What are programs?

Nick Sumner
wsumner@sfu.ca
What is a program?

- A program *communicates* a set of *instructions* for performing a task.
What is a program?

- A program *communicates* a set of *instructions* for performing a task
  - Do you always communicate in the same way?
What is a program?

- A program *communicates* a set of *instructions* for performing a task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?
What is a program?

- A program *communicates* a set of *instructions* for performing a task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?

- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities
What is a program?

- A program *communicates* a set of *instructions* for performing a task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?

- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities

- **Different programs have different requirements**
  - Performance over all
  - Security!
What is a program?

- A program *communicates* a set of *instructions* for performing a task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?

- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities

- Different programs have different requirements
  - Performance over all
  - Security!

- We cannot reason about programs in only one way
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze.
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze

- **Difficult models:**
  - Compiled binaries

```
1001101
0101011
1101011
0001110
frob.exe
```
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze.

- **Difficult models:**
  - Compiled binaries
    - Difficult to even separate code from data

```
1001101
0101011
1101011
0001110
frob.exe
```
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze

- **Difficult models:**
  - Compiled binaries
    - Difficult to even separate code from data
    - Often used in reverse engineering or security tasks

```
1001
0101
1101
0001110
frob.exe
```
Program Representation

• Before we can reason about programs, we must have a vocabulary and a *model* to analyze

• **Difficult models:**
  – Compiled binaries
    • Difficult to even separate code from data
    • Often used in reverse engineering or security tasks

Why might binaries be good for security tasks?
Program Representation

• Before we can reason about programs, we must have a vocabulary and a *model* to analyze

• **Difficult models:**
  – Compiled binaries
  – Source code
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze

- **Difficult models:**
  - Compiled binaries
  - Source code
    - Very language specific
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze

- **Difficult models:**
  - Compiled binaries
  - Source code
    - Very language specific
    - Relationships can be hard to extract

```
Foo.c
Bar.c
Baz.c
```
Program Representation

● Before we can reason about programs, we must have a vocabulary and a *model* to analyze

● **Difficult models:**
  – Compiled binaries
  – *Source code*
    ● Very language specific
    ● Relationships can be hard to extract
    ● Often used when relating to comments or specs
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze.

- Difficult models:
  - Compiled binaries
  - Source code

- *A good* representation should make explicit the relationships you want to analyze.
Program Representation

Core graph representations for analysis:

1) Abstract Syntax Trees
2) Control Flow Graphs
3) Program Dependence Graphs
4) Call Graphs
5) Points-to Graphs
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
  - *Internal* nodes are operators, statements, etc.

```python
for i in range(5,10):
    a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
  - Internal nodes are operators, statements, etc.
  - Leaves are values, variables, operands

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns

```python
for i in range(5, 10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

- **Used for syntax analysis & transformation:**
  - Simple bug patterns
  - Style checking

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns
  - Style checking
  - Refactoring

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- **Used for syntax analysis & transformation:**
  - Simple bug patterns
  - Style checking
  - Refactoring
  - Training prediction/completion models

```python
for i in range(5,10):
a[i] = i * 5
```
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns
  - Style checking
  - Refactoring
  - Training prediction/completion models

But the same program may still be spelled many ways.
2) Control Flow Graphs

- Express the possible decisions and possible paths through a program

```python
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```
2) Control Flow Graphs

- Express the possible decisions and possible paths through a program

```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```
2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
  - *Basic Blocks* (Nodes) are straight line code

```python
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```

```
cond = ...
if cond:
    a = foo()
else:
    a = bar()
print(a)
```
2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
  - *Basic Blocks* (Nodes) are straight line code
  - *Edges* show how decisions can lead to different basic blocks

```python
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```
2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
  - **Basic Blocks** (Nodes) are straight line code
  - **Edges** show how decisions can lead to different basic blocks
  - **Paths** through the graph are potential paths through the program

```python
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```python
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```

The 'while' is gone

