

LAST TIME: CONTROL DEPENDENCE

REVIEW: LAST LECTURE

FOCUS THE ANALYSIS ON WHAT WE CARE ABOUT

Control Dependence Graph (CDG)

- Shows what program statements most immediately decide which others execute

implicit flow

```
if (x) {
```

```
    y = true;
```

```
    else { y = false; }
```



DOM/FDOM INTUITION

REVIEW: LAST LECTURE

DOMINATION INTUITION

$\text{DOM}(X,Y)$ – Paths **to** Y must go through X

You cannot get to Y without going through X

X “guards” Y



FORWARD DOMINATION INTUITION

$\text{FDOM}(X,Y)$ – Paths **from** X must go through Y

You cannot avoid Y after going through X

X “is destined for” Y



IMMEDIACY

REVIEW: LAST LECTURE

Immediate

DOMINATION INTUITION

| $\text{DOM}(X,Y)$ – Paths to Y must go through X
with no intervening node that
paths *must* go through to Y

X “is the closest guard of” Y



Immediate

FORWARD DOMINATION INTUITION

| $\text{FDOM}(X,Y)$ – Paths from X must go through Y
With no intervening node that
paths *must* go through from X

X’s “first guaranteed successor is” Y



CONTROL DEPENDENCE INTUITION

REVIEW: LAST LECTURE

We'd like to express that getting to Y
depends on what happens in X

$Y \text{ CD } X \Leftrightarrow$ there is a CFG-path from X to Y omitting IFDOM(X)

It's possible to get from X to Y But it's not guaranteed

CONTROL DEPENDENCE GRAPH

DEPENDENCE GRAPH REVIEW

BONUS EXERCISE

DEPENDENCE RELATIONS REVIEW

Draw the Control Dependence Graph for the following program

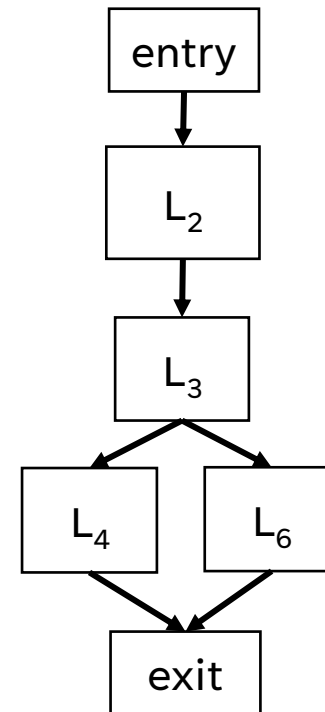
```
1 int main(){
2     i = getchar();
3     if ( i == 1 ){
4         printf("hi!");
5     } else {
6         i = 1;
7     }
8 }
```

BONUS EXERCISE

DEPENDENCE RELATIONS REVIEW

Draw the Control Dependence Graph of instructions for the following program

```
1 int main(){
2     i = getchar();
3     if ( i == 1 ){
4         printf("hi!");
5     } else {
6         i = 1;
7     }
8 }
```

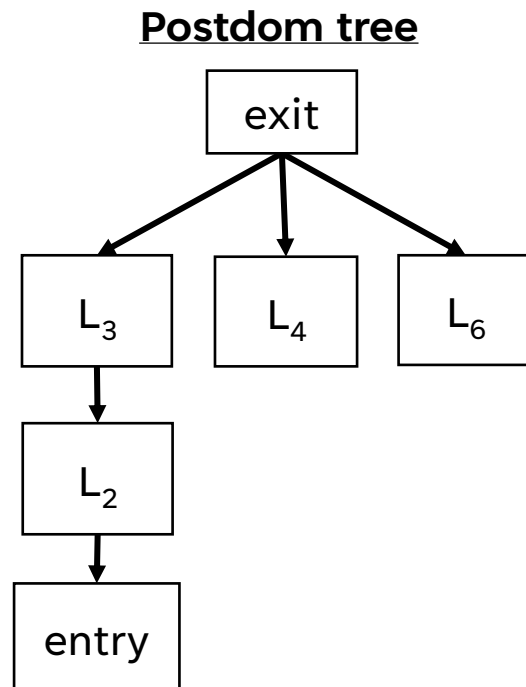
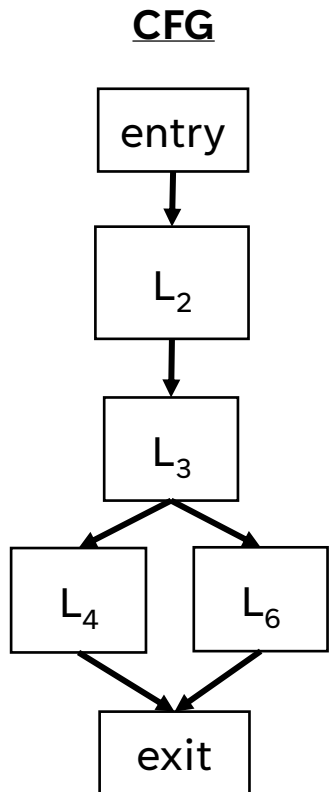


$Y \text{ CD } X \Leftrightarrow$ there is a CFG-path from X to Y omitting $\text{IFDOM}(X)$

BONUS EXERCISE

DEPENDENCE RELATIONS REVIEW

Draw the Control Dependence Graph of instructions for the following program



IFDOM(entry, L₂)

IFDOM(L₂, L₃)

IFDOM(L₃, exit)

IFDOM(L₄, exit)

