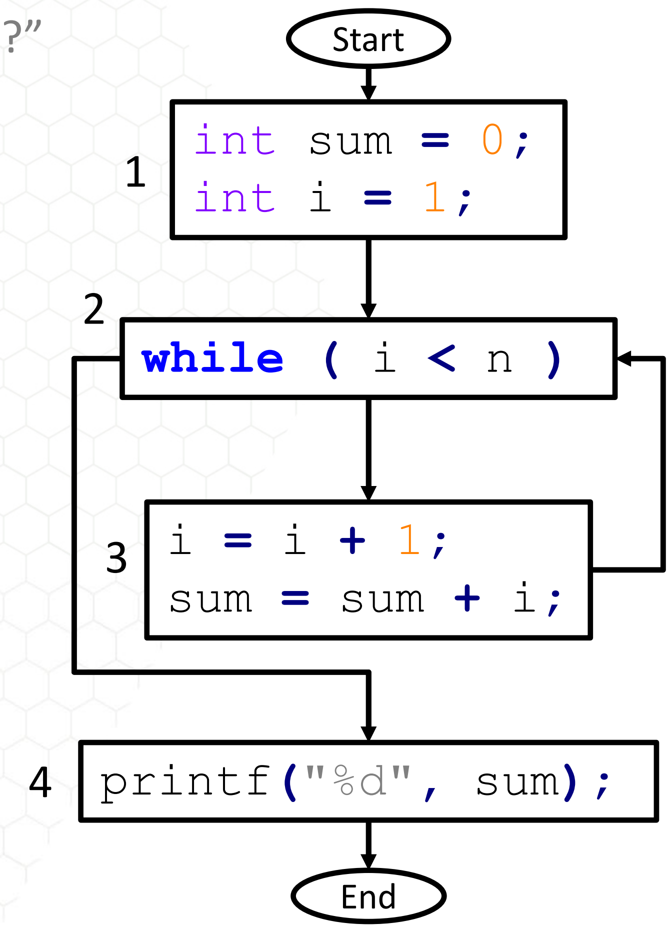
CREATING THE NEXT®

DOMINATORS ALLOW FOR **BACKWARD** REASONING

- Dominators allow algorithms to determine backward control flow
- Put simply: “Who needs to execute for block X to execute?”

- DOM(START) = {}
- DOM(1) = {1}
- DOM(2) = {1, 2}
- DOM(3) = {1, 2, 3}
- DOM(4) = {1, 2, 4}
- DOM(END) = {1, 2, 4}
- Notice that Start and End are not true nodes!

- Notice that DOM(END) = Blocks executed for ANY input

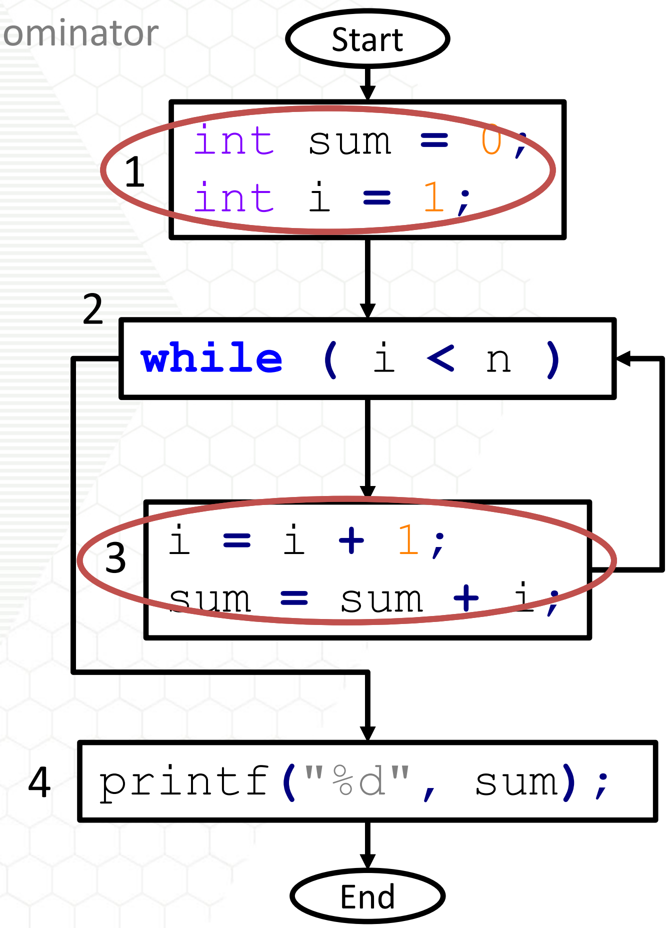


CREATING THE NEXT®

POST-DOMINATOR

- X post-dominates Y if **all** possible program path from Y to End has to pass through X
  - Similar strict post-dominator & immediate post-dominator

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```



PDOM(3) = {2, 3, 4}
SPDOM(3) = {2, 4}
IPDOM(3) = {2}

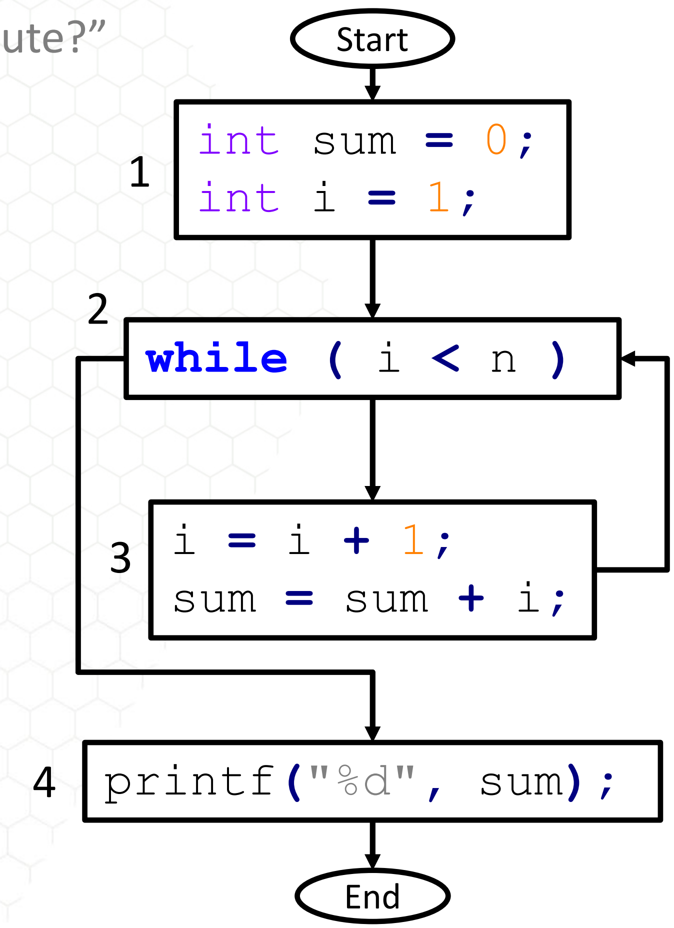
CREATING THE NEXT®

POST-DOMINATORS ALLOW FOR **FORWARD** REASONING

- Post-dominators allow algorithms to determine forward control flow
- Put simply: “If block X executes, then who else must execute?”

- PDOM(START) = {1, 2, 4}
- PDOM(1) = {1, 2, 4}
- PDOM(2) = {2, 4}
- PDOM(3) = {2, 3, 4}
- PDOM(4) = {4}
- PDOM(END) = {}

- Notice that PDOM(START) = DOM(END). Why??



CREATING THE NEXT®

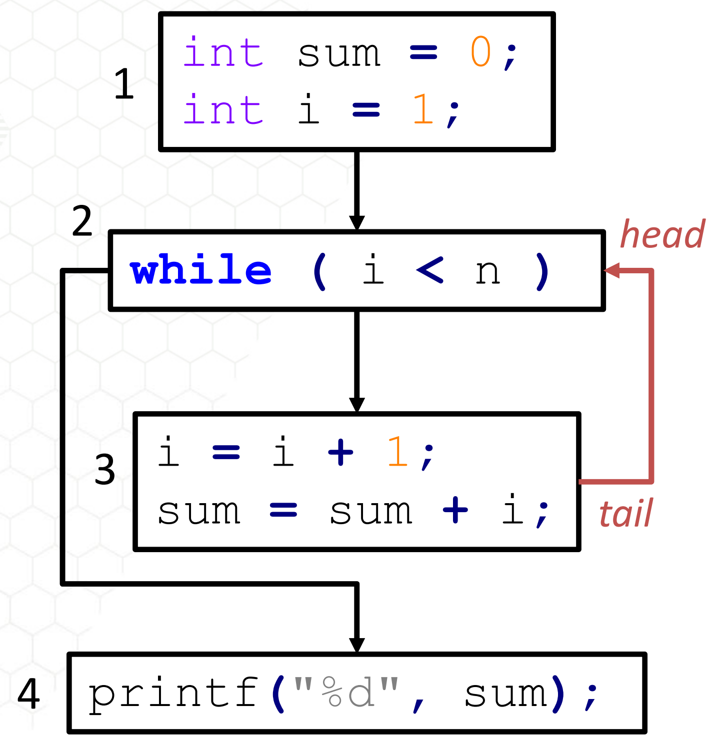
BACK EDGES

- Dominators/Post-dominators allow us to define characteristics of the CFG
- The most common: A back edge is an edge whose head dominates its tail
- A “closed loop back edge” is an edge whose head dominates AND post-dominates its tail
- What would be different if block 3 looked like this?

