Program Analysis (static analysis)

Hyungsub Kim

Purdue University

About me

- A PhD student in Purdue CS
  - Joined in 2018
  - Working on how to apply static and dynamic analysis to robotic vehicle security
  - Published papers into security conferences (NDSS, S&P, USENIX, ACSAC)

Details of research topics:

1) Find bugs (fuzzing)
2) Automatically patch the bugs
3) Verify the fixed bugs

Outline

- Intro
- Terminology
- Static Analysis

Goal (1)

1. Understanding terms in program analysis techniques
   - Path-sensitive, flow-sensitive
   - Intra-procedural, Inter-procedural
   - Static single assignment (SSA), pointer analysis

But why should we care about these terms?

Goal (2)

1. Understanding terms in program analysis techniques
   - Path-sensitive, flow-sensitive
   - Intra-procedural, Inter-procedural
   - Static single assignment (SSA), pointer analysis

load and store operations recursively. For pointers, to identify data flow via pointer reference/dereference operators, we perform an inter-procedural, path-insensitive, and flow-sensitive points-to analysis [62]. More precisely, the profiling engine operates in three steps: (1) performs Andersen's pointer analysis [8] to identify aliases of the parameter variables, (2) transforms the code to its single static assignment form [59] and builds the data-flow graph (DFG), and (3) collects the def-use chain of the identified parameter variable from the built DFG.

Can you understand this paragraph?

<A paragraph on a paper in NDSS 2021>

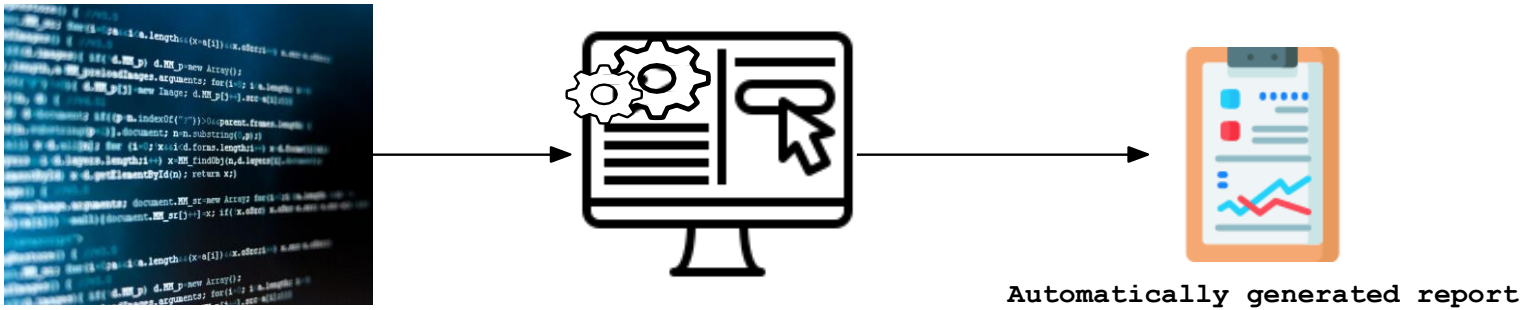
Goal (3)

2. Understanding how each technique is used for improving security in software

What is Program Analysis

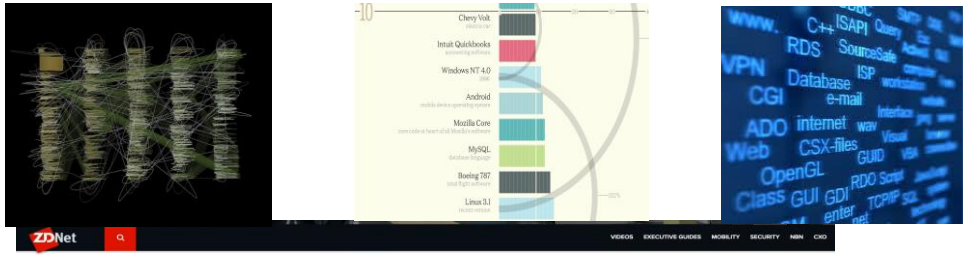
- A process of automatically analyzing behaviors of a program

- Applications:
  - Program understanding
  - Compiler optimizations
  - Bug finding



Why should we automate this analysis?

- Modern system software
  - Extremely large and complex but error-prone



More Complex!

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

Memory Leaks

Buffer Overflows

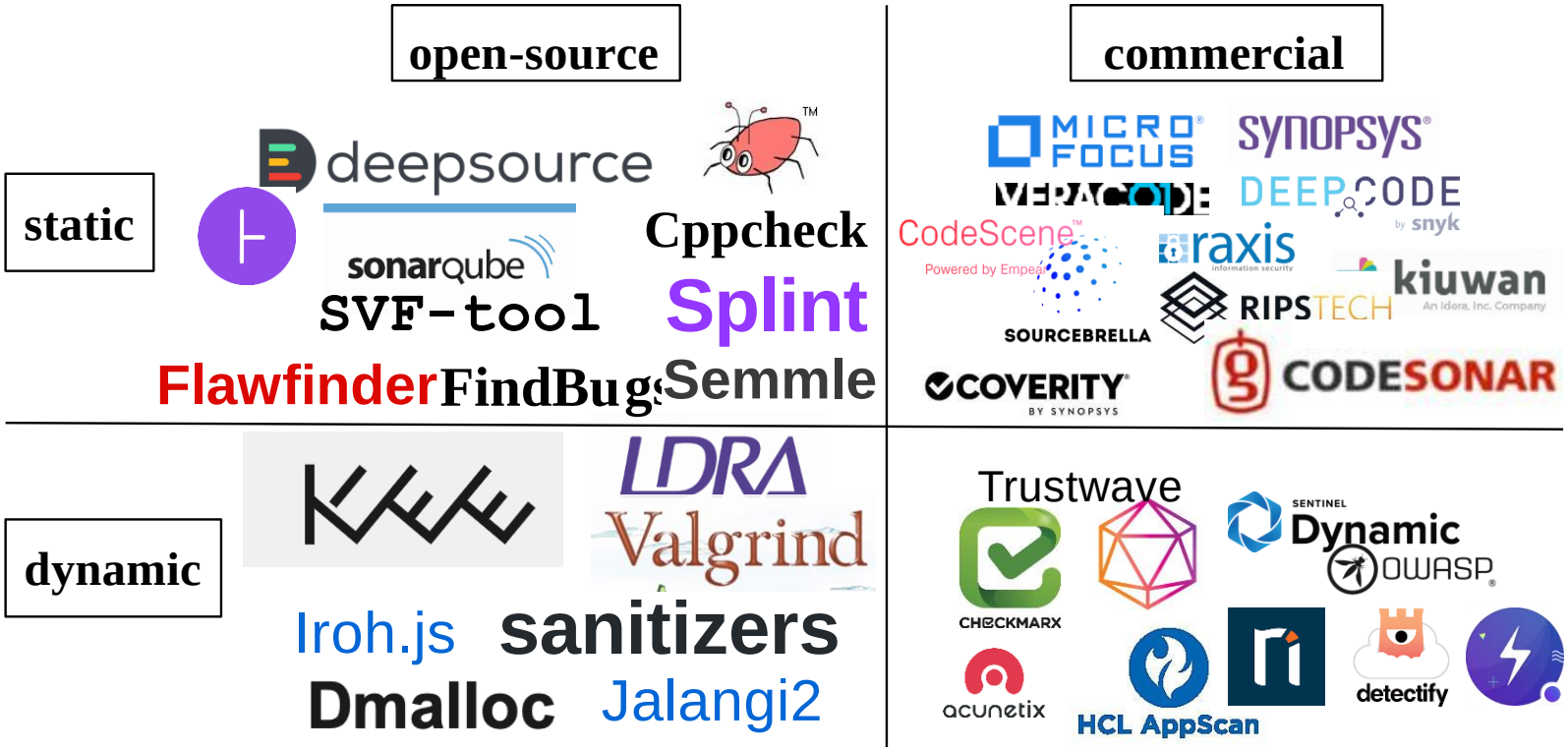
Null Pointers

Use-After-Frees

Data-races

More Buggy!

Existing Program Analysis Tools



Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it*

  + Catch bugs earlier during software development

  - False alarms due to over-approximation

Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it*

  + Catch bugs earlier during software development

  - False alarms due to over-approximation

Dynamic Analysis

- *Analyze a program at runtime*

  + Zero or very low false alarm rates

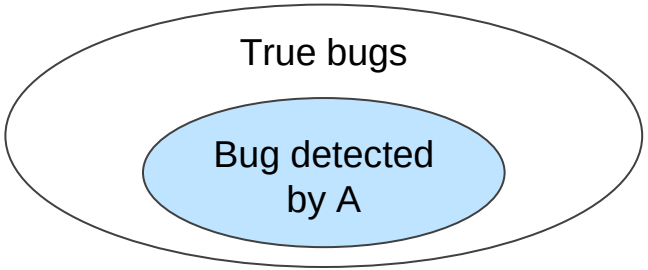
  - May miss bugs (false negative) due to under-approximation

Outline

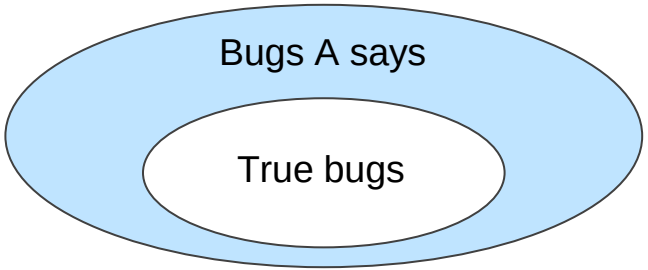
- Intro
- Terminology
- Static Analysis

Characterizing Program Analyses

- Soundness
  - If analysis `A` says that `X` is buggy, then `X` is buggy.



