EXERCISE 23

POINTS-TO ANALYSIS REVIEW

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

Draw the points-to graph of the following snippet:

Assignment	Constraint
a = &b	a ⊇ { b }
a = b	a ⊇ b
a = *b	a ⊇ *b
*a = b	*a ⊇ b

EXERCISE 23 SOLUTION

POINTS-TO ANALYSIS REVIEW

Assignment	Constraint
a = &b	a ⊇ { b }
a = b	a ⊇ b
a = *b	a ⊇ *b
*a = b	*a ⊇ b

EXERCISE 23

POINTS-TO ANALYSIS REVIEW

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

Draw the points-to graph of the following snippet:

Assignment	Constraint
a = &b	a ⊇ { b }
a = b	a ⊇ b
a = *b	a ⊇ *b
*a = b	*a ⊇ b

EXERCISE 23 SOLUTION

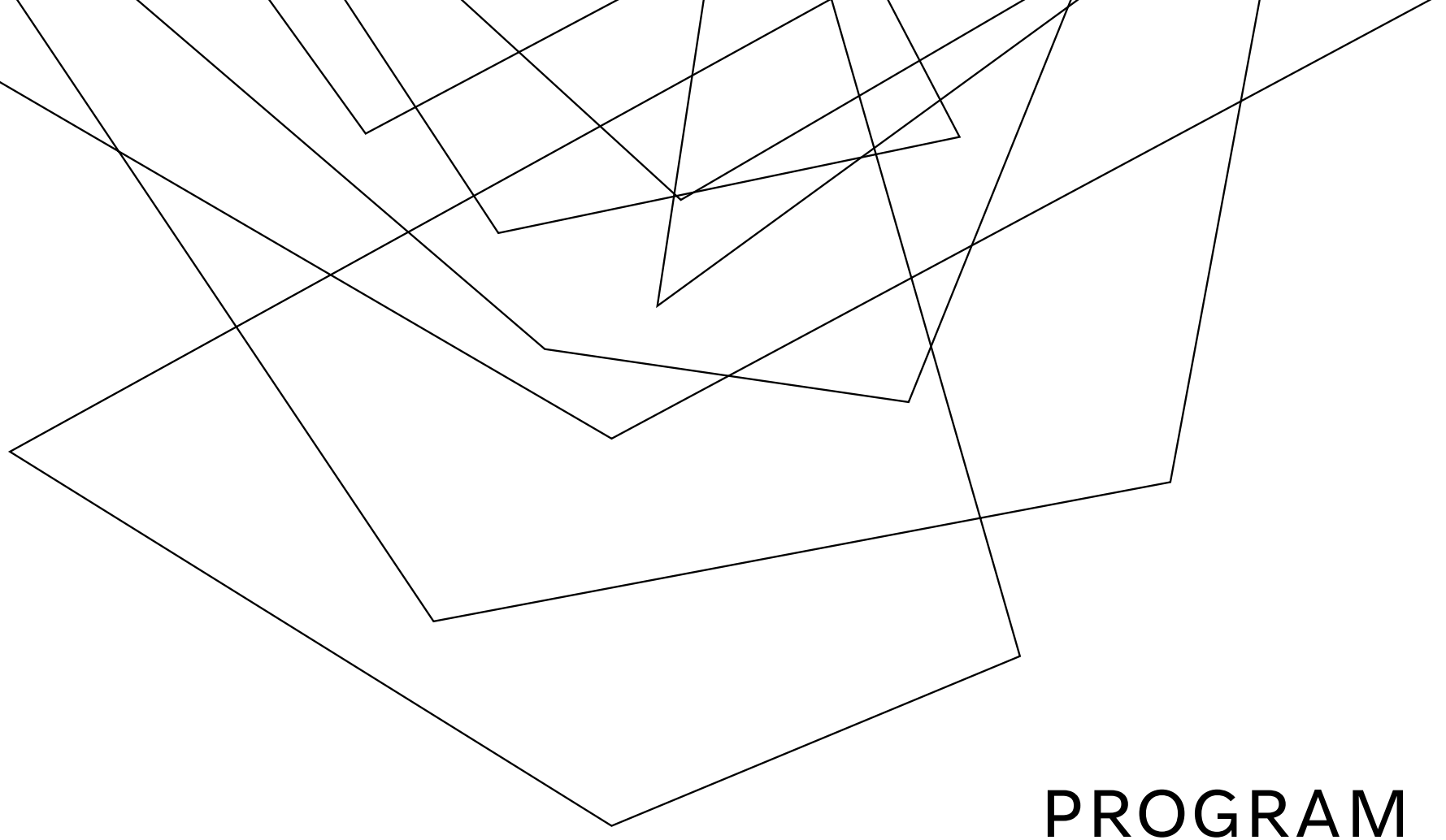
POINTS-TO ANALYSIS REVIEW

Assignment	Constraint
a = &b	a ⊇ { b }
a = b	a ⊇ b
a = *b	a ⊇ *b
*a = b	*a ⊇ b

ADMINISTRIVIA AND ANNOUNCEMENTS

Quiz 2 on Monday

Review session: Friday at 6:30 PM, Location TBA



PROGRAM INSTRUMENTATION

EECS 677: Software Security Evaluation

Drew Davidson

ANDERSEN'S ALGORITHM

REVIEW: LAST LECTURE

REACHABILITY FORMULATION

