# EXERCISE #18

## *REVIEW INTERPROCEDURAL ANALYSIS*

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

*Draw the exploded supergraph for the following program:*

```
int baz(int arg){
        if (arg > 2){
                baz(arg-1);
        }
        return 1;
}

int bar(){
        baz(3);

}

int foo(){
        int a = bar();
        int b = bar()
        return a + b;
}

int main(){
        foo();
}
```

# EXERCISE #18: SOLUTION

*REVIEW INTERPROCEDURAL ANALYSIS*

# ADMINISTRIVIA AND ANNOUNCEMENTS

# SUMMARY FUNCTIONS

EECS 677: Software Security Evaluation

Drew Davidson

# LAST TIME: INTERPROCEDURAL ANALYSIS

## REVIEW: LAST LECTURE

### CONSIDER THE EFFECT OF MULTIPLE FUNCTIONS

**Simplistic**

– Function call overturns all global / aliased facts

**Supergraph / Context String**

– 1-CFA (use a call-chain of 1)

```
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

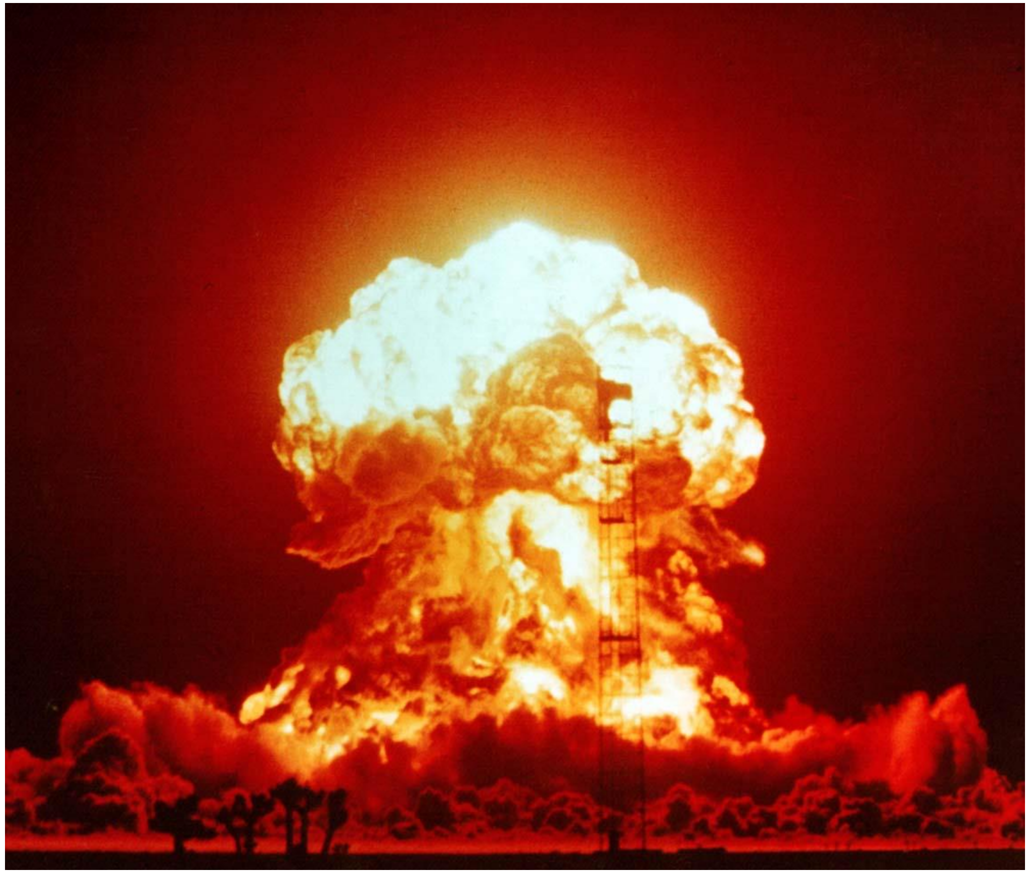
## SUPERGRAPHS

### 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

*Function summary*

# 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*

```
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**

```
int A,B,C,D;
void baz(){
  D = B;
  bar();
}
void bar(){
  B = 2;
  cout << C;
  baz();
}
void foo(){
   A = 1;
   bar();
}
int main(){
   foo();
   foo();
}
```