```
sum = 0
i = 1
if i < N
    i = i + 1
    sum = sum + i
print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```python
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```python
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```python
def compute_sum(N):
    sum = 0
    i = 1
    while i < N:
        i = i + 1
        sum = sum + i
    print(sum)
```
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```

Why is the 'if' in a separate block?
2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```plaintext
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```

What would the CFG of the equivalent 'for' look like?
3) Program Dependence Graph (PDG)

- A *Program Dependence Graph* captures how instructions can influence each other
3) Program Dependence Graph (PDG)

- A *Program Dependence Graph* captures how instructions can influence each other
- Instruction X depends on Y if Y *can influence* X
3) Program Dependence Graph (PDG)

- A **Program Dependence Graph** captures how instructions can influence each other.

- **Instruction X depends on Y if Y can influence X**
  - Nodes are instructions
  - An edge $Y \rightarrow X$ shows that $Y$ influences $X$
3) Program Dependence Graph (PDG)

- A *Program Dependence Graph* captures how instructions can influence each other
- Instruction X depends on Y if Y *can influence* X
- 2 main types of influence:
  - Data dependence
  - Control dependence
Data Dependence

X data depends on Y if
- There exists a path from Y to X in the CFG
Data Dependence

X data depends on Y if
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X
Data Dependence

X data depends on Y if

- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X
Data Dependence

X data depends on Y if

- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X
Data Dependence

X data depends on Y if
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X

1) \( a = \ldots \)
2) \( b = \ldots \)
\[ \quad \]
3) \( a = \ldots \)
4) \( c = a \)
\[ \quad \]
\[ a = \ldots \quad b = \ldots \]
\[ \ldots = b + a \]
Data Dependence

X data depends on Y if

- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X
Control Dependence

Preliminary: X dominates Y if
• every path from the entry node to Y passes X
  – strict, normal, & immediate dominance
Control Dependence

Preliminary: X dominates Y if
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

Entry

X

Y
Control Dependence

Preliminary: X dominates Y if
• every path from the entry node to Y passes X
  – strict, normal, & immediate dominance
Control Dependence

Preliminary: X dominates Y if
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)

DOM(6) = ?  IDOM(6) = ?
Control Dependence

Preliminary: X dominates Y if
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

```
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```

DOM(6)={1,2,3,6}  IDOM(6)= ?
Control Dependence

Preliminary: X dominates Y if
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

```plaintext
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```

DOM(6)={1,2,3,6}  IDOM(6)=3
Control Dependence

Preliminary: X dominates Y if
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

```
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```

DOM(6)={1,2,3,6}  IDOM(6)=3
Control Dependence

Preliminary: X post dominates Y if
• every path from the Y to exit passes X
  – strict, normal, & immediate dominance
Control Dependence

Preliminary: X post dominates Y if
- every path from the Y to exit passes X
  - strict, normal, & immediate dominance

1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)

PDOM(5) = ?    IPDOM(5) = ?
Control Dependence

Preliminary: X *post* dominates Y if
- every path from the Y *to exit* passes X
  - strict, normal, & immediate dominance

\[
PDOM(5)=\{3,5,6\} \quad IPDOM(5)=?\]

```
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```
Control Dependence

Preliminary: X post dominates Y if

- every path from the Y to exit passes X
  - strict, normal, & immediate dominance

1) $\text{sum} = 0$
2) $i = 1$
3) while $i < N$:
   4) $i = i + 1$
   5) $\text{sum} = \text{sum} + i$
   6) $\text{print}(\text{sum})$

\[ \text{PDOM}(5) = \{3, 5, 6\} \quad \text{IPDOM}(5) = 3 \]
Control Dependence

Preliminary: X *post* dominates Y if
- every path from the Y to exit passes X
  - strict, normal, & immediate dominance

```plaintext
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```

PDOM(5)=\{3,5,6\}  IPDOM(5)=3
Control Dependence (Finally)

Y is control dependent on X iff
Control Dependence (Finally)

Y is control dependent on X iff

- **Definition 1:**
  
  X directly decides whether Y executes
Control Dependence (Finally)

Y is control dependent on X iff

- **Definition 1:**
  X directly decides whether Y executes

- **Definition 2:**
  - There exists a path from X to Y s.t. Y post dominates every node between X and Y.
  - Y does not strictly post dominate X
Control Dependence (Finally)

Y is control dependent on X iff

- **Definition 1:**
  X directly decides whether Y executes

- **Definition 2:**
  - There exists a path from X to Y s.t. Y post dominates every node between X and Y.
  - Y does not strictly post dominate X
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