3
```
i = i + 1;
sum = sum + i;
if(sum > 100)
    break;
```

DOM(2) = {1, 2}
DOM(3) = {1, 2, 3}

PDOM(2) = {2, 4}
PDOM(3) = {2, 3, 4}



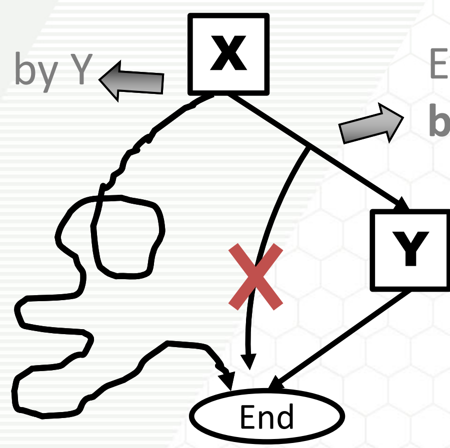
- Most importantly, Dominators & Post-dominators allow us to define **Control Dependence**
- Y is control dependent on X **iff** X directly determines whether Y executes
  - In general, statements inside each branch of a predicate are control dependent on the predicate
- Both criteria must hold:

A path from X to End exists that does not pass Y or X==Y

& No such path exists for nodes in the path between X and Y

1) X is not strictly post-dominated by Y
2) There exists a path from X to Y such that every node on that path other than X is post-dominated by Y

X is **not** strictly post-dominated by Y



Every node on this path **must be** post-dominated by Y

Then Y is control-dependent on X

CONTROL DEPENDENCE - EXAMPLE

- Y is control dependent on X **iff** X directly determines whether Y executes
  1) X is not strictly post-dominated by Y
  2) There exists a path from X to Y such that every node on that path other than X is post-dominated by Y

Tricky!! CD(2) = {2}

2->3->2->4->End

X = 2, Y = 2 (2nd iteration)
SPDOM(2) = {4}

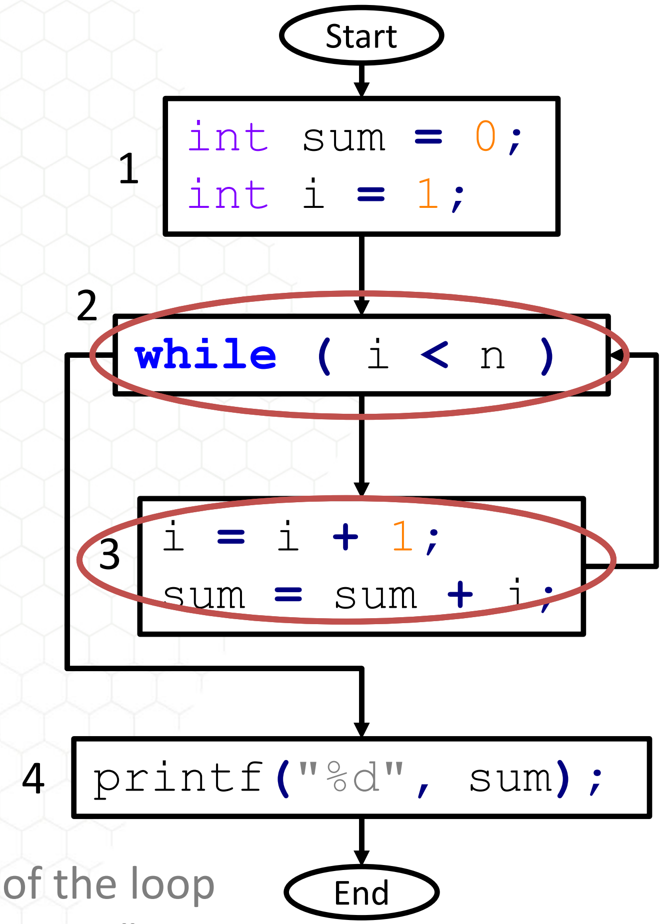
2 ∉ SPDOM(2) &
2->2 = {3, 2}, and
2 ∈ PDOM(3)
2 ∈ PDOM(2)

X = 2, Y = 3
3 ∉ SPDOM(2) &
2->3 = {3}, 3 ∈ PDOM(3)

CD(3) = {2}

Why not CD(3) = {1, 2}?

X = 1, Y = 3
3 ∉ SPDOM(1) &
1->3 = {2,3}, but
3 ∉ PDOM(2)
3 ∈ PDOM(3)

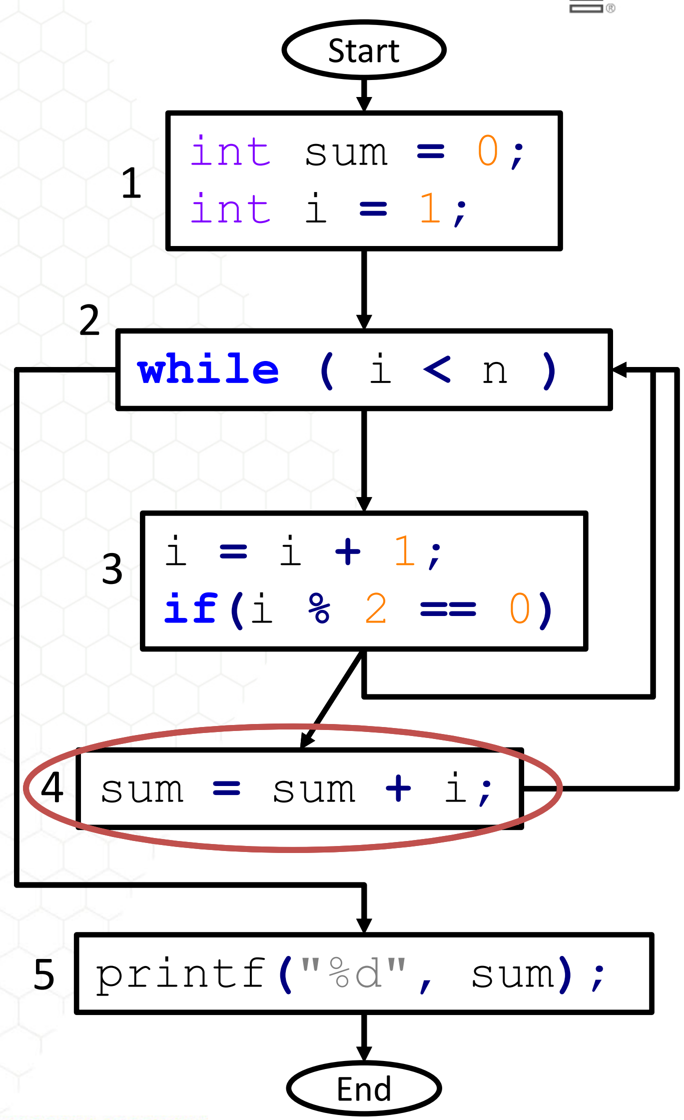


- May seem confusing, but this is really just the "unrolling" of the loop
- In fact, "loop unrolling" is the concrete term for "loop induction"
- Algorithm Analysis in Binary Analysis … Mind = Blown

