EXERCISE #19

CALL GRAPHS REVIEW

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

Draw the callgraph that CHA would produce for the following program:

```
class SupClass{
public:
        virtual int fun(SupClass * in){
                in->fun();
                return 1;
        }
};
class SubA : public SupClass{
        int fun(SupClass * in){
                return 2;
        }
};
class SubB : public SupClass{
        int fun(SupClass * in){
                return 3;
        }
};
int main(){
        SupClass * s = new SubA();
        s->fun();
}
```

ADMINISTRIVIA AND ANNOUNCEMENTS

INTERPROCEDURAL ANALYSIS

EECS 677: Software Security Evaluation

Drew Davidson

CLASS PROGRESS

EXPLORING ANALYSES UNDERLYING OUR EVALUATION/ENFORCEMENT NEEDS

Intraprocedural analysis: Within a function

Interprocedural analysis: Between functions

LAST TIME: CALL GRAPHS

REVIEW: LAST LECTURE

DETERMINE WHERE A (POSSIBLY INDIRECT) CALL MIGHT GO

Motivation

- Powers some forms of CFI

Implementation

- Consider ALL functions
- Consider functions in the “cone” (CHA)
- Consider functions in the cone that might be used (RTA, MTA, FTA, XTA)

OVERVIEW

WE'VE SEEN THE NECESSITY OF MULTI-FUNCTION ANALYSIS IN REAL-WORLD PROGRAMS

TIME TO CONSIDER HOW IT IS DONE

WORST-CASE ASSUMPTIONS

NAÏVE APPROACH

CREATE SIMPLE, "SAFE" OVER-APPROXIMATION

What constitutes "being safe" depends on your analysis

- **Example 1, confidentiality:** Assume a function call tags all reachable data as confidential
- **Example 2, integrity:** Assume a function call tags all reachable data as untrusted

JUSTIFICATION

NAÏVE APPROACH

OUR GENERAL PHILOSOPHY: "DO NO HARM" GUARANTEES

Recall our notions of soundness and completeness:
- Sound: no false positives ("tells no lie")
- Complete: no false negatives ("omits no truth")