# 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 }

```
int A,B,C,D;
void baz(){
  D = B;
  bar();
}
void bar(){
  B = 2;
  cout << C;
  baz();
}
void foo(){
   A = 1;
   bar();
}
int main(){
   foo();
   foo();
}
```

# INTERPROCEDURAL MOD/REF ANALYSIS

## SUMMARY FUNCTIONS

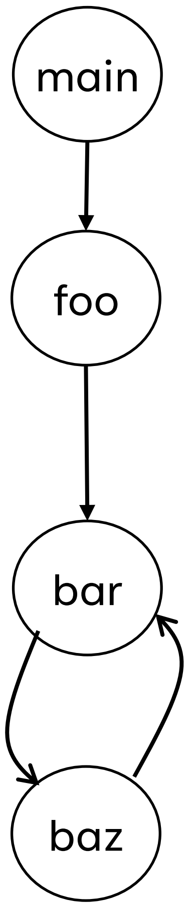
### Basic idea (let's ignore parameters, pointers, and Locals for now)

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




```
int A,B,C,D;
void baz(){
  D = B;
  bar();
}
void bar(){
  B = 2;
  cout << C;
  baz();
}
void foo(){
   A = 1;
   bar();
}
int main(){
   foo();
   foo();
}
```

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

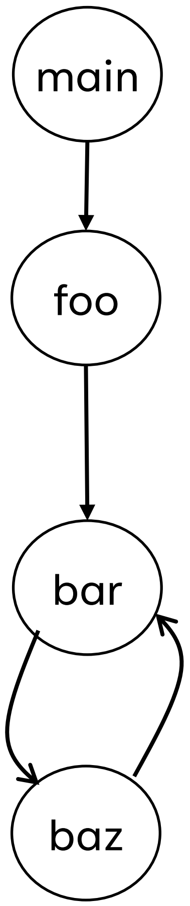
### BASIC IDEA (LET'S IGNORE PARAMETERS, POINTERS, AND LOCALS FOR NOW)

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

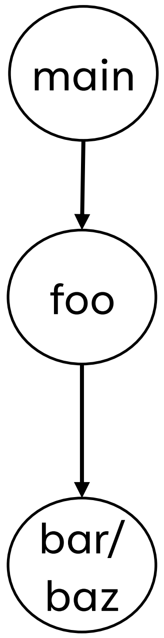
**Build the simple call graph**

**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)**

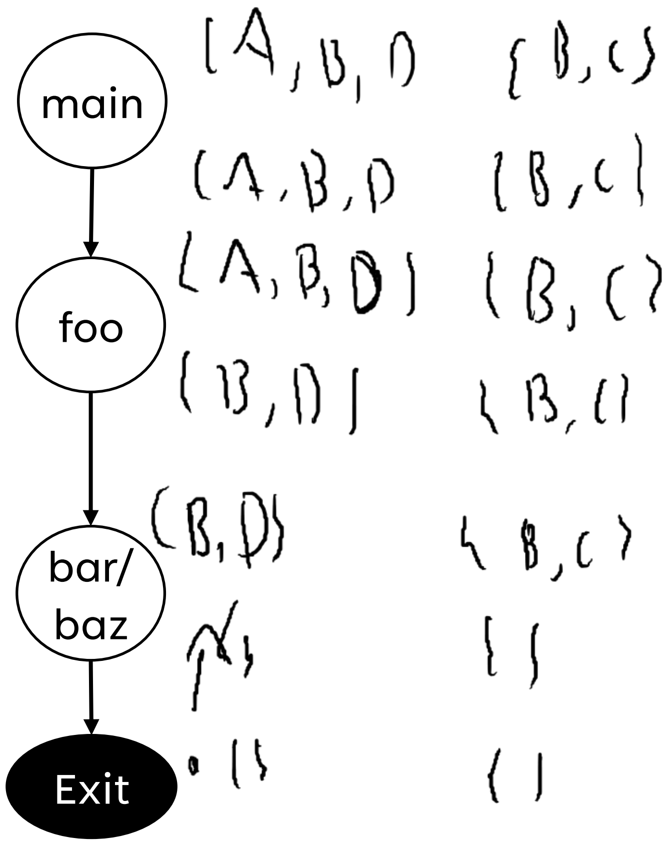
**Build the simple call graph**

**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

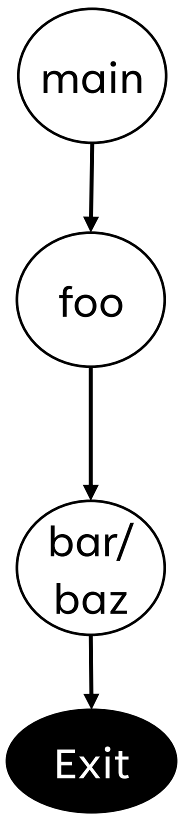
### Basic idea (let's ignore parameters, pointers, and Locals for now)

