EXERCISE #18

INTERPROCEDURAL ANALYSIS REVIEW

Write your name and answer the following on a piece of paper

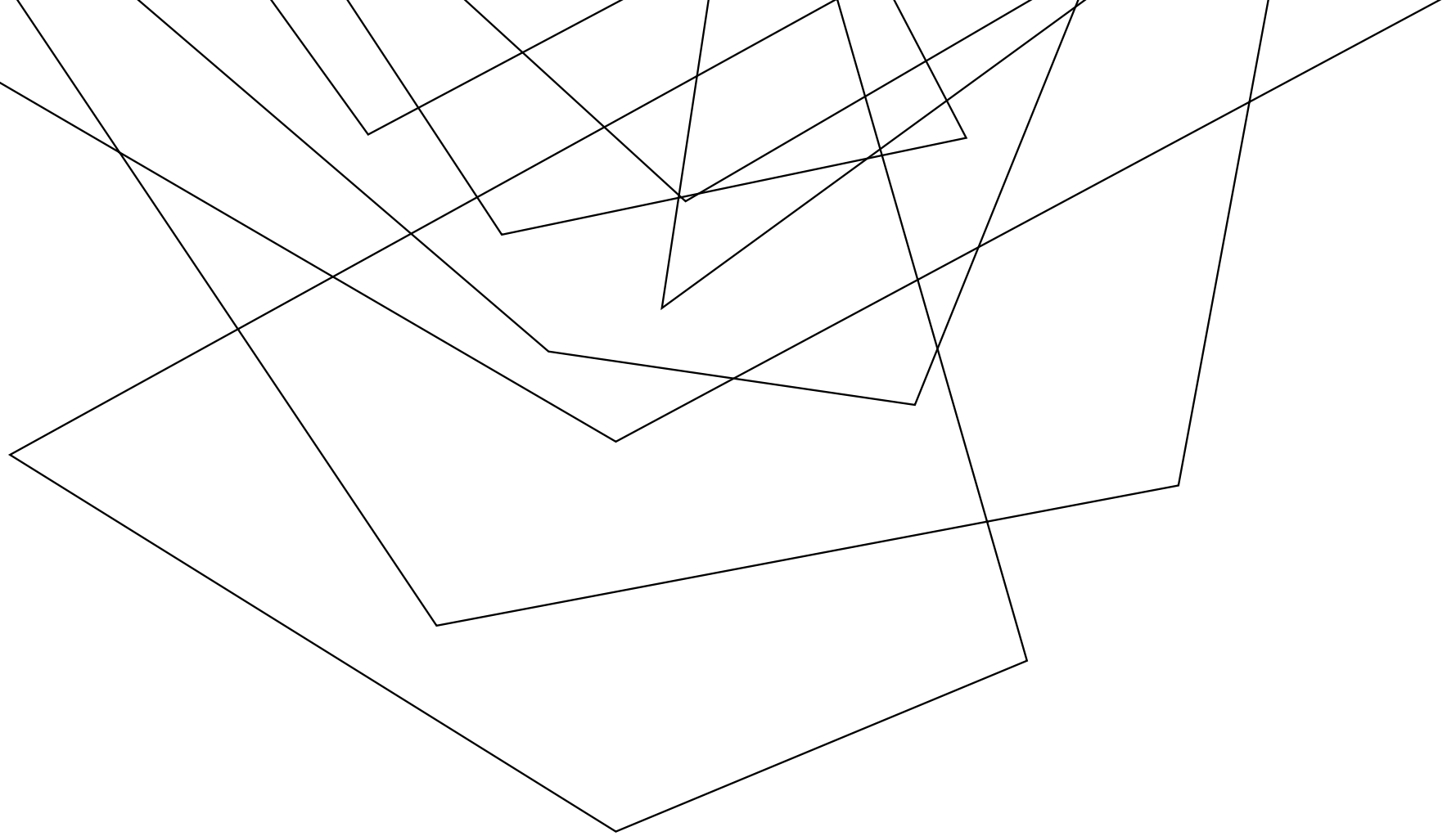
Draw the exploded supergraph of the following program:

```
 1: int baz(int arg){
 2:     if (arg > 2){
 3:         baz(arg-1);
 4:     }
 5: }
 6:
 7: int bar(){
 8:     baz(3);
 9: }
10:
11: int foo(){
12:     int a = bar();
13:     int b = bar();
14:     return a + b;
15: }
16:
17: int main(){
18:     foo();
19: }
```

EXERCISE #18 SOLUTION

INTERPROCEDURAL ANALYSIS REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS



SUMMARY FUNCTIONS

EECS 677: Software Security Evaluation

Drew Davidson

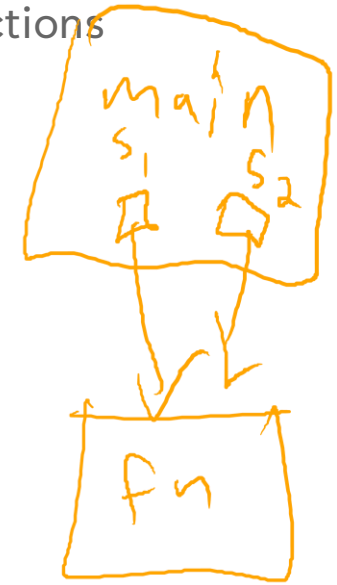
LAST TIME: INTERPROCEDURAL ANALYSIS

REVIEW: LAST LECTURE

CALL GRAPHS

Simple:
- nodes are functions
- edges are caller-callee relations

Composite:
- Sub-nodes are call sites, grouped into functions
- edges are call site-callee relations




```c
int g;
int v1;
int v2;
int fn(int a){
        if (a > 1){
                return 0;
        }
        return 1;
}

int main(){
        g = 1;
        v1 = fn(1);
        v1 = fn(v1);
        v2 = v1 / g;
        return v2 / v1;
}
```

LAST TIME: EXTENDING DATAFLOW

REVIEW: LAST LECTURE

CONSIDER THE EFFECT OF MULTIPLE FUNCTIONS

Abject Pessimism

- Function call overturns all global / aliased facts

Supergraph / Context String

- Stitch together multiple CFGs according to the composite call graph targets


```
int g;
int v1;
int v2;
int fn(int a){
        if (a > 1){
                return 0;
        }
        return 1;
}

int main(){
        g = 1;
        v1 = fn(1);
        v1 = fn(v1);
        v2 = v1 / g;
        return v2 / v1;
}
```

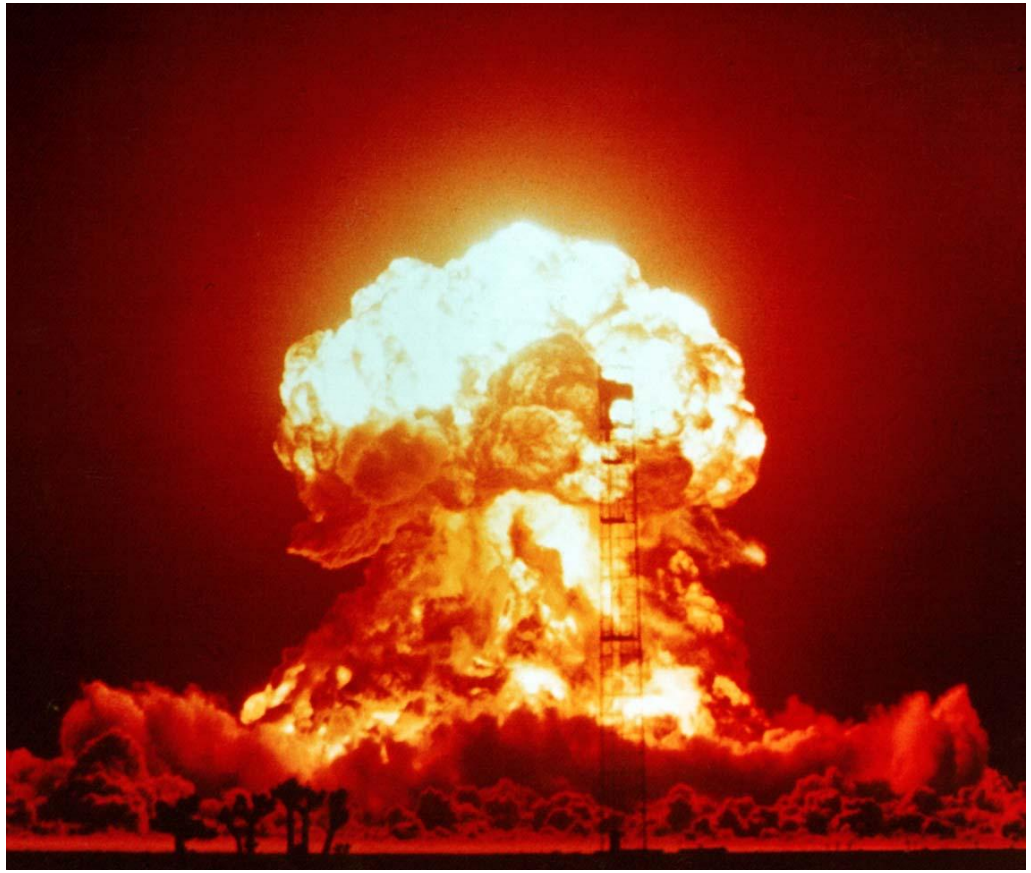
EXPLODING SUPERGRAPHS

The Exploded Supergraph... Explodes

For large programs, the supergraph may be too large (and exploding the supergraph certainly will not help)

What can we do in the presence of such limitations?

