Information Flow Analysis via Path Condition Refinement

Mana Taghdiri, Gregor Snelting, Carsten Sinz
Karlsruhe Institute of Technology, Germany

FAST – September 16, 2010

Problem statement

- Information Flow Control (IFC)
  - Looks for a flow between two points in a program
  - Analyzes program's source code
  - Either discovers a potential (illegal) flow or guarantees that no flow is possible

- Problem
  - Sound IFC methods produce false alarms
  - False alarms have devastating effects on practicability

- Our goal
  - To improve precision while maintaining soundness

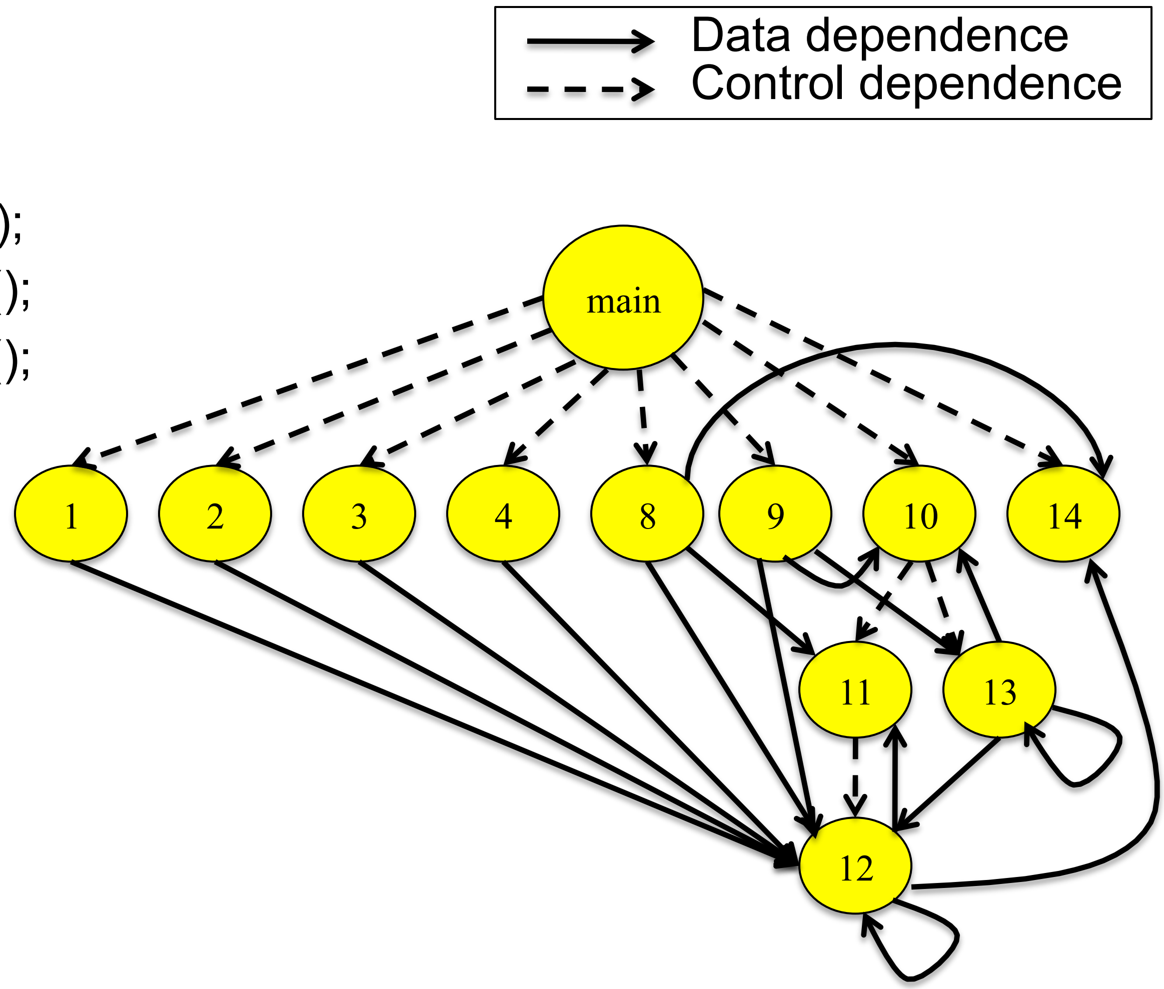
PDG-based IFC

- Program dependence graph (PDG)
  - Nodes represent program statements
  - Edges give data-dependency
    - Statement s is data-dependent on statement t iff s uses a variable assigned in t
  - Edges give control-dependency
    - Statement s is control-dependent on statement t iff s is executed only if an expression in t has a specific value

- PDG can be used for information flow problem
  - Information flow is possible only if there is a PDG path b/w the points
  - Flow-sensitive, context-sensitive, object-sensitive
  - Fewer false alarms, but more expensive than security type systems

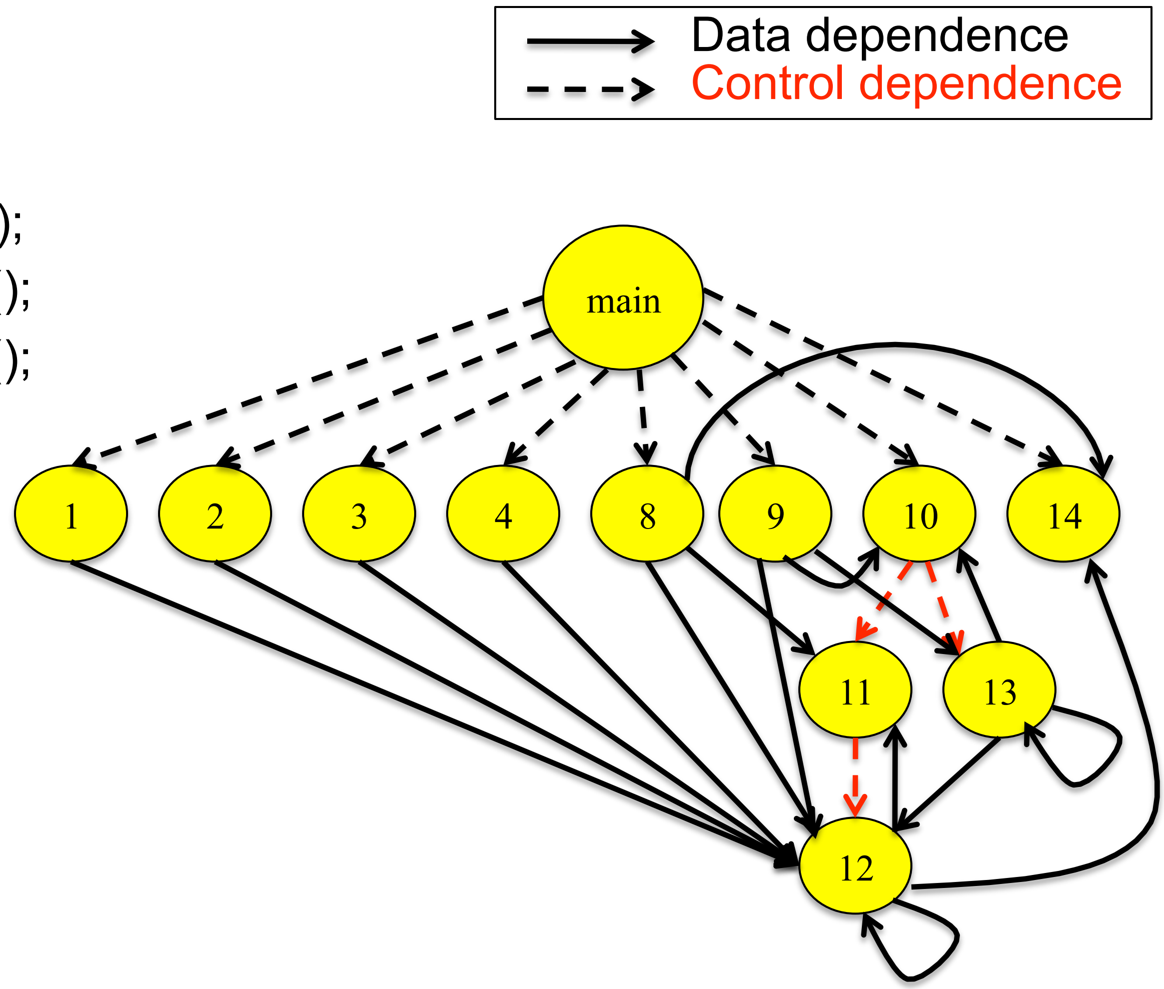
Example – PDG

```
0.  int main(String[] argv) {
1.      int[] a = new int[3];
2.      a[0] = System.in.read();
3.      a[1] = System.in.read();
4.      a[2] = System.in.read();
5.      assert(a[0] > 0);
6.      assert(a[1] > 0);
7.      assert(a[2] > 0);
8.      int sum = 0;
9.      int i = 0;
10.     while (i < 3) {
11.         if (sum == 0)
12.             sum = sum+a[i];
13.         i = i+1; }
14.     System.out.println(sum); }
```