**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

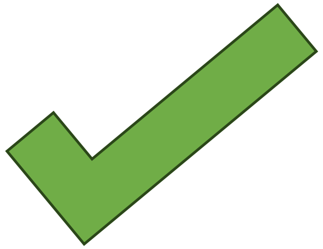
### Basic idea (let's ignore parameters, pointers, and Locals for now)

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

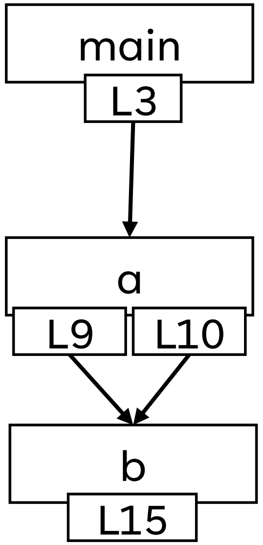
### FULL IDEA

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

Build the compound call graph

Repurpose the call graph for dataflow algorithm!

Run the dataflow algorithm to saturation




```
void main() {
  int v1;
  call a(v1);
}


void a( int f1){
  int v2, v3, v4, v5;
  call b(v2, v3);
  call b(v4, v5);
}

void b( int f2, int f3){
  cout << f3;
  b(g1,g2);
}
```

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

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

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

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

# LAST TIME: GMOD & GREF COMPUTATION

**REVIEW: LAST LECTURE**

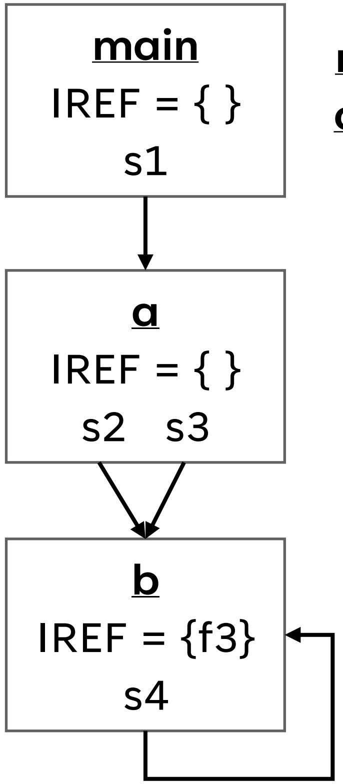
## Globals, Locals & Value-Passing

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)
}
```




| node or call site | GREF set |
|---|---|
| main | g2 |
| a | g2, v3, v5 |
| b | f3, g2 |
| s1 | g2 |
| s2 | v3, g2 |
| s3 | v5, g2 |
| s4 | g2 |

# LECTURE OUTLINE

Intuition

MOD/REF analysis

- Global only
- Globals, Locals and args

Abstract Summaries

# ABSTRACT SUMMARIES

## SUMMARY FUNCTIONS

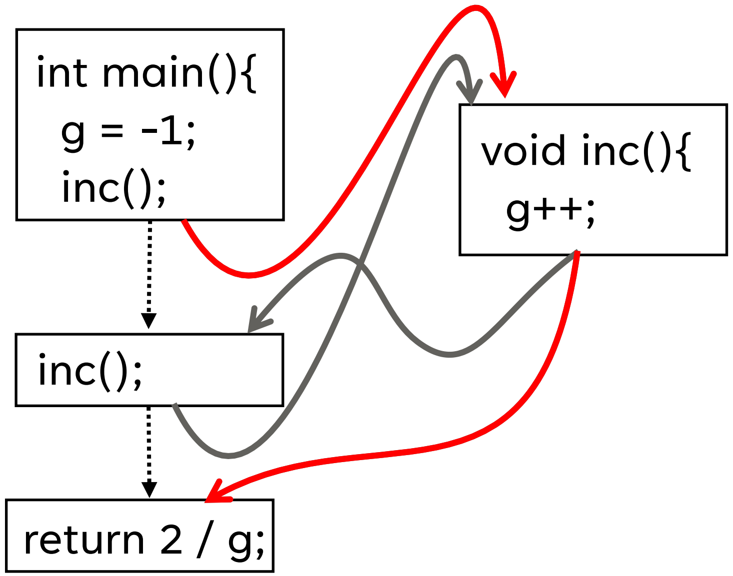
### LET'S RECALL THE PROBLEM THAT GOT US INTO THIS MESS

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