Gather a lightweight, over-approxmation of the effect of a function call



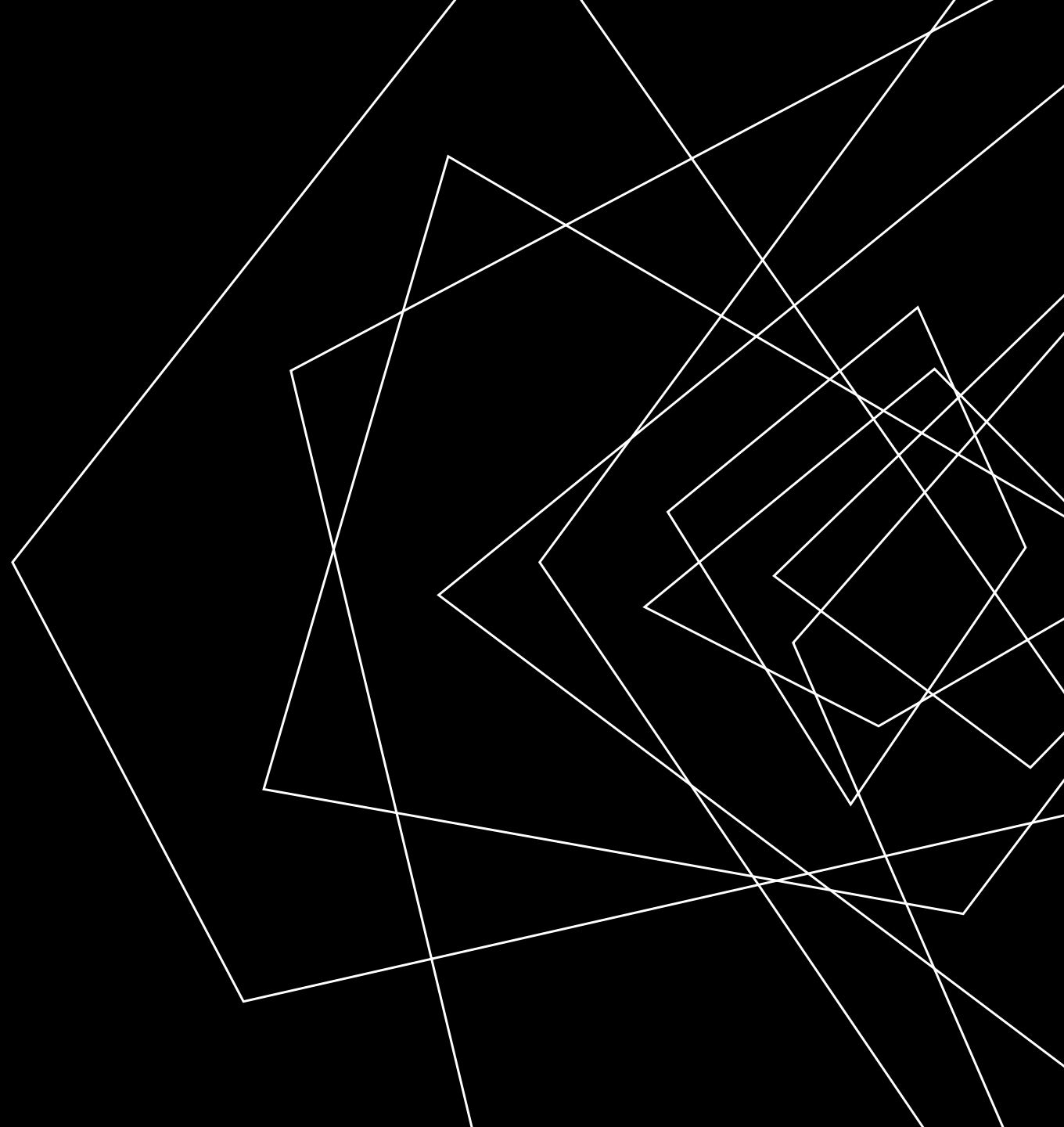
LECTURE OUTLINE

Intuition

MOD/REF analysis

- Global only
- Globals, Locals and args

Abstract Summaries



INTUITION
SUMMARY FUNCTIONS

Tracking context is expensive

Maybe our analysis can get by without it

Coarse-Grained analysis need only capture coarse-grained function information

“Summarize” the information we need to know

INTUITION
SUMMARY FUNCTIONS

Tracking context is expensive

Maybe our analysis can get by without it

Coarse-Grained analysis need only capture coarse-grained function information

"Summarize" the information we need to know

Function summary

```
int main () {
  global1 = SOURCE();
  foo();
  SINK(global2);
}
```

Does foo...
reference (i.e. read) global1?
Modify (i.e. write) global2?