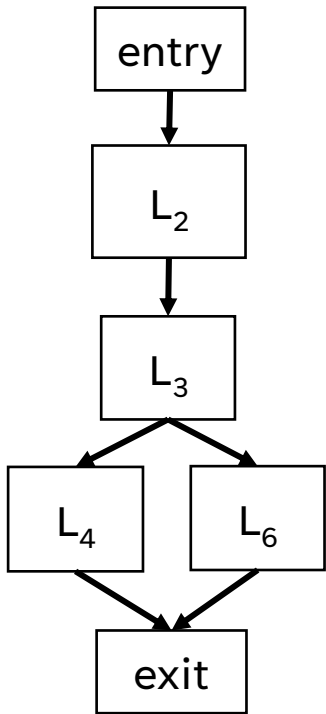
IFDOM(L₆, exit)

$Y \text{ CD } X \Leftrightarrow$ there is a CFG-path from X to Y omitting IFDOM(X)

BONUS EXERCISE

DEPENDENCE RELATIONS REVIEW

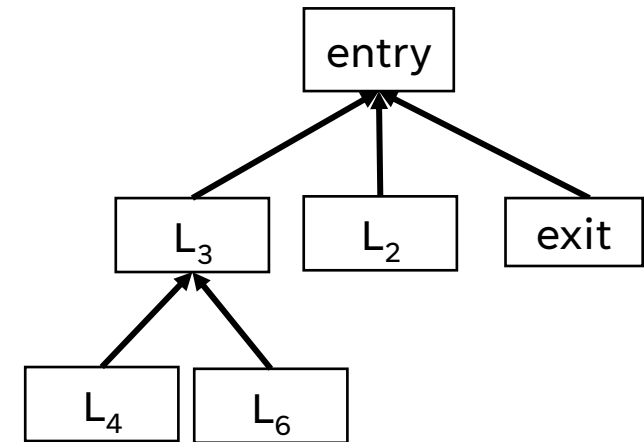
Draw the Control Dependence Graph of Basic Blocks for the following program



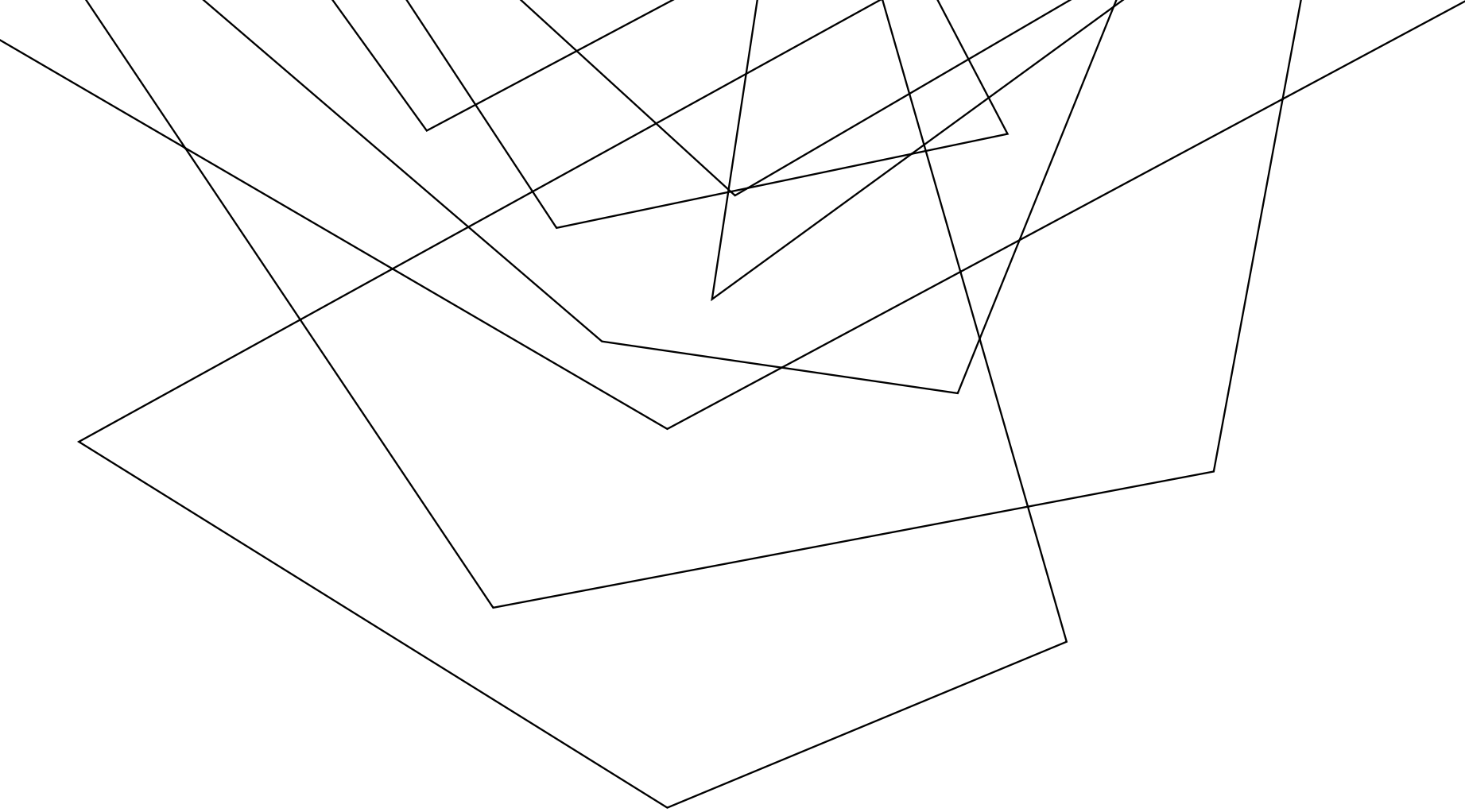
IFDOM(entry, L₂)
IFDOM(L₂, L₃)
IFDOM(L₃, exit)
IFDOM(L₄, exit)
IFDOM(L₆, exit)