ANYTHING THAT **CAN** GO WRONG **WILL** GO WRONG

- MURPHY'S LAW

"BEING SAFE" REQUIRES FORMULATING ANALYSIS GOAL

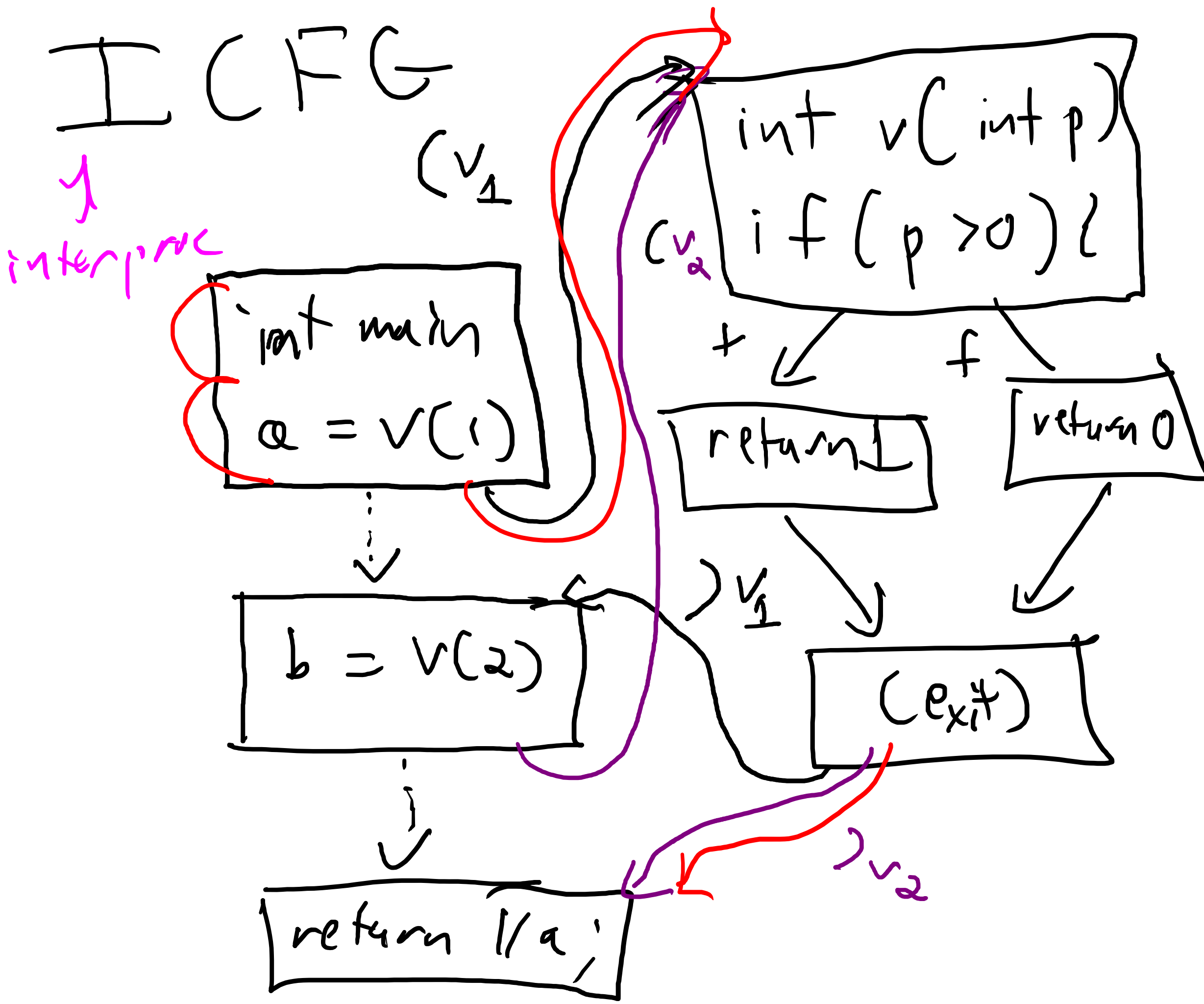
bug hunting:
- Report buggy programs
- Safe means complete analysis

program verification:
- Report clean programs
- Safe means sound analysis

```
int a;
int b;

int v( int p ){
    if (p>0){
      return 1;
    else {
      return 0;
    }
}
int main( ){

  a = v(1);

  b = v(2);

    return 1/a;
}
```

ICFG

← "Supergraph"



+ no extra machinery

− extra imprecision

STITCH TOGETHER CFGS
SUPERGRAPHS

USE THE ICFG (AKA "SUPERGRAPH")