```
int g;

void inc(){
        g++;
}

void main(){
        g = -1;
        inc();
        inc();
        return 2 / g;
}
```

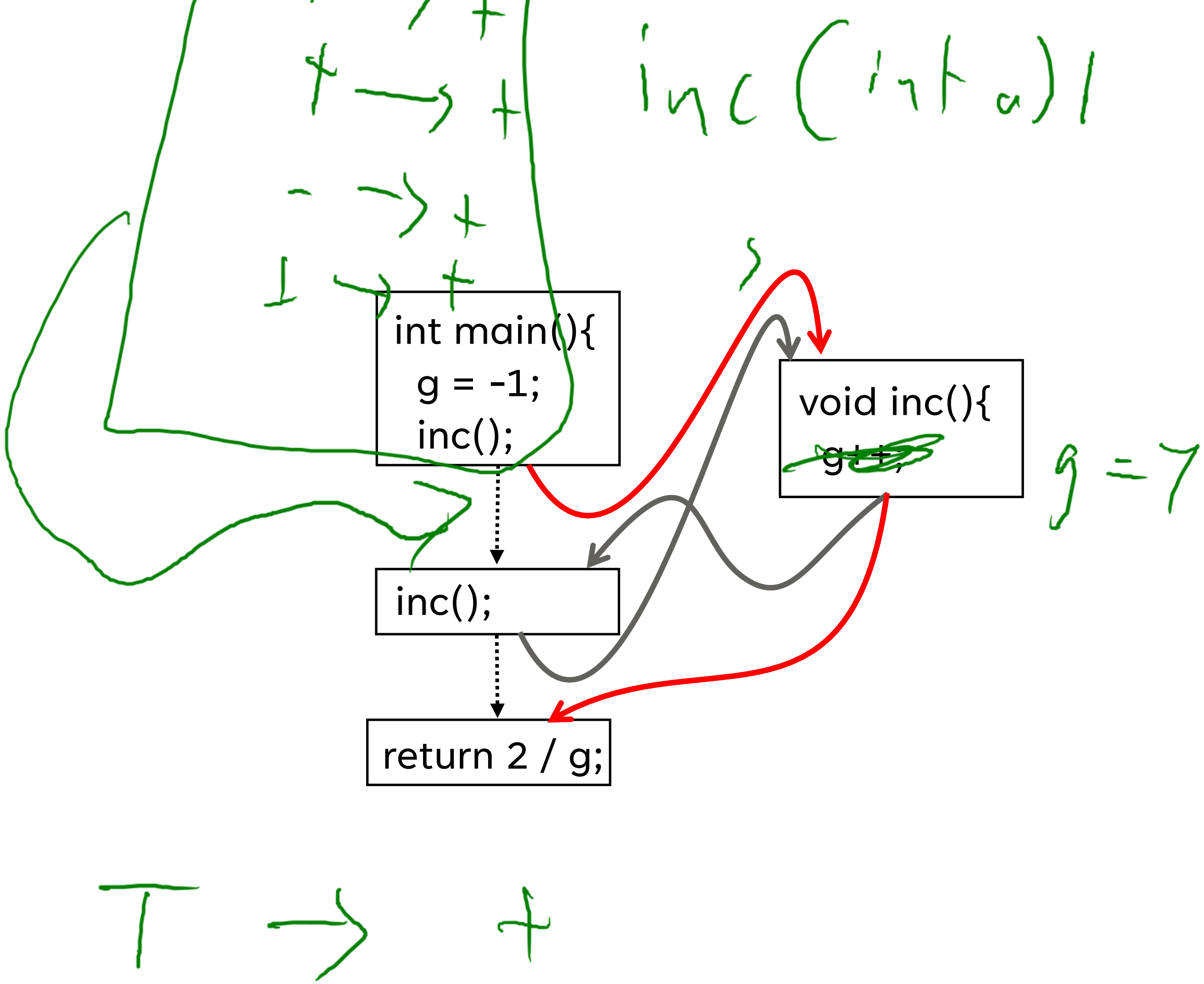


# ABSTRACT SUMMARIES

## SUMMARY FUNCTIONS

### What if the callee isn't so tricky

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



# WRAP-UP