Y **X**
L₃ CD L₂ ?
Path(X, Y) omitting IFDOM(X)
Path(L₂, L₃) omitting IFDOM(L₂)
Path(L₂, L₃) omitting L₃ **No!**

Y **X**
L₄ CD L₃ ?
Path(X, Y) omitting IFDOM(X)
Path(L₃, L₄) omitting IFDOM(L₃)
Path(L₃, L₄) omitting L₃ **Yes!**



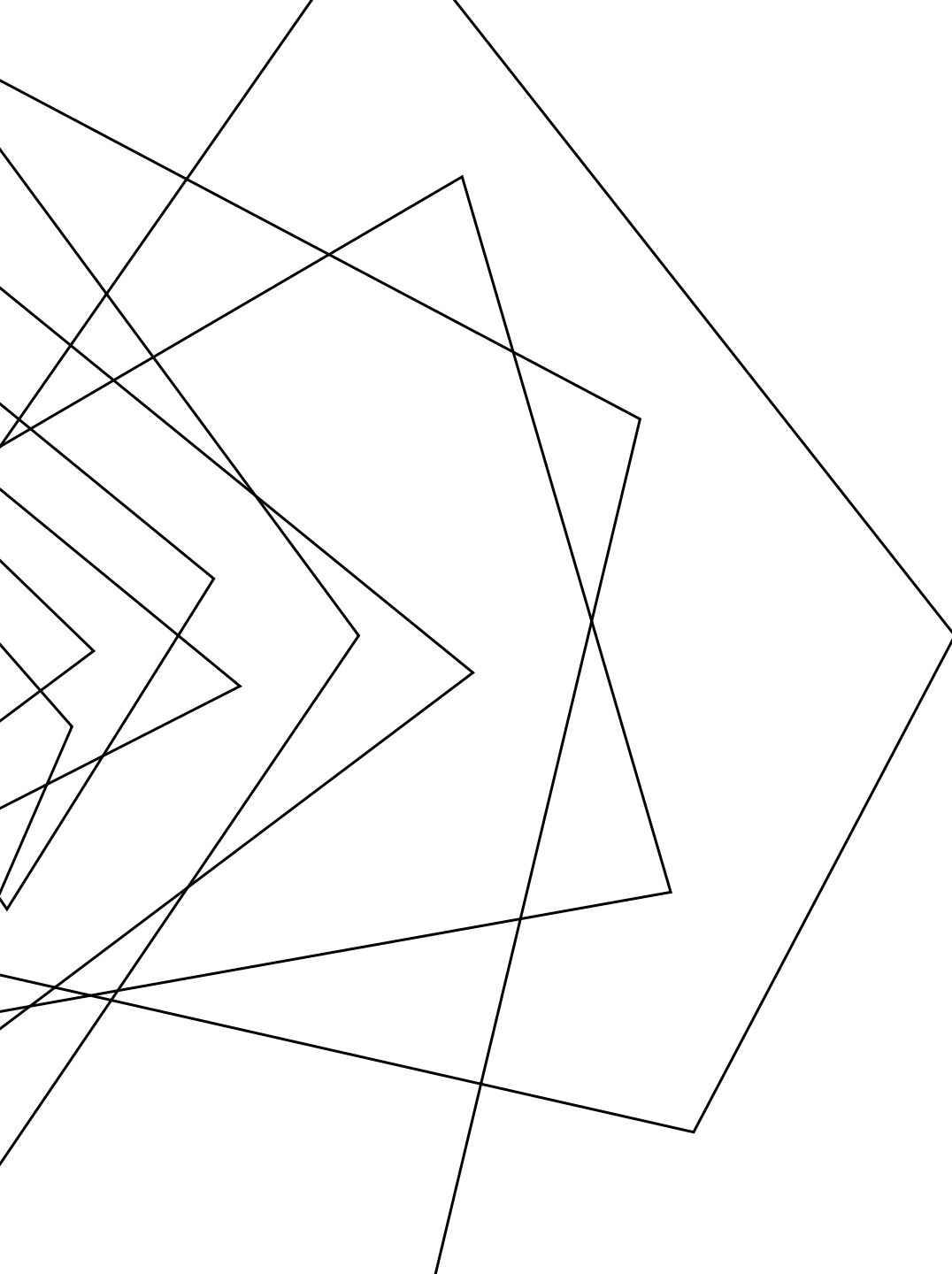
$Y \text{ CD } X \Leftrightarrow$ there is a CFG-path from X to Y omitting IFDOM(X)