Interprocedural control flow graph

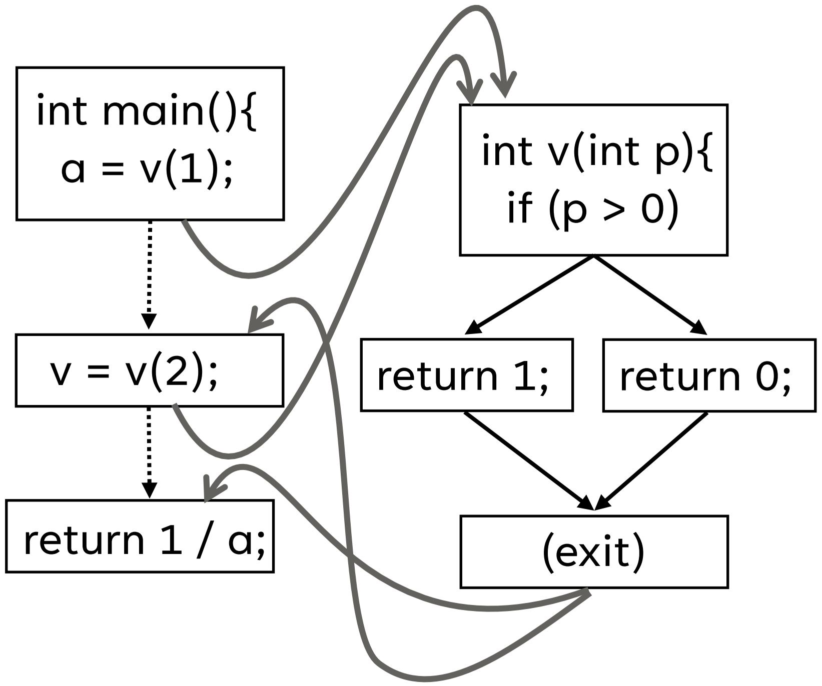
Benefits

- No extra machinery required

Detriments

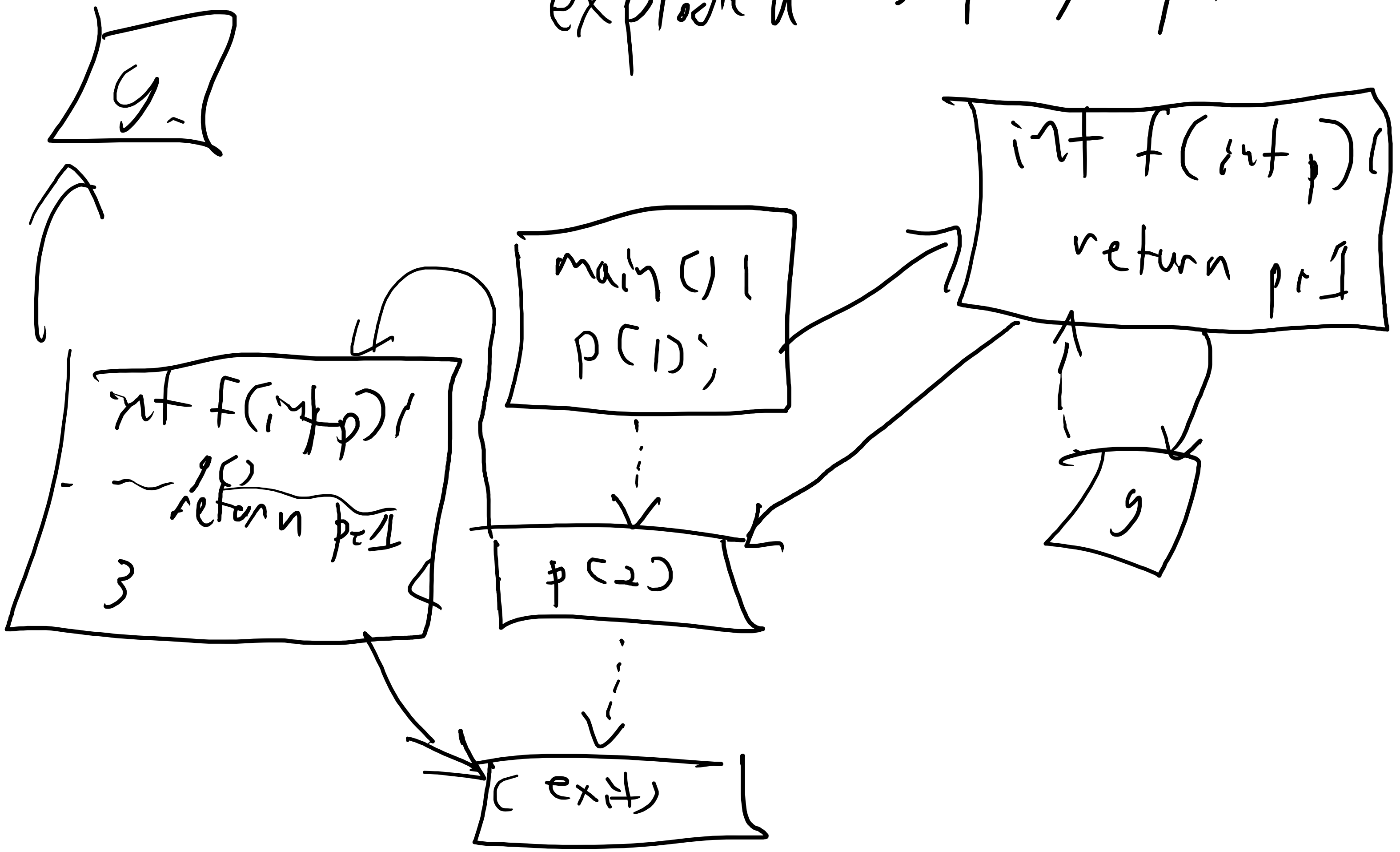
- Extra imprecision

```
int a;
int b;
int v(int p){
        if (p > 0)
                return 1;
        else
                return 0;
}

int main(){
        a = v(1);
        b = v(2);
        return 1 / a;
}
```