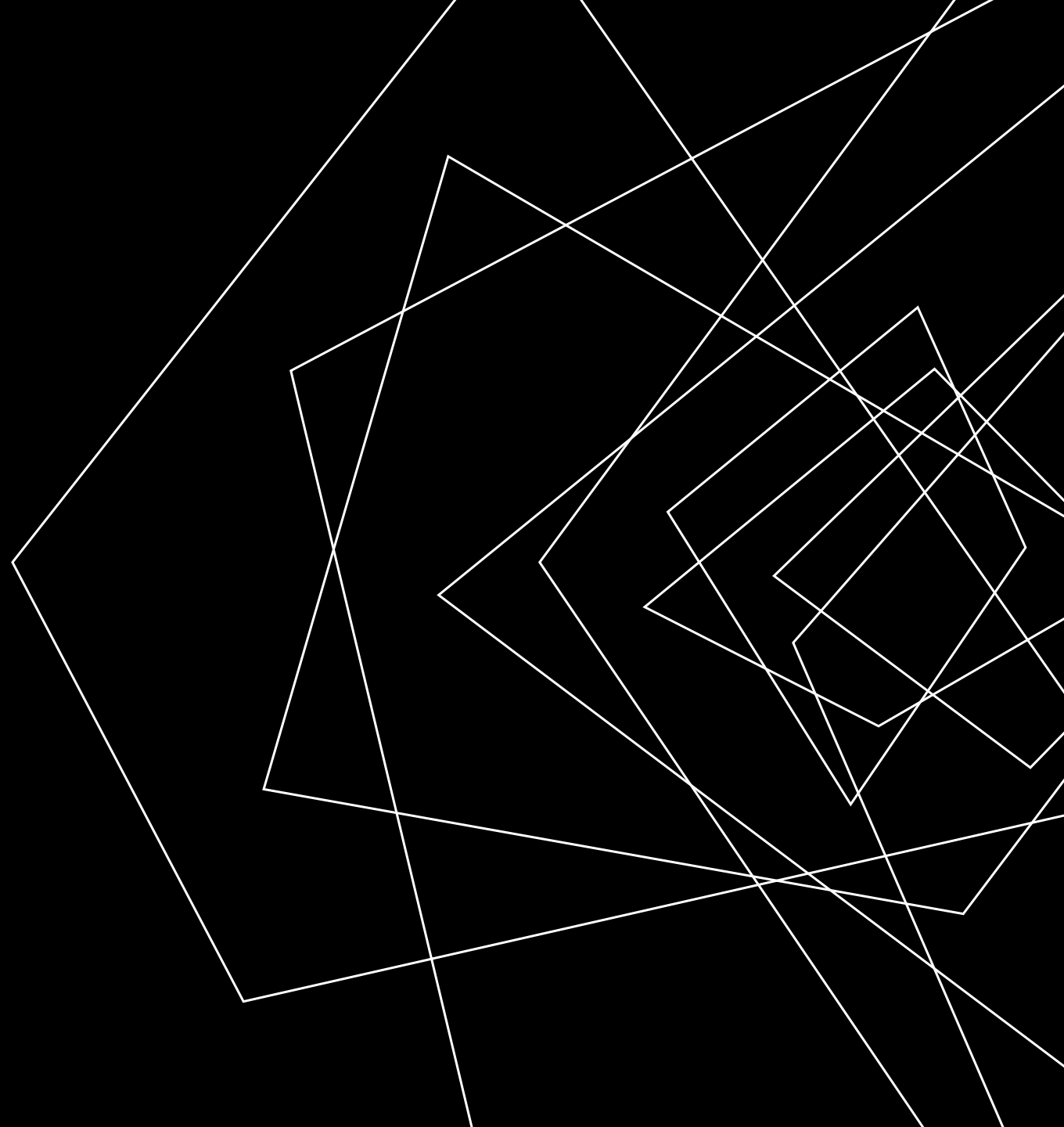
LECTURE OUTLINE

Intuition

MOD/REF analysis

- Global only
- Globals, Locals and args

Abstract Summaries



INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

LET US ATTEMPT TO COMPUTE 2 SETS

GMOD(P) – The set of variables that might be modified as a result of calling P

GREF(P) – The set of variables that might be referenced as a result of calling P

Also includes variables mod/ref'ed by P's callees!

BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation

```
1:  int A,B,C,D;
2:  void baz(){
3:    D = B;
4:    bar();
5:  }
6:  void bar(){
7:    B = 2;
8:    cout << C;
9:    baz();
10: }
11: void foo(){
12:    A = 1;
13:    bar();
14: }
15: int main(){
16:    foo();
17:    foo();
18: }
19:
```

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation

"Good enough" initial approximation:
Simple statement scan

IMOD(main) = { } IREF(main) = { }
IMOD(foo) = { A } IREF(foo) = { }
IMOD(bar) = { B } IREF(bar) = { C }
IMOD(baz) = { D } IREF(baz) = { B }

```
 1: int A,B,C,D;
 2: void baz(){
 3:   D = B;
 4:   bar();
 5: }
 6: void bar(){
 7:   B = 2;
 8:   cout << C;
 9:   baz();
10: }
11: void foo(){
12:    A = 1;
13:    bar();
14: }
15: int main(){
16:    foo();
17:    foo();
18: }
19:
```

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

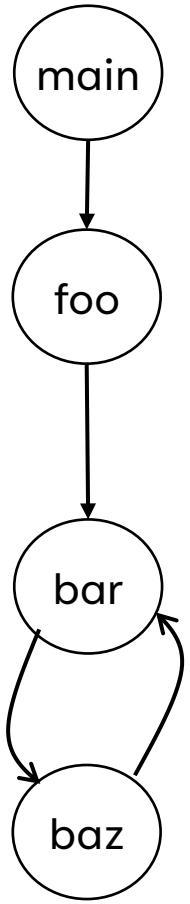
BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation




```
 1: int A,B,C,D;
 2: void baz(){
 3:   D = B;
 4:   bar();
 5: }
 6: void bar(){
 7:   B = 2;
 8:   cout << C;
 9:   baz();
10: }
11: void foo(){
12:    A = 1;
13:    bar();
14: }
15: int main(){
16:    foo();
17:    foo();
18: }
19:
```

