Secure Information Flow

Program Slicing

Winter Term 2014/15

Advanced Lecture (9 CP)

Christian Hammer

Program Slicing in the PDG

- Which statements can influence the the slicing criterion?
- Defined slightly different from Weiser's original
- Slicing criterion is just a node v in the dependence graph
- Equivalent to criterion $(v, \text{Ref}(v) \cup \text{Def}(v))$

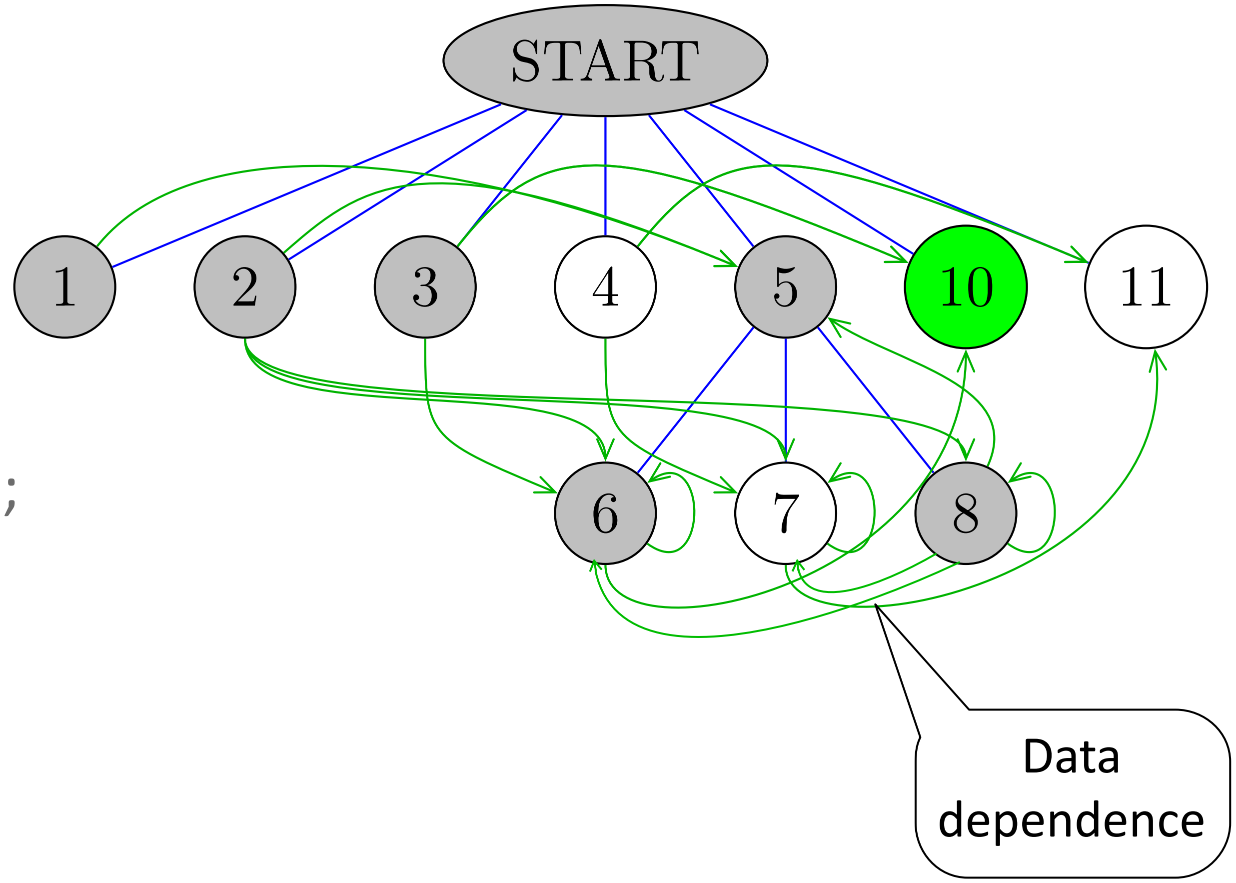
- Intraprocedural Backward Slice:
  $\text{BS}(v) = \{x \in \text{PDG} \mid x \rightarrow^* v\}$
- Simple graph-reachability problem based on transitivity of data and control dependence

Example for Backwards Slice

```
read(n);
i = 1;
sum = 0;
prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i++;
}
write(sum);
write(prod);
```

Example for Backwards Slice

```
(1)  read(n);
(2)  i = 1;
(3)  sum = 0;
(4)  prod = 1;
(5)  while (i <= n) {
(6)    sum = sum + i;
(7)    prod = prod * i;
(8)    i++;
(9)  }
(10) write(sum);
(11) write(prod);
```