CONTROL DEPENDENCE IS **NOT** SYNTACTICALLY EXPLICIT

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        if(i % 2 == 0)
            continue;
        sum = sum + i;
    }
    printf("%d", sum);
}
```

CD(4) = ?



CONTROL DEPENDENCE IS **NOT** SYNTACTICALLY EXPLICIT

- Y is control dependent on X **iff** X directly determines whether Y executes
  1) X is not strictly post-dominated by Y
  2) There exists a path from X to Y such that every node on that path other than X is post-dominated by Y

X = 3, Y = 4
SPDOM(3) = {2, 5}

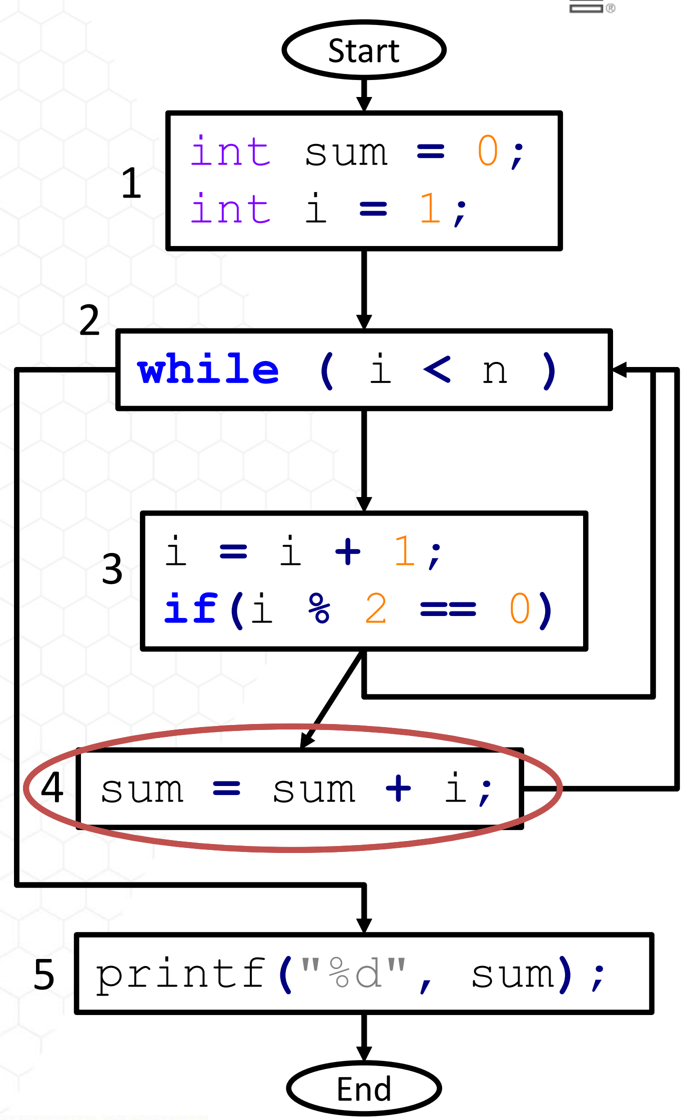
4 ∉ SPDOM(3) &
3->4 = {4}, 4 ∈ PDOM(4)

So CD(4) = {3}

X = 2, Y = 4
SPDOM(2) = {5}

4 ∉ SPDOM(2) &
2->4 = {3, 4}, but
4 ∉ PDOM(3)
4 ∈ PDOM(4)

So 4 is **not** control dependent on 2!

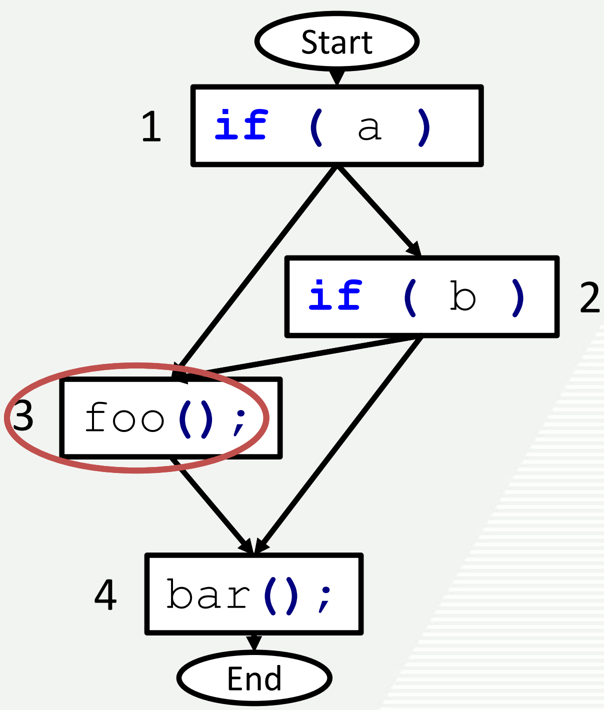


CREATING THE NEXT®

CONTROL DEPENDENCE IS VERY TRICKY!

- Y is control dependent on X **iff** X directly determines whether Y executes
  1) X is not strictly post-dominated by Y
  2) There exists a path from X to Y such that every node on that path other than X is post-dominated by Y

- Can one statement be control dependent on two predicates?




```
if ( a || b )
    foo();
bar();
```

```
    cmp rax, 0
    jne .L2
    cmp rbx, 0
    je  .L3
.L2:
    call    foo
.L3:
    call    bar
```

You didn't think we were finished with assembly did you??

X = 2, Y = 3
SPDOM(2) = {4}

3 ∉ SPDOM(2) &
2->3 = {3}, 3 ∈ PDOM(3)

So CD(3) = {2} ... but wait

X = 1, Y = 3
SPDOM(1) = {4}

3 ∉ SPDOM(1) &
1->3 = {3}, 3 ∈ PDOM(3)

So CD(3) = {1, 2}

Georgia Tech

DATA DEPENDENCE

WHERE DID ALL MY DATA GO?

CREATING THE NEXT®

DATA DEPENDENCE

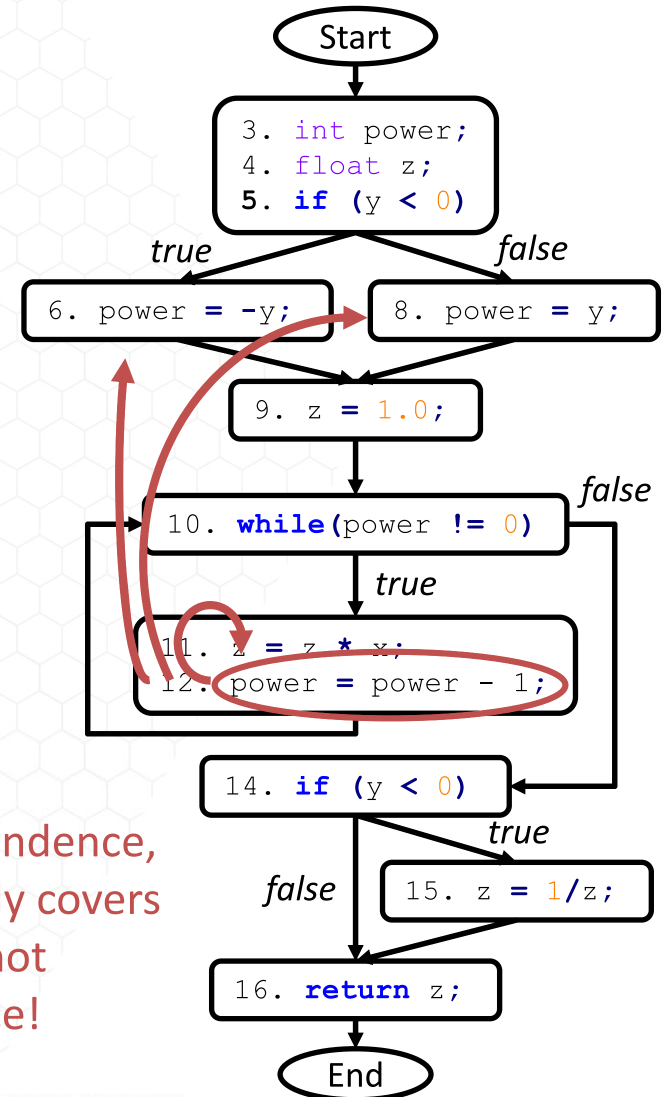
- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined
- Data dependence is calculated **per statement**
  - Rarely will results be aggregated per basic block

```
1.   float pow(int x, int y)
2.   {
3.       int power;
4.       float z;
5.       if (y < 0)
6.           power = -y;
7.       else
8.           power = y;
9.       z = 1.0;
10.      while(power != 0) {
11.          z = z * x;
12.          power--;
13.      }
14.      if (y < 0)
15.          z = 1/z;
16.      return z;
17.  }
```

DD(12) = {6, 8, 12}

But notice, NOT on the variable y! Why?

Just like control dependence, data dependence only covers direct dependence, not transitive dependence!



CREATING THE NEXT®

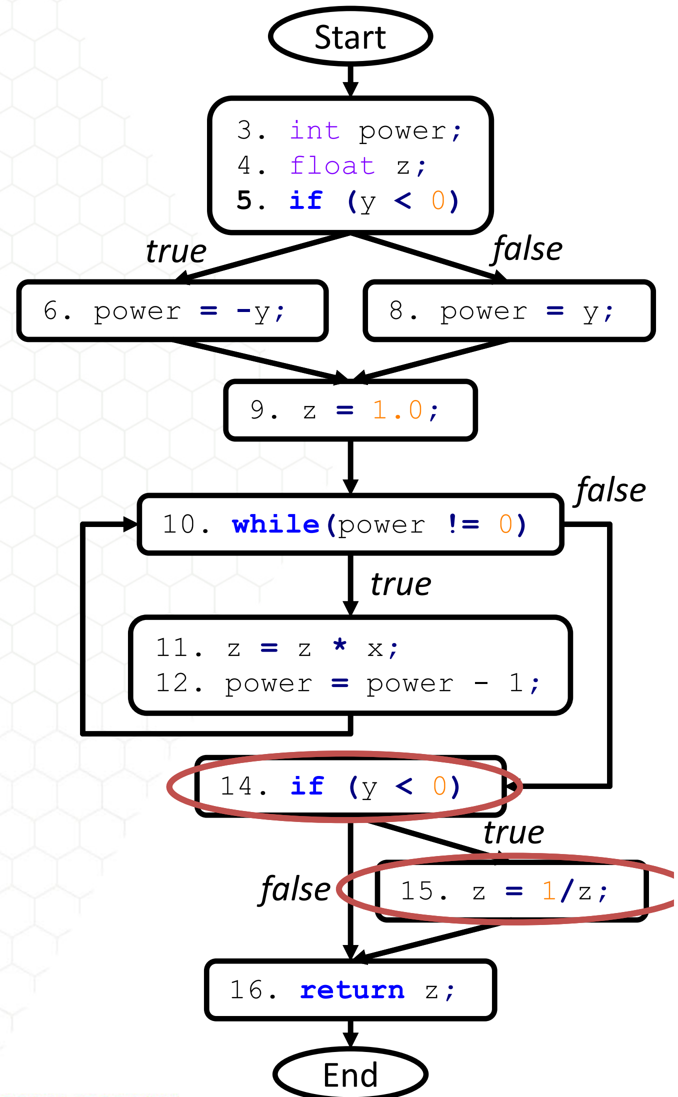
DATA DEPENDENCE

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined
- Data dependence is calculated **per statement**
  - Rarely will results be aggregated per basic block