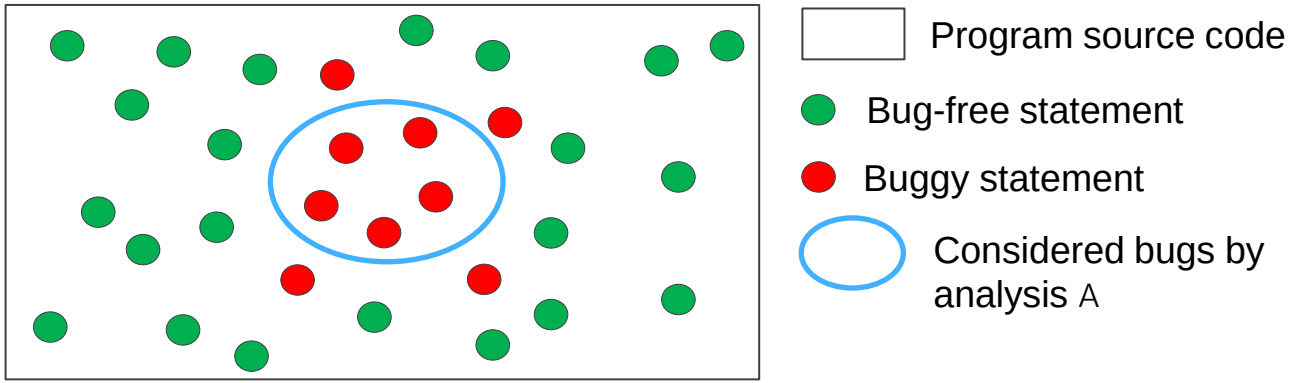
- Completeness
  - If `X` is buggy, then analysis `A` says `X` is buggy.



Sound vs. Complete (1)

- Is analysis A sound? Yes
  - Why? If analysis A says that X is buggy, then X is buggy.

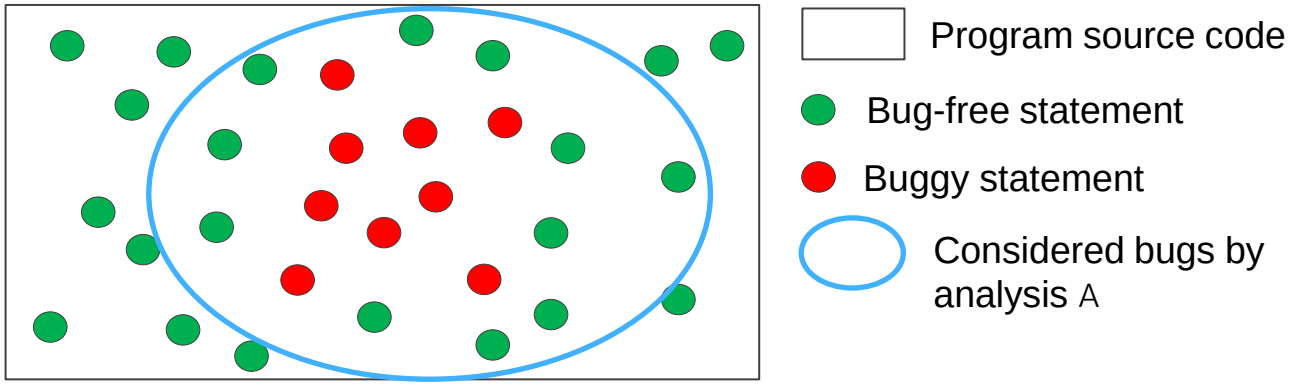
- Is analysis A complete? No
  - Why? If X is buggy, then analysis A says X is buggy.



Sound vs. Complete (2)

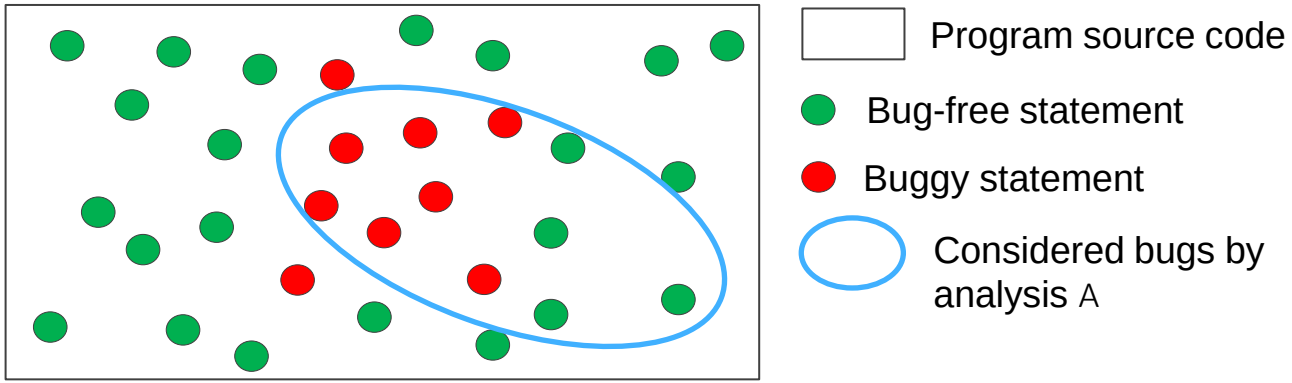
- Is analysis A sound? No
  - Why? If analysis A says that X is buggy, then X is buggy.

- Is analysis A complete? Yes
  - Why? If X is buggy, then analysis A says X is buggy.



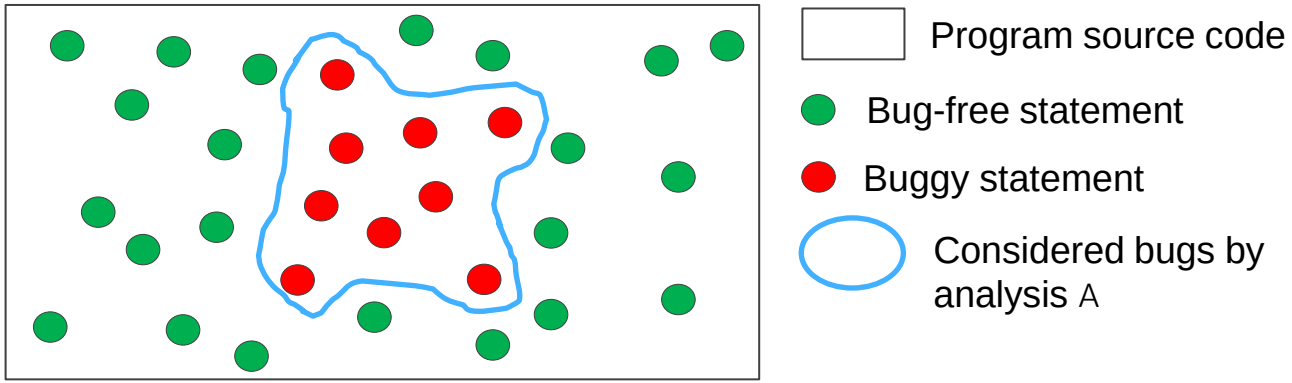
Sound vs. Complete (3)

- Is analysis A sound? No
  - Why? If analysis A says that X is buggy, then X is buggy.
- Is analysis A complete? No
  - Why? If X is buggy, then analysis A says X is buggy.



Sound vs. Complete (4)

- Is analysis A sound? Yes
  - Why? If analysis A says that X is buggy, then X is buggy.
- Is analysis A complete? Yes
  - Why? If X is buggy, then analysis A says X is buggy.



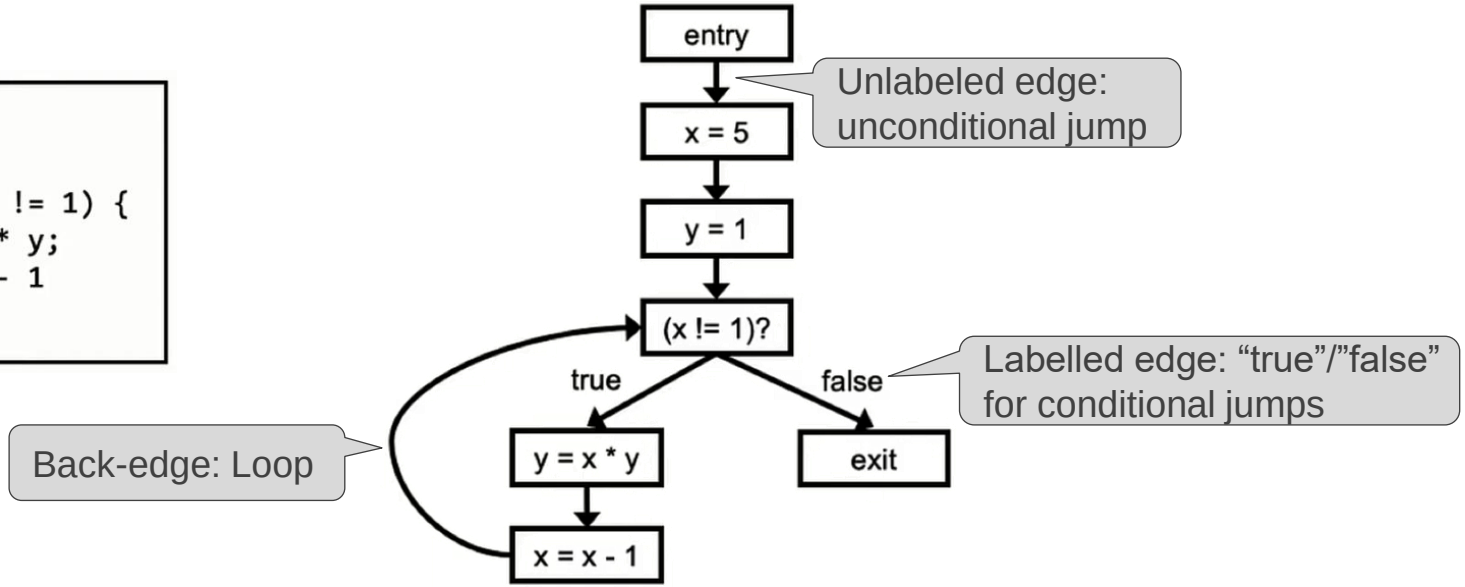
Program Representations

- Original representations of programs
  - Source code
  - Binaries
- They are hard for machines to analyze
- Software is translated into certain representations before analyses are applied.