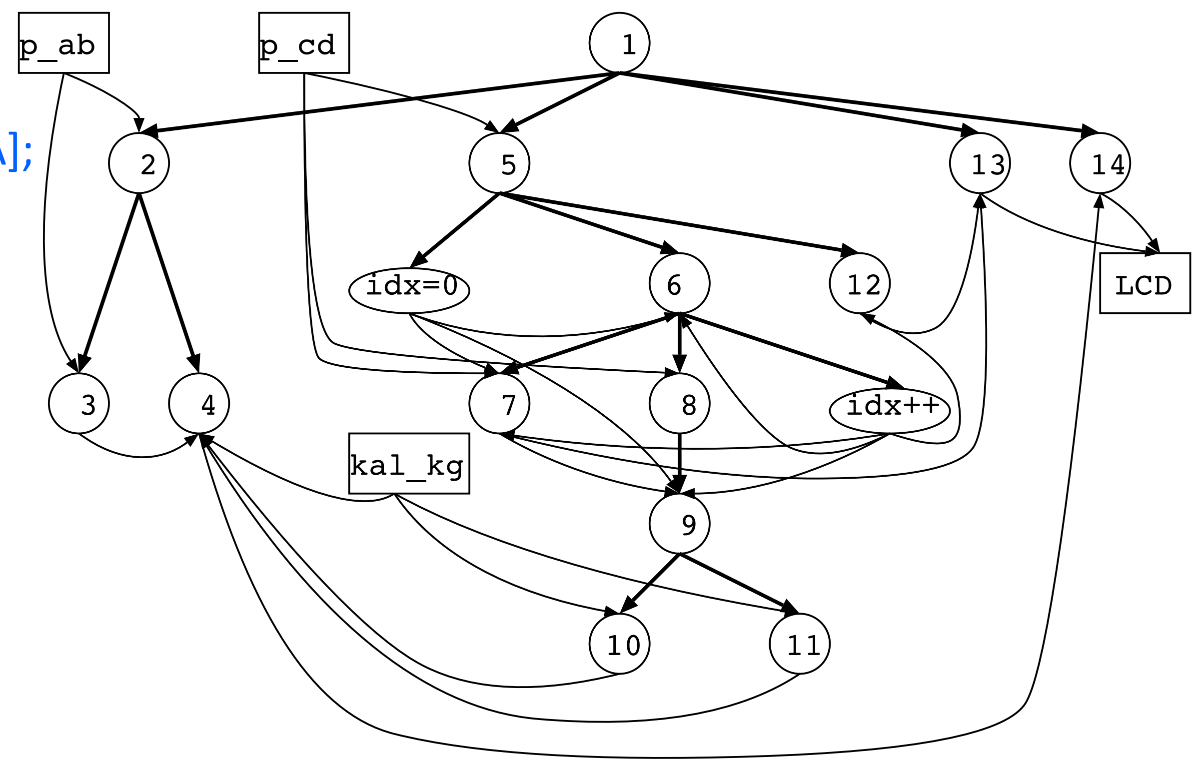


Correctness of Slicing

- Theorem:
  A backward slice from a node v contains all the statements that could potentially influence the computation of variables defined or used at v.
- Proof: Wasserrab PhD, formalized in Isabelle

Scale

```
(1)  while(TRUE) {
(2)    if ((p_ab[CTRL2] & 0x10)==0) {
(3)      u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)      u_kg = u * kal_kg;}
(5)    if ((p_cd[CTRL2] & 0x01) != 0) {
(6)      for (idx=0;idx<7;idx++) {
(7)        e_puf[idx] = p_cd[PA];
(8)        if ((p_cd[CTRL2] & 0x10) != 0) {
(9)          switch(e_puf[idx]) {
(10)           case '+': kal_kg *= 1.1; break;
(11)           case '-': kal_kg *= 0.9; break; } } }
(12)       e_puf[idx] = '\0'; }
(13)   printf("Artikel: %07.7s\n",e_puf);
(14)   printf(" %6.2f kg ",u_kg);
(15) }
```



Scale

```
while(TRUE) {
  if ((p_ab[CTRL2] & 0x10)==0) {
    u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
    u_kg = u * kal_kg;}
  if ((p_cd[CTRL2] & 0x01) != 0) {
    for (idx=0;idx<7;idx++) {
      e_puf[idx] = p_cd[PA];
      if ((p_cd[CTRL2] & 0x10) != 0) {
        switch(e_puf[idx]) {
          case '+': kal_kg *= 1.1; break;
          case '-': kal_kg *= 0.9; break; } } }
      e_puf[idx] = '\0'; }
  printf("Artikel: %07.7s\n",e_puf);
  printf(" %6.2f kg ",u_kg);
}
```