```
1.   float pow(int x, int y)
2.   {
3.        int power;
4.        float z;
5.        if (y < 0)
6.             power = -y;
7.        else
8.             power = y;
9.        z = 1.0;
10.       while(power != 0) {
11.            z = z * x;
12.            power--;
13.       }
14.       if (y < 0)
15.            z = 1/z;
16.       return z;
17.  }
```

DD(14) = {Arg2}

DD(15) = {9, 11}



DATA DEPENDENCE ON BINARIES

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
.text:0000000000000000 pow              proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal    = dword ptr -1Ch
.text:0000000000000000 var_Y            = dword ptr -18h
.text:0000000000000000 var_X            = dword ptr -14h
.text:0000000000000000 var_Z            = dword ptr -8
.text:0000000000000000 var_Power        = dword ptr -4
.text:0000000000000000
.text:0000000000000000                  push    rbp
.text:0000000000000001                  mov     rbp, rsp
.text:0000000000000004                  mov     [rbp+var_X], edi
.text:0000000000000007                  mov     [rbp+var_Y], esi
.text:000000000000000A                  cmp     [rbp+var_Y], 0
.text:000000000000000E                  jns     short loc_1A    ; if ( y < 0 )
.text:0000000000000010                  mov     eax, [rbp+var_Y]
.text:0000000000000013                  neg     eax             ; power = -y
.text:0000000000000015                  mov     [rbp+var_Power], eax
.text:0000000000000018                  jmp     short loc_20
.text:000000000000001A ; ------------------------------------------------------
```

reg. read

Remember: NOT transitive dependence!

DD(.text:000013) = {.text:000010}

CREATING THE NEXT®

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
.text:0000000000000000 pow              proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal    = dword ptr -1Ch
.text:0000000000000000 var_Y            = dword ptr -18h
.text:0000000000000000 var_X            = dword ptr -14h
.text:0000000000000000 var_Z            = dword ptr -8
.text:0000000000000000 var_Power        = dword ptr -4
.text:0000000000000000
.text:0000000000000000                  push    rbp
.text:0000000000000001                  mov     rbp, rsp
.text:0000000000000004                  mov     [rbp+var_X], edi
.text:0000000000000007                  mov     [rbp+var_Y], esi
.text:000000000000000A                  cmp     [rbp+var_Y], 0
.text:000000000000000E                  jns     short loc_1A    ; if ( y < 0 )
.text:0000000000000010                  mov     eax, [rbp+var_Y]
.text:0000000000000013                  neg     eax             ; power = -y
.text:0000000000000015                  mov     [rbp+var_Power], eax
.text:0000000000000018                  jmp     short loc_20
.text:000000000000001A ; ------------------------------------------------
```

DD(.text:000010) = {
.text:000007,
.text:000001
}

reg. read

mem. read

?

Georgia Tech

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
.text:0000000000000000 pow              proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal    = dword ptr -1Ch
.text:0000000000000000 var_Y            = dword ptr -18h
.text:0000000000000000 var_X            = dword ptr -14h
.text:0000000000000000 var_Z            = dword ptr -8
.text:0000000000000000 var_Power        = dword ptr -4
.text:0000000000000000
.text:0000000000000000                  push    rbp
.text:0000000000000001                  mov     rbp, rsp
.text:0000000000000004                  mov     [rbp+var_X], edi
.text:0000000000000007                  mov     [rbp+var_Y], esi
.text:000000000000000A                  cmp     [rbp+var_Y], 0
.text:000000000000000E                  jns     short loc_1A      ; if ( y < 0 )
.text:0000000000000010                  mov     eax, [rbp+var_Y]
.text:0000000000000013                  neg     eax               ; power = -y
.text:0000000000000015                  mov     [rbp+var_Power], eax
.text:0000000000000018                  jmp     short loc_20
.text:000000000000001A ; -----------------------------------------------------
```

reg. read (rflags)

Be careful of **implicit** data flows!

DD(.text:00000E) = {.text:00000A}

CREATING THE NEXT®

DATA DEPENDENCE ON BINARIES

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
.text:0000000000000000 pow             proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal   = dword ptr -1Ch
.text:0000000000000000 var_Y           = dword ptr -18h
.text:0000000000000000 var_X           = dword ptr -14h
.text:0000000000000000 var_Z           = dword ptr -8
.text:0000000000000000 var_Power       = dword ptr -4
.text:0000000000000000
.text:0000000000000000                 push    rbp
.text:0000000000000001                 mov     rbp, rsp
.text:0000000000000004                 mov     [rbp+var_X], edi
.text:0000000000000007                 mov     [rbp+var_Y], esi
.text:000000000000000A                 cmp     [rbp+var_Y], 0
.text:000000000000000E                 jns     short loc_1A     ; if ( y < 0 )
.text:0000000000000010                 mov     eax, [rbp+var_Y]
.text:0000000000000013                 neg     eax              ; power = -y
.text:0000000000000015                 mov     [rbp+var_Power], eax
.text:0000000000000018                 jmp     short loc_20
.text:000000000000001A ; ---------------------------------------------------
```

Values which come from outside the function are marked as the START node

DD(.text:000000) = {START}

CREATING THE NEXT®

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
.text:0000000000000000 pow             proc near
.text:0000000000000000
.text:0000000000000000 var_ReturnVal   = dword ptr -1Ch
.text:0000000000000000 var_Y           = dword ptr -18h
.text:0000000000000000 var_X           = dword ptr -14h
.text:0000000000000000 var_Z           = dword ptr -8
.text:0000000000000000 var_Power       = dword ptr -4
.text:0000000000000000
.text:0000000000000000                 push    rbp
.text:0000000000000001                 mov     rbp, rsp
.text:0000000000000004                 mov     [rbp+var_X], edi
.text:0000000000000007                 mov     [rbp+var_Y], esi
.text:000000000000000A                 cmp     [rbp+var_Y], 0
.text:000000000000000E                 jns     short loc_1A     ; if ( y < 0 )
.text:0000000000000010                 mov     eax, [rbp+var_Y]
.text:0000000000000013                 neg     eax              ; power = -y
.text:0000000000000015                 mov     [rbp+var_Power], eax
.text:0000000000000018                 jmp     short loc_20
.text:000000000000001A ; ---------------------------------------------------
```

reg. read (rbp)

DD(.text:000004) = {.text:000001, START}

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

DD(.text:0040577A) = { }

```
.text:0040575F          lea     eax, [ebp+cbData]
.text:00405765          push    eax                 ; lpcbData
.text:00405766          lea     eax, [ebp+Data]
.text:0040576C          push    eax                 ; lpData
.text:0040576D          push    ebx                 ; lpType
.text:0040576E          push    ebx                 ; lpReserved
.text:0040576F          push    offset aCdkey_0 ; "CDKey"
.text:00405774          push    [ebp+phkResult] ; hKey
.text:0040577A  ──────► call    RegQueryValueExA
.text:00405780          test    eax, eax
.text:00405782          jnz     short loc_4057B3
```

DATA DEPENDENCE ON BINARIES

Georgia Tech

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

Don't forget about implicit flows!!
Calls redefine the RAX/EAX register!

DD(.text:00405780) = {.text:0040577A}

```
.text:0040575F                 lea     eax, [ebp+cbData]
.text:00405765                 push    eax             ; lpcbData
.text:00405766                 lea     eax, [ebp+Data]
.text:0040576C                 push    eax             ; lpData
.text:0040576D                 push    ebx             ; lpType
.text:0040576E                 push    ebx             ; lpReserved
.text:0040576F                 push    offset aCdkey_0 ; "CDKey"
.text:00405774                 push    [ebp+phkResult] ; hKey
.text:0040577A                 call    RegQueryValueExA
.text:00405780                 test    eax, eax
.text:00405782                 jnz     short loc_4057B3
```

reg. read (eax)

CREATING THE NEXT®

- X is data dependent on Y **iff**
    1) There is a variable V that is defined at Y and used at X
    2) There exists a path of nonzero length from Y to X along which V is not re-defined

DD(.text:004010BB) = { }

Constant offset value!
Just like "mov eax, 0x1234"

```
.text:004010B8          push    ebp
.text:004010B9          mov     ebp, esp
.text:004010BB  ----->  mov     eax, off_429014
.text:004010C0          push    esi
.text:004010C1          cmp     byte ptr [eax], 0
```

CREATING THE NEXT®

- X is data dependent on Y **iff**
  1) There is a variable V that is defined at Y and used at X
  2) There exists a path of nonzero length from Y to X along which V is not re-defined

DD(.text:004010C1) = {.text:004010BB, ?? }

```
.text:004010B8                push    ebp
.text:004010B9                mov     ebp, esp
.text:004010BB                mov     eax, off_429014
.text:004010C0                push    esi
.text:004010C1                cmp     byte ptr [eax], 0
```

reg. read (eax)

?