IMOD(main) = { } IREF(main) = { }

IMOD(foo) = { A } IREF(foo) = { }

IMOD(bar) = { B } IREF(bar) = { C }

IMOD(baz) = { D } IREF(baz) = { B }

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

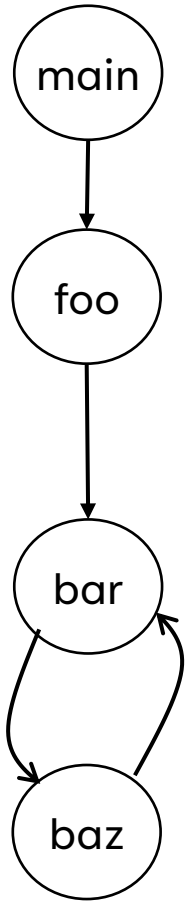
BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

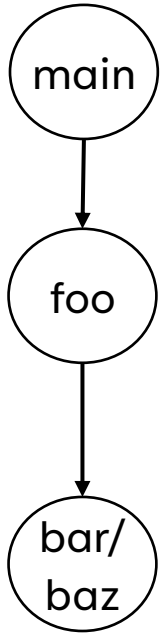
Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation



(optimization)
collapse cycles



IMOD(main) = { } IREF(main) = { }

IMOD(foo) = { A } IREF(foo) = { }

IMOD(bar) = { B } IREF(bar) = { C }

IMOD(baz) = { D } IREF(baz) = { B }

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

GMOD: $f_P(S) = S \cup IMOD(P)$
GREF: $f_P(S) = S \cup IREF(P)$
Init GMOD: {}
Init GREF: {} Join = Union

BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

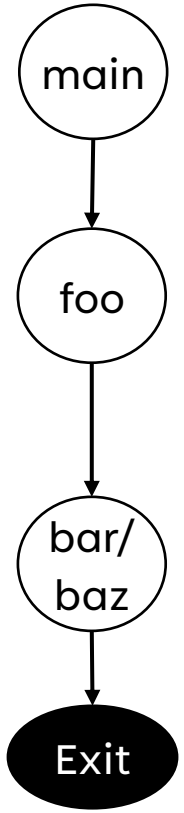
Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation

(optimization) collapse cycles

Add a dummy exit node targeted by all leaves



IMOD(main) = { } IREF(main) = { }
IMOD(foo) = { A } IREF(foo) = { }
IMOD(bar) = { B } IREF(bar) = { C }
IMOD(baz) = { D } IREF(baz) = { B }

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

GMOD: $f_P(S) = S \cup IMOD(P)$
GREF: $f_P(S) = S \cup IREF(P)$
Init GMOD: {}
Init GREF: {} Join = Union

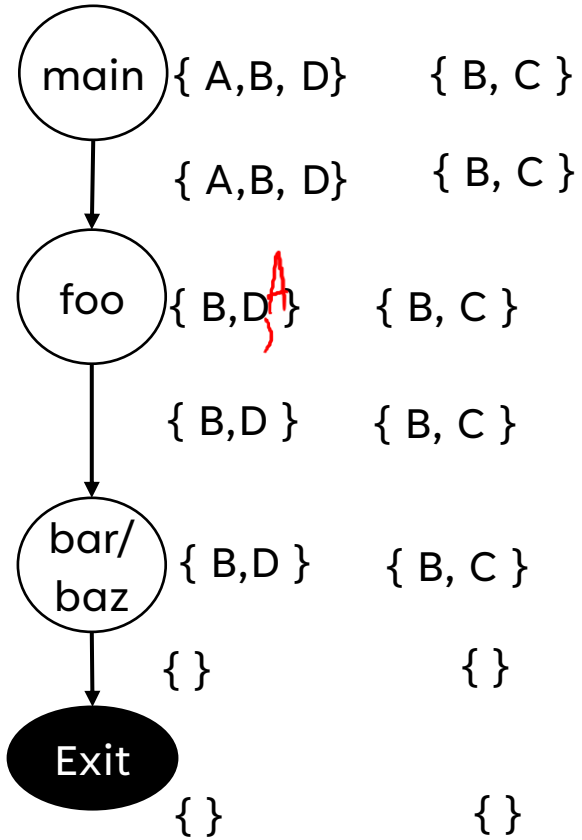
BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation



IMOD(main) = { } IREF(main) = { }
IMOD(foo) = { A } IREF(foo) = { }
IMOD(bar) = { B } IREF(bar) = { C }
IMOD(baz) = { D } IREF(baz) = { B }

INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

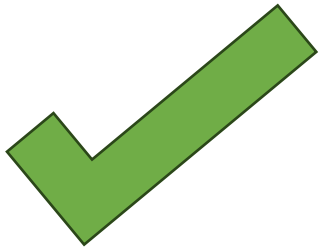
Basic idea (let's ignore parameters, pointers, and Locals for now)