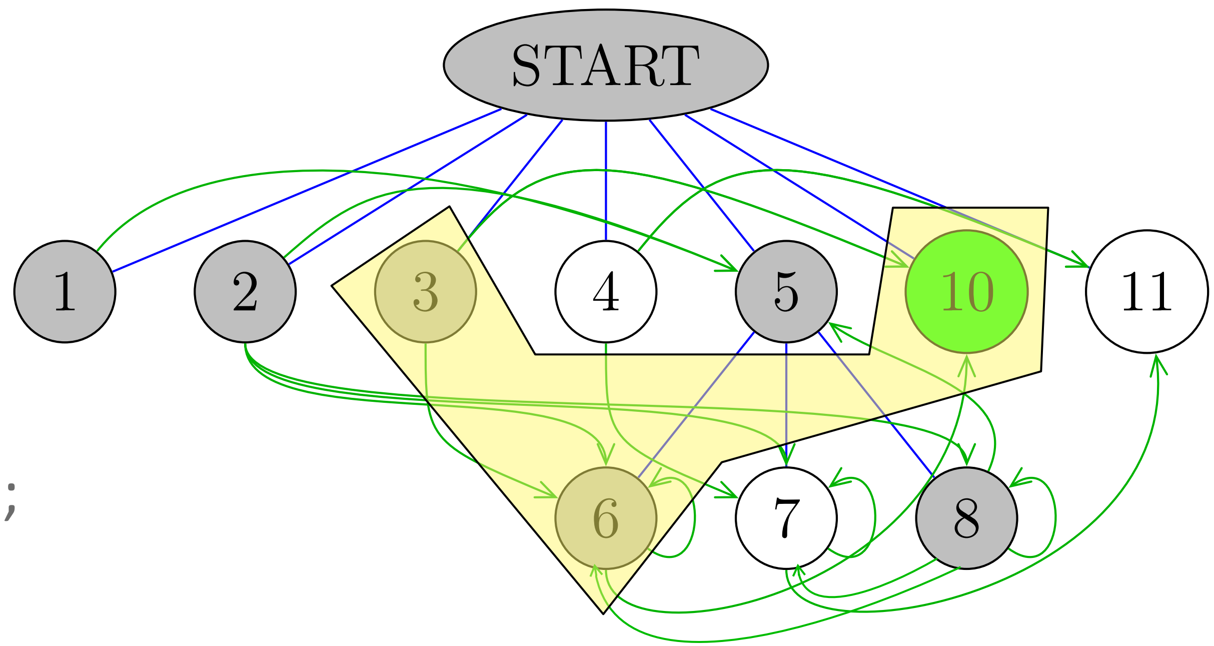
Calibration factor can be changed!

Forward Slicing

- Backward slicing: which statements can influence the the slicing criterion?
- Forward slicing: which statement can be influenced by the slicing criterion?
- Intraprocedural Forward Slice:
  $\mathrm{FS}(v) = \{x \in \mathrm{PDG} \mid v \to^* x\}$
- What is transitively reachable from the slicing criterion?

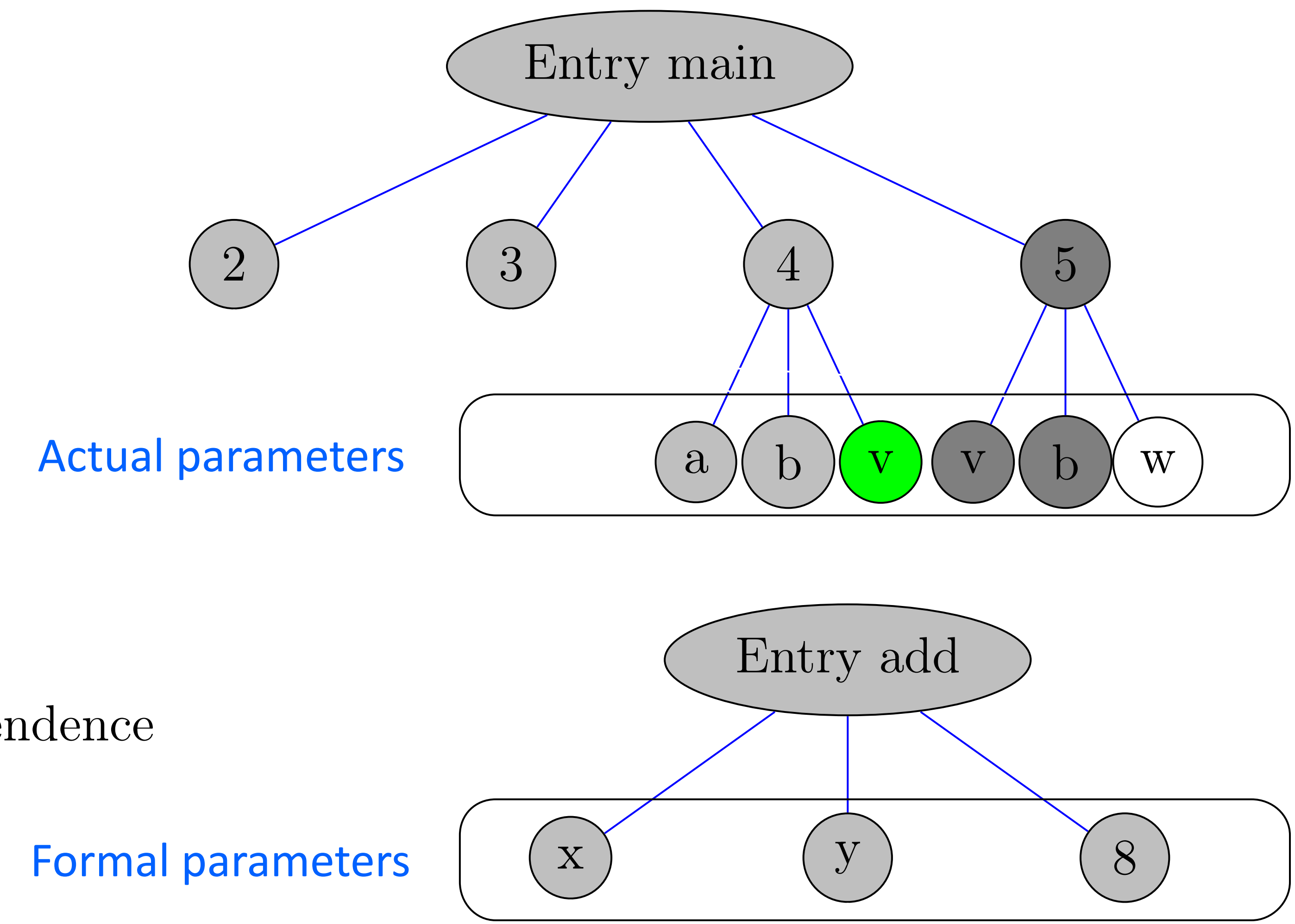
Example

```
(1)   read(n);
(2)   i = 1;
(3)   sum = 0;
(4)   prod = 1;
(5)   while (i <= n) {
(6)     sum = sum + i;
(7)     prod = prod * i;
(8)     i++;
(9)   }
(10)  write(sum);
(11)  write(prod);
```