Handling global data dependence is implementation specific!
Options:
1) Only track global data globally
2) Track all data globally
3) Note global dependencies at the START node & patch later

CREATING THE NEXT®

DATA DEP. MODELLING WITH DU CHAINS

- DU chains (Def-Use chains) link the definition of a variable and the use of the variable
- Pro: very fast to collect data dependencies, easy to program (table data structure)
- Con: must compute & store every variable, cannot omit unused variables

```
float pow(int x, int y)          // D: x, y
{
    int power;
    float z;
    if (y < 0)                   // U: y
        power = -y;              // D: power  U: y
    else
        power = y;               // D: power  U: y
    z = 1.0;                     // D: z
    while(power != 0) {          // U: power
        z = z * x;               // D: z  U: x, z
        power--;                 // D: power  U: power
    }
    if (y < 0)                   // U: y
        z = 1/z;                 // D: z  U: z
    return z;                    // U: z
}
```

DATA DEP. MODELLING WITH DU CHAINS

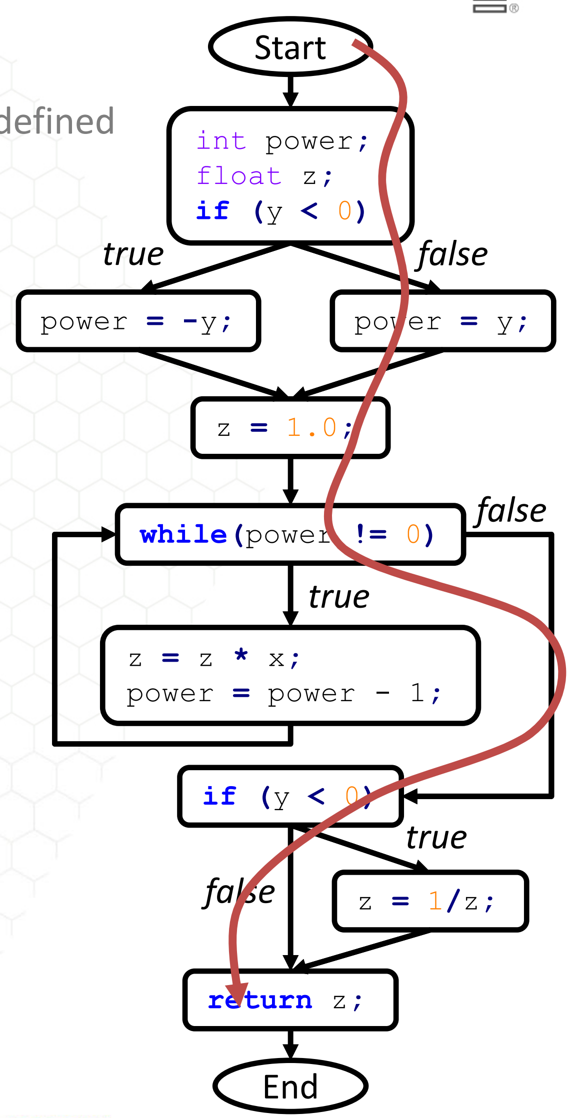
- X is data dependent on Y **iff**

1) There is a variable V that is defined at Y and used at X
2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
1.  float pow(int x, int y)          // D: x, y
2.  {
3.      int power;
4.      float z;
5.      if (y < 0)                    // U: y
6.          power = -y;               // D: power  U: y
7.      else
8.          power = y;                // D: power  U: y
9.      z = 1.0;                      // D: z
10.     while(power != 0) {           // U: power
11.         z = z * x;                // D: z  U: x, z
12.         power--;                  // D: power  U: power
13.     }
14.     if (y < 0)                    // U: y
15.         z = 1/z;                  // D: z  U: z
16.     return z;                     // U: z
17. }
```

DD(16) = {9}

DD(16) = ?



DATA DEP. MODELLING WITH DU CHAINS

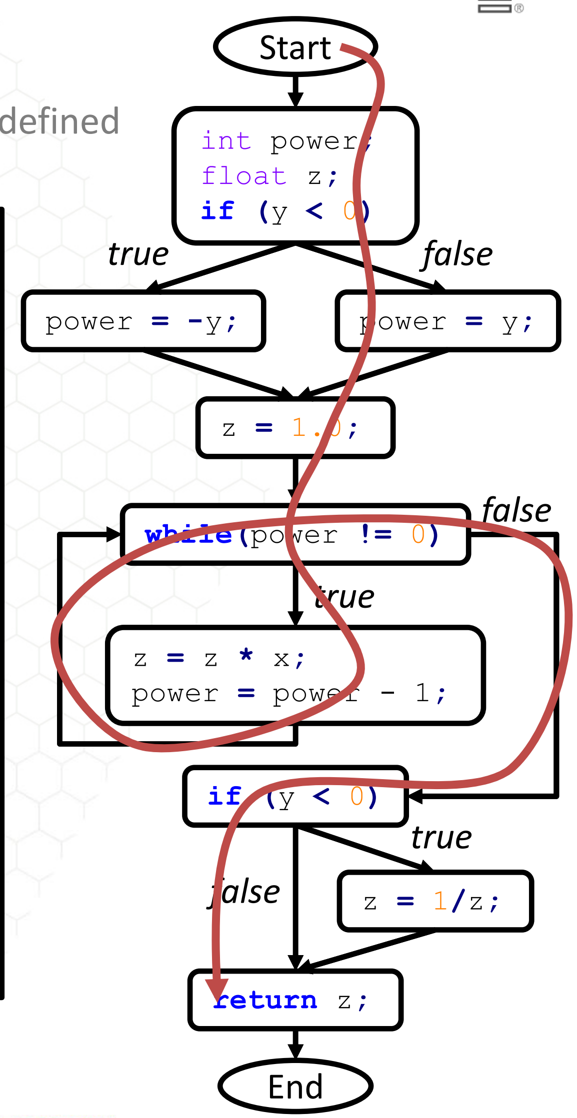
- X is data dependent on Y **iff**

1) There is a variable V that is defined at Y and used at X
2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
1.  float pow(int x, int y)       // D: x, y
2.  {
3.      int power;
4.      float z;
5.      if (y < 0)                 // U: y
6.          power = -y;            // D: power  U: y
7.      else
8.          power = y;             // D: power  U: y
9.      z = 1.0;                   // D: z
10.     while(power != 0) {        // U: power
11.         z = z * x;             // D: z  U: x, z
12.         power--;               // D: power  U: power
13.     }
14.     if (y < 0)                 // U: y
15.         z = 1/z;               // D: z  U: z
16.     return z;                  // U: z
17. }
```

DD(16) = {9}

DD(16) = {9, 11}



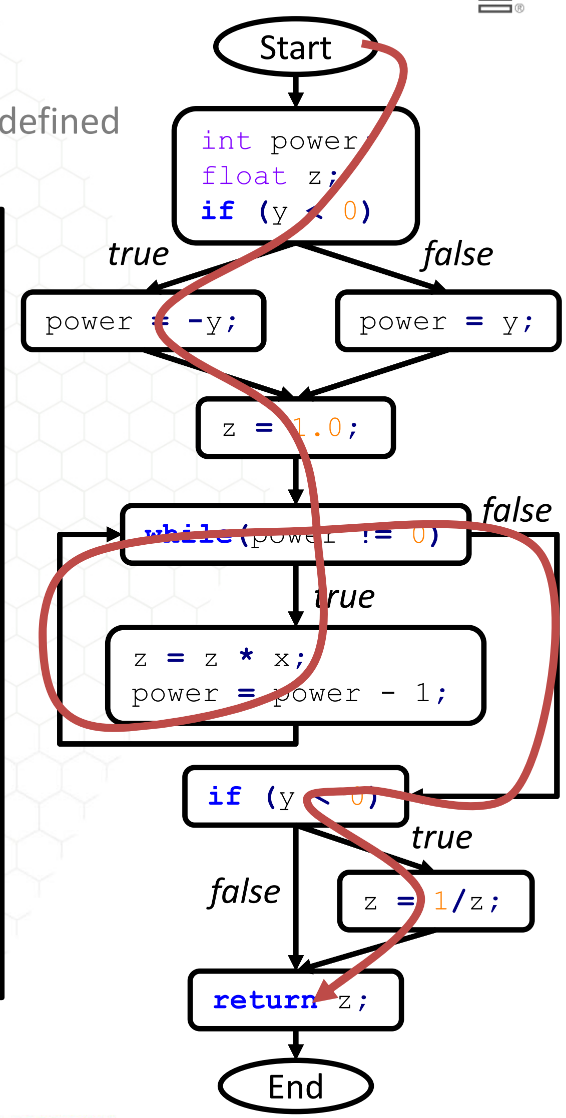
DATA DEP. MODELLING WITH DU CHAINS

- X is data dependent on Y **iff**

1) There is a variable V that is defined at Y and used at X
2) There exists a path of nonzero length from Y to X along which V is not re-defined

```
1.  float pow(int x, int y)        // D: x, y
2.  {
3.      int power;
4.      float z;
5.      if (y < 0)                 // U: y
6.          power = -y;            // D: power  U: y
7.      else
8.          power = y;             // D: power  U: y
9.      z = 1.0;                   // D: z
10.     while(power != 0) {        // U: power
11.         z = z * x;             // D: z  U: x, z
12.         power--;               // D: power  U: power
13.     }
14.     if (y < 0)                 // U: y
15.         z = 1/z;               // D: z  U: z
16.     return z;                  // U: z
17. }
```