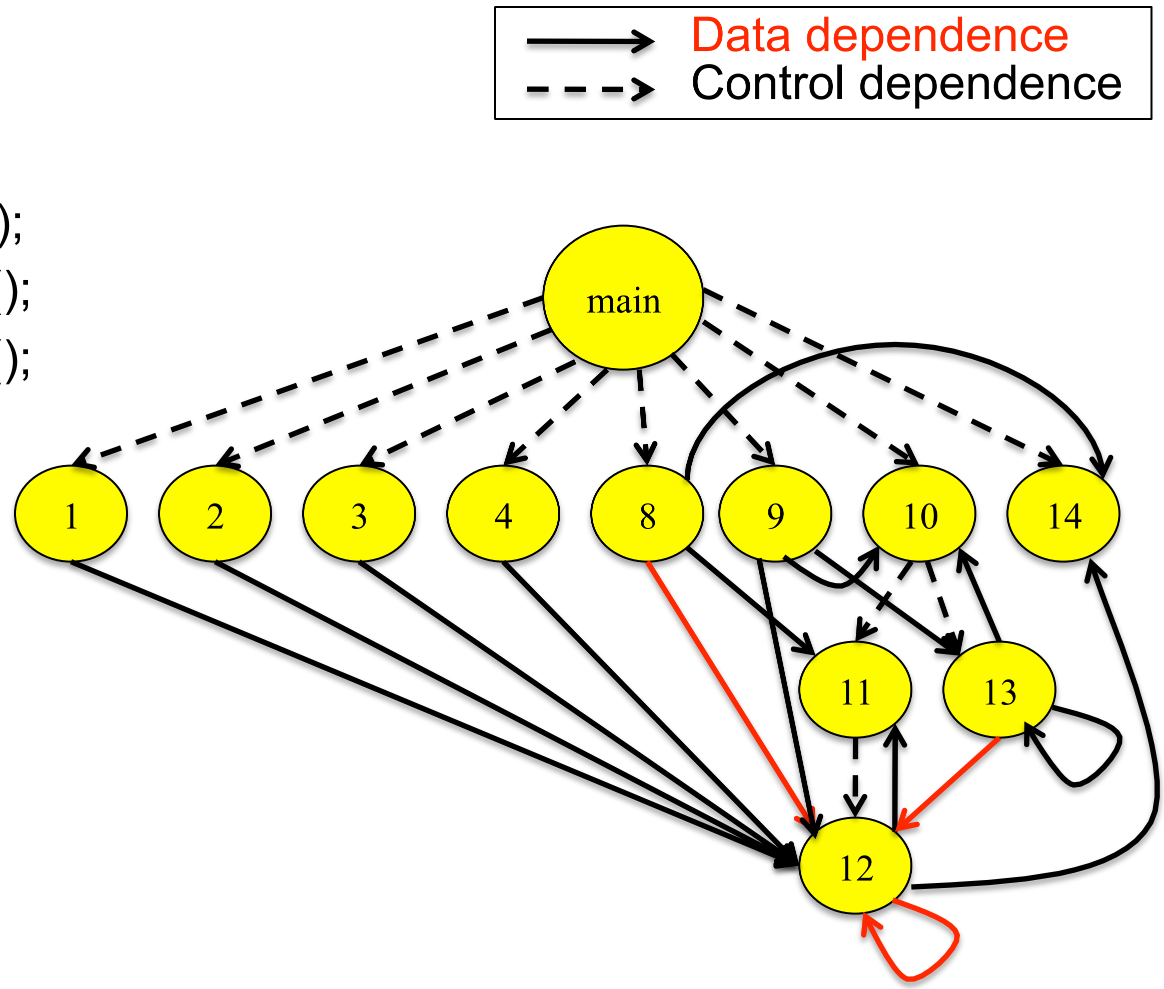
Example – PDG

```
0.  int main(String[] argv) {
1.      int[] a = new int[3];
2.      a[0] = System.in.read();
3.      a[1] = System.in.read();
4.      a[2] = System.in.read();
5.      assert(a[0] > 0);
6.      assert(a[1] > 0);
7.      assert(a[2] > 0);
8.      int sum = 0;
9.      int i = 0;
10.     while (i < 3) {
11.         if (sum == 0)
12.             sum = sum+a[i];
13.         i = i+1; }
14.     System.out.println(sum); }
```



Example – PDG

```
0.  int main(String[] argv) {
1.      int[] a = new int[3];
2.      a[0] = System.in.read();
3.      a[1] = System.in.read();
4.      a[2] = System.in.read();
5.      assert(a[0] > 0);
6.      assert(a[1] > 0);
7.      assert(a[2] > 0);
8.      int sum = 0;
9.      int i = 0;
10.     while (i < 3) {
11.         if (sum == 0)
12.             sum = sum+a[i];
13.         i = i+1; }
14.     System.out.println(sum); }
```



Example – PDG-based IFC

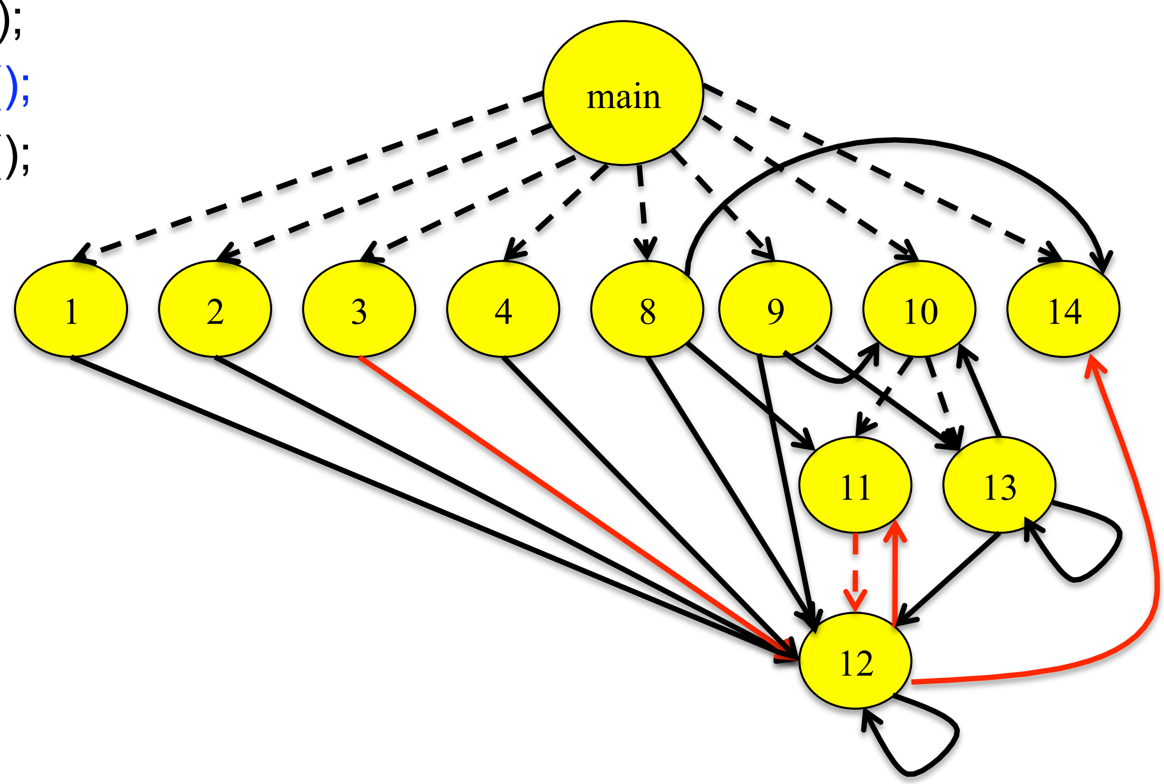
```
int main(String[] argv) {
    int[] a = new int[3];
    a[0] = System.in.read();        // PUBLIC
    a[1] = System.in.read();        // SECRET
    a[2] = System.in.read();        // SECRET
    assert(a[0] > 0);
    assert(a[1] > 0);
    assert(a[2] > 0);
    int sum = 0;
    int i = 0;
    while (i < 3) {
        if (sum == 0)
            sum = sum+a[i];
        i = i+1; }
    System.out.println(sum); }
```

Is there a flow from a[1] to sum?

Example

```
0.  int main(String[] argv) {
1.      int[] a = new int[3];
2.      a[0] = System.in.read();
3.      a[1] = System.in.read();
4.      a[2] = System.in.read();
5.      assert(a[0] > 0);
6.      assert(a[1] > 0);
7.      assert(a[2] > 0);
8.      int sum = 0;
9.      int i = 0;
10.     while (i < 3) {
11.         if (sum == 0)
12.             sum = sum+a[i];
13.         i = i+1; }
14.     System.out.println(sum); }
```

Is there a flow from a[1] to sum?



YES!, but this is just a false alarm

PC-based IFC

- Path conditions (PC) are built on top of PDG
  - A flow is possible only along PDG paths
  - Characterizes the conditions over program variables that must be satisfied for a path to be taken

- Example:

```
1.   a[i+3] = x;
2.   if (i>10 && j<5)
3.        y = a[2*j-42];
```

- PDG cotains a path 1 → 3
- $PC(1, 3) \equiv (i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42) \equiv false$
- So flow is impossible

PC-based IFC

- Path conditions are more precise than PDG alone
  - If a path condition PC(x, y) is not satisfiable, it is guaranteed that there is no flow from x to y, even if the PDG contains a path x → y

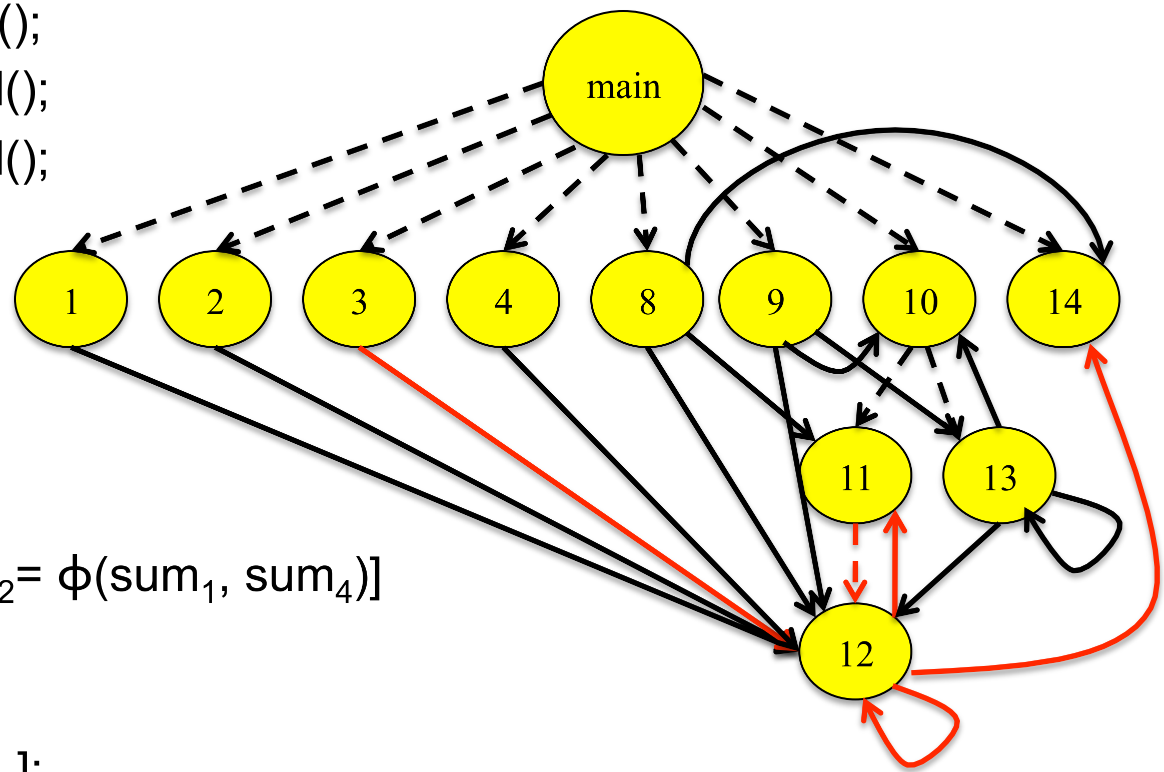
- Solving a path condition produces a witness for flow
  - Variables in PC are existentially quantified
  - A constraint solver can find a solution
  - A solution satisfying PC provides values for program inputs
  - PDG gives only a binary answer: either a flow is possible, or no flow exists