Control-Flow Graph

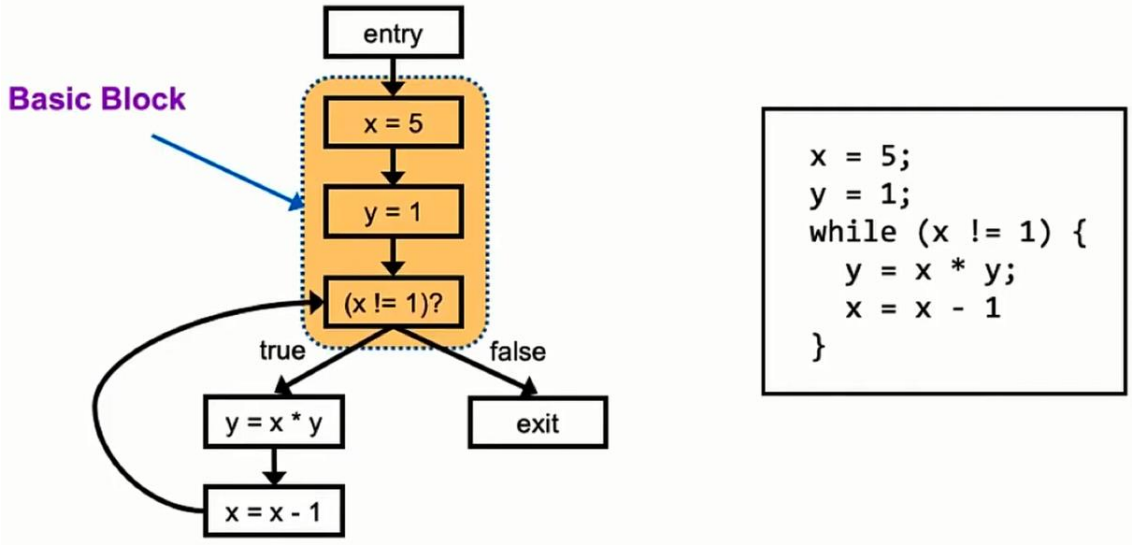
- Directed graph
  - Edge: summarizing flow of graph
  - Node: a statement in a program

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```



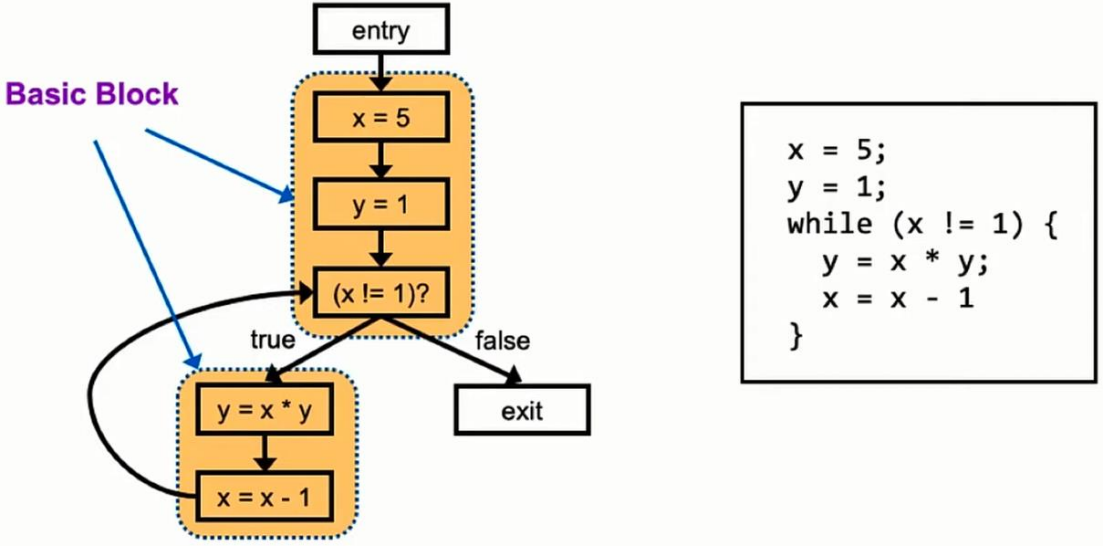
Basic Block (1)

- Definition
  - Group statements without intervening control flow



Basic Block (2)

- Definition
  - Group statements without intervening control flow




```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Call Graph

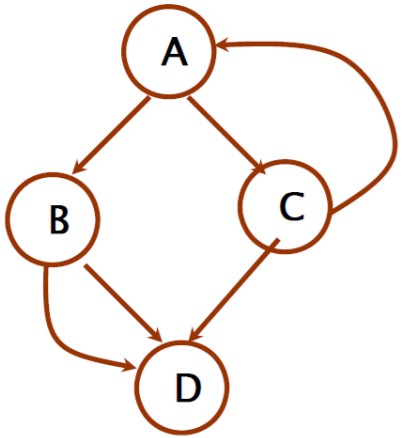
- Node
  - Represents a function
- Edge
  - Represents a function invocation

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

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

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

void D ( ) {
}
```

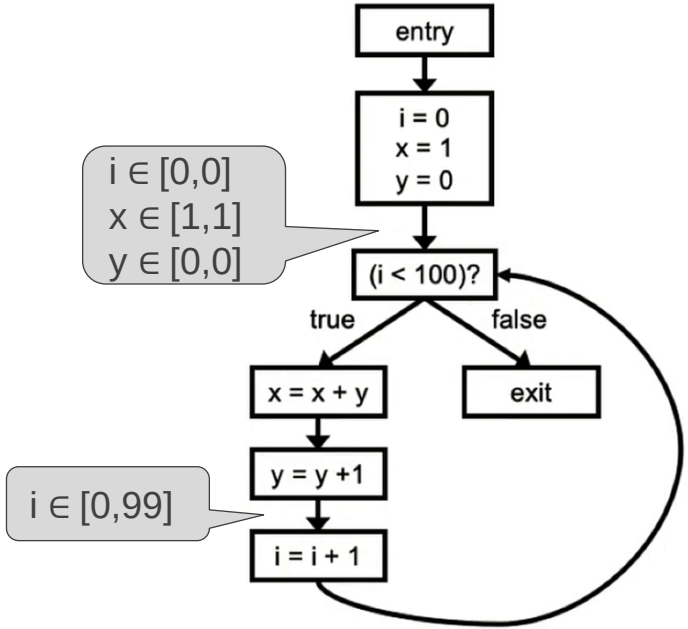


Outline

- Intro
- Terminology
- Static Analysis

Interval Analysis

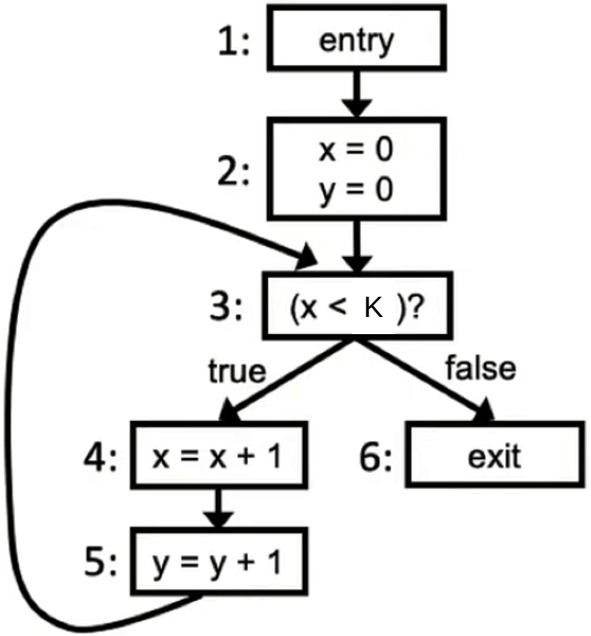
- Goal
  - For each integer variable at each program point
  - Find a lower/upper bounds on its possible values



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis
K is more than 0 (i.e., K > 0)

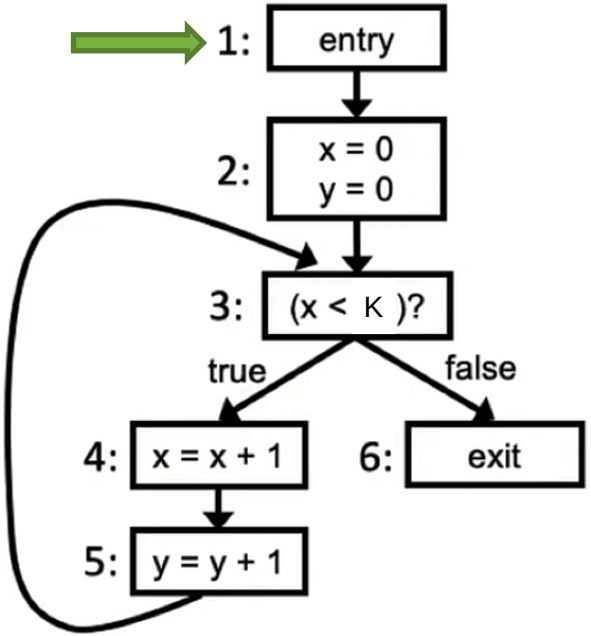
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]				
2	x ∈[⊥] y ∈[⊥]				
3	x ∈[⊥] y ∈[⊥]				
4	x ∈[⊥] y ∈[⊥]				
5	x ∈[⊥] y ∈[⊥]				