DD(16) = {9, 11}   DD(16) = {9, 11, 15}



- Def-Use chains link the definition of a data location and the use of the data location
  - Registers or memory!

```
.text:00 pow               proc near
.text:00
.text:00 var_ReturnVal     = dword ptr -1Ch
.text:00 var_Y             = dword ptr -18h
.text:00 var_X             = dword ptr -14h
.text:00 var_Z             = dword ptr -8
.text:00 var_Power         = dword ptr -4
.text:00
.text:00 push    rbp                 ; D: rsp, [rsp] U: rsp, rbp
.text:01 mov     rbp, rsp            ; D: rbp U: rsp
.text:04 mov     [rbp+var_X], edi    ; D: [rbp+var_X] U: rbp, edi
.text:07 mov     [rbp+var_Y], esi    ; D: [rbp+var_Y] U: rbp, esi
.text:0A cmp     [rbp+var_Y], 0      ; D: rflags, U: rbp, [rbp+var_Y]
.text:0E jns     short loc_1A        ; D: U: rflags
.text:10 mov     eax, [rbp+var_Y]    ; D: eax U: rbp, [rbp+var_Y]
.text:13 neg     eax                 ; D: eax U: eax
.text:15 mov     [rbp+var_Power], eax ; D: [rbp+var_Power] U: rbp, eax
.text:18 jmp     short loc_20        ; D: U:
```

CREATING THE NEXT®

DATA DEP. MODELLING ON BINARIES

- Be careful! Data dependence in terms of source lines is easy to see
  - Because our brains naturally follow the control flow!!

- Consider:

```
1.  int func(int y)
2.  {
3.      int x = 1;
4.      if (y == 1)
5.          x = 2;
6.      else
7.          x += 1;
8.      return x;
9.  }
```

- The if statement (line 4) is dependent on the argument (line 1)

- The x += 1 (line 7) is dependent on the initialization of x (line 3)

- The return is dependent on both paths of the if statement (lines 5 and 7), since either are possible

- Data Dep. **must consider the CFG** of the basic blocks traversed before the current block!

```
int func(int y)
{
      int x = 1;
      if (y == 1):
            x = 2;
      else:
            x += 1;
      return x;
}
```

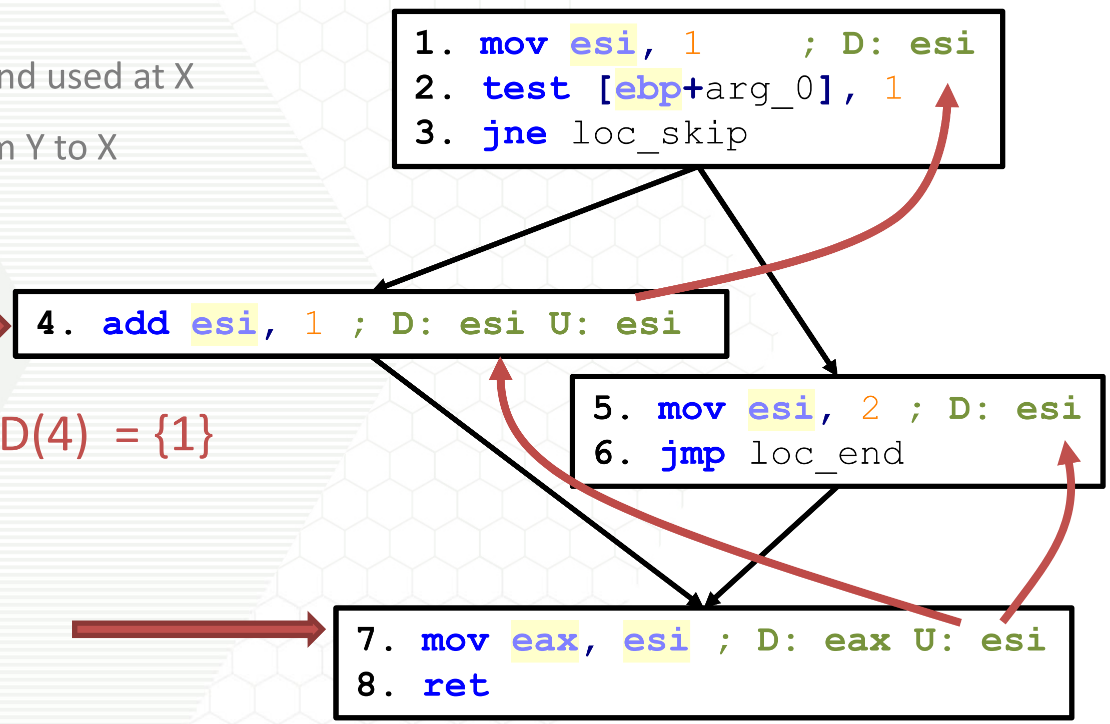
```
loc_start:
    ; stack setup
    mov esi, 1          ; D: esi
    test [ebp+arg_0], 1
    jne loc_skip
    mov esi, 2          ; D: esi
    jmp loc_end

loc_skip:
    add esi, 1    ; D: esi U: esi

loc_end:
    mov eax, esi  ; D: eax U: esi
    ; clean up stack
    ret
```

- For example: A **linear parse of the DU Chain (not following the CFG)** would incorrectly say:
  - The `add esi, 1` is data dependent on `mov esi, 2`
  - The `mov eax, esi` is only data dependent on `add esi, 1`
- When computing data dep, you MUST follow the possible paths in the CFG!

- Data Dep. **must consider the CFG** of the basic blocks traversed before the current block!

- X is data dependent on Y **iff**

1) There is a variable V that is defined at Y and used at X
2) There exists a path of nonzero length from Y to X along which V is not re-defined



DD(4) = {1}

DD(7) = ? DD(7) = {4, 5}

KEEPING PUSHES AND POPS STRAIGHT!

- Problem: Pushes and Pops Def and Use the same stuff!!

```
.text:00 push    rax                  ; D: rsp, [rsp]    U: rsp, rax
.text:01 push    rbx                  ; D: rsp, [rsp]    U: rsp, rbx
.text:02 pop     rcx                  ; D: rsp, rcx      U: rsp, [rsp]
.text:03 push    rdx        reg. read ; D: rsp, [rsp]    U: rsp, rdx
.text:04 pop     rsi        (rsp)     ; D: rsp, rsi      U: rsp, [rsp]
.text:05 pop     rdi                  ; D: rsp, rdi      U: rsp, [rsp]
```

- Bigger Problem: **Linear parse of the DU Chain (even following the CFG) will be incorrect**

KEEPING PUSHES AND POPS STRAIGHT!

- Problem: Pushes and Pops Def and Use the same stuff!!

```
.text:00 push    rax          ; D: rsp, [rsp]    U: rsp, rax
.text:01 push    rbx          ; D: rsp, [rsp]    U: rsp, rbx
.text:02 pop     rcx          ; D: rsp, rcx      U: rsp, [rsp]
.text:03 push    rdx          ; D: rsp, [rsp]    U: rsp, rdx
.text:04 pop     rsi          ; D: rsp, rsi      U: rsp, [rsp]
.text:05 pop     rdi          ; D: rsp, rdi      U: rsp, [rsp]
```

mem read ([rsp])

- Bigger Problem: **Linear parse of the DU Chain (even following the CFG) will be incorrect**

KEEPING PUSHES AND POPS STRAIGHT!

- Problem: Pushes and Pops Def and Use the same stuff!!

```
.text:00 push    rax   mem read   ; D: rsp, [rsp]     U: rsp, rax
.text:01 push    rbx   ([rsp])    ; D: rsp, [rsp]     U: rsp, rbx
.text:02 pop     rcx              ; D: rsp, rcx       U: rsp, [rsp]
.text:03 push    rdx              ; D: rsp, [rsp]     U: rsp, rdx
.text:04 pop     rsi              ; D: rsp, rsi       U: rsp, [rsp]
.text:05 pop     rdi              ; D: rsp, rdi       U: rsp, [rsp]
```

- Bigger Problem: **Linear parse of the DU Chain (even following the CFG) will be incorrect**
- **Solution:** Create a “Shadow Stack” --- A model stack which tracks who pushed what!