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the simple call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation



This is a pretty big restriction,
we should remove it

Good news:
GMOD computation is the same

Bad news:
We'll need to use the compound call graph
More elaborate IMOD computation
Can't collapse cycles

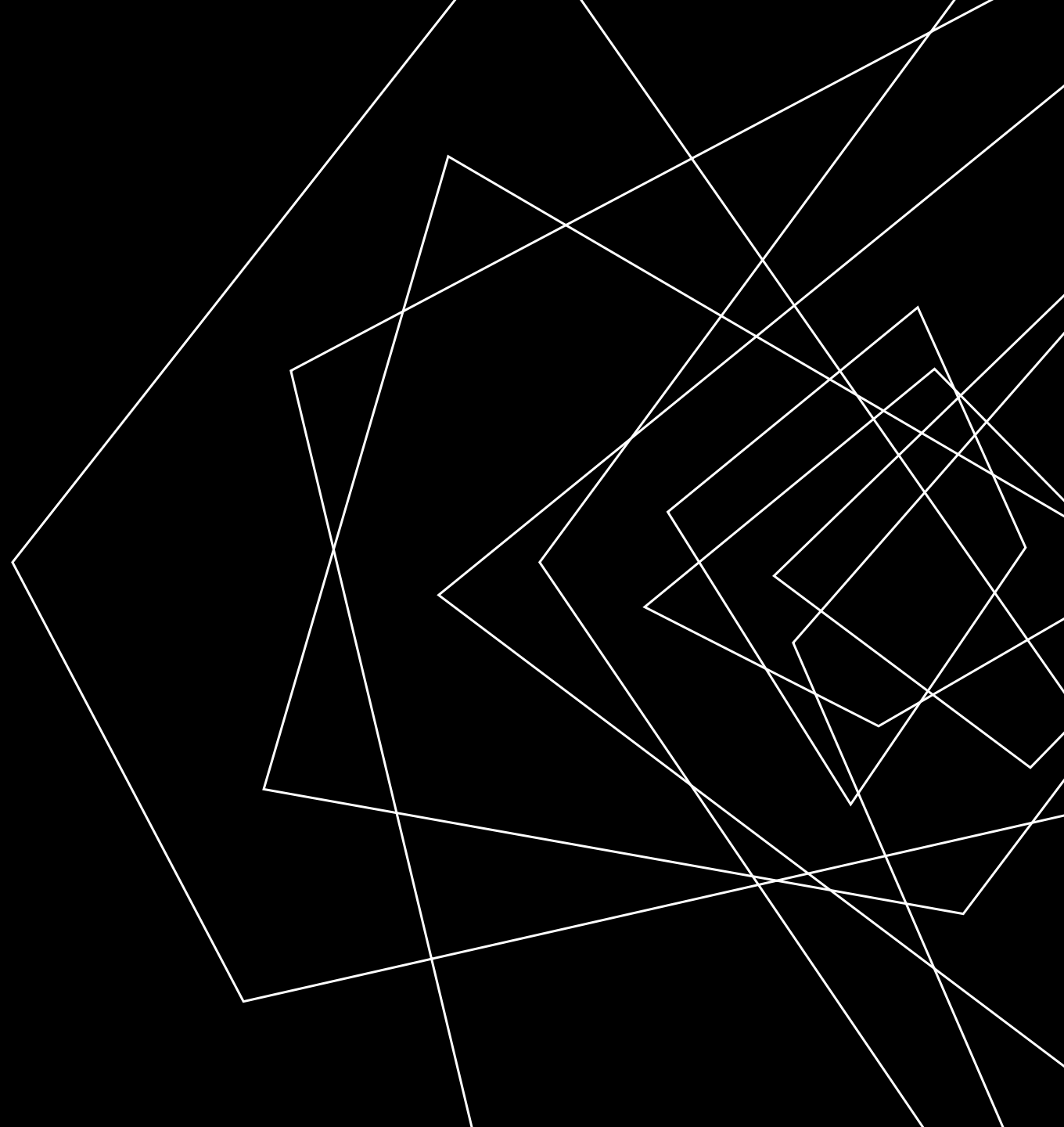
LECTURE OUTLINE

Intuition

MOD/REF analysis

- Global only
- Globals, Locals and args

Abstract Summaries



INTERPROCEDURAL MOD/REF ANALYSIS

SUMMARY FUNCTIONS

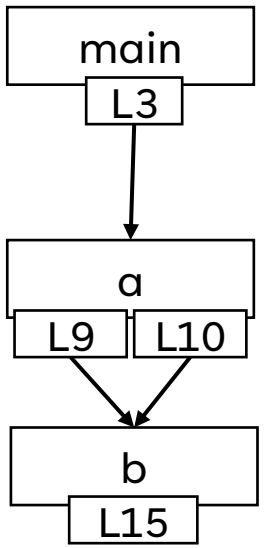
FULL IDEA

Build IMOD(P) and IREF(P) – the variables immediately modified in P (ignoring callees)