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis

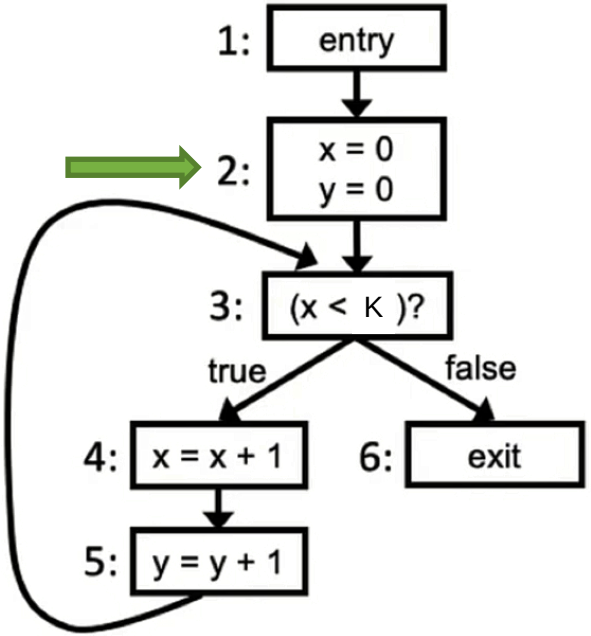
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]			
2	x ∈[⊥] y ∈[⊥]				
3	x ∈[⊥] y ∈[⊥]				
4	x ∈[⊥] y ∈[⊥]				
5	x ∈[⊥] y ∈[⊥]				



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis

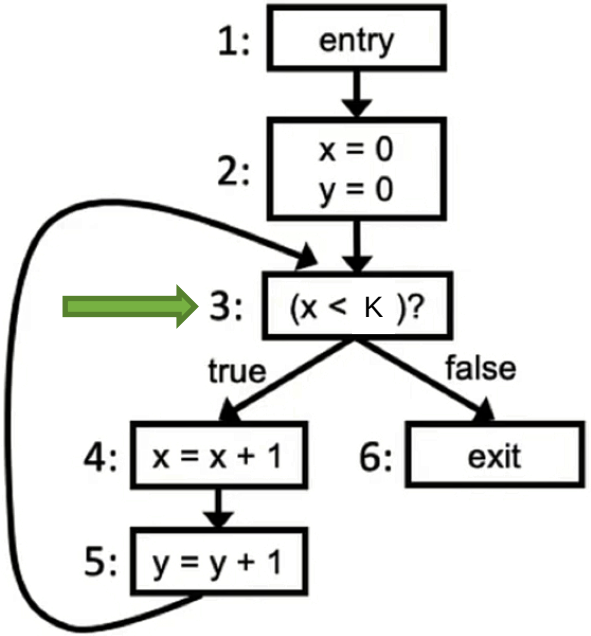
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]			
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
3	x ∈[⊥] y ∈[⊥]				
4	x ∈[⊥] y ∈[⊥]				
5	x ∈[⊥] y ∈[⊥]				



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis

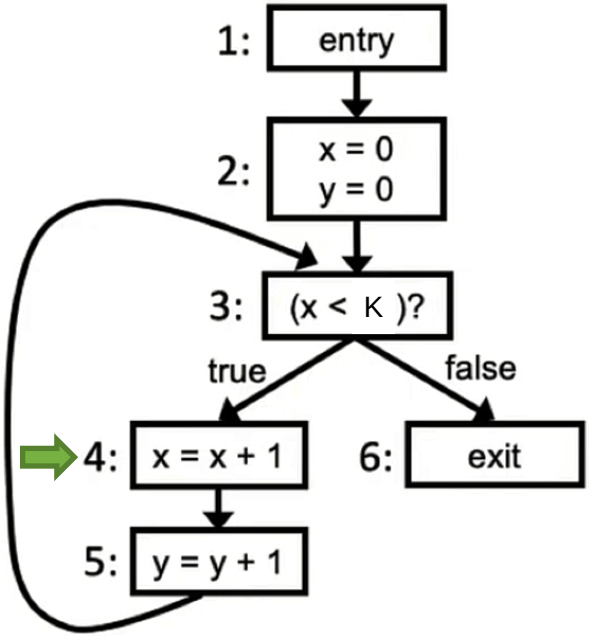
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]			
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
4	x ∈[⊥] y ∈[⊥]				
5	x ∈[⊥] y ∈[⊥]				



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis

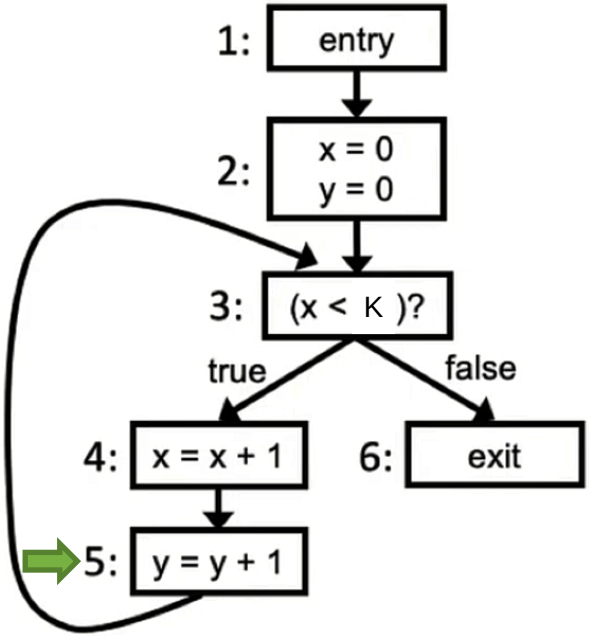
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]			
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]			
5	x ∈[⊥] y ∈[⊥]				



Interval Analysis Example

∞: Infinite
⊥: Undecided by a program analysis

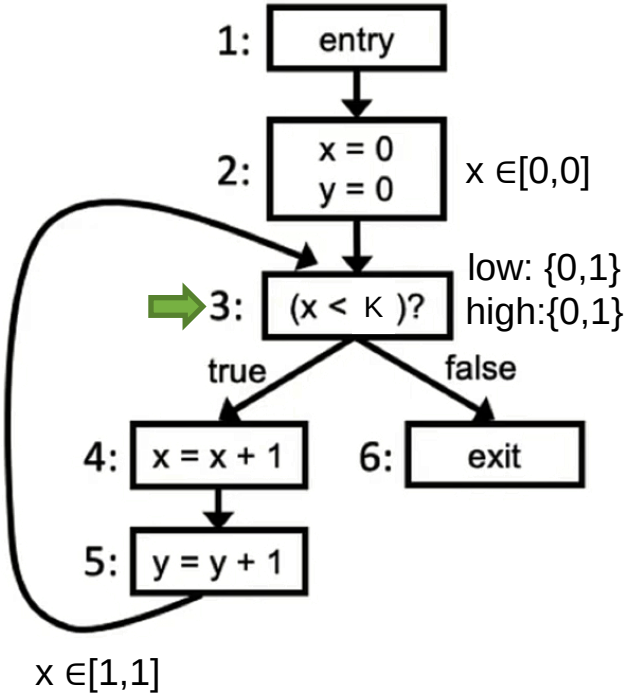
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]			
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]			
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]			
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]			



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

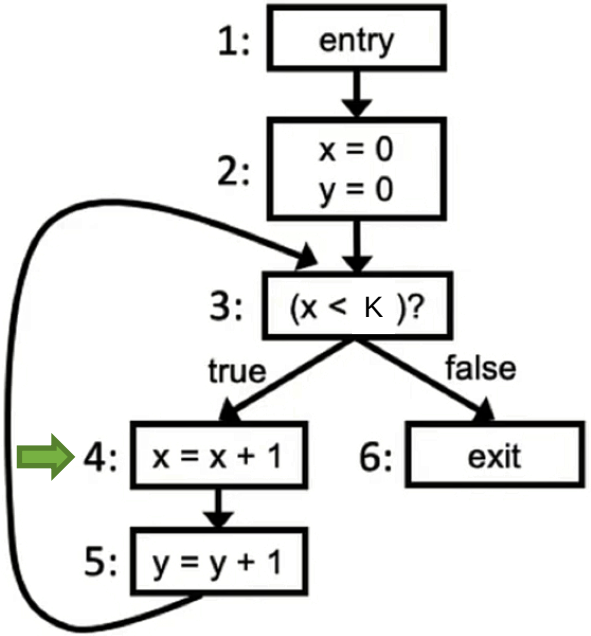
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]		
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]		
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]		
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]			
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]			



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

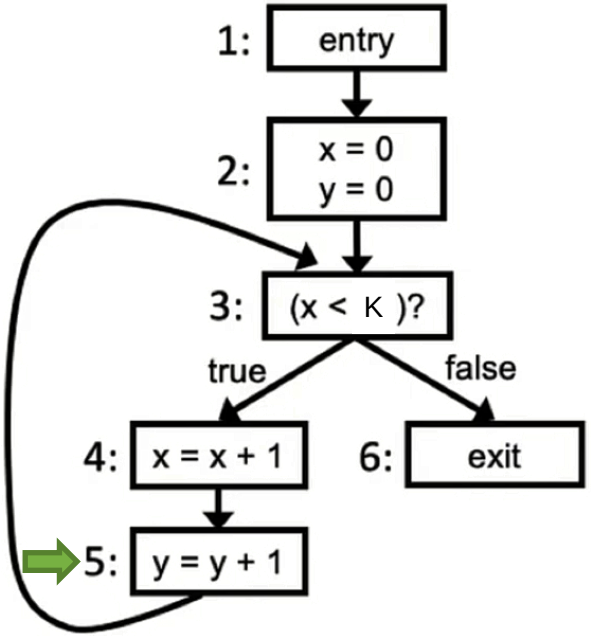
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]		
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]		
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]		
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]		
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]			