1) sum = 0  
2) i = 1  
3) while i < N:  
4) i = i + 1  
5) sum = sum + i  
6) print(sum)
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```python
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
```

What is CD(5)? CD(3)
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```python
1) sum = 0
2) i = 1
3) while i < N:
   4)   i = i + 1
   5)   if 0 == i%2:
   6)     continue
   7)   sum = sum + i
8) print(sum)
```
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```python
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) if 0 == i%2:
      6) continue
   7) sum = sum + i
8) print(sum)
```
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```
1) sum = 0
2) i = 1
3) while i < N:
   4) i = i + 1
   5) if 0 == i%2:
      6) continue
   7) sum = sum + i
8) print(sum)
```

What is CD(7)?
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

1) if X or Y:
2) print(X)
3) print(Y)

What is CD(2)?
Control Dependence

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

1) if X or Y:
   2) print(X)
   3) print(Y)

What is CD(2)?

Diagram:

1A) if X:
   1B) if Y:
   2) print(X)
   3) print(Y)
3) Program Dependence Graph (PDG)

The PDG is the combination of:
- The control dependence graph
- The data dependence graph
3) Program Dependence Graph (PDG)

The PDG is the combination of

- The control dependence graph
- The data dependence graph

Recall: Edges identify potential influence
3) Program Dependence Graph (PDG)

The PDG is the combination of

- The control dependence graph
- The data dependence graph

Recall: Edges identify potential influence

- **Debugging:** What may have caused a bug?
3) Program Dependence Graph (PDG)

The PDG is the combination of
- The control dependence graph
- The data dependence graph

Recall: Edges identify potential influence

- **Debugging**: What may have caused a bug?
- **Security**: Can sensitive information leak?
3) Program Dependence Graph (PDG)

The PDG is the combination of
- The control dependence graph
- The data dependence graph

Recall: Edges identify *potential influence*

- **Debugging**: What may have caused a bug?
- **Security**: Can sensitive information leak?
- **Testing**: How can I reach a statement?
- ...

...
4) Call Graph (Multigraph)

- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls
4) Call Graph (Multigraph)

- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls
4) Call Graph (Multigraph)

- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls

```
foo()  bar()  baz()
```

```
foo calls bar & baz
bar calls bam
```
4) Call Graph (Multigraph)

- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls

What does this capture?
4) Call Graph (Multigraph)

- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls

How should we handle function pointers?
5) Points-to Graphs

Pointers / indirection create two difficult problems:
5) Points-to Graphs

Pointers / indirection create two difficult problems:

- **Aliasing**
  - Multiple variables may denote the same memory location
5) Points-to Graphs

Pointers / indirection create two difficult problems:

- **Aliasing**
  - Multiple variables may denote the same memory location

- **Ambiguity**
  - One variable may potentially denote several different targets in memory.
5) Points-to Graphs

Pointers / indirection create two difficult problems:

- **Aliasing**
  - Multiple variables may denote the same memory location

- **Ambiguity**
  - One variable may potentially denote several different targets in memory.

```c
x.lock()  
...  
y.unlock()  
x = password  
...  
broadcast(y)
```
5) Points-to Graphs

Points-to graphs capture this points-to relation
5) Points-to Graphs

Points-to graphs capture this points-to relation

- The relation \((p,x)\) where \(p\) MAY/MUST point to \(x\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
• The relation \((p,x)\) where \(p\) MAY/MUST point to \(x\)
  – Both MAY and MUST information can be useful
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation \((p, x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...:
5) \(q = p\)
6) \(r.f = t\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation \((p, x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...:
5) \(q = p\)
6) \(r.f = t\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation \((p,x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...:
5) \(q = p\)
6) \(r.f = t\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation \((p,x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...
5) \(q = p\)
6) \(r.f = t\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

1) \( r = C() \)
2) \( p.f = r \)
3) \( t = C() \)
4) if ...:
5) \( q = p \)
6) \( r.f = t \)
5) Points-to Graphs

Points-to graphs capture this points-to relation

- The relation \((p, x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...:
5) \(q = p\)
6) \(r.f = t\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation \((p, x)\) where \(p\) MAY/MUST point to \(x\)
  - Both MAY and MUST information can be useful