- But path conditions may also produce false alarms
  - Due to conservative abstractions

Example – SSA form

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. $a_1[0]$ = **System.in.read**();
3. $a_2[1]$ = **System.in.read**();
4. $a_3[2]$ = **System.in.read**();
5. **assert**($a_3[0] > 0$);
6. **assert**($a_3[1] > 0$);
7. **assert**($a_3[2] > 0$);
8. **int** $sum_1 = 0$;
9. **int** $i_1 = 0$;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2 < 3$) {
11. **if** ($sum_2 == 0$)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

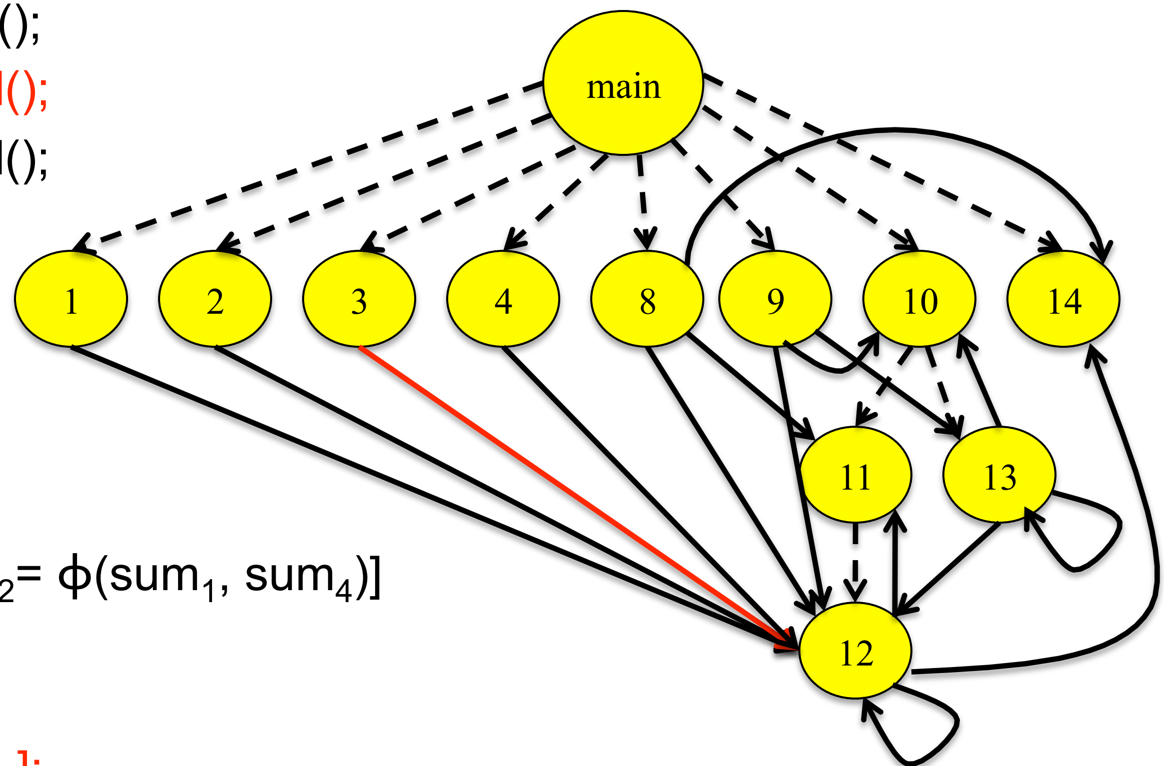
Is there a flow from a[1] to sum?



Example – path condition

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. $a_1[0]$ = **System.in.read**();
3. $a_2[1]$ = **System.in.read**();
4. $a_3[2]$ = **System.in.read**();
5. **assert**($a_3[0] > 0$);
6. **assert**($a_3[1] > 0$);
7. **assert**($a_3[2] > 0$);
8. **int** $sum_1 = 0$;
9. **int** $i_1 = 0$;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2 < 3$) {
11. **if** ($sum_2 == 0$)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

Is there a flow from a[1] to sum?

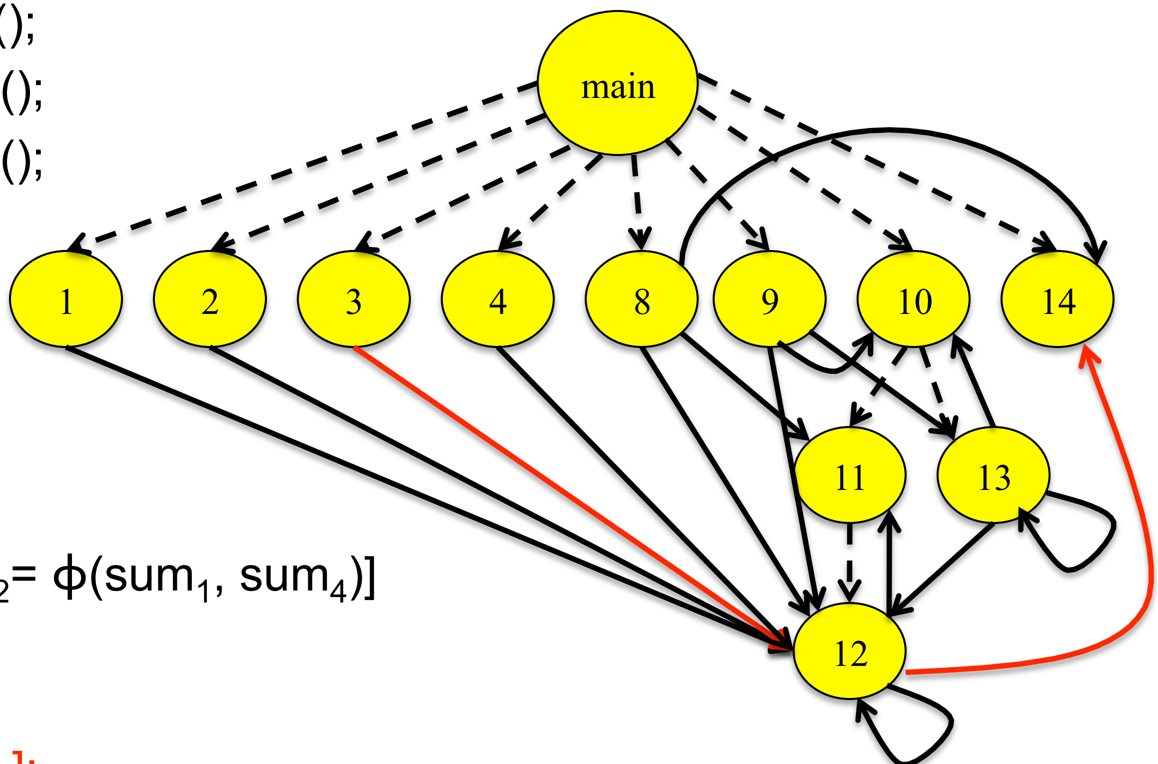


PC (3, 14) ≡ ($i_2 = 1$)

Example – path condition

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. $a_1[0]$ = **System.in.read**();
3. $a_2[1]$ = **System.in.read**();
4. $a_3[2]$ = **System.in.read**();
5. **assert**($a_3[0] > 0$);
6. **assert**($a_3[1] > 0$);
7. **assert**($a_3[2] > 0$);
8. **int** $sum_1 = 0$;
9. **int** $i_1 = 0$;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2 < 3$) {
11. **if** ($sum_2 == 0$)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

Is there a flow from a[1] to sum?