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

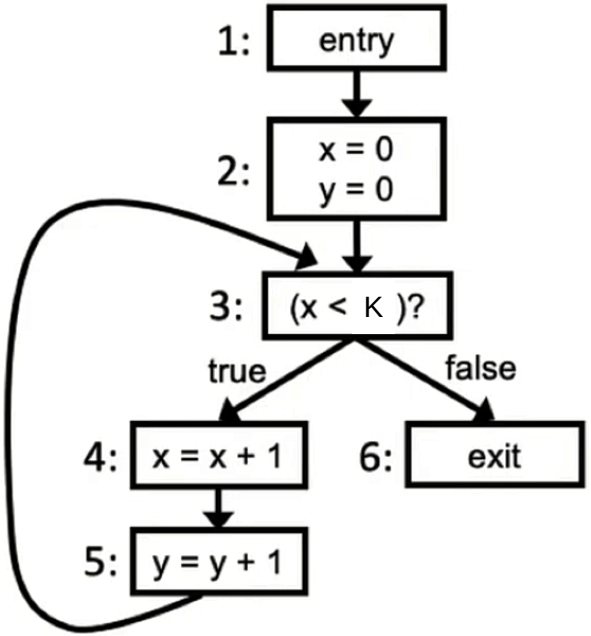
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]		
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]		
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]		
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]		
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]	x ∈[1,2] y ∈[1,2]		



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

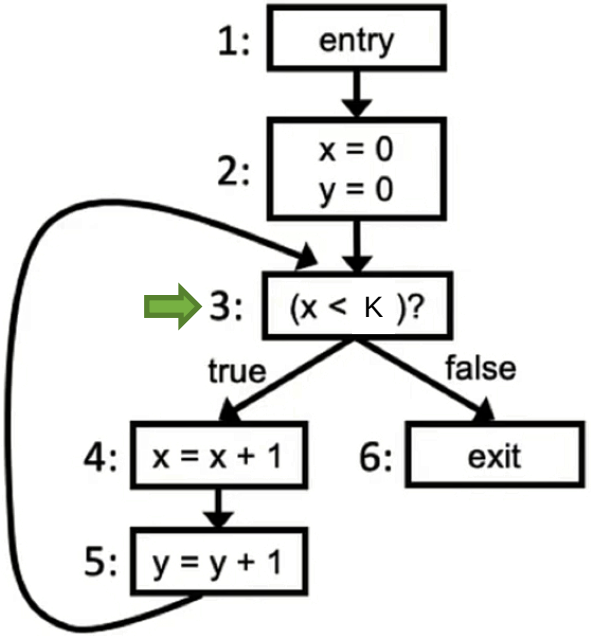
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]	x ∈[0,2] y ∈[0,2]	
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]	x ∈[1,3] y ∈[0,2]	
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]	x ∈[1,2] y ∈[1,2]	x ∈[1,3] y ∈[1,3]	



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

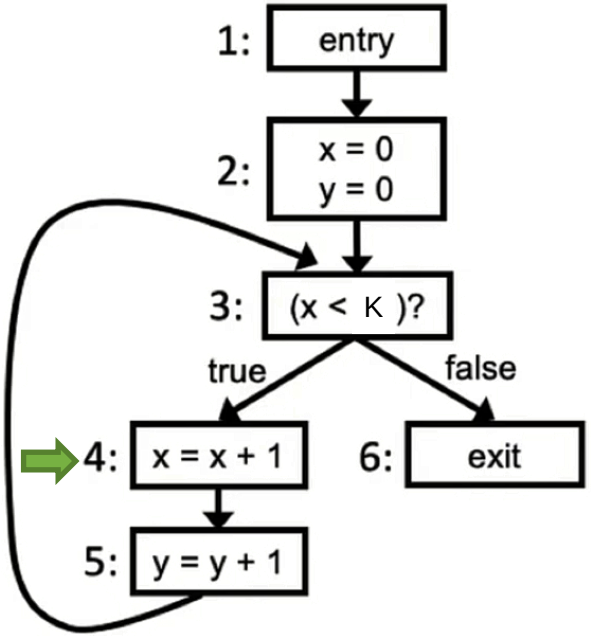
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]	x ∈[0,2] y ∈[0,2]	x ∈[0,k-1] y ∈[0,k-1]
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]	x ∈[1,3] y ∈[0,2]	
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]	x ∈[1,2] y ∈[1,2]	x ∈[1,3] y ∈[1,3]	



Interval Analysis Example

Keep iterating statements in a loop (i.e., nodes from 3 to 5)

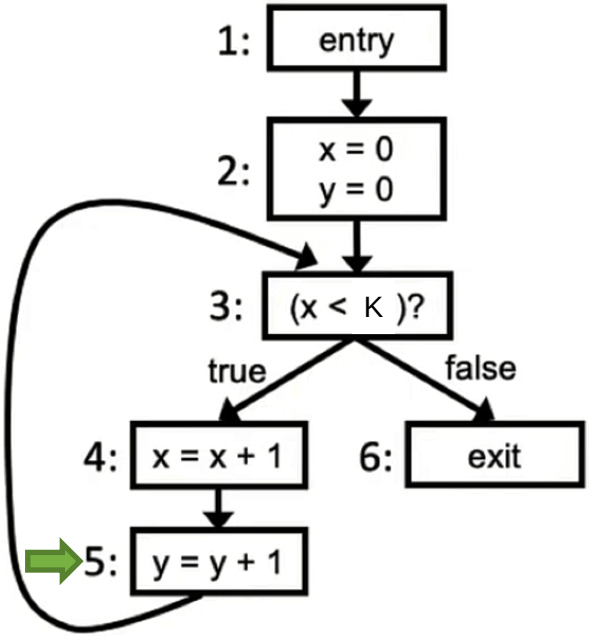
Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]	x ∈[0,2] y ∈[0,2]	x ∈[0,k-1] y ∈[0,k-1]
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]	x ∈[1,3] y ∈[0,2]	x ∈[1,k] y ∈[0,k-1]
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]	x ∈[1,2] y ∈[1,2]	x ∈[1,3] y ∈[1,3]	



Interval Analysis Example

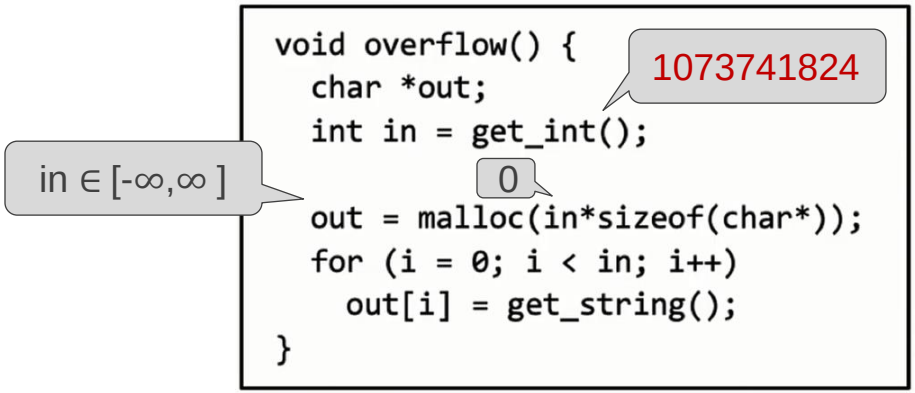
Keep iterating statements in a loop (i.e., nodes from 3 to 5)

Node	Iter #0	Iter #1	Iter #2	Iter #3	Iter #k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈[-∞,∞] y ∈[-∞,∞]
2	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]	x ∈[0,0] y ∈[0,0]
3	x ∈[⊥] y ∈[⊥]	x ∈[0,0] y ∈[0,0]	x ∈[0,1] y ∈[0,1]	x ∈[0,2] y ∈[0,2]	x ∈[0,k-1] y ∈[0,k-1]
4	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[0,0]	x ∈[1,2] y ∈[0,1]	x ∈[1,3] y ∈[0,2]	x ∈[1,k] y ∈[0,k-1]
5	x ∈[⊥] y ∈[⊥]	x ∈[1,1] y ∈[1,1]	x ∈[1,2] y ∈[1,2]	x ∈[1,3] y ∈[1,3]	x ∈[1,k] y ∈[1,k]



Interval Analysis

- Applications 1
  - Detecting integer overflow

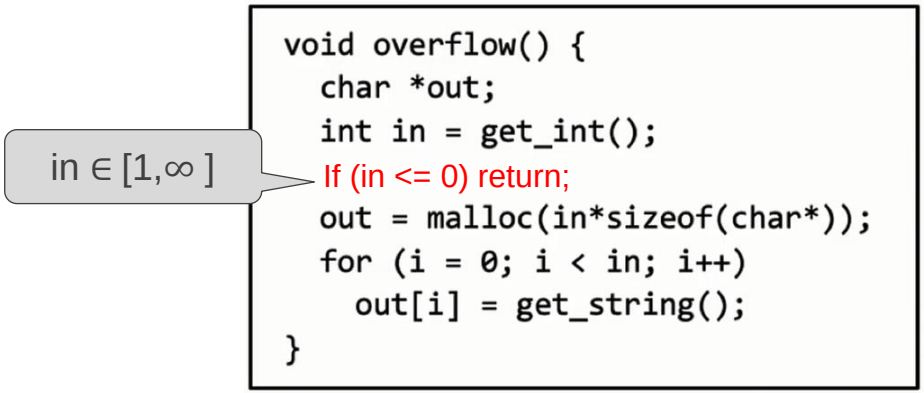


CVE-2019-3855
In LibSSH, an attacker can exploit to execute code on the client system when a user connects to the server

CVE-2019-8099
In Adobe Acrobat, an attacker can use to steal information

Interval Analysis

- Applications 1
  - Detecting integer overflow



Interval Analysis

- Applications 2
  - Detecting index-out-of-bounds

index ∈ [-∞,∞]

```
int main () {
  char *items[] = {"boat", "car", "truck", "train"};
  int index = get_int();

  printf("You selected %s\n", items[index]);
}
```

Interval Analysis

- Applications 2
  - Detecting index-out-of-bounds

index ∈ [0,3]

```
int main () {
  char *items[] = {"boat", "car", "truck", "train"};
  int index = get_int();
  If (index < 0 || index > 3) return;
  printf("You selected %s\n", items[index]);
}
```