PROGRAM SLICING

EECS 677: Software Security Evaluation

Drew Davidson

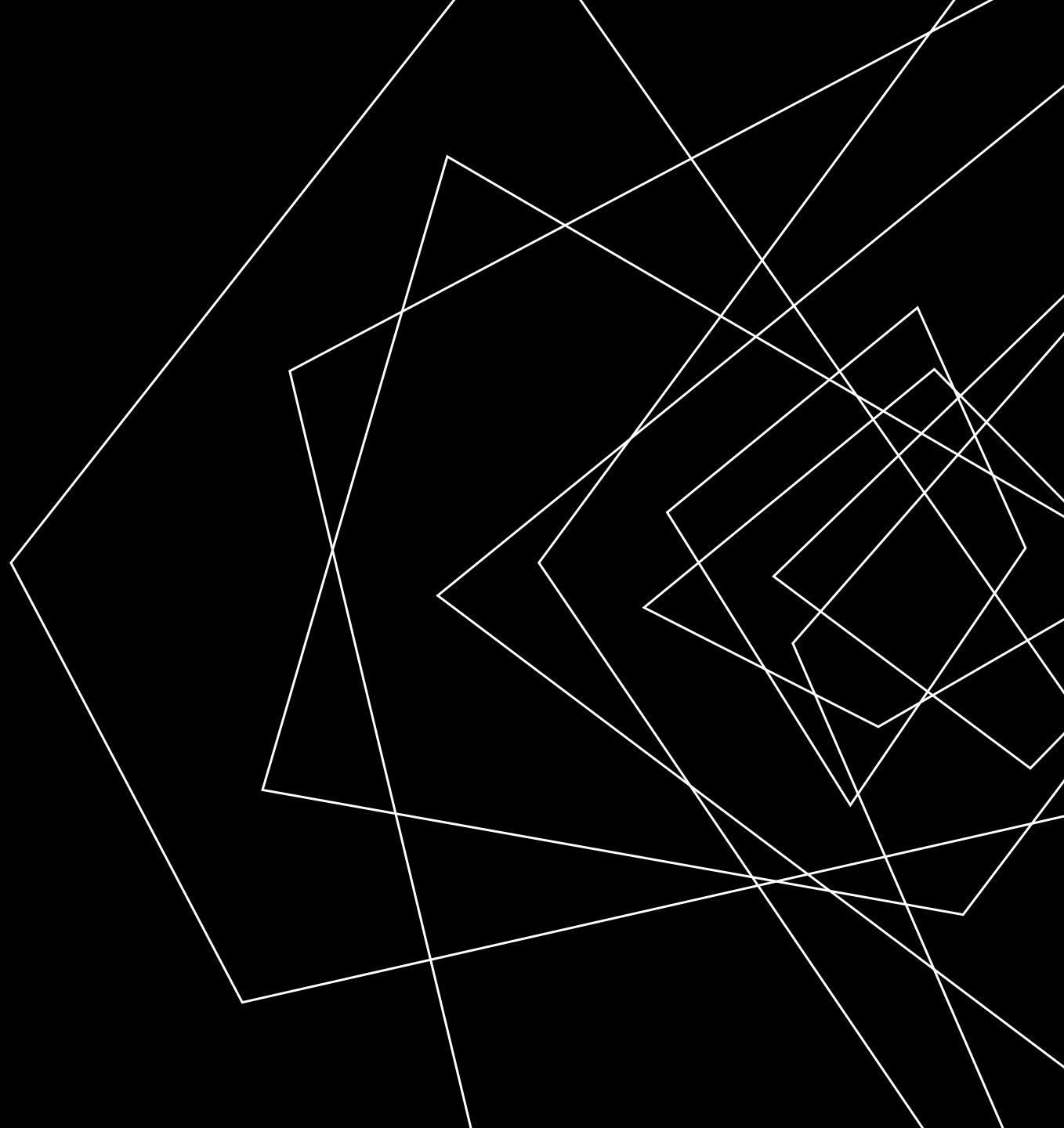


OVERVIEW

EXTENDING THE DEPENDENCE RELATION
AND SHOWING ITS USE

LECTURE OUTLINE

- Data Dependence
- PDGs
- Slicing

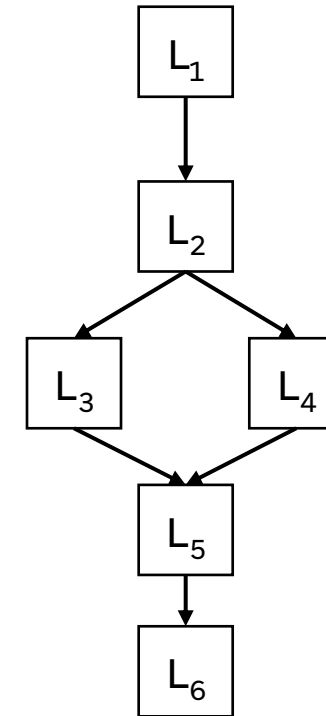


DATA DEPENDENCE

DEPENDENCE RELATIONS

Influence is more than control, it's also what values mattered to your behavior

```
1: READ i;  
2: if ( i == 1)  
3:   PRINT "hi!"  
   else  
4:   i = 1;  
5: PRINT i;  
6: end
```



Note here: a value at L_1 might have set a value at L_5 , but it's not control dependent!

THE DATA DEPENDENCE GRAPH

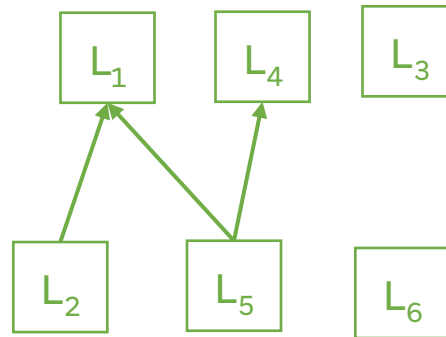
DEPENDENCE RELATIONS

Depiction of the *reaching definitions* of each statement

Procedure

```
1: READ i;  
2: if ( i == 1)  
3:   PRINT "hi!"  
   else  
4:   i = 1;  
5: PRINT i;  
6: end
```