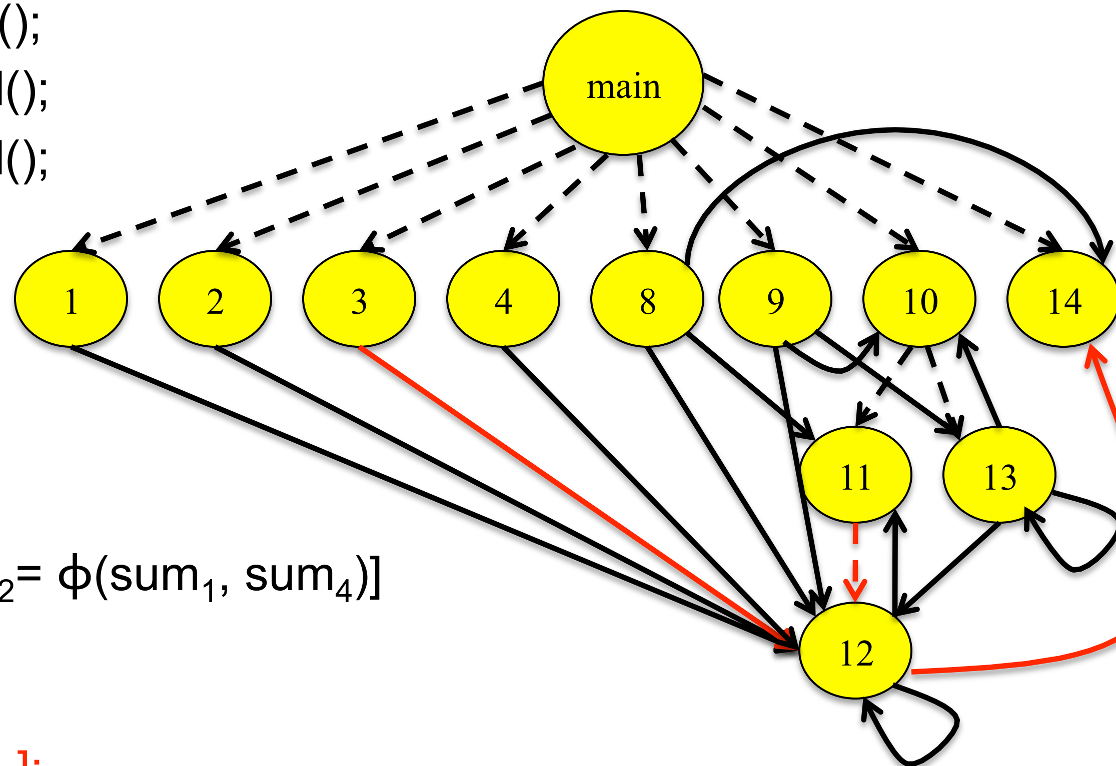


$PC(3, 14) \equiv \ldots \wedge (sum_2 = sum_3)$

Example – path condition

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. a_1[0] = **System.in.read**();
3. a_2[1] = **System.in.read**();
4. a_3[2] = **System.in.read**();
5. **assert**(a_3[0] > 0);
6. **assert**(a_3[1] > 0);
7. **assert**(a_3[2] > 0);
8. **int** sum_1 = 0;
9. **int** i_1= 0;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2$ < 3) {
11. **if** (sum_2 == 0)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

Is there a flow from a[1] to sum?



PC (3, 14) ≡ … $\wedge$ (sum_2 = 0)

Example – path condition

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. $a_1[0]$ = **System.in.read**();
3. $a_2[1]$ = **System.in.read**();
4. $a_3[2]$ = **System.in.read**();
5. **assert**($a_3[0] > 0$);
6. **assert**($a_3[1] > 0$);
7. **assert**($a_3[2] > 0$);
8. **int** $sum_1 = 0$;
9. **int** $i_1 = 0$;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2 < 3$) {
11. **if** ($sum_2 == 0$)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

Is there a flow from a[1] to sum?

$$
\begin{aligned}
PC(3, 14) \equiv\ & (sum_1 = 0) \wedge (i_1 = 0) \\
& \wedge (i_2 = 1) \wedge (sum_2 = sum_3) \\
& \wedge (i_2 < 3) \wedge (sum_2 = 0) \\
& \wedge (a_3[0] > 0) \wedge (a_3[1] > 0) \wedge (a_3[2] > 0) \\
& \wedge (i_2 = i_1 \vee i_2 = i_3) \\
& \wedge (sum_2 = sum_1 \vee sum_2 = sum_4) \\
& \wedge (sum_4 = sum_2 \vee sum_4 = sum_3)
\end{aligned}
$$

Satisfying solution (potential witness):

a[0] = 15, a[1] = 5, a[2] = 42,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

But, this is a false alarm

False alarms in PC

- Path conditions are
  - Flow-sensitive
  - Context-sensitive
  - Object-sensitive

- But yet, may produce false alarms
  - To preserve soundness, path conditions make conservative abstractions
  - Cannot distinguish between different iterations of a loop
  - Static analysis alone cannot analyze loops precisely

CEGAR-based IFC

- Counterexample-guided abstraction refinement (CEGAR)
  - Increases precision while preserving scalability
  - Successfully used in software model checking and data structure analysis, but never for IFC
  - Follows a fully automatic solve-validate-refine loop

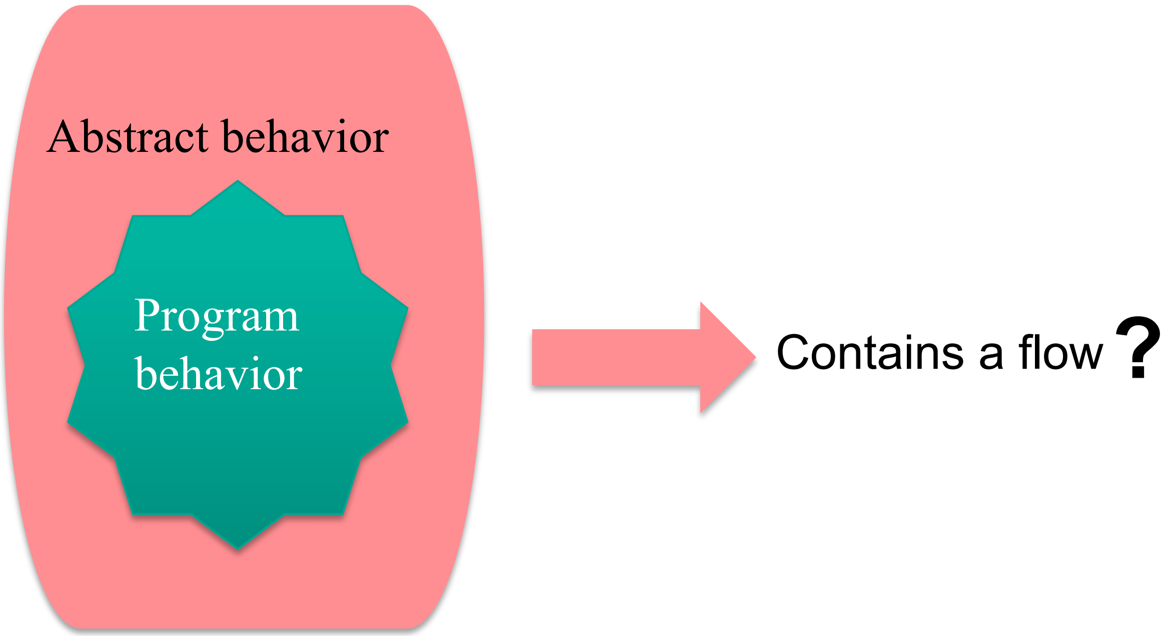
- Goal:
  - Generate a flow whose execution truly demonstrates the security breach
  - Reduce the number of false alarms

- Approach:
  - Keep looking for better witnesses until existence of the flow is established or refuted
  - Make path conditions more precise iteratively as needed

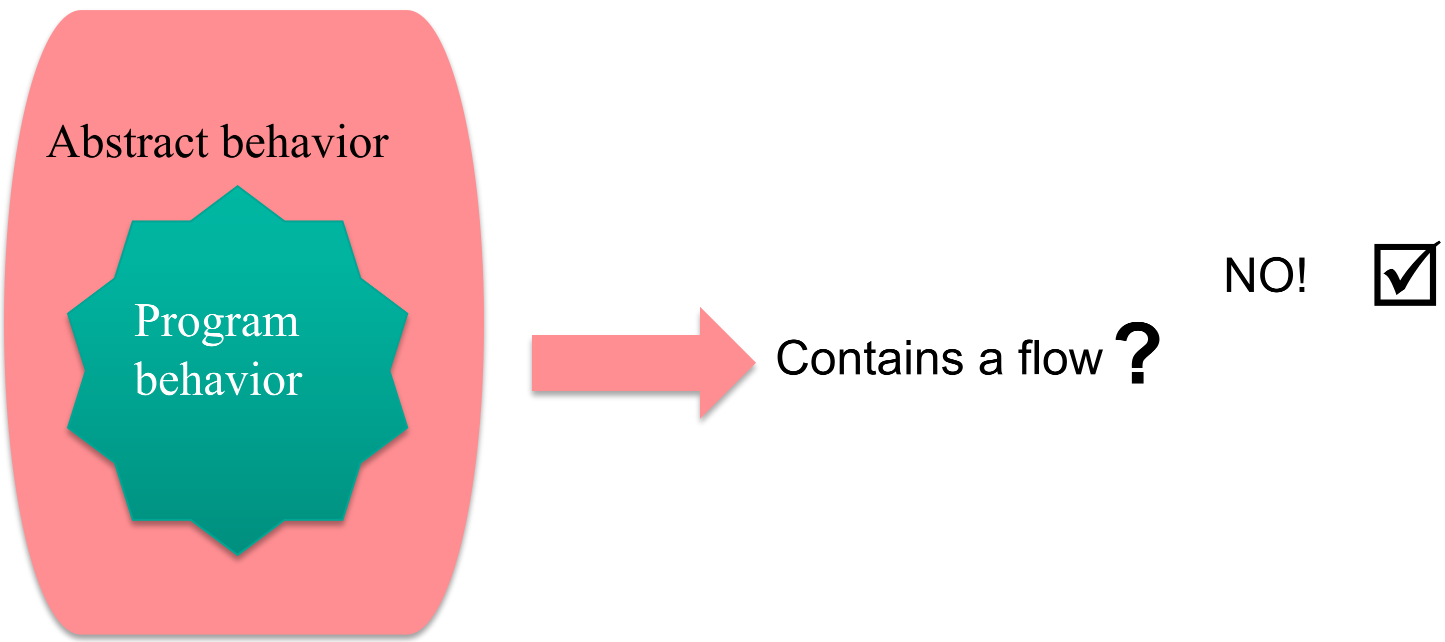
CEGAR – incremental analysis

Contains a flow ?