Interprocedural Analysis

```
main() {
  a=3;
  b=4;
  v=add(a,b);
  w=add(v,b);
}
add(x,y) {
  return x+y;
}
```

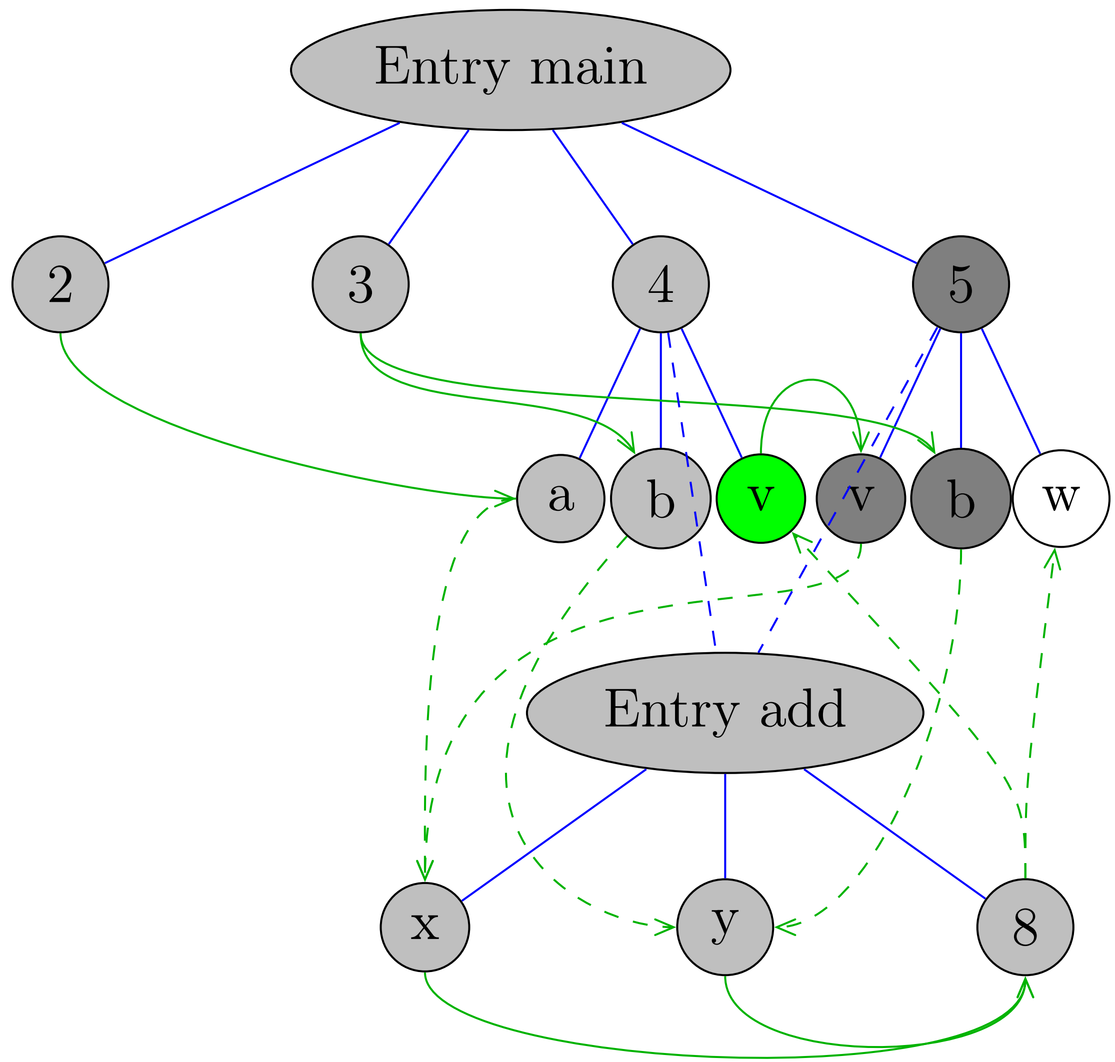


control dependence

Interprocedural Analysis: Parameter Passing

```
main() {
  a=3;
  b=4;
  v=add(a,b);
  w=add(v,b);
}
add(x,y) {
  return x+y;
}
```