exploded supergraph

```
int f(int p){
  g();
  return p+1;

}

main() {
     p(1);
     p(2);

}
```



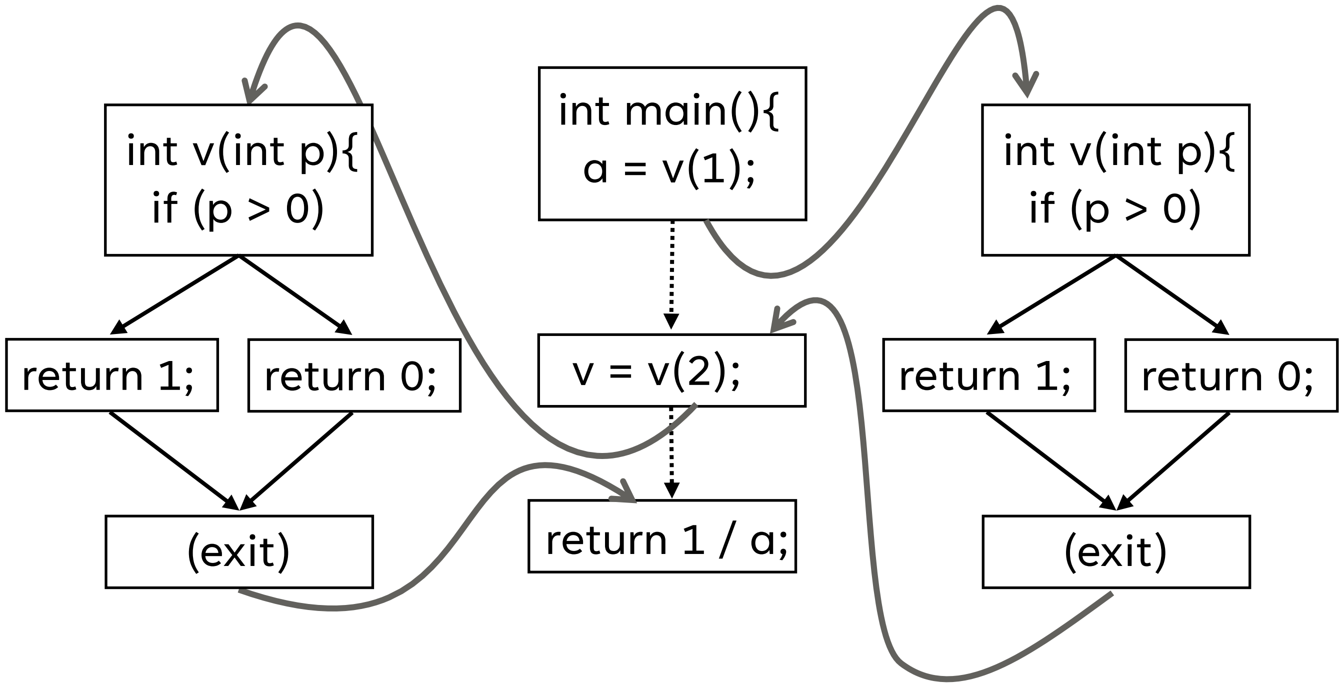
STITCH TOGETHER CFGS

SUPERGRAPHS

The Exploded Supergraph

Make a copy of the callee for each call site

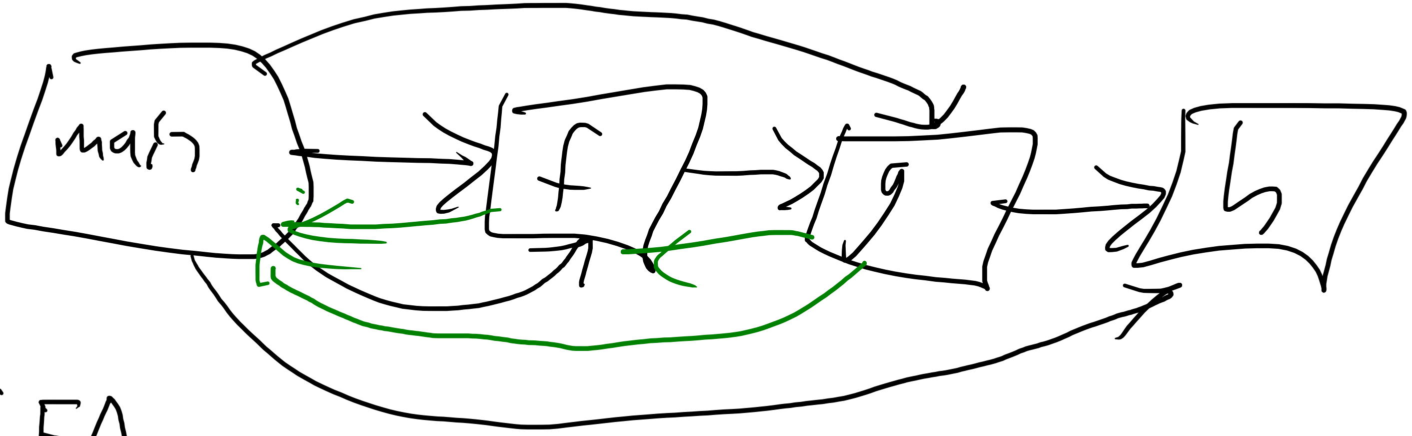
```
int a;
int b;
int v(int p){
        if (p > 0)
                return 1;
        else
                return 0;
}

int main(){
        a = v(1);
        b = v(2);
        return 1 / a;
}
```



How much context to keep?