Interval Analysis

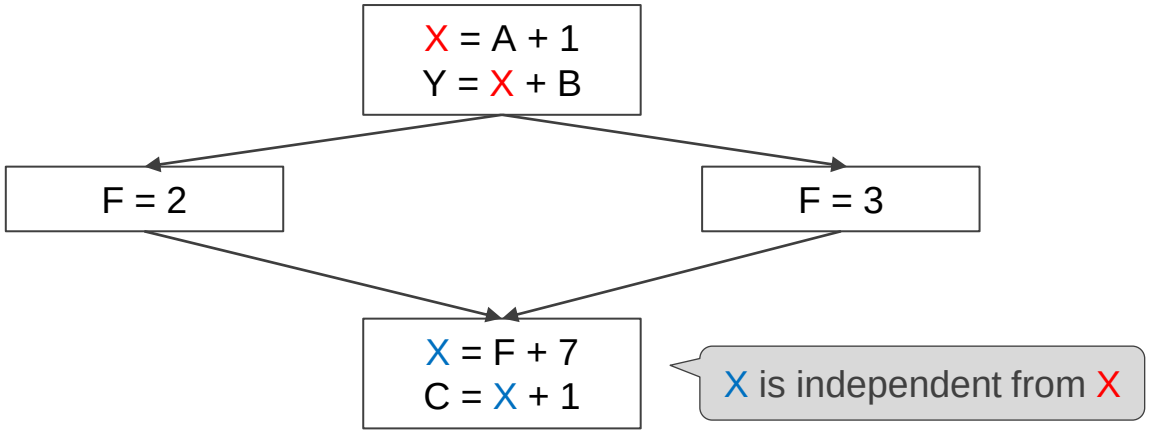
- Applications 3
  - Detecting divide-by-zero

```
int averageResponseTime(int totalTime, int numRequests) {
  return totalTime / numRequests;
}
```

numRequests ∈ [-∞,∞]

Same Variable Name May Be Unrelated

- The values in reused storage locations
  - May be probably independent
- Problem of this situation
  - Unrelated uses of same variable are mixed together
    - This complicates program analysis

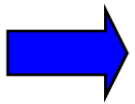


Static Single Assignment (SSA)

- Idea
  - Each variable be assigned exactly once, and every variable be defined before it is used
- Why?
  - Explicitly express different definitions of variables

Original

```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```

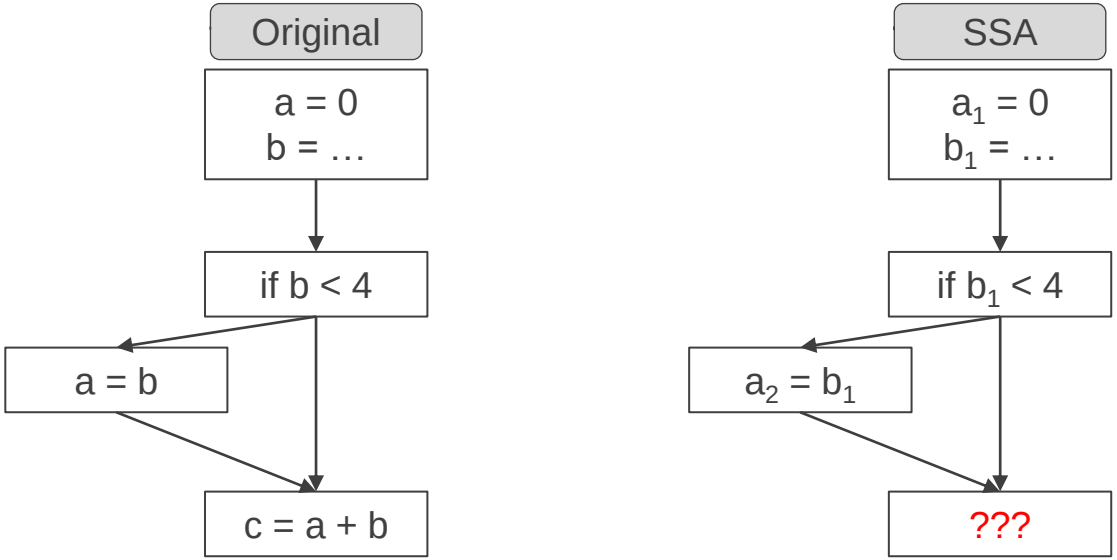


SSA

```
a1 ← x + y
b1 ← a1 + x
a2 ← b1 + 2
c1 ← y + 1
a3 ← c1 + a2
```

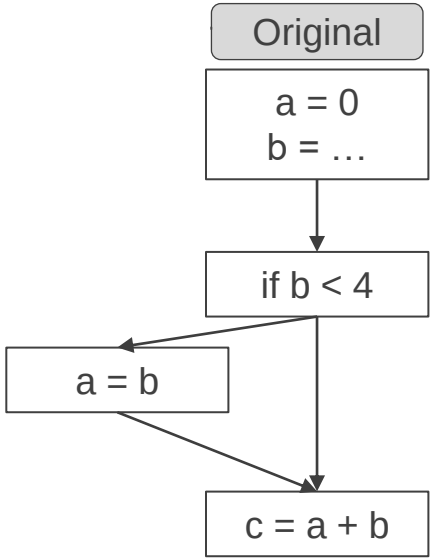
Merge Points (SSA)

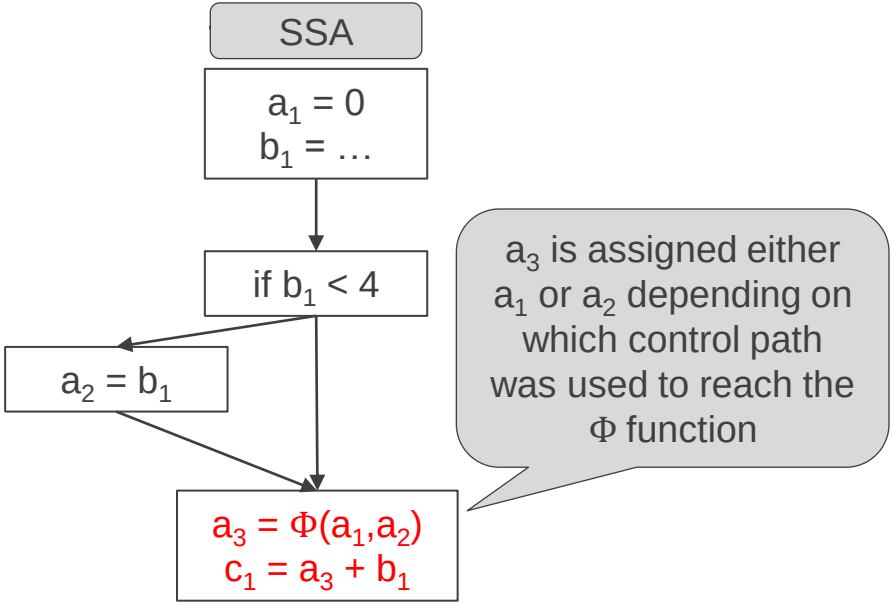
- Issue
  - How to handle merge points in the flowgraph?



Merge Points (SSA)

- Issue
  - How to handle merge points in the flowgraph?
- Solution
  - Φ-function



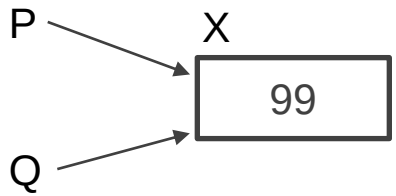


Pointer Analysis

- What memory locations can a pointer expression refer to?
- Alias analysis
  - When do two pointer expressions refer to the same storage location?

```
int X = 99;
P = &X;
Q = P;

*P and *Q alias
```



Pointer Operations in C

- Recall C pointer semantics
  - &a: Address of a
  - *a: Object pointed to by a
  - *(&a) = a: Converse operators

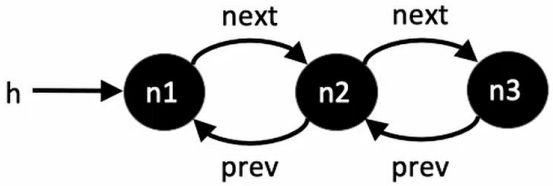
	Referencing • Create location	**Dereferencing read** • Access location • Indirect read	**Dereferencing write** • Access location • Indirect write	**Aliasing** • Copy pointer
C	a = &b	int *b = &c a = *b	int *a = &c *a = b	a = b
JAVA	a = new A()	a = b.f	a.f = b	a = b

Why Is Pointer Analysis Hard?

- Issue
  - There are infinite many ways to express the same data.

```
class Node {
  int data;
  Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```



h.data
h.next.prev.data
h.next.next.prev.prev.data
h.next.prev.next.prev.data

And many more ...