data dependence
control dependence
parameter edges
call dependence



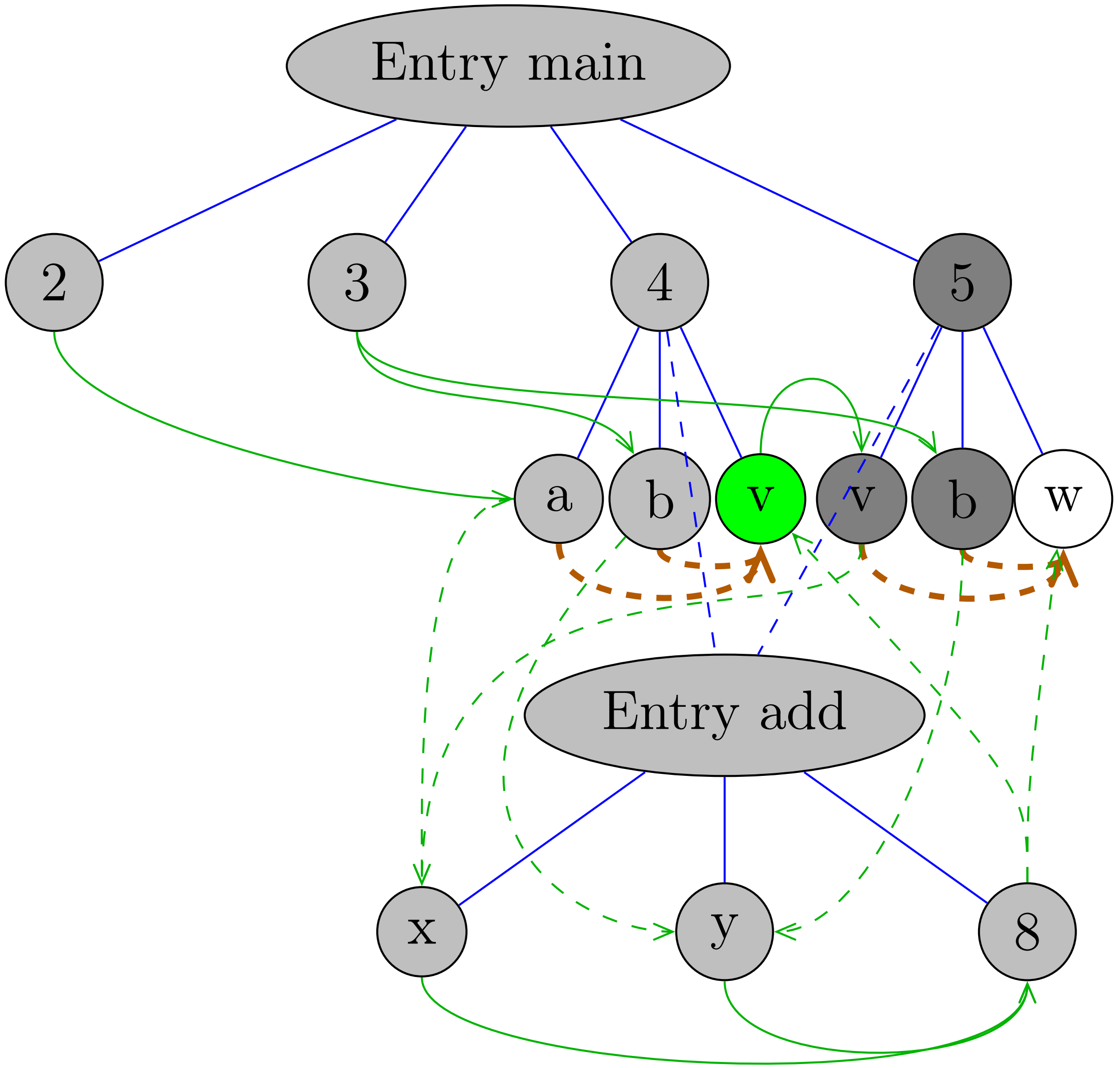
Interprocedural Slicing

- Backward slicing from v would contain the whole graph except for w.
- But 5 and its children are not influencing the definition of v in line 4.
- This is called context-insensitive program slicing.
- It may contain spurious nodes (imprecise, in dark grey)
- Idea: only return to same call site where we left the method

Interprocedural Analysis: Summary Edges

```
main() {
  a=3;
  b=4;
  v=add(a,b);
  w=add(v,b);
}
add(x,y) {
  return x+y;
}
```

- data dependence
- control dependence
- summary edge
- parameter edges
- call dependence