- Recursion: unbounded depth of context

call string length



k – CFA

0: 0-CFA ⇐ context-insensitive

1-CFA ⇐ track caller, not caller's caller

Categorizing Static Dataflow

flow-sensitive ⇐ tracks order of statements within a function

context-sensitive ⇐ tracks the call string

CALL STRINGS

SUPERGRAPHS

PROVIDE A WAY TO SPECIFY DEGREE OF CONTEXT

Recursion: Unbounded depth of context

Call string depth

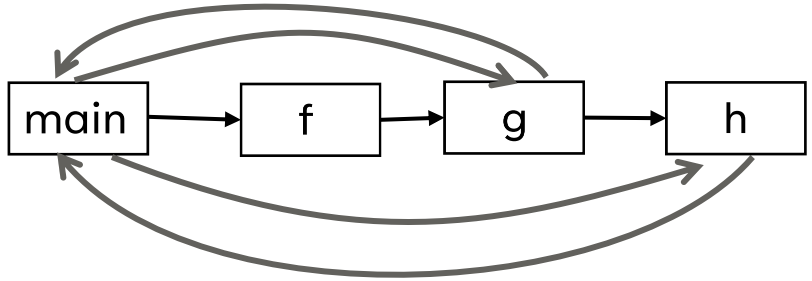
X-CFA, where X is the length of the call string