Build the composite call graph

Repurpose the call graph for dataflow algorithm

Run the dataflow algorithm to saturation




```
 1 void main() {
 2   int v1;
 3   call a(v1);
 4 }
 5
 6
 7 void a( int f1){
 8   int v2, v3, v4, v5;
 9   call b(v2, v3);
10   call b(v4, v5);
11 }
12
13 void b( int f2, int f3){
14   cout << f3;
15   b(g1,g2);
16 }
```

IMOD(main) = { } IREF(main) = { }

IMOD(foo) = { A } IREF(foo) = { }

IMOD(bar) = { B } IREF(bar) = { C }

IMOD(baz) = { D } IREF(baz) = { B }

GMOD & GREF COMPUTATION

REVIEW: LAST LECTURE

GLOBALS, LOCALS & VALUE-PASSING

GREF will change, GMOD doesn't need to change

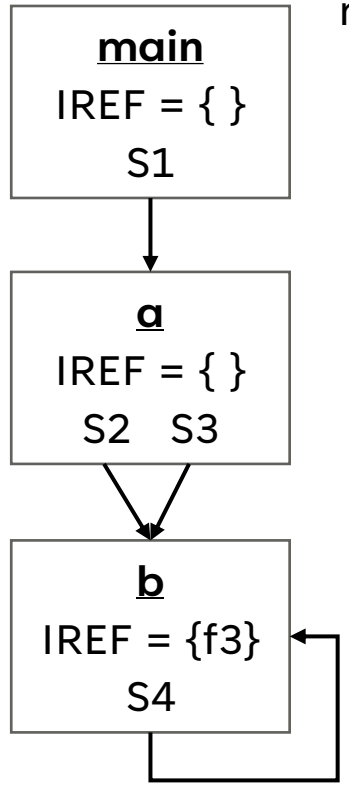
Init all node GREF sets to their IREF sets
Init all call site GREF sets to empty
Put all nodes and call sites on a worklist
Iterate until the worklist is empty.

Each time a node n is removed from the worklist, its current GREF set is computed. If that set doesn't match its previous value, then add all call sites to n to the worklist (if not present).
Each time a call site s is removed from the worklist, its current GREF set is computed. If that set doesn't match its previous value, then the node that contains s is added to the worklist (if not present).

```
void main() {
S1: call  a(v1)
}

void a( f1 ) {
S2: call b(v2, v3)
S3: call b(v4, v5)
}

void b( f2, f3 ) {
      print f3;
S4: call b(g1, g2)
}
```



main	{ g2 }
S1	{ g2 }
a	{ g2, v3, v5 }
S2	{ v3, g2 }
S3	{ v5, g2 }
b	{ f3, g2 }
S4	{ v5, g2 }