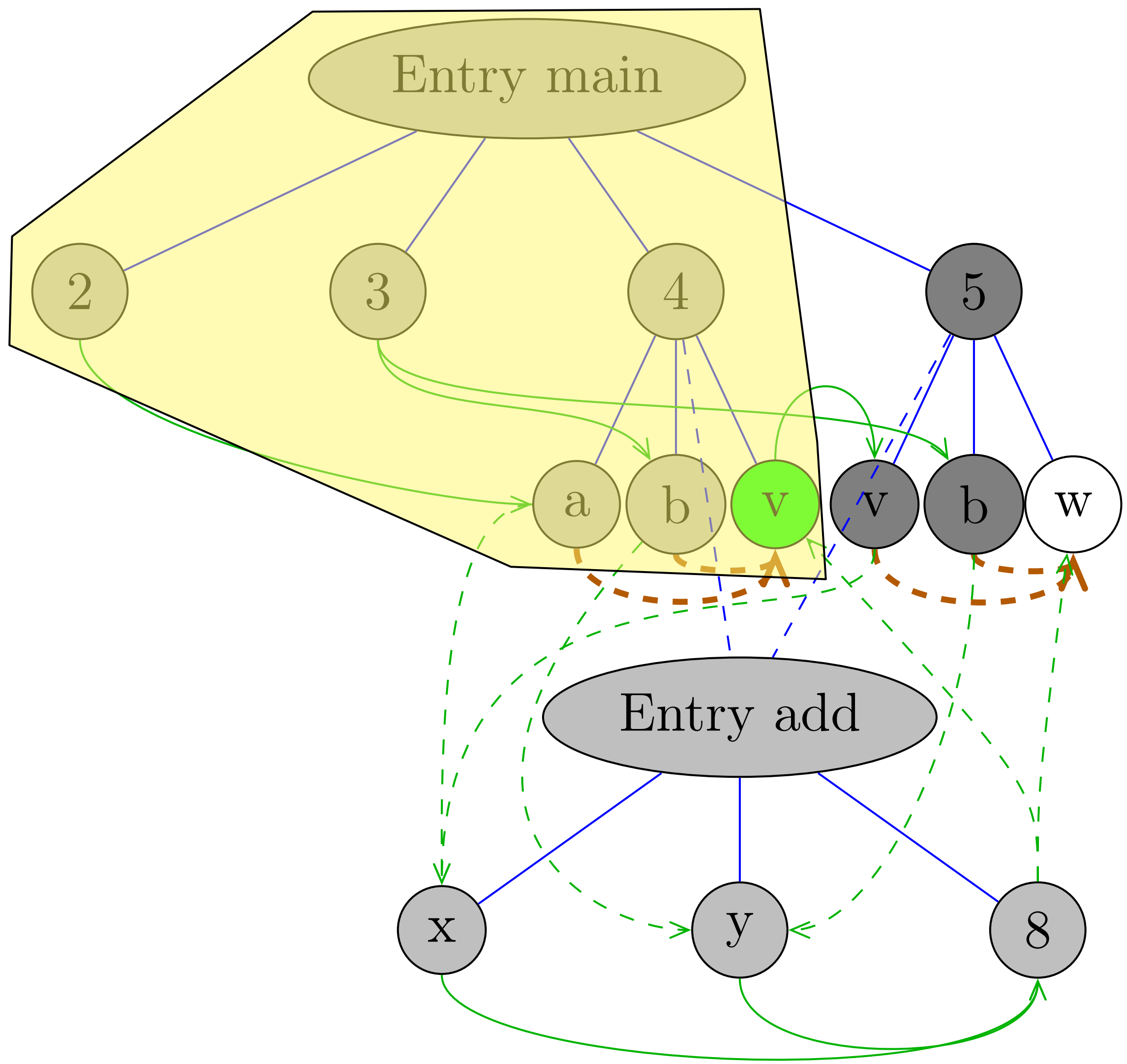


Two-Phase Slicing

- In the first phase:
  Do not descend into called methods, mark omitted edges for later phase. Traverse summary edges instead.
- In the second phase:
  Starting with the omitted edges, do not reascend into calling method. Still traverse summary edges.