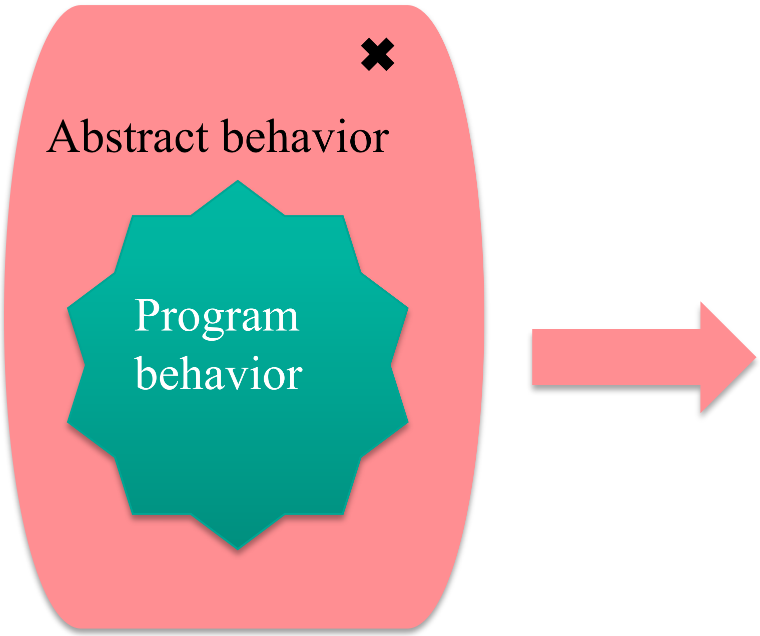
CEGAR – incremental analysis



CEGAR – incremental analysis



CEGAR – incremental analysis

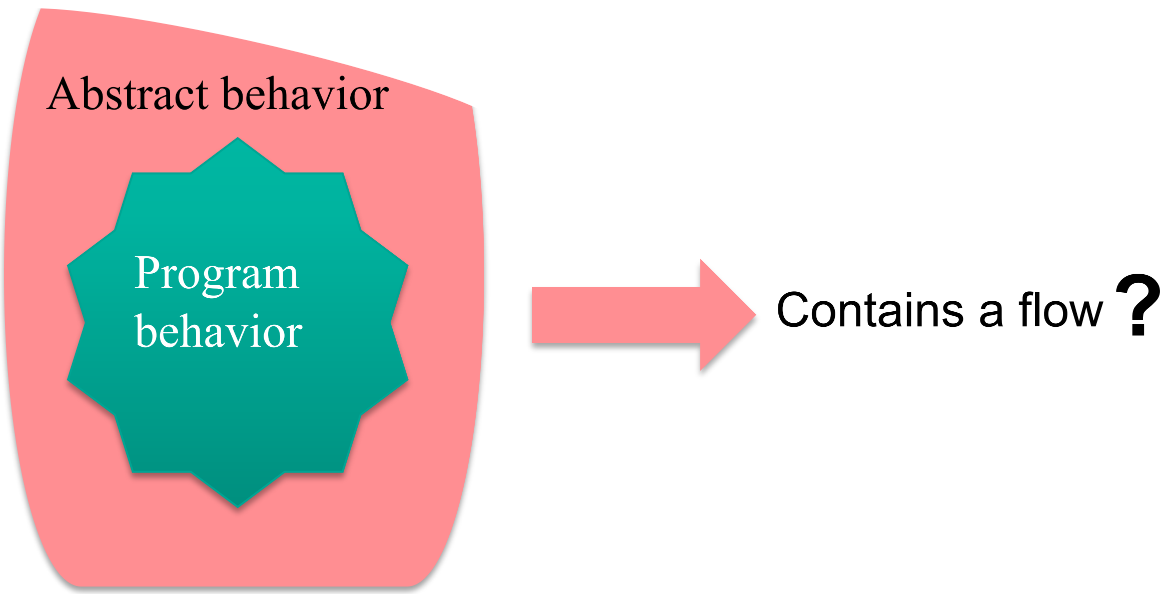


Contains a flow**?**

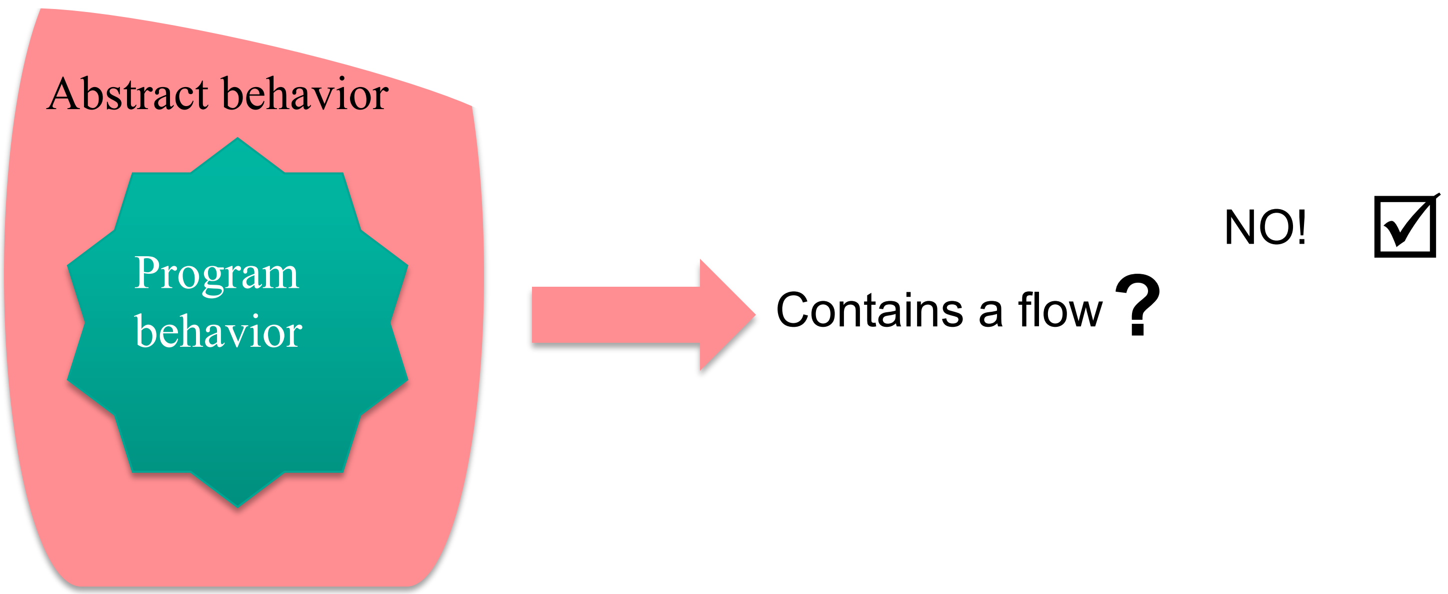
NO! ☑

Yes …

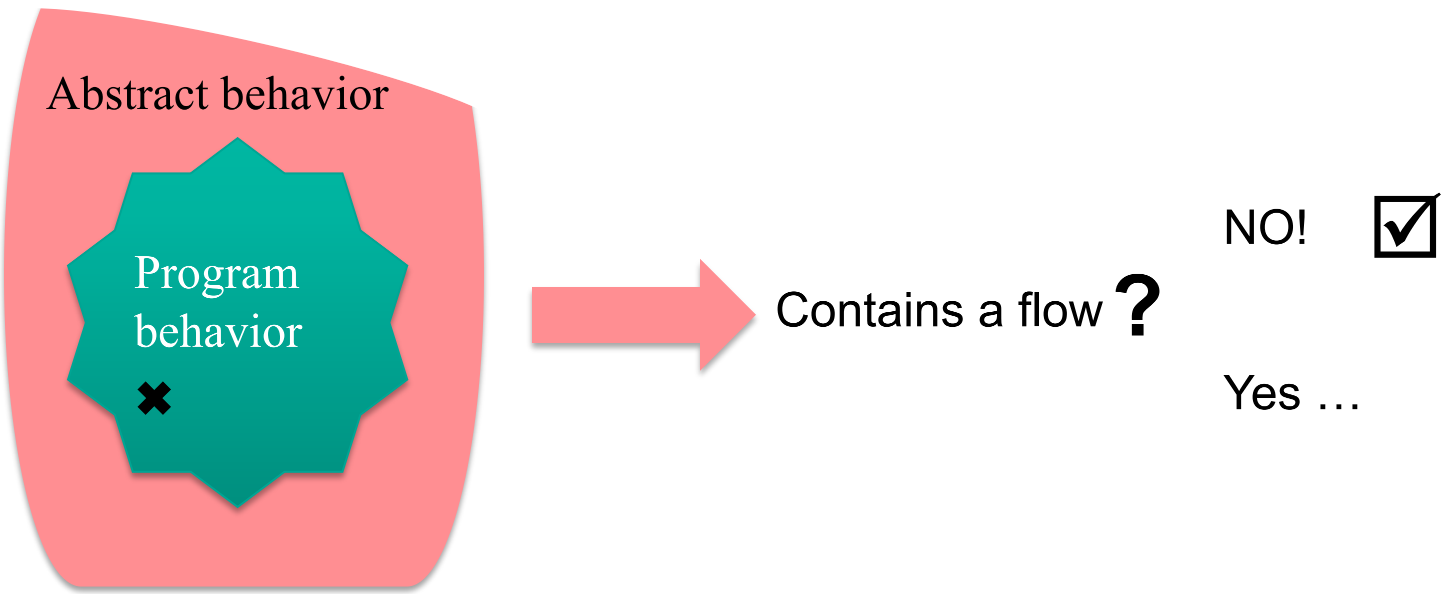
CEGAR – incremental analysis



CEGAR – incremental analysis

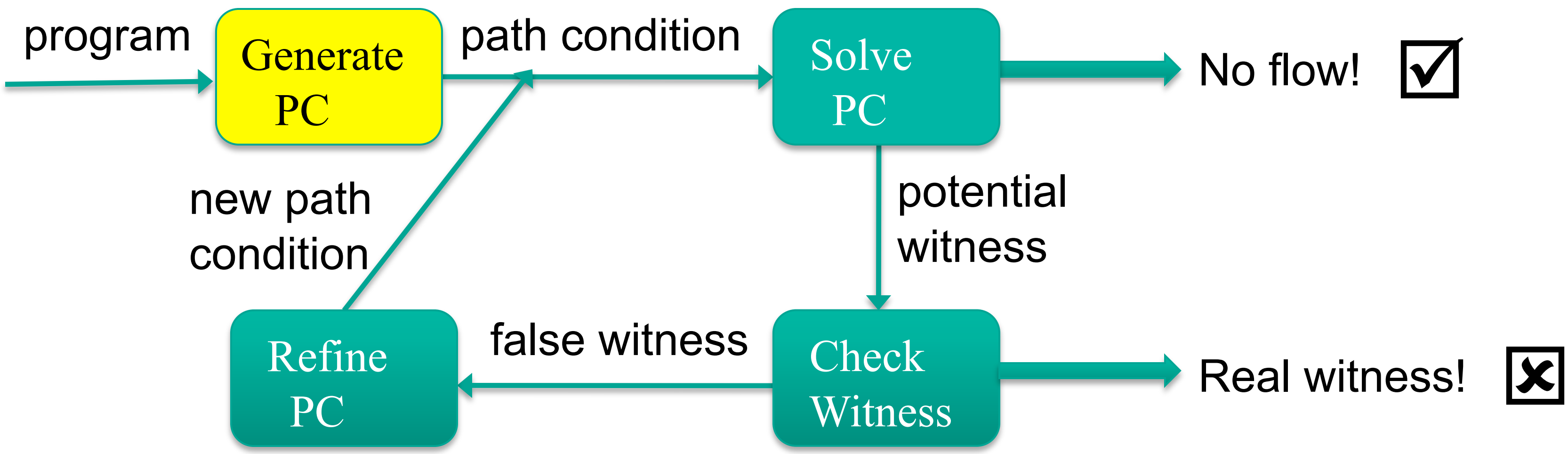


CEGAR – incremental analysis



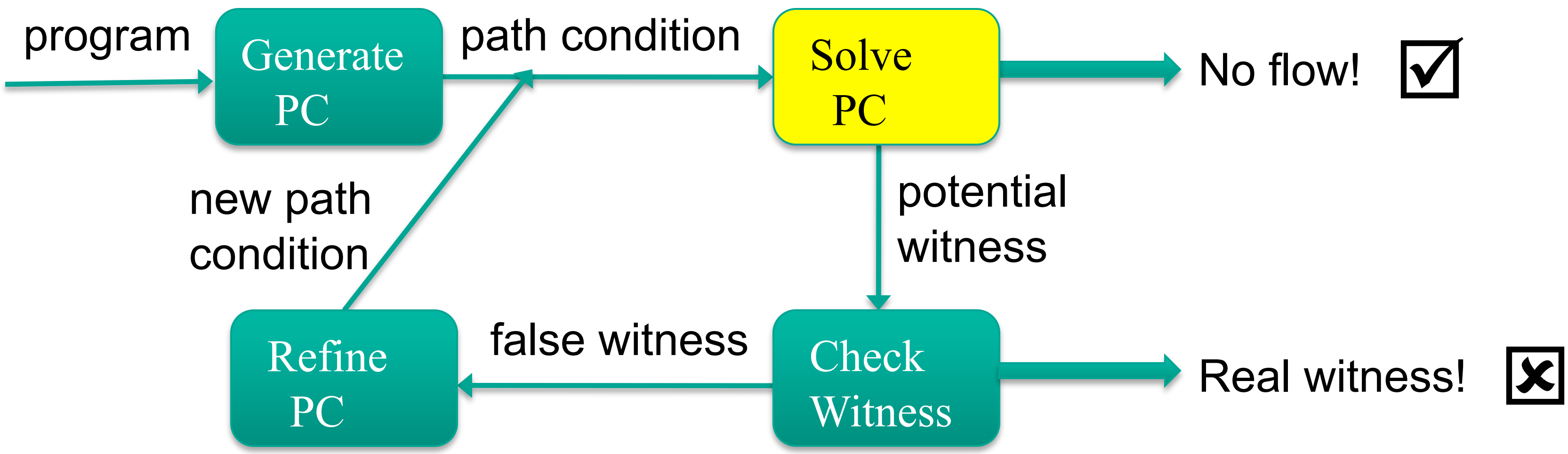
CEGAR-based IFC

- Initial path condition is an over-approximation of conditions necessary for flow



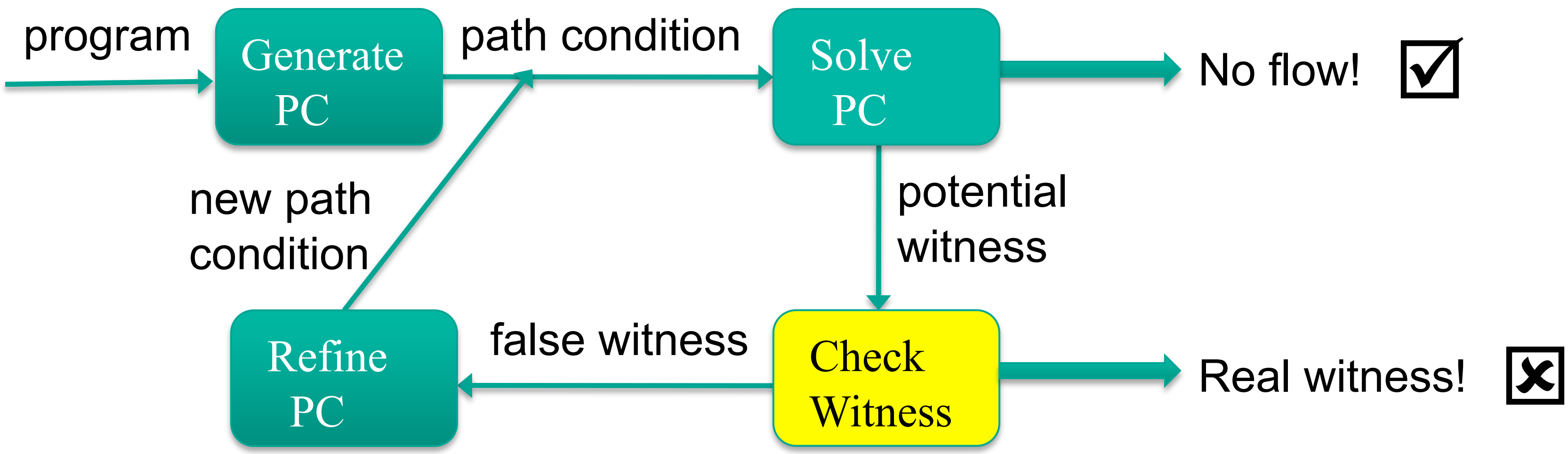
CEGAR-based IFC

- We use an off-the-shelf SMT (SAT Modulo Theory) solver to check path conditions
- The solver is complete: if a solution exists, it is guaranteed to find it
- The solver returns only one solution at a time
- Path conditions are sound: if they are unsatisfiable, it is guaranteed that no flow exists



CEGAR-based IFC

- A solution to PC represents a potential witness for information flow
- A solution is checked for validity by running the program on the inputs given by the solution
- The execution is monitored and compared to the solution
- A contradictory program state denotes that the witness is spurious



Example – potential witness

```
int main(String[] argv) {
    int[] a = new int[3];
    a1[0] = System.in.read();
    a2[1] = System.in.read();
    a3[2] = System.in.read();
    assert(a3[0] > 0);
    assert(a3[1] > 0);
    assert(a3[2] > 0);
    int sum1 = 0;
    int i1= 0;
    while [i2= ϕ(i1, i3); sum2= ϕ(sum1, sum4)]
          (i2 < 3) {
        if (sum2 == 0)
            sum3 = sum2+a3[i2];
        [sum4= ϕ(sum2, sum3)]  i3 = i2+1; }
    System.out.println(sum2); }
```

Is there a flow from a[1] to sum?

Initial PC solution (potential witness):

$a[0] = 15, a[1] = 5, a[2] = 42,$
$i_1 = 0, i_2 = i_3 = 1,$
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