DDG



THE DATA DEPENDENCE GRAPH

DEPENDENCE RELATIONS

Depiction of the *reaching definitions* of each statement

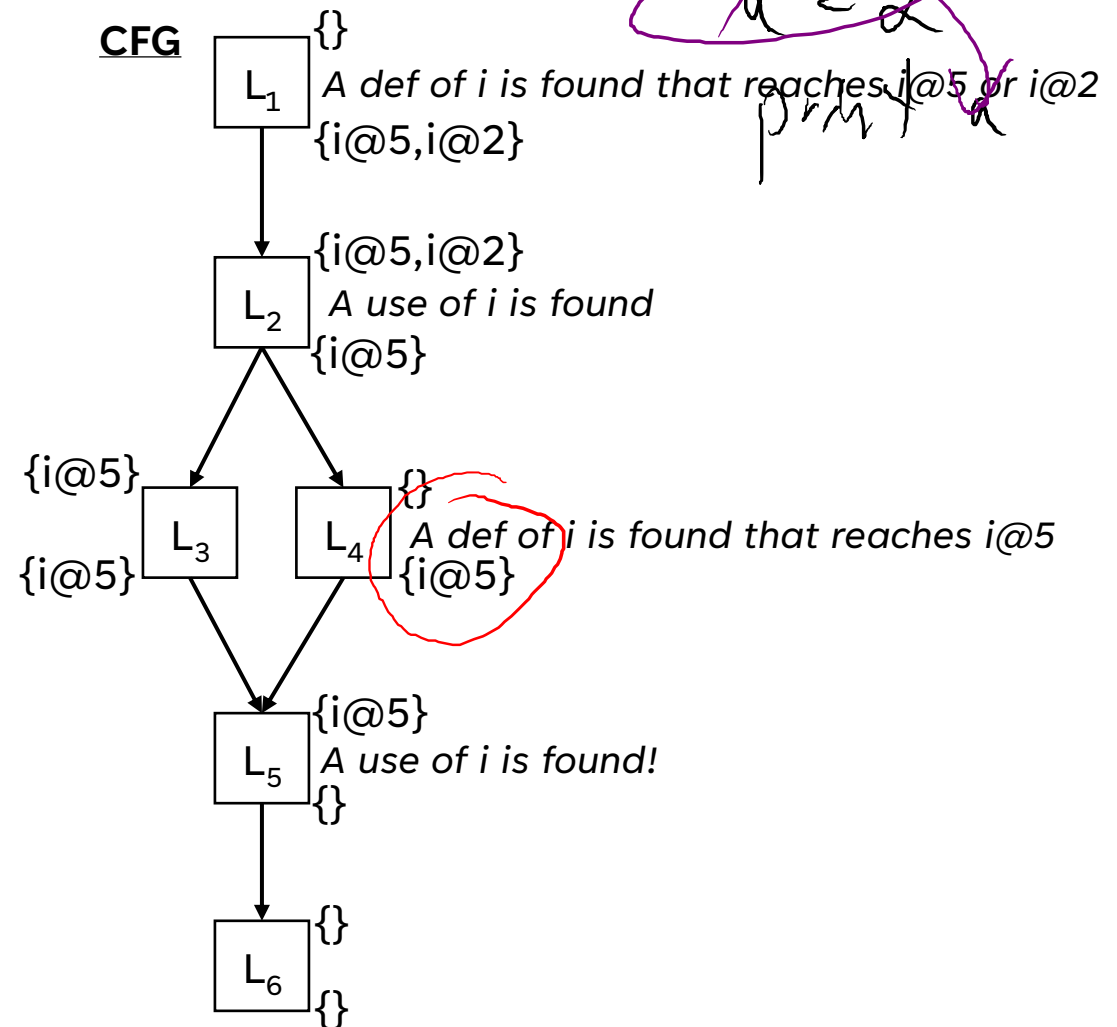
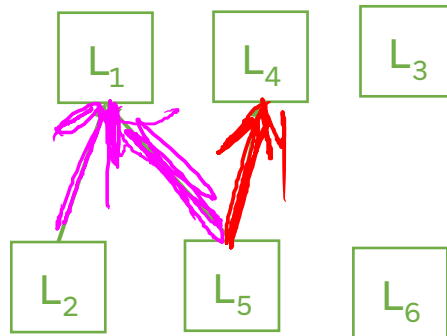
Procedure

```

1: READ i;
2: if ( i == 1)
3:   PRINT "hi!"
   else
4:   i = 1;
5: PRINT i;
6: end

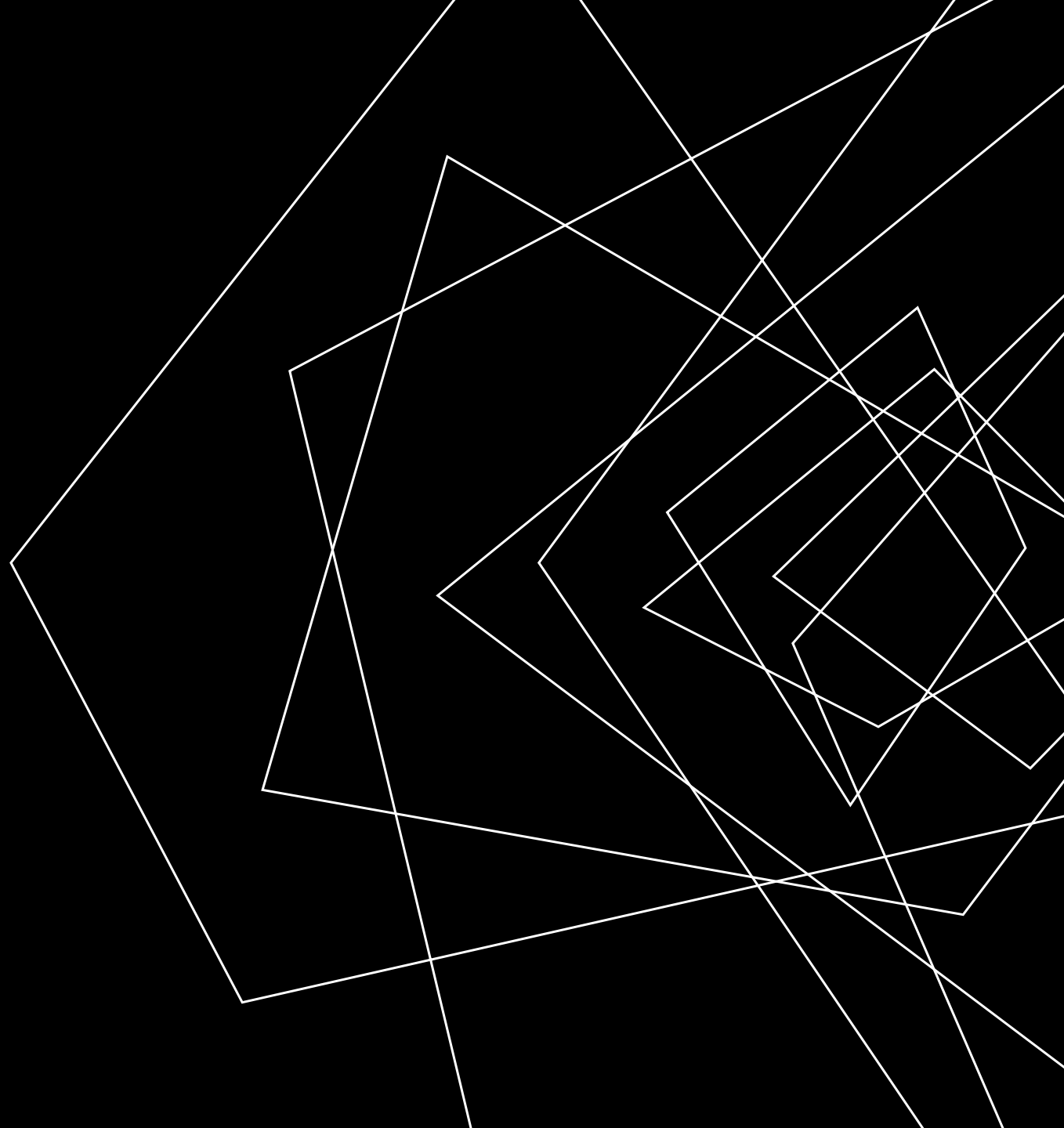
```