Interprocedural Analysis

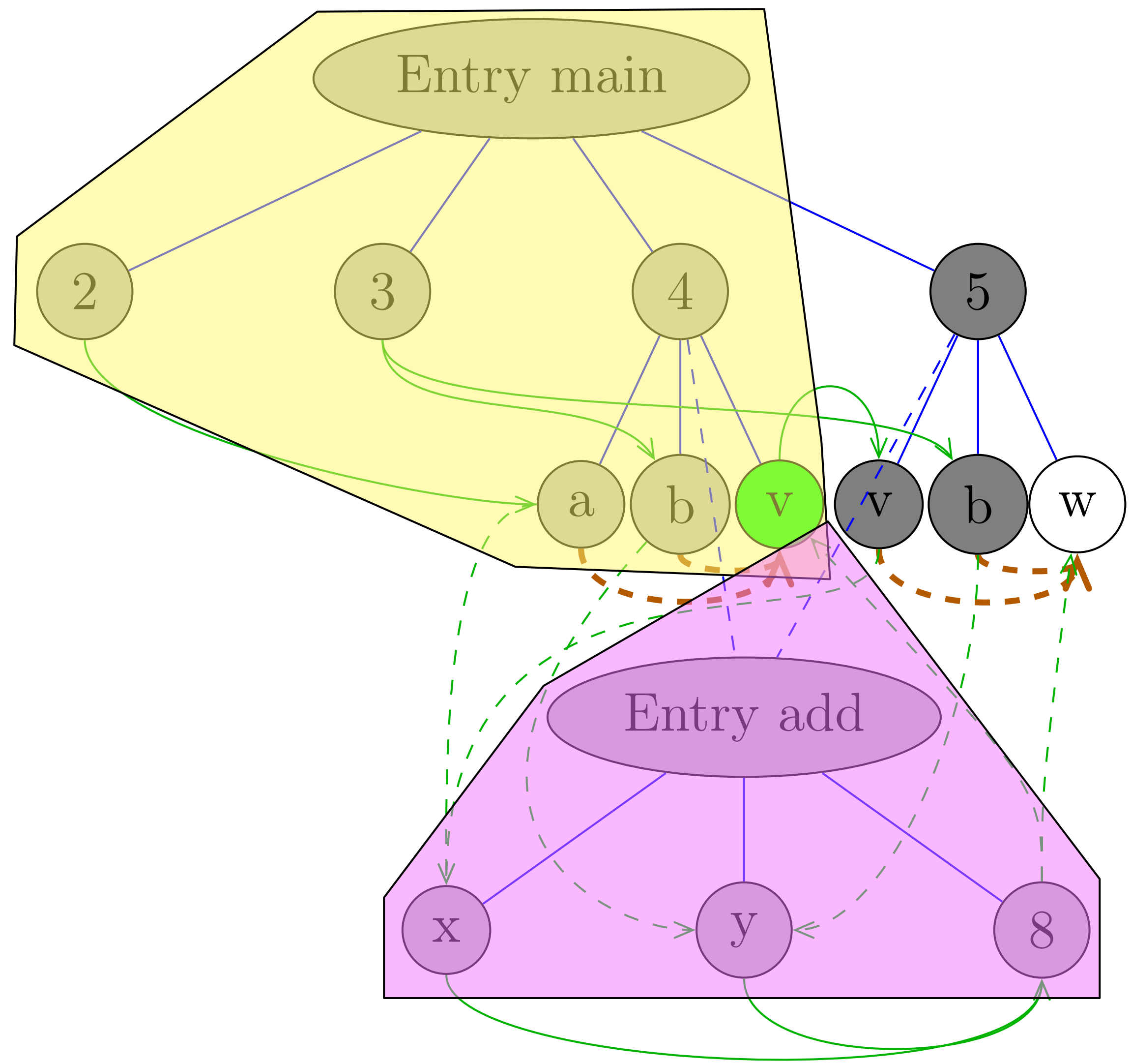
```
main() {
  a=3;
  b=4;
  v=add(a,b);
  w=add(v,b);
}
add(x,y) {
  return x+y;
}
```



- data dependence
- control dependence
- summary edge
- parameter edges
- call dependence

Interprocedural Analysis

```
main() {
  a=3;
  b=4;
  v=add(a,b);
  w=add(v,b);
}
add(x,y) {
  return x+y;
}
```



- data dependence
- control dependence
- summary edge
- parameter edges
- call dependence

CISPA Center for IT-Security, Privacy and Accountability

$W = \emptyset$, *worklist*
$P = \emptyset$
foreach $n \in N$ which is a formal-out node **do**
  $W = W \cup \{(n, n)\}$
  $P = P \cup \{(n, n)\}$
Iteration
while $W \neq \emptyset$ *worklist is not empty* **do**
  $W = W / \{(n, m)\}$ *remove one element from the worklist*
  if n is a formal-in node **then**
    foreach $n' \xrightarrow{pi} n$ which is a parameter-in edge **do**
      foreach $m \xrightarrow{po} m'$ which is a parameter-out-edge **do**
        if n' and m' belong to the same call site **then**
          $E = E \cup n' \xrightarrow{su} m'$ *add a new summary edge*
          foreach $(m', x) \in P \wedge (n', x) \notin P$ **do**
            $P = P \cup \{(n', x)\}$
            $W = W \cup \{(n', x)\}$
  else
    foreach $n' \xrightarrow{dd,cd,su} n$ **do**
      if $(n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
return G *the SDG*

Example Program

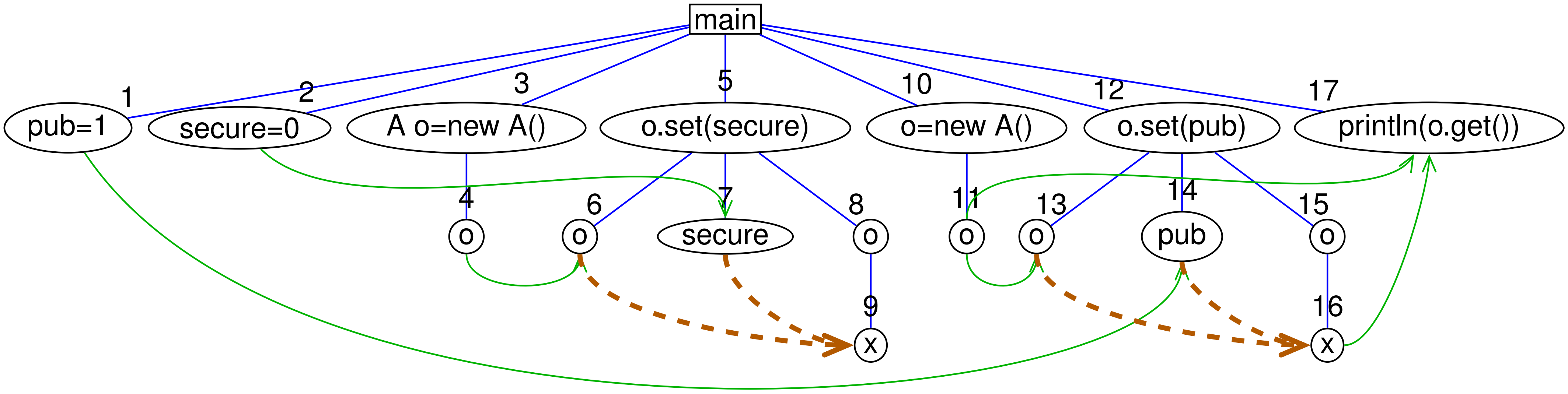
```
class A {
  int x;
  void set() { x = 0; }
  void set(int i) { x = i; }
  int get() { return x; }
}
class B extends A {
  void set() { x = 1; }
}

class InfFlow {
 static void main(String[] a) {
   // 1: no information flow
   int sec = 0, pub = 1;
   A o = new A();
   o.set(sec);
   o = new A();
   o.set(pub);
   System.out.println(o.get());

   // 2: dynamic dispatch
   if (sec==0 && a[0].equals("007"))
     o = new B();
   o.set();
   System.out.println(o.get());

   // 3: instanceof
   o.set(42);
   System.out.println(o instanceof B);
}}
```