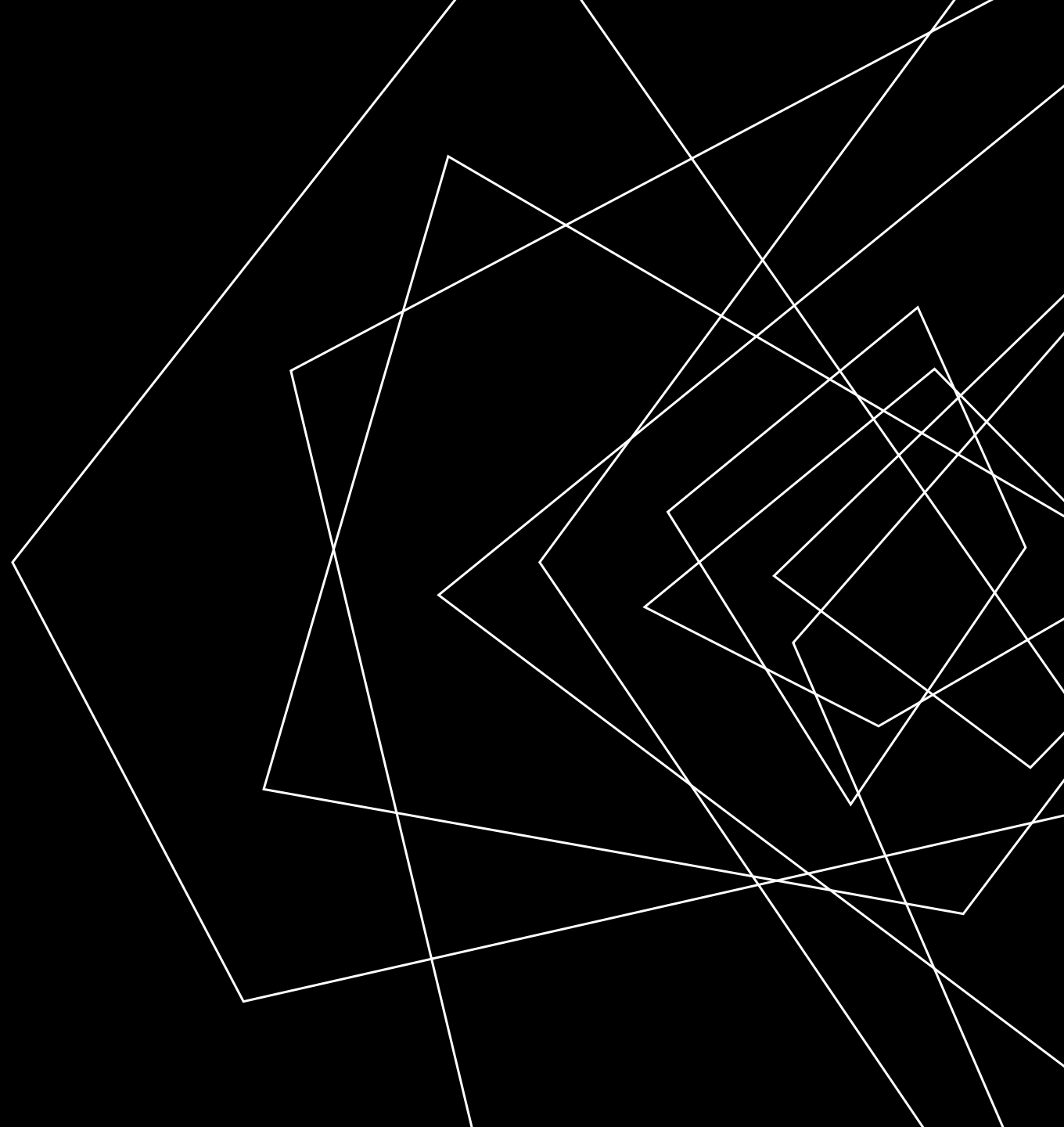
LECTURE OUTLINE

Intuition

MOD/REF analysis

- Global only
- Globals, Locals and args

Abstract Summaries



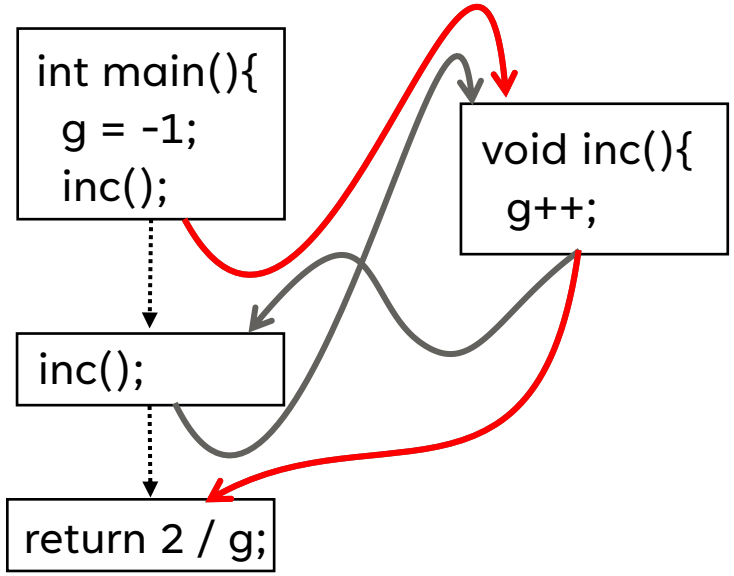
ABSTRACT SUMMARIES

SUMMARY FUNCTIONS

LET'S RECALL THE PROBLEM THAT GOT US INTO THIS MESS

Summarize callee analysis (rather than include it in the analysis)

```
 1 int g;
 2
 3 void inc(){
 4         g++;
 5 }
 6
 7 void main(){
 8         g = -1;
 9         inc();
10         inc();
11         return 2 / g;
12 }
```

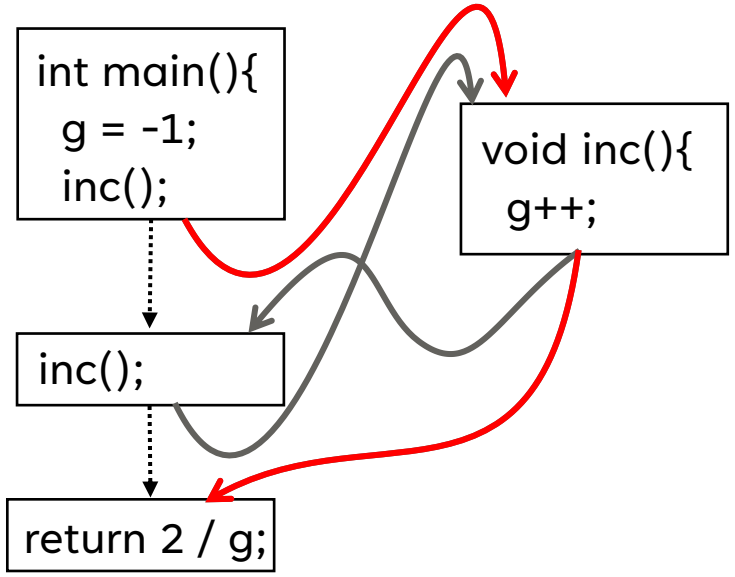


ABSTRACT SUMMARIES

SUMMARY FUNCTIONS

WHAT IF THE CALLEE ISN'T SO TRICKY

Summarize callee analysis (rather than include it in the analysis)



WRAP-UP