DDG



LECTURE OUTLINE

- Data Dependence
- PDGs
- Slicing



THE PROGRAM DEPENDENCE GRAPH

DEPENDENCE RELATIONS

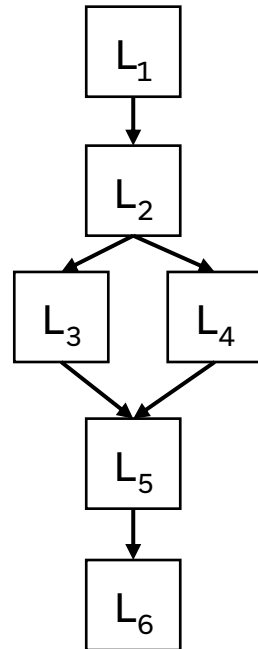
An overlay of the CDG + DDG = PDG

```

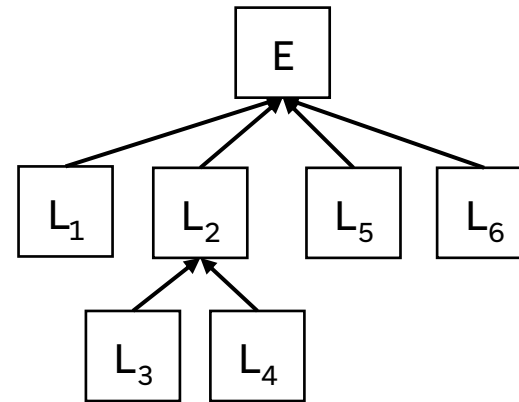
1: READ i;
2: if ( i == 1)
3:   PRINT "hi!"
   else
4:   i = 1;
5: PRINT i;
6: end