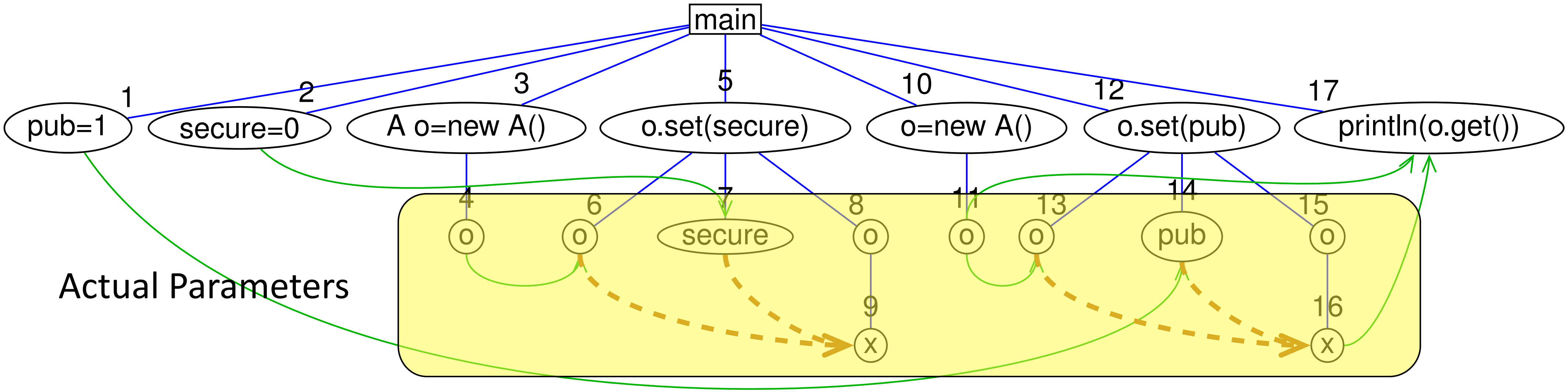
SDG for first part of the Program



- Summary edges for transitive flow of parameters
- Allows context-sensitive slicing
- Object-sensitive slice of first println(o.get()) does *not* contain secure
- Program part is guaranteed to be secure (noninterferent)

CISPA
Center for IT-Security, Privacy and Accountability

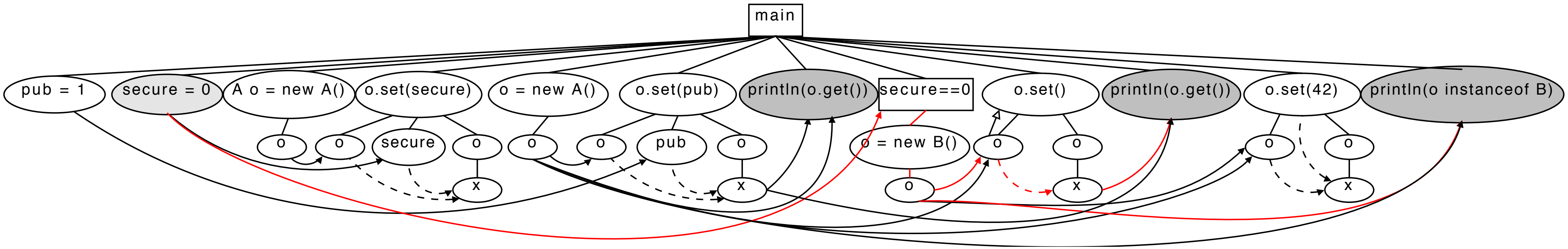
SDG for first part of the Program



- Summary edges for transitive flow of parameters
- Allows context-sensitive slicing
- Object-sensitive slice of first println(o.get()) does *not* contain secure
- Program part is guaranteed to be secure (noninterferent)

CISPA Center for IT-Security, Privacy and Accountability

SDG for complete Program



- Slice from second println(o.get()) contains secure
- Slice from instanceof also contains secure
- Both parts of the program potentially insecure
- Input “007” triggers illegal flow

References

- Hammer, C. Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs. Ph.D. Thesis, Universität Karlsruhe (TH), Fak. f. Informatik, 2009. (also available at the CS Library)

C|ISPA
Center for IT-Security, Privacy and Accountability

SAARLAND UNIVERSITY
COMPUTER SCIENCE