0. **int main**(**String**[] argv) {
1. **int**[] a = **new int**[3];
2. a_1[0] = **System.in.read**(); [a[0] = 15]
3. a_2[1] = **System.in.read**(); [.., a[1] = 5]
4. a_3[2] = **System.in.read**(); [.., a[2] = 42]
5. **assert**(a_3[0] > 0);
6. **assert**(a_3[1] > 0);
7. **assert**(a_3[2] > 0);
8. **int** sum_1 = 0;
9. **int** i_1= 0;
10. **while** [$i_2 = \phi(i_1, i_3)$; $sum_2 = \phi(sum_1, sum_4)$] ($i_2 < 3$) {
11. **if** (sum_2 == 0)
12. $sum_3 = sum_2 + a_3[i_2]$;
13. [$sum_4 = \phi(sum_2, sum_3)$] $i_3 = i_2 + 1$; }
14. **System.out.println**(sum_2); }

potential witness:
a[0] = 15, a[1] = 5, a[2] = 42,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();          [a[0] = 15]
     a_2[1] = System.in.read();         [.., a[1] = 5]
     a_3[2] = System.in.read();         [.., a[2] = 42]
     assert(a_3[0] > 0);
     assert(a_3[1] > 0);
     assert(a_3[2] > 0);
     int sum_1 = 0;                     [.., sum_1 = 0]
     int i_1= 0;                        [.., i_1 = 0]
     while [i_2= φ(i_1, i_3); sum_2= φ(sum_1, sum_4)]
            (i_2 < 3) {
        if (sum_2 == 0)
            sum_3 = sum_2+a_3[i_2];
        [sum_4= φ(sum_2, sum_3)]  i_3 = i_2+1; }
    System.out.println(sum_2); }
```

potential witness:
$a[0] = 15, a[1] = 5, a[2] = 42,$
$i_1 = 0, i_2 = i_3 = 1,$
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();        [a[0] = 15]
    a_2[1] = System.in.read();        [.., a[1] = 5]
    a_3[2] = System.in.read();        [.., a[2] = 42]
    assert(a_3[0] > 0);
    assert(a_3[1] > 0);
    assert(a_3[2] > 0);
    int sum_1 = 0;                    [.., sum_1 = 0]
    int i_1= 0;                       [.., i_1 = 0]
    while [i_2= ϕ(i_1, i_3); sum_2= ϕ(sum_1, sum_4)]        [.., sum_2 = 0, i_2 = 0]
          (i_2 < 3) {
       if (sum_2 == 0)
          sum_3 = sum_2+a_3[i_2];
       [sum_4= ϕ(sum_2, sum_3)]  i_3 = i_2+1; }
    System.out.println(sum_2); }
```

potential witness:

$a[0] = 15$, $a[1] = 5$, $a[2] = 42$,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();        [a[0] = 15]
    a_2[1] = System.in.read();        [.., a[1] = 5]
    a_3[2] = System.in.read();        [.., a[2] = 42]
    assert(a_3[0] > 0);
    assert(a_3[1] > 0);
    assert(a_3[2] > 0);
    int sum_1 = 0;                    [.., sum_1 = 0]
    int i_1= 0;                       [.., i_1 = 0]
    while [i_2= ϕ(i_1, i_3); sum_2= ϕ(sum_1, sum_4)]        [.., sum_2 = 0, i_2 = 0]
            (i_2 < 3) {
        if (sum_2 == 0)
            sum_3 = sum_2+a_3[i_2];                          [.., sum_3 = 15]
        [sum_4= ϕ(sum_2, sum_3)]  i_3 = i_2+1; }             [.., i_3 = 1]
    System.out.println(sum_2); }
```

potential witness:
$a[0] = 15$, $a[1] = 5$, $a[2] = 42$,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();        [a[0] = 15]
    a_2[1] = System.in.read();        [.., a[1] = 5]
    a_3[2] = System.in.read();        [.., a[2] = 42]
    assert(a_3[0] > 0);
    assert(a_3[1] > 0);
    assert(a_3[2] > 0);
    int sum_1 = 0;                    [.., sum_1 = 0]
    int i_1= 0;                       [.., i_1 = 0]
    while [i_2= ϕ(i_1, i_3); sum_2= ϕ(sum_1, sum_4)]        [.., sum_2 = 15, i_2 = 1]
          (i_2 < 3) {
       if (sum_2 == 0)
          sum_3 = sum_2+a_3[i_2];                           [.., sum_3 = 15]
       [sum_4= ϕ(sum_2, sum_3)]  i_3 = i_2+1; }             [.., i_3 = 2]
    System.out.println(sum_2); }
```

potential witness:
$a[0] = 15$, $a[1] = 5$, $a[2] = 42$,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution

int main(**String**[] argv) {
int[] a = **new int**[3];
a_1[0] = **System.in.read**(); [a[0] = 15]
a_2[1] = **System.in.read**(); [.., a[1] = 5]
a_3[2] = **System.in.read**(); [.., a[2] = 42]
assert(a_3[0] > 0);
assert(a_3[1] > 0);
assert(a_3[2] > 0);
int sum_1 = 0; [.., sum_1 = 0]
int i_1= 0; [.., i_1 = 0]
while [i_2= $\phi(i_1, i_3)$; sum_2= $\phi(sum_1, sum_4)$] (i_2 < 3) { [.., sum_2 = 15, i_2 = 2]
if (sum_2 == 0)
sum_3 = sum_2+a_3[i_2]; [.., sum_3 = 15]
[sum_4= $\phi(sum_2, sum_3)$] i_3 = i_2+1; } [.., i_3 = 3]
System.out.println(sum_2); }

potential witness:
a[0] = 15, a[1] = 5, a[2] = 42,
i_1 = 0, i_2 = i_3 = 1,
sum_1 = sum_2 = sum_3 = sum_4 = 0

Example – program execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();                              [a[0] = 15]
    a_2[1] = System.in.read();                              [.., a[1] = 5]
    a_3[2] = System.in.read();                              [.., a[2] = 42]
    assert(a_3[0] > 0);
    assert(a_3[1] > 0);
    assert(a_3[2] > 0);
    int sum_1 = 0;                                          [.., sum_1 = 0]
    int i_1= 0;                                             [.., i_1 = 0]
    while [i_2= φ(i_1, i_3); sum_2= φ(sum_1, sum_4)]        [.., sum_2 = 15, i_2 = 3]
          (i_2 < 3) {
      if (sum_2 == 0)
          sum_3 = sum_2+a_3[i_2];                           [.., sum_3 = 15]
      [sum_4= φ(sum_2, sum_3)]  i_3 = i_2+1; }              [.., i_3 = 3]
    System.out.println(sum_2); }
```

potential witness:

$a[0] = 15$, $a[1] = 5$, $a[2] = 42$,

$i_1 = 0$, $i_2 = i_3 = 1$,

$sum_1 = sum_2 = sum_3 = sum_4 = 0$

Example – program execution – contradiction

```
int main(String[] argv) {
    int[] a = new int[3];
    a_1[0] = System.in.read();            [a[0] = 15]
    a_2[1] = System.in.read();            [.., a[1] = 5]
    a_3[2] = System.in.read();            [.., a[2] = 42]
    assert(a_3[0] > 0);
    assert(a_3[1] > 0);
    assert(a_3[2] > 0);
    int sum_1 = 0;                        [.., sum_1 = 0]
    int i_1= 0;                           [.., i_1 = 0]
    while [i_2= φ(i_1, i_3); sum_2= φ(sum_1, sum_4)]
            (i_2 < 3) {
        if (sum_2 == 0)
            sum_3 = sum_2+a_3[i_2];       [.., sum_3 = 15]
        [sum_4= φ(sum_2, sum_3)]  i_3 = i_2+1; }   [.., i_3 = 3]
    System.out.println(sum_2); }
```

Loop (line 10) values:

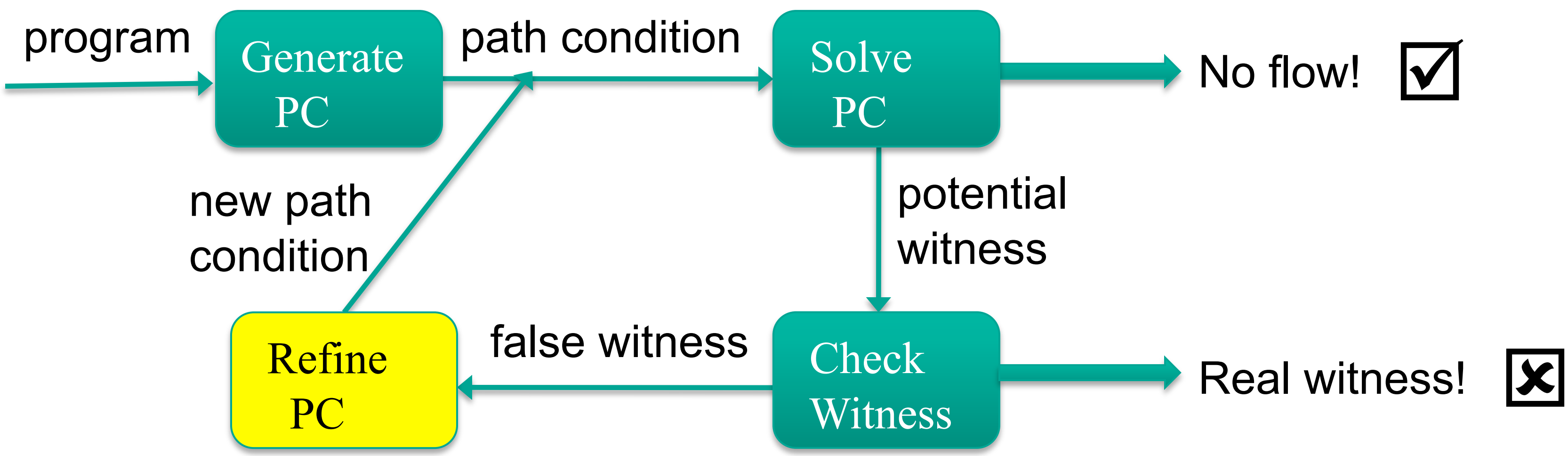
- [.., sum_2 = 0, i_2 = 0]
- [.., sum_2 = 15, i_2 = 1]
- [.., sum_2 = 15, i_2 = 2]
- [.., sum_2 = 15, i_2 = 3]

potential witness:
a[0] = 15, a[1] = 5, a[2] = 42,
$i_1 = 0$, $i_2 = i_3 = 1$,
$sum_1 = sum_2 = sum_3 = sum_4 = 0$

This witness is spurious!!

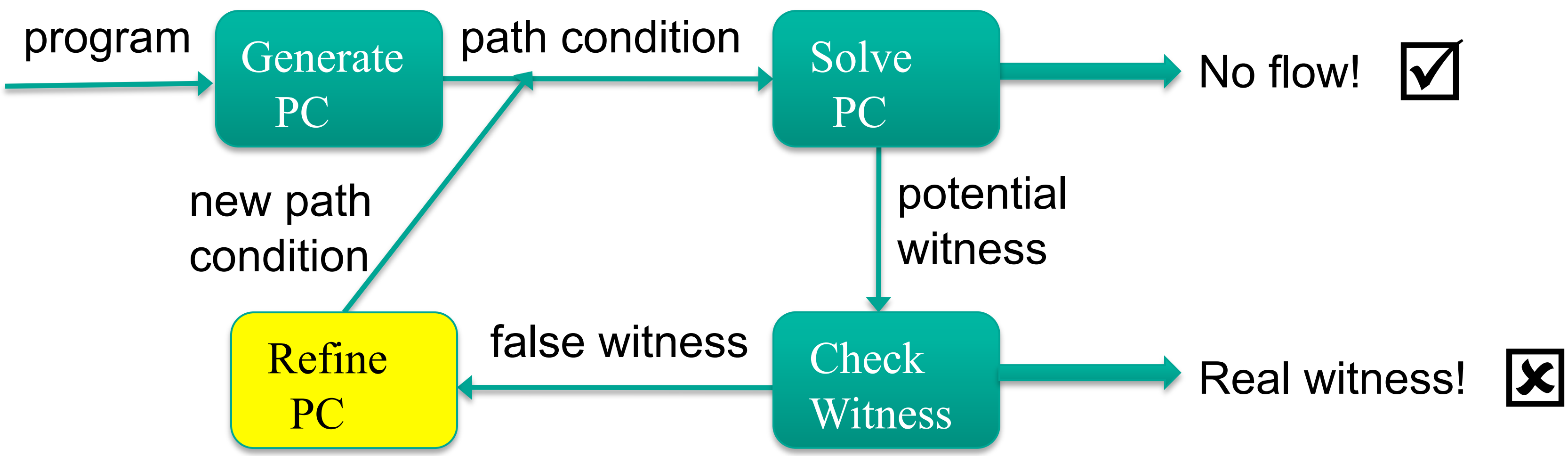
CEGAR-based IFC

- To refine the path condition, run the code symbolically, following the previous concrete execution of the previous stage
- During the symbolic execution, compute symbolic values of variables
- Also compute the symbolic control conditions taken in this path (guards)



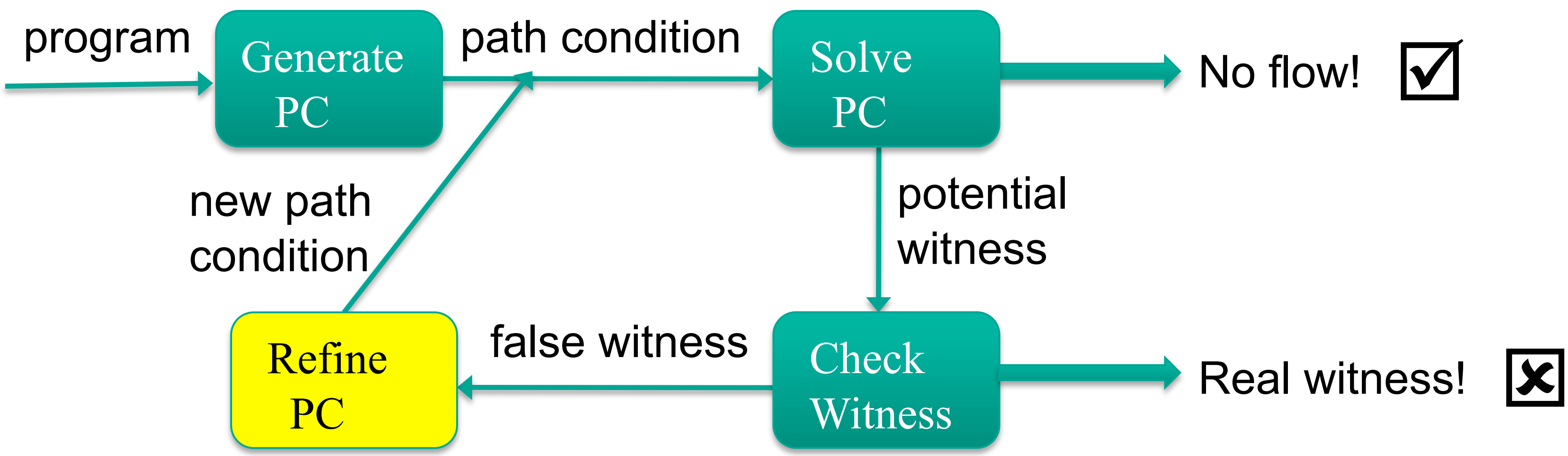
CEGAR-based IFC

- Construct a formula that gives the symbolic values of program variables at the contradictory program point
- Use the SMT solver again to solve this formula wrt the current witness
  - The witness provides a partial solution for the formula
  - This formula is unsatisfiable



CEGAR-based IFC

- Since the SMT solver cannot find a solution, it generates a proof of unsatisfiability
  - Encodes why the current witness is invalid
- Refined PC = old PC conjoined with the unsatisfiability proof



Example – symbolic execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a [0] = System.in.read();
    a [1] = System.in.read();
    a [2] = System.in.read();
    assert(a [0] > 0);
    assert(a [1] > 0);
    assert(a [2] > 0);
    int sum = 0;
    int i = 0;
    while (i < 3) {
        if (sum == 0)
            sum = sum+a [i];
        i = i+1; }
    System.out.println(sum); }
```