Step 1: Extract pointer-related operations
Step 2: Saturate points-to graph
Step 3: Compute node reachability

Assignment	Constraint	Meaning
a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
a = b	a ⊇ b	pts(a) ⊇ pts(b)
a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

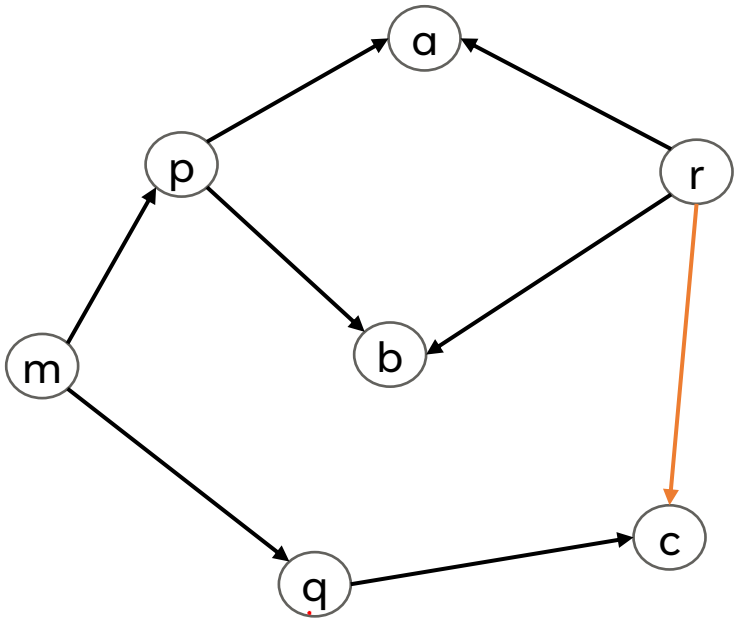
ANDERSEN'S ALGORITHM: REACHABILITY

REVIEW: LAST LECTURE

REACHABILITY FORMULATION

Step 1: List pointer-related operations
Step 2: Saturate points-to graph
Step 3: Compute node reachability

Program	Constraints
p = &a	p ⊇ {a}
p = &b	p ⊇ {b}
m =&p;	m ⊇ {p}
r = *m;	r ⊇ *m
q = &c;	q ⊇ {c}
m = &q	m ⊇ {q}



Assignment	Constraint	Meaning
a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
a = b	a ⊇ b	pts(a) ⊇ pts(b)
a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

Initial	Final
pts(a) = { }	pts(a) = { }
pts(b) = { }	pts(b) = { }
pts(m) = { }	pts(m) = { p, q }
pts(p) = { }	pts(p) = { a, b }
pts(q) = { }	pts(q) = { c }
pts(r) = { }	pts(r) = { a, b, c }