- 0-CFA: Context-insensitive
- 1-CFA: Tracker caller (but not caller's caller)

ANOTHER WAY TO TUNE STATIC ANALYSIS

Flow-sensitive: Unbounded depth of context

Context-sensitive: Track the call string



Approach 3: Summary functions

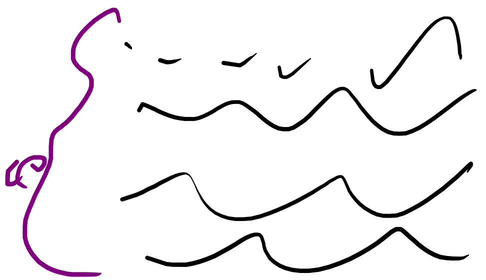
3.a: rely on human annotations

3.b: "worst case" summary for individual functions

```
main() {
    y = secret
    f();
    Leak(g);
}
```

```
f() {
```

no reference to g



```
}
```

g ~~mod~~ ref = global effect on ~~modifying~~ used variable by calling some function

SUMMARY FUNCTIONS

SUPERGRAPHS

Big Idea

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

Manual Manifestation

Ask the user to provide information

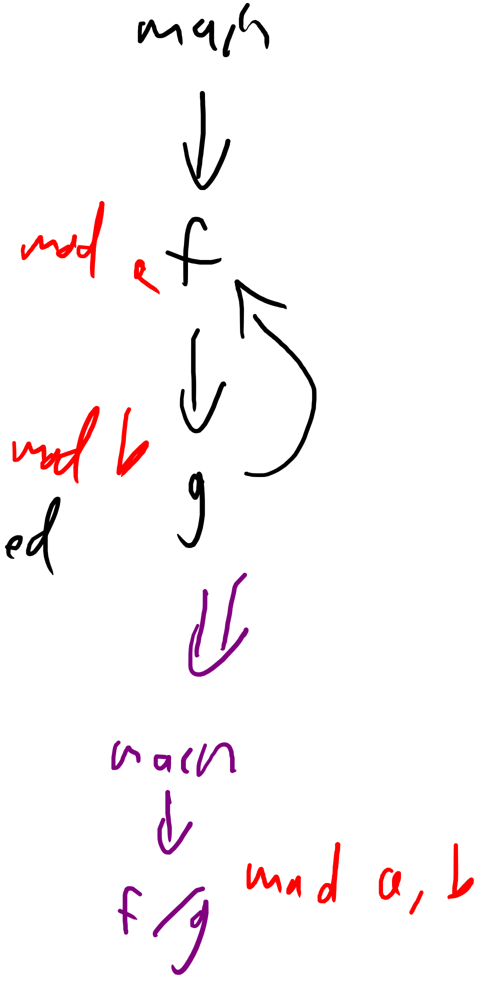
Automatic Manifestation

Create a lightweight inference
One version: GMOD and GREF
- What variables are (transitively) modified as a result of a function call?
- What variables are (transitively) referenced as a result of a function call?

Automatically Building Summary Functions

Version I: programs only have globals

1) construct call graph
2) Initialize G-mod(F) set of variables directly changed in statements of F
   G-ref(F) " " " " read in statements of F
3) Collapse SCCs of the callgraph
4) Add a dummy edge from leaves to dummy exit
5) Backwards dataflow enhanced callgraph



GMOD AND GREF

SUPERGRAPHS

VERSION 1: GLOBALS ONLY

Step 1: Construct Call Graph, normalize program assignments
Step 2: Initialize GMod and Gref
- GMod: initialize to variables on the LHS of assignments
- Gref: initialize to variabels on the RHS of statements

Step 3: Collapse SCCs
Step 4: Add a dummy edge from leaves to dummy exit
Step 5: Do a backwards dataflow on the augmented callgraph