KEEPING PUSHES AND POPS STRAIGHT!

- Problem: Pushes and Pops Def and Use the same stuff!!

```
.text:00 push   rax     ; D: rsp, [rsp]     U: rsp, rax
.text:01 push   rbx     ; D: rsp, [rsp]     U: rsp, rbx
.text:02 pop    rcx     ; D: rsp, rcx       U: rsp, [rsp]
.text:03 push   rdx     ; D: rsp, [rsp]     U: rsp, rdx
.text:04 pop    rsi     ; D: rsp, rsi       U: rsp, [rsp]
.text:05 pop    rdi     ; D: rsp, rdi       U: rsp, [rsp]
```

- Bigger Problem: **Linear parse of the DU Chain (even following the CFG) will be incorrect**
- **Solution:** Create a "Shadow Stack" --- A model stack which tracks who pushed what!

DD(.text:00) = { prior rax, rsp }
DD(.text:01) = { .text:00, prior rbx }
DD(.text:02) = { .text:01, .text:01 }

Shadow Stack

.text:00
.text:01

CREATING THE NEXT®

KEEPING PUSHES AND POPS STRAIGHT!

- Problem: Pushes and Pops Def and Use the same stuff!!

```
.text:00 push    rax                 ; D: rsp, [rsp]    U: rsp, rax
.text:01 push    rbx                 ; D: rsp, [rsp]    U: rsp, rbx
.text:02 pop     rcx                 ; D: rsp, rcx      U: rsp, [rsp]
.text:03 push    rdx                 ; D: rsp, [rsp]    U: rsp, rdx
.text:04 pop     rsi                 ; D: rsp, rsi      U: rsp, [rsp]
.text:05 pop     rdi                 ; D: rsp, rdi      U: rsp, [rsp]
```

- Bigger Problem: **Linear parse of the DU Chain (even following the CFG) will be incorrect**
- **Solution:** Create a "Shadow Stack" --- A model stack which tracks who pushed what!

DD(.text:00) = { prior rax, rsp }
DD(.text:01) = { .text:00, prior rbx }
DD(.text:02) = { .text:01, .text:01 }
DD(.text:03) = { .text:02, prior rdx }
DD(.text:04) = { .text:03, .text:03 }
DD(.text:05) = { .text:04, .text:00 }

Shadow Stack

.text:00
.text:03

- Aliasing --- the kryptonite to the data dependence superhero!
    - Alias: A variable which can refer to multiple memory locations/objects

C loves aliases

```
int x, y, z;
int * p;
x = 5;
y = 10;
z = 8;
p = &x;
p = p + z;
z = *p;
```

What is the value of z?

Assembly loves aliases

```
push    rbp
mov     rbp, rsp
push    rdi
sub     rsp, 16
...
cmp     QWORD PTR [rbp], 0
...
mov     rax, QWORD PTR [rsp+24]
```

What is being moved into RAX?

```
void func(int d) {
    int b;
    int c;
    if ( d == 0 )
        ...
    int a = d;
}
```

The original code had no aliases!

STATIC VALUE TRACKING

Assembly loves aliases

```
push    rbp
mov     rbp, rsp
push    rdi
sub     rsp, 16
...
cmp     QWORD PTR [rbp], 0
...
mov     rax, QWORD PTR [rsp+24]
```

What is being moved into RAX?

RAX = 0x0!

Value Tracking

RBP:	0x3FF8
RSP:	0x3FE0
RDI:	0x0
RAX:	0x0
[0x3FF8]:	0x0

[0x3FE0 + 24 = [0x3FF8]

[0x3FF8]

- Similar to Shadow Stack tracking, but now you track specific values of registers/memory
- You may have to assume "initial values" at function or program start
- As you step through the code, parse the instructions and update the tracked values
- When you need to know a value, hopefully you have tracked it!

CREATING THE NEXT®

PROGRAM DEPENDENCE GRAPHS

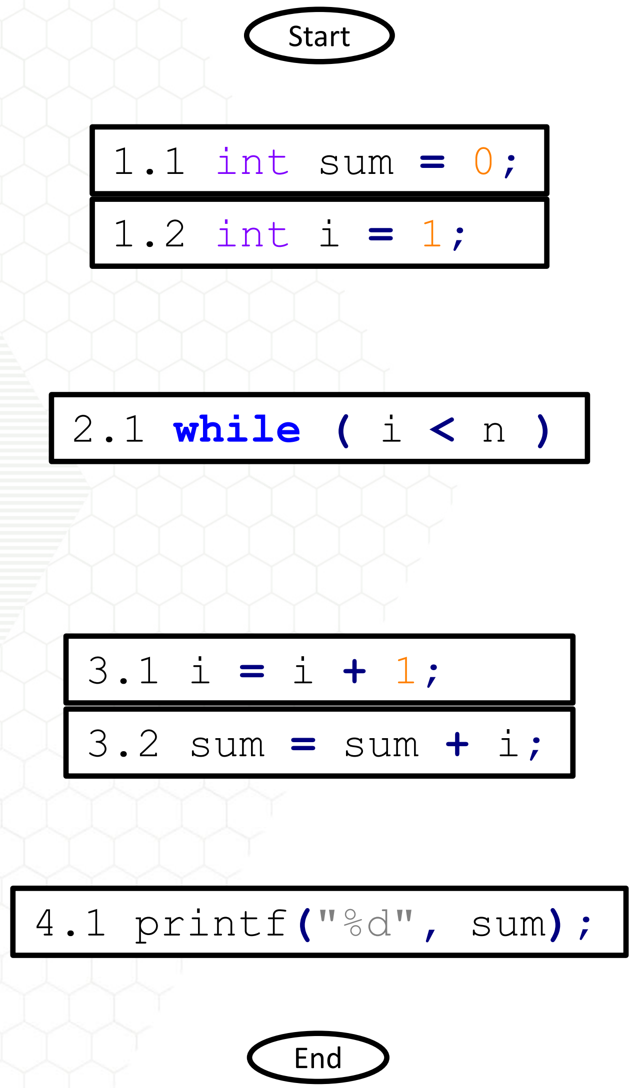
CREATING THE NEXT®

PROGRAM DEPENDENCE GRAPH (PDG)

- The second most widely used program representation
- Represents the union of the two types of dependences
  - Data dependence
  - Control dependence
- Optional (but valuable) reading:
  - Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren.
    "The Program Dependence Graph And Its Use In Optimization."
    ACM Transactions on Programming Languages and Systems (TOPLAS) 9.3. (1987): 319-349.
  - Susan Horwitz, Thomas Reps, and David Binkley.
    "Interprocedural slicing using dependence graphs."
    ACM Transactions on Programming Languages and Systems (TOPLAS) 12.1 (1990): 26-60.

PROGRAM DEPENDENCE GRAPH

- A program dependence graph PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks "super-nodes"



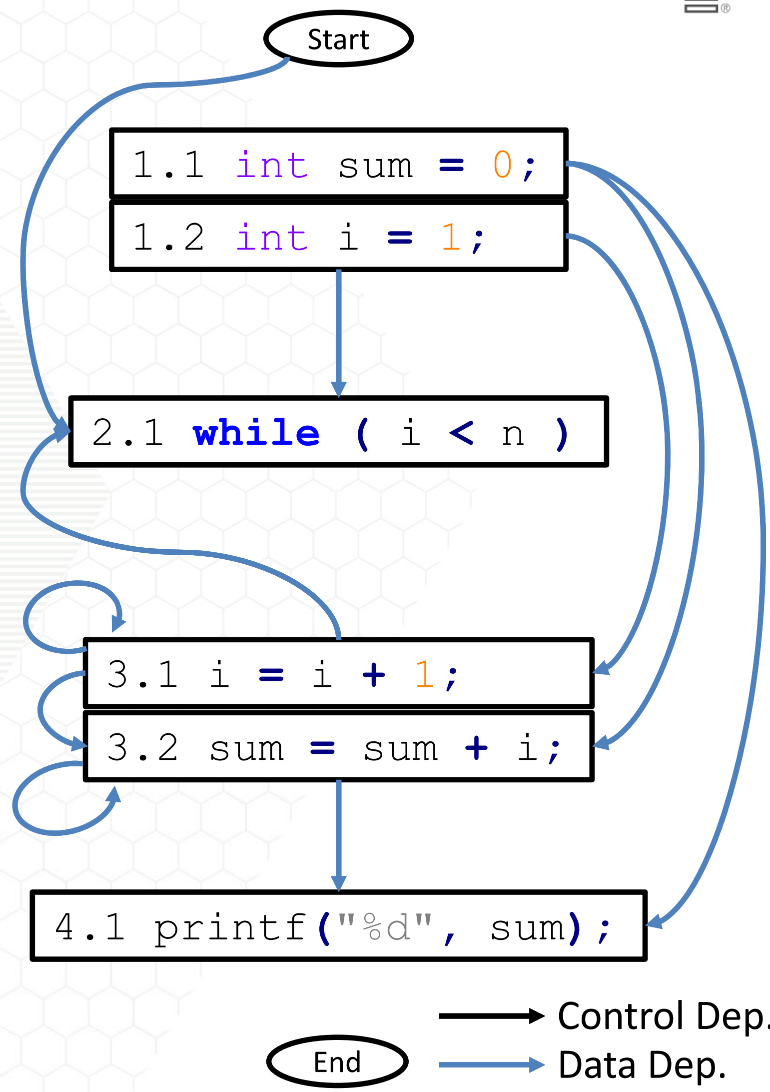

```c
void sumUp(int n) {
     int sum = 0;
     int i = 1;
     while ( i < n ) {
          i = i + 1;
          sum = sum + i;
     }
     printf("%d", sum);
}
```