1) \(r = C()\)
2) \(p.f = r\)
3) \(t = C()\)
4) if ...:
5) \(q = p\)
6) \(r.f = t\)

\(\text{p.f.f MUST ALIAS t}\)
5) Points-to Graphs

Points-to graphs capture this points-to relation
- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

1) r = C()
2) p.f = r
3) t = C()
4) if ...:
5) q = p
6) r.f = t

\[\text{p.f.f MUST ALIAS t}\]
\[\text{q MAY ALIAS p}\]
Execution Representations

- *Program* representations are *static*
  - All possible program behaviors at once
  - Usually projected onto the CFG
Execution Representations

- **Program** representations are *static*
  - All possible program behaviors at once
  - Usually projected onto the CFG

- **Execution** representations are *dynamic*
  - Only the behavior of a single real execution
Execution Representations

- **Program** representations are *static*
  - All possible program behaviors at once
  - Usually projected onto the CFG

- **Execution** representations are *dynamic*
  - Only the behavior of a single real execution
  - Multiple instances of an instruction occur multiple times
Control Flow Trace

1) `sum = 0`
2) `i = 1`
3) `if i < N`
4) `i = i + 1`
5) `sum = sum + i`
6) `print(sum)`

All Equivalent

1) `sum = 0`
2) `i = 1`
3) `if i < N`
4) `i = i + 1`
5) `sum = sum + i`
6) `print(sum)`
Control Flow Trace

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)

All Equivalent

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)
1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)

TTF

1 2 3 4 5 1 3 2 4 2 5 2 3 3 6 1
1 3 1 4 1 3 2 4 2 3 3 6 1

All Equivalent

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

1) \text{sum} = 0
2) \text{i} = 1
3) \text{if } \text{i} < N
4) \text{i} = \text{i} + 1
5) \text{sum} = \text{sum} + \text{i}
6) \text{print}(\text{sum})
Dynamic Dependence Graph

1) \( \text{sum} = 0 \)
2) \( i = 1 \)
3) if \( i < N \)
   
   4) \( i = i + 1 \)
   5) \( \text{sum} = \text{sum} + i \)
6) print(\text{sum})
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

1) \( \text{sum} = 0 \)
2) \( i = 1 \)

3) if \( i < N \)

4) \( i = i + 1 \)
5) \( \text{sum} = \text{sum} + i \)

6) \text{print}(\text{sum})
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
   4) i = i + 1
   5) sum = sum + i
6) print(sum)
Dynamic Dependence Graph

Notably a bit difficult for people to wade through.
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
6) print(sum)

If only we could focus on the parts that interest us...

Notably a bit difficult for people to wade through.
Dynamic Dependence Graph

1) $\text{sum} = 0$
2) $i = 1$
3) if $i < N$
4) $i = i + 1$
5) $\text{sum} = \text{sum} + i$
6) print($\text{sum}$)

Slicing (static or dynamic) computes a transitive closure of dependences
Dynamic Dependence Graph

1) sum = 0
2) i = 1
3) if i < N
   4) i = i + 1
   5) sum = sum + i
6) print(sum)

1) sum = 0
2) i = 1
3) if i < N
   4) i = i + 1
   5) sum = sum + i

Slicing (static or dynamic) computes a transitive closure of dependences

Note: potential influences are missed!
Dynamic Dependence Graphs

Capture a notion of observed influence
Dynamic Dependence Graphs

Capture a notion of *observed* influence

- **Debugging**: What caused a bug?
Dynamic Dependence Graphs

Capture a notion of *observed* influence

- **Debugging**: What caused a bug?
- **Security**: How did sensitive information leak?
Dynamic Dependence Graphs

Capture a notion of *observed* influence

- **Debugging**: What caused a bug?
- **Security**: How did sensitive information leak?
- **Testing**: What tests need to be run based on a change?
- ...


Dynamic Dependence Graphs

Capture a notion of *observed* influence

- **Debugging:** What caused a bug?
- **Security:** How did sensitive information leak?
- **Testing:** What tests need to be run based on a change?
- ...

Prioritizing, pruning, & bundling information is often critical when applying slicing
Summary

- Different tasks may benefit from representing programs in different ways
- Thinking of the right representation for the task you have is important