Context Sensitivity

- Consider calling context

```
int foo (int i) {

   return i;
}
…
y1 = foo (1);
y2 = foo (2);
```

With context sensitivity

- More precise
- We have one i per call site of foo
- y1 is 1
- y2 is 2

Without context sensitivity

- Less precise, but faster
- We have one i total
- y1 is {1, 2}
- y2 is {1, 2}

Flow Sensitivity

- Consider control flow and order of execution

```
x = 2;
y = x;
x = 3;
```

With flow sensitivity

- y is 2

Without flow sensitivity

- y is {2, 3}

Path Sensitivity

- Consider properties inferred from order of execution

Line	
1:	x = 0;
2:	if (P) {
3:	x = 1;
4:	}
5:	y = 2;
6:	
7:	If (P) {
8:	y = x;
9:	}

With path sensitivity

- y is {1, 2}
- Records that x = 0 when P = false
- Knows that line 8 is executed only if P = true (i.e., x ≠ 0 at line 8)

Without path sensitivity

- y is {0, 1, 2}
- Less precise

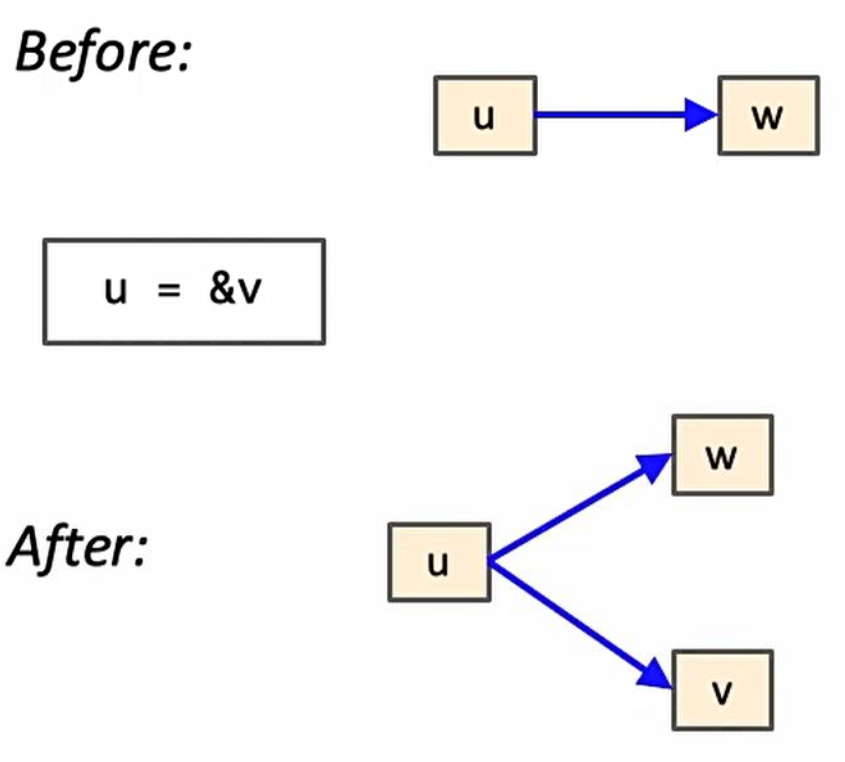
Approximation to the Rescue

- Pointer analysis problem is undecidable
  - We must sacrifice some combinations of
    - Soundness, completeness, termination
- Many sound approximate algorithms for pointer analysis
  - Differ in two key aspects
    - How to abstract the heap
    - How to abstract control-flow

Pointer Analysis Algorithm

- Andersen's Points-To Analysis
  - Asymptotic performance is $O(n^3)$
    - Where 'n' is the number of nodes in the graph
  - Context-insensitive, flow-insensitive, path-insensitive
  - Four collecting rules
    - Referencing
    - Copy
    - Dereferencing (indirect) read
    - Dereferencing (indirect) write

Rule for Referencing



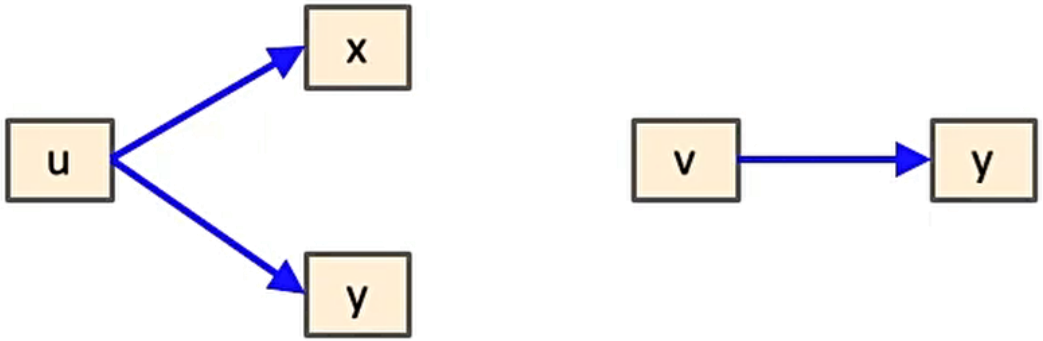

```
if (user_input == true)
   u = &w;
else
    u = &v;
```

Rule for Copy

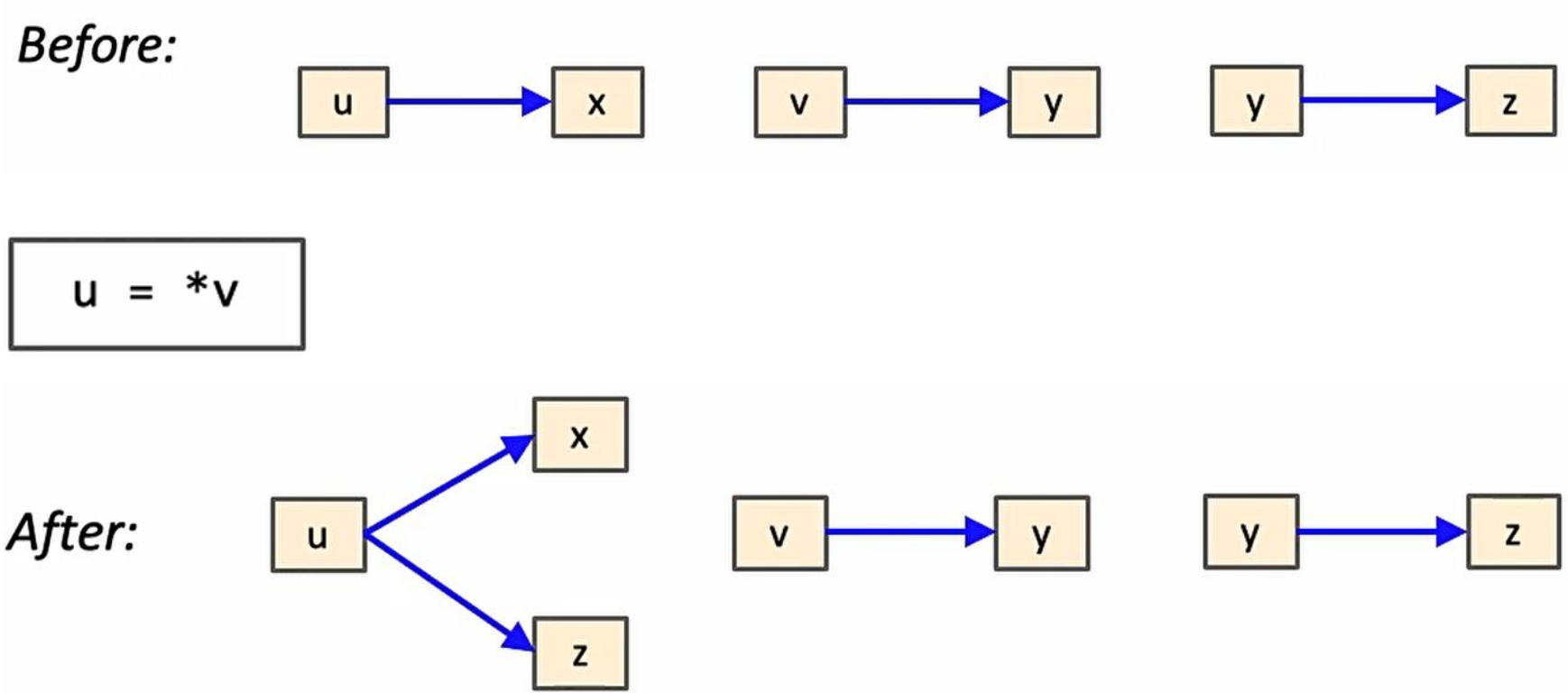
Before:

u = v

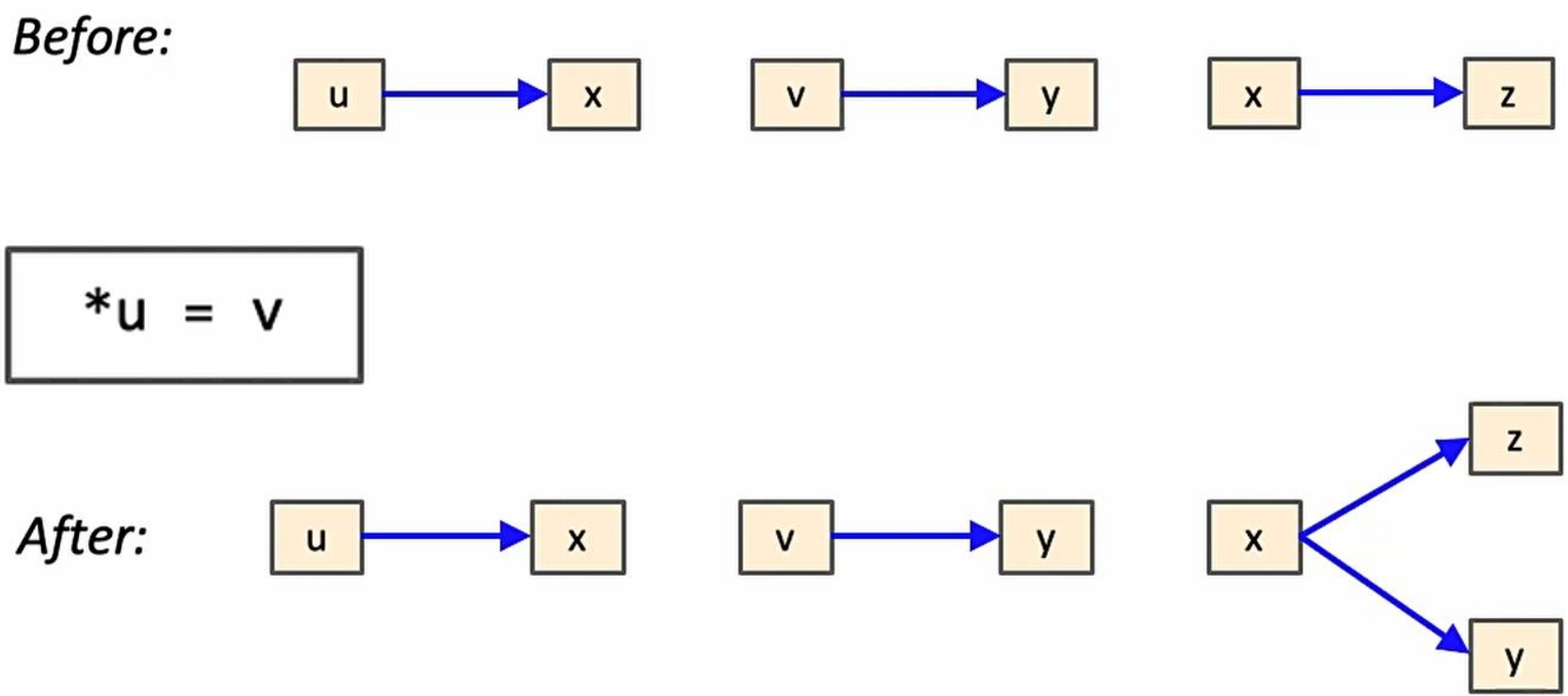
After:



Rule for Indirect Read



Rule for Indirect Write



Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```

Recall: Andersen's Algorithm

graph = empty

repeat:

    for (each statement s in program)

        apply rule corresponding to s on graph

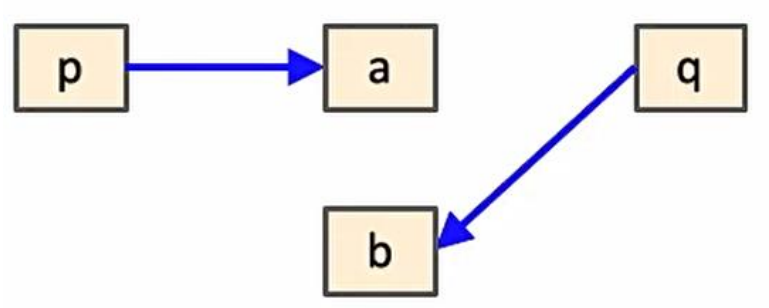
until graph stops changing

Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```

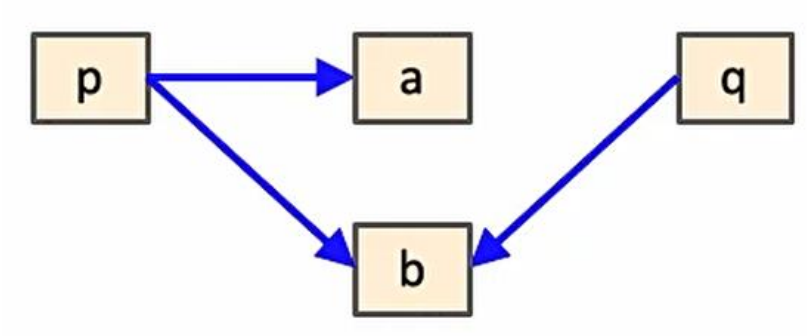
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



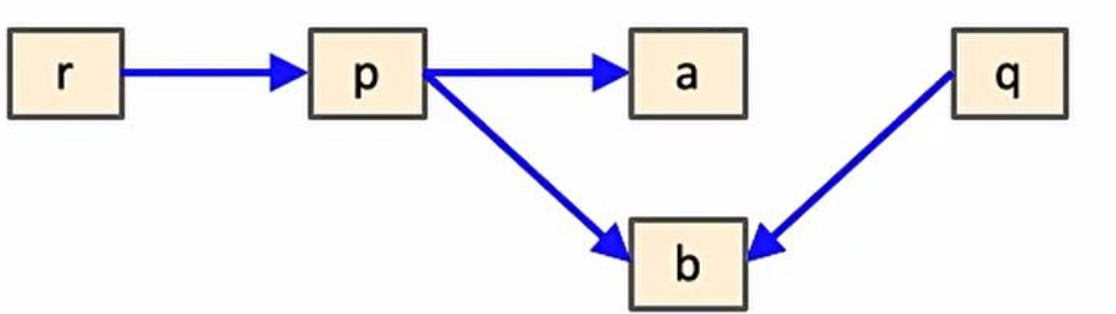
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



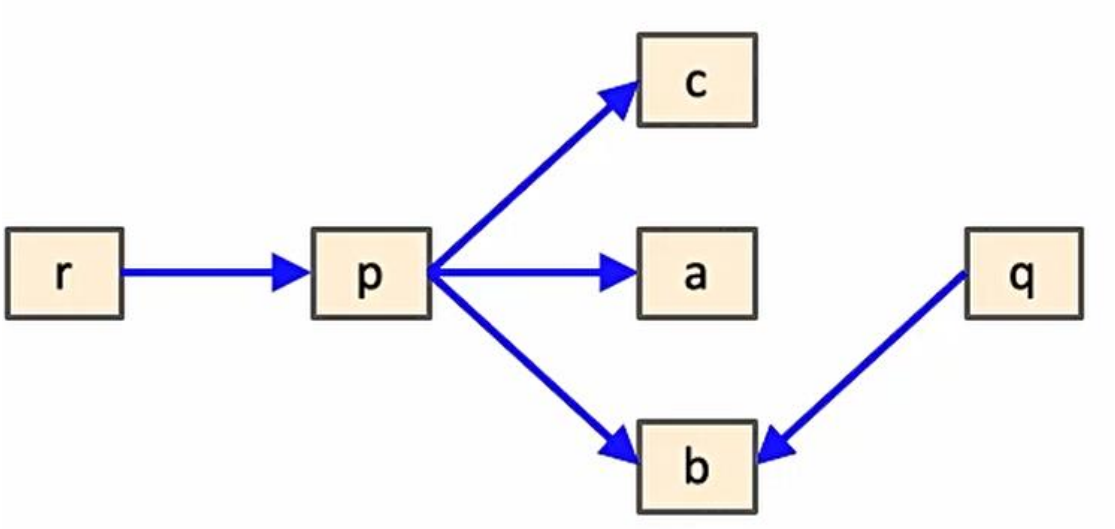
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



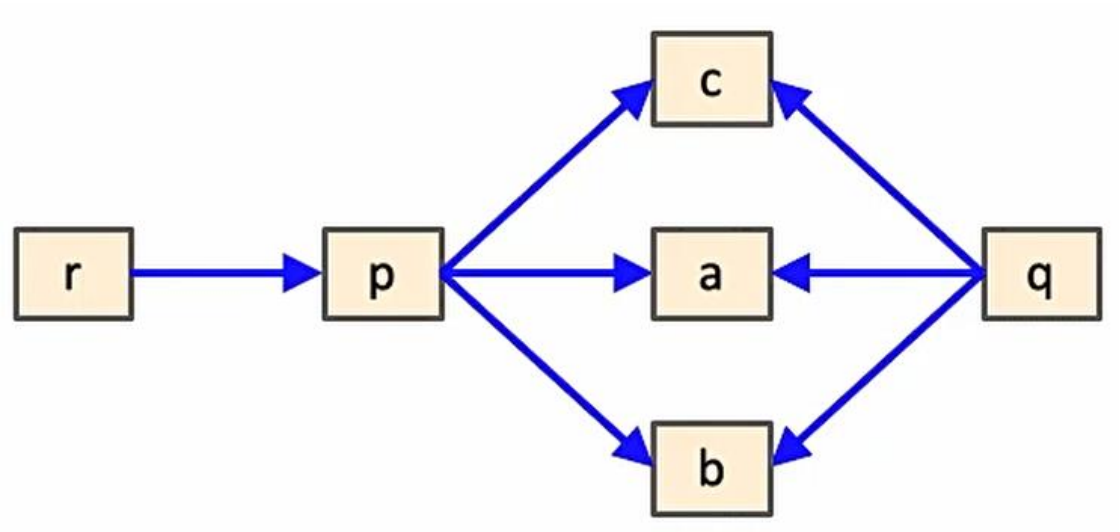
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



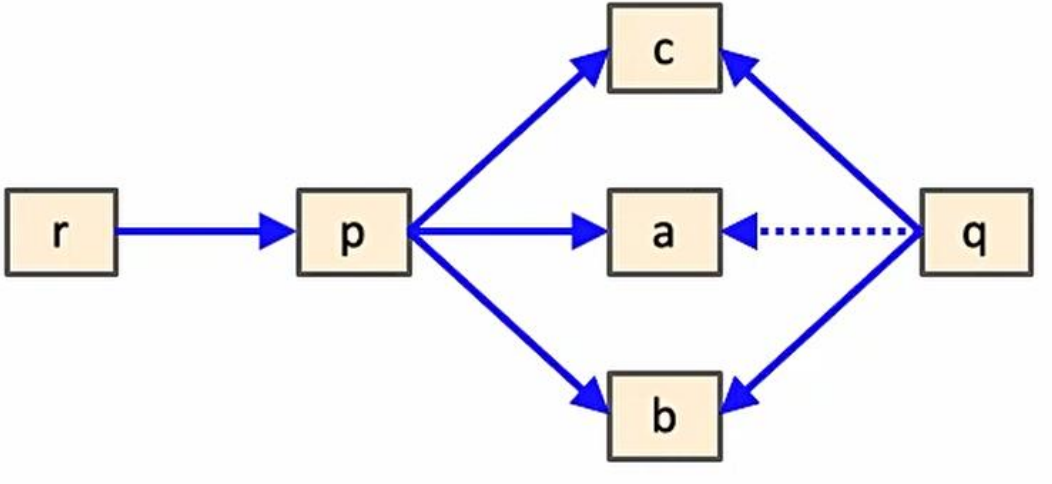
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



Imprecision in Andersen's analysis: **q** never points to **a** in a concrete execution.

Static Analysis Tools

- LLVM
  - To convert a program into a language-independent intermediate representation (IR)
- SVF[1)]
  - Analysis tool for LLVM-based languages
  - Pointer alias analysis
  - Memory SSA form construction
  - Data value-flow tracking

1) https://github.com/SVF-tools/SVF
https://github.com/SVF-tools/SVF-Teaching

Thank you! Questions?

kim2956@purdue.edu