CREATING THE NEXT®

PROGRAM DEPENDENCE GRAPH

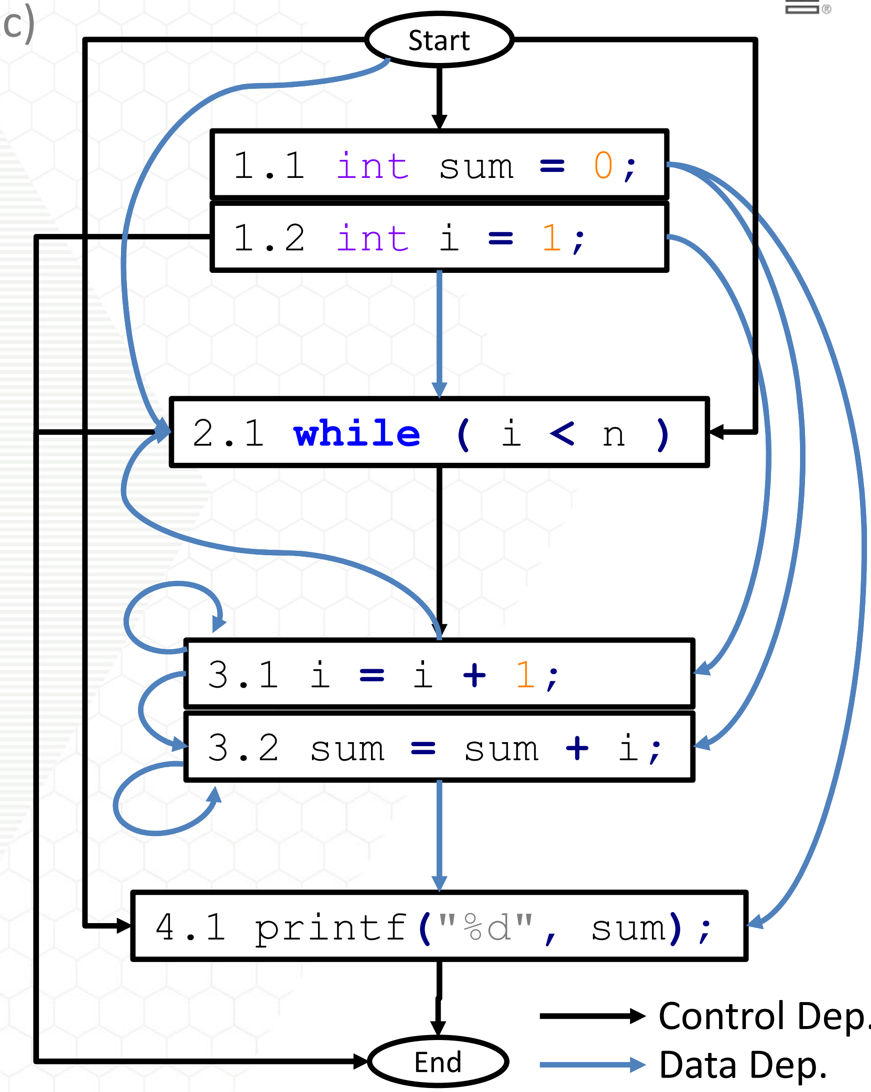
- A program dependence graph PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks “super-nodes”
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - Recall: X is data dependent on Y iff
    (1) There exists a variable v that is defined at Y and used at X
    (2) There exists a path of nonzero length from Y to X along which v is not re-defined

```
void sumUp(int n) {
     int sum = 0;
     int i = 1;
     while ( i < n ) {
          i = i + 1;
          sum = sum + i;
     }
     printf("%d", sum);
}
```



PROGRAM DEPENDENCE GRAPH

- A program dependence graph PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks "super-nodes"
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - A finite set Ec of edges (i, j) representing that (super-)node n_j is control dependent on node n_i
  - Recall: Y is control-dependent on X iff X directly determines whether Y executes:
    (1) X is not strictly post-dominated by Y
    (2) There exists a path from X to Y s.t.
    every node in the path other than X and Y is post-dominated by Y
- Used to represent the set of all program statements involved in reaching any single execution point



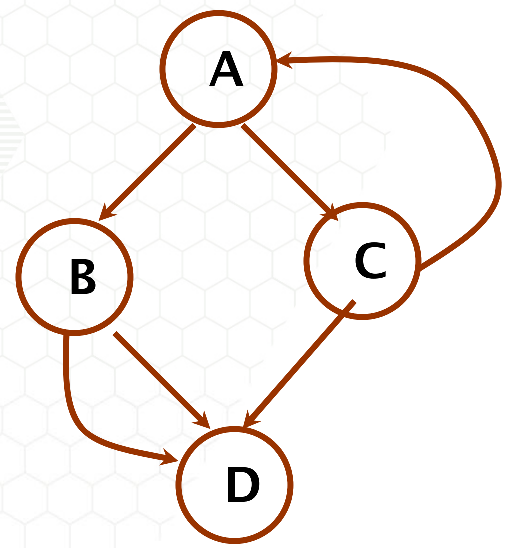
CREATING THE NEXT®

CALL GRAPH (CG)

- Simplest case: Nodes represent functions; each edge represents a function invocation
- Used to perform higher-level Interprocedural Analysis

```
void A( ) {
    B( );
    C( );
}

void C ( ) {
    D( );
    A( );
}
```

```
void B( ) {
L1:    D( );
L2:    D( );
}

void D ( ) {
}
```

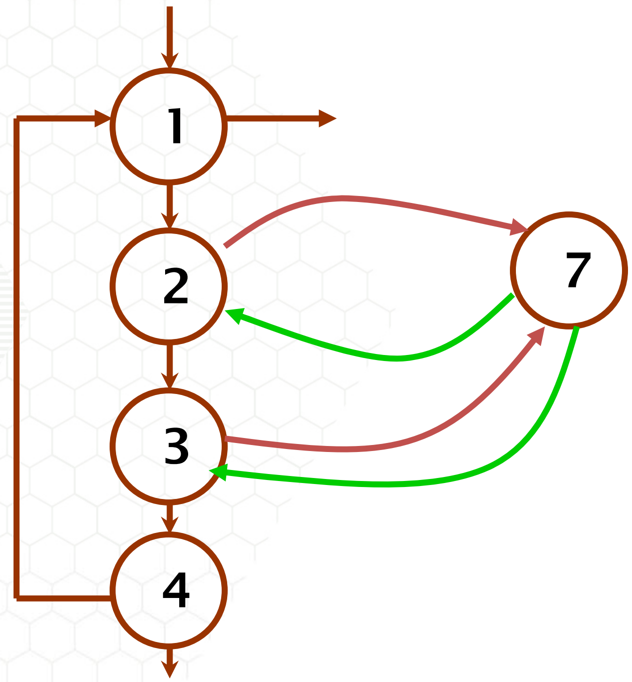


CREATING THE NEXT®

SUPER CONTROL FLOW GRAPH (SCFG)

- Interprocedural control flow graph
  - Additional edges are added connecting each call site to the beginning of the procedure it calls
  - The return statement links back to the call site

```
for (i=0; i<n; i++) {
    t1= f(0);
    t2 = f(243);
    x[i] = t1 + t2;
}
int  f (int v) {
    return (v+1);
}
```



- Rarely used in static analysis due to path explosion!
- Sometimes used in dynamic analysis due to ambiguity of function calls
  - JMP to some far away code? Fetch a return address and JMP to it?
  - JMP to a new function entry (no return address push) & then do a “double” return at RET?

- Interprocedural analysis is a very deep rabbit hole!
- Excellent material: https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec05-Interprocedural.pdf

- Interprocedural analysis can provide VERY precise static analysis
- But path explosion makes global reasoning nearly impossible!

- For this class, we will selectively cover interprocedural analysis as it applies to dynamic analysis

- As we will see, dynamic analysis cannot reason locally (due to limited knowledge)
- Therefore, everything is global until we realize it is local! ☺

CREATING THE NEXT®

QUESTIONS?

Georgia Tech

CREATING THE NEXT®