Symbolic values:

[a[0] = A0]
[.., a[1] = A1]
[.., a[2] = A2]

[.., sum = 0]
[.., i = 0]

Symbolic guards:

{(A0 > 0)}
{.. ∧ (A1 > 0)}
{.. ∧ (A2 > 0)}

sum = **if** guard **then** 0 **else** ANY
i = **if** guard **then** 0 **else** ANY

Example – symbolic execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a [0] = System.in.read();
    a [1] = System.in.read();
    a [2] = System.in.read();
    assert(a [0] > 0);
    assert(a [1] > 0);
    assert(a [2] > 0);
    int sum = 0;
    int i = 0;
    while (i < 3) {
        if (sum == 0)
            sum = sum+a [i];
        i = i+1; }
    System.out.println(sum); }
```

Symbolic values:

[a[0] = A0]
[.., a[1] = A1]
[.., a[2] = A2]

[.., sum = 0]
[.., i = 0]

[.., sum = A0]
[.., i = 1]

Symbolic guards:

$\{(A0 > 0)\}$
$\{.. \wedge (A1 > 0)\}$
$\{.. \wedge (A2 > 0)\}$

$\{.. \wedge (0 < 3)\}$
$\{.. \wedge (0 = 0)\}$

Example – symbolic execution

```
int main(String[] argv) {
    int[] a = new int[3];
    a [0] = System.in.read();
    a [1] = System.in.read();
    a [2] = System.in.read();
    assert(a [0] > 0);
    assert(a [1] > 0);
    assert(a [2] > 0);
    int sum = 0;
    int i = 0;
    while (i < 3) {
        if (sum == 0)
            sum = sum+a [i];
        i = i+1; }
    System.out.println(sum); }
```

Symbolic values:

[a[0] = A0]
[.., a[1] = A1]
[.., a[2] = A2]
[.., sum = 0]
[.., i = 0]

Symbolic guards:

{(A0 > 0)}
{.. ∧ (A1 > 0)}
{.. ∧ (A2 > 0)}
{.. ∧ (0 < 3)}
{.. ∧ (0 = 0)}

sum = **if** guard **then** A0 **else** ANY
i = **if** guard **then** 1 **else** ANY

Example – PC refinement

Symbolic formula:

$(a[0] = A0) \wedge (a[1] = A1) \wedge (a[2] = A2) \wedge$ (Initial symbolic values)

let $c = (A0 > 0) \wedge (A1 > 0) \wedge (A2 > 0)$ **and** $c' = c \wedge \neg(A0 = 0)$

in $((sum_2 = \text{if } c \text{ then } 0 \text{ else ANY}) \wedge (i_2 = \text{if } c \text{ then } 0 \text{ else ANY}))$ (1st iter)

$\vee ((sum_2 = \text{if } c \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c \text{ then } 1 \text{ else ANY}))$ (2nd iter)

$\vee ((sum_2 = \text{if } c' \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c' \text{ then } 2 \text{ else ANY}))$ (3rd iter)

$\vee ((sum_2 = \text{if } c' \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c' \text{ then } 3 \text{ else ANY}))$ (4th iter)

Current witness:

a[0] = 15, a[1] = 5, a[2] = 42,

$i_1 = 0, i_2 = i_3 = 1, sum_1 = sum_2 = sum_3 = sum_4 = 0$

FALSE

Example – PC refinement

Symbolic formula:

$(a[0] = A0) \wedge (a[1] = A1) \wedge (a[2] = A2) \wedge$ (Initial symbolic values)

let $c = (A0 > 0) \wedge (A1 > 0) \wedge (A2 > 0)$ **and** $c' = c \wedge \neg(A0 = 0)$

in $((sum_2 = \text{if } c \text{ then } 0 \text{ else ANY}) \wedge (i_2 = \text{if } c \text{ then } 0 \text{ else ANY}))$ (1^{st} iter)

$\vee ((sum_2 = \text{if } c \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c \text{ then } 1 \text{ else ANY}))$ (2^{nd} iter)

$\vee ((sum_2 = \text{if } c' \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c' \text{ then } 2 \text{ else ANY}))$ (3^{rd} iter)

$\vee ((sum_2 = \text{if } c' \text{ then } A0 \text{ else ANY}) \wedge (i_2 = \text{if } c' \text{ then } 3 \text{ else ANY}))$ (4^{th} iter)

Current witness:

$a[0] = 15, a[1] = 5, a[2] = 42,$

$i_1 = 0, i_2 = i_3 = 1, sum_1 = sum_2 = sum_3 = sum_4 = 0$

Proof $\equiv ((a[0] > 0) \wedge (a[1] > 0) \wedge (a[2] > 0)) \Rightarrow ((sum_2 = 0 \wedge i_2 = 0) \vee sum_2 = a[0])$

Path condition refinement

- Proof of unsatisfiability is
  - A formula weaker than the one solved
  - Is still unsatisfiable wrt the current witness

- That is,
  - If $F \land Sol \equiv false$,
  - Then $Proof \land Sol \equiv false$, and $F \Rightarrow Proof$

- Refined PC is the previous PC augmented with the proof
  - Refined PC = old PC $\land$ Proof
  - It guarantees that the current solution will never be found again

Example – PC refinement

$$PC'(3, 14) \equiv PC(3, 14) \land \text{Proof}$$
$$\equiv (a[0] > 0) \land (a[1] > 0) \land (a[2] > 0) \land (i_1 = 0)$$
$$\land (i_2 = i_3 = 1) \land (sum_1 = sum_2 = sum_3 = sum_4 = 0)$$
$$\land ((a[0] > 0) \land (a[1] > 0) \land (a[2] > 0) \Rightarrow$$
$$((sum_2 = 0 \land i_2 = 0) \lor (sum_2 = a[0])))$$
$$\equiv \text{false}$$

No information flow is possible!

Final words

- CEGAR-based IFC is expected to be much more precise than PDG- and PC-based IFC
  - Incorporates a fully automatic solve-validate-refine loop
  - Increases the precision of the analysis on demand
  - Exploits constraint solving, concrete and symbolic executions, unsat proof
- Number of iterations depends on the richness of proofs and order of solutions returned by the solver
  - A minimal proof is ideal, but not available
  - Existing proofs have been shown to be good enough in practice
- In certain cases, termination requires a time-out threshold
  - Because witness execution may not terminate
  - Because PC can become intractable

Final words

- Status of the project
  - PDG-based IFC implemented for full Java bytecode
    - Handles around 50 kLoc
  - Path condition implemented for imperative Java
    - Handles a few kLoc
  - CEGAR implementation just started
    - Scalability reports will be available in the future

Related work – security type systems

- Opened the door to language-based security analysis
- Compositional, scalable
- But require user-provided annotations
- Examples include JIF and Mobius
- Mobius augments type systems with proof-carrying code
  - In PCC, code is accompanied by a certificate that can be efficiently checked by the code consumer that the code conforms to security policies
  - Certificates can be produced via traditional verification, static analysis, type systems

- Compared to PDGs
  - PDGs are much more expensive, less scalable
  - But fewer false alarms (????)

Related work – CEGAR

- So far applied very successfully to
  - Software model checking
    - Involves predicate abstraction
    - Detects reachability of the error states in the context of temporal safety properties
    - Examples include SLAM, BLAST
  - Data structure analysis
    - Uses proof of unsatisfiability
    - Checks functional properties of structure-rich code
    - But purely static, based on SAT solving, and finite domain analysis
    - Examples include Karun
- CEGAR provides a scalable analysis by
  - Analyzing code incrementally, only on demand
  - Performing refinements locally