```

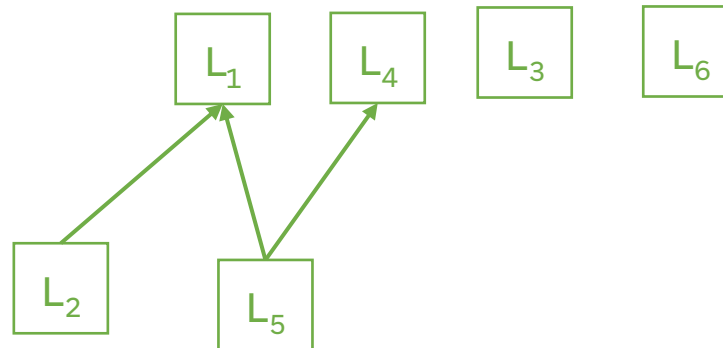
CFG



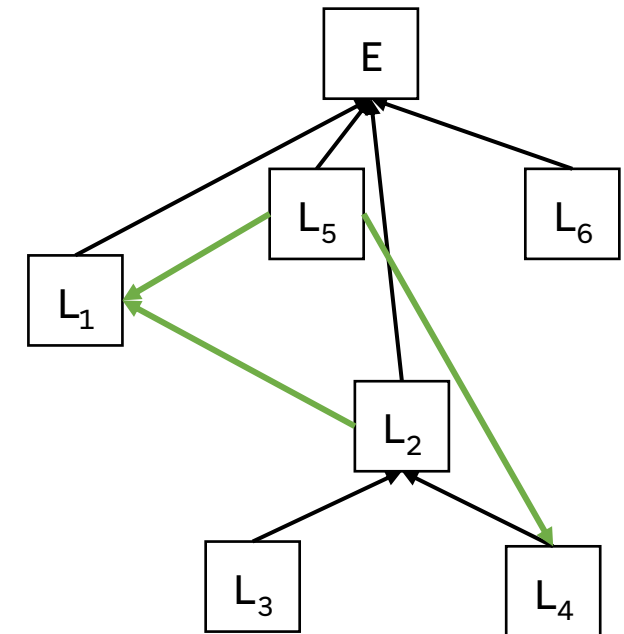
CDG



DDG

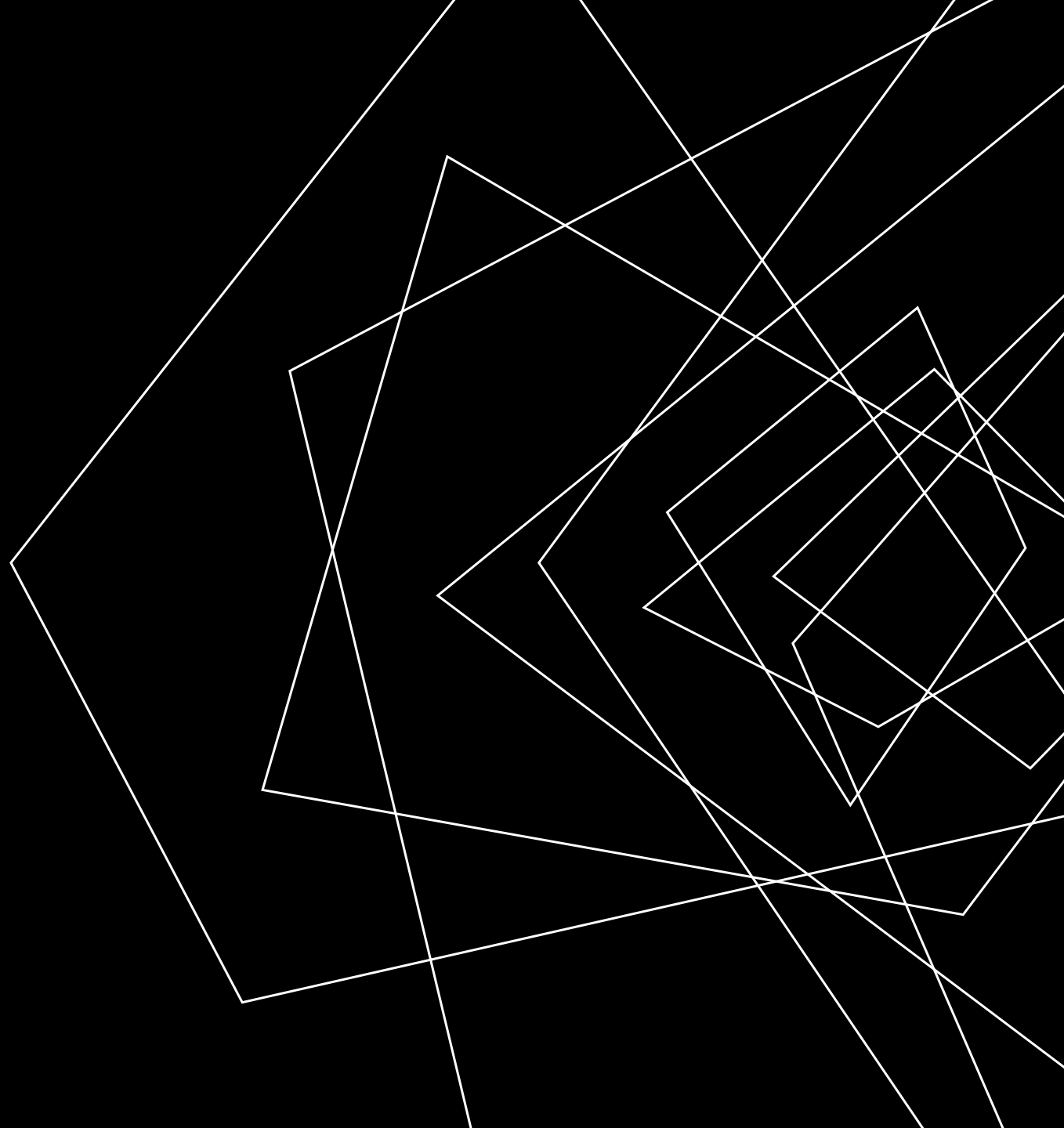


PDG



LECTURE OUTLINE

- Data Dependence
- PDGs
- Slicing



THE “SUB-PROGRAM” CONCEPT

PROGRAM SLICING

**BIG IDEA: IGNORE “IRRELEVANT”
FUNCTIONALITY FOR A PARTICULAR CASE**

Control Dependence Graph (CDG)

- Shows what program statements depend on each other

Program Dependence Graph (PDG)

- At minimum: A CDG enriched with data dependence information

THE SLICE OF THE PROGRAM

PROGRAM SLICING

FORWARD SLICE

Everything **influenced by** statement K
Forward reachability in the PDG

Forward Slice

```
x = net_read()
if (x > 2){
    x = 2;
}
array[x] = 4;
```

Program

```
x = rand()
y = rand()
x = net_read()
if (y == 1){
    printf("hello");
}
if (x > 2){
    x = 2;
}
array[x] = 4;
```

BACKWARDS SLICE

Everything that **influences** statement K
Backward reachability in the PDG

Backward Slice

```
y = rand()
if (y == 1){
    printf("hello");
}
```

SLICE EXECUTION

PROGRAM SLICING

DO WE NEED OUR SLICED SUBPROGRAM TO
BE EXECUTABLE?

If so, we may need to include additional
instructions

OUTPUT DEPENDENCE

PROGRAM SLICING

DO WE NEED OUR SLICED SUBPROGRAM TO
PERFORM IDENTICALLY TO THE ORIGINAL?

If so, we'll need additional output dependence
edges

SLICING SUMMARY

PROGRAM SLICING

STATIC SLICING HAS SOME PROMISING APPLICATIONS

It's not a one-size-fits-all scalability panacea
Any (sound) slicing is likely a benefit!

SOME APPLICATIONS BEYOND ANALYSIS

Automatic parallelization

Software metrics (how big of a change is this refactor?)



ANALYSIS TOOLS

SWITCHING GEARS

WE'VE COVERED SEVERAL POPULAR
ANALYSIS TECHNIQUES FOR IMPERATIVE
PROGRAMMING

Let's talk a bit about their tooling



LLVM: STATIC SLICING

ANALYSIS TOOLS

<https://github.com/mchalupa/dg>

README.md

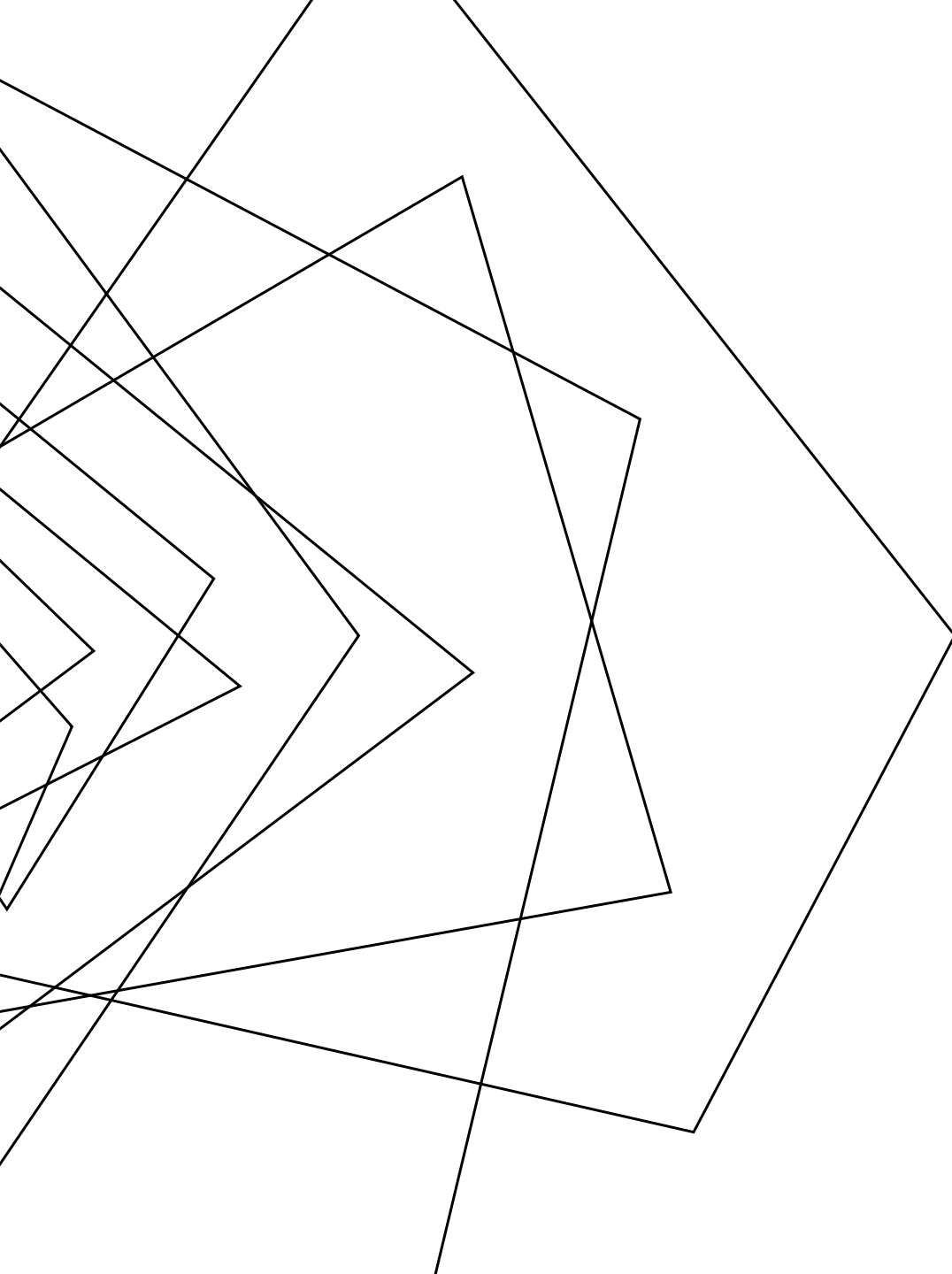
DG

 Linux CI passing  macOS CI passing

DG is a library containing various bits for program analysis. However, the main motivation of this library is program slicing. The library contains implementation of a pointer analysis, data dependence analysis, control dependence analysis, and an analysis of relations between values in LLVM bitcode. All of the analyses target LLVM bitcode, but most of them are written in a generic way, so they are not dependent on LLVM in particular.

Further, DG contains an implementation of dependence graphs and a [static program slicer](#) for LLVM bitcode. Some documentation can be found in the [doc/](#) directory.

- [Downloading DG](#)
- [Compiling DG](#)
- [Using llvm-slicer](#)
- [Other tools](#)



NEXT TIME

DEALING WITH “REAL” PROGRAMS

- POINTERS

- (AFTER THAT) CLASSES

- (AFTER THAT) INTERPROCEDURAL ANALYSIS