POINTS TO AND TYPE SAFTEY

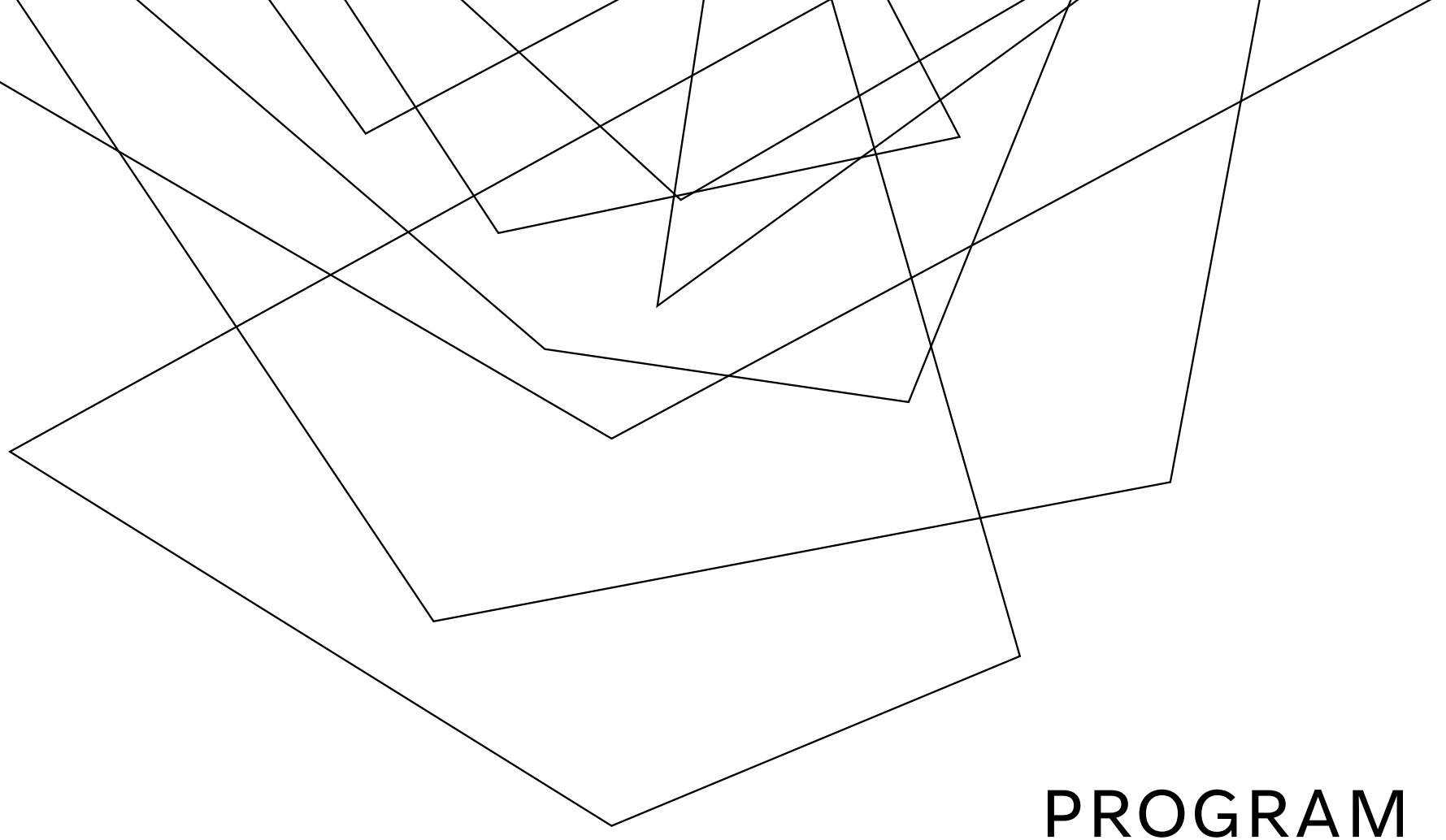
REVIEW: LAST LECTURE

A "FEATURE" OF THE ANALYSIS

Our points-to relationships are somewhat contrived

Would a program ever actually have both of these statements?

```
*a = b;
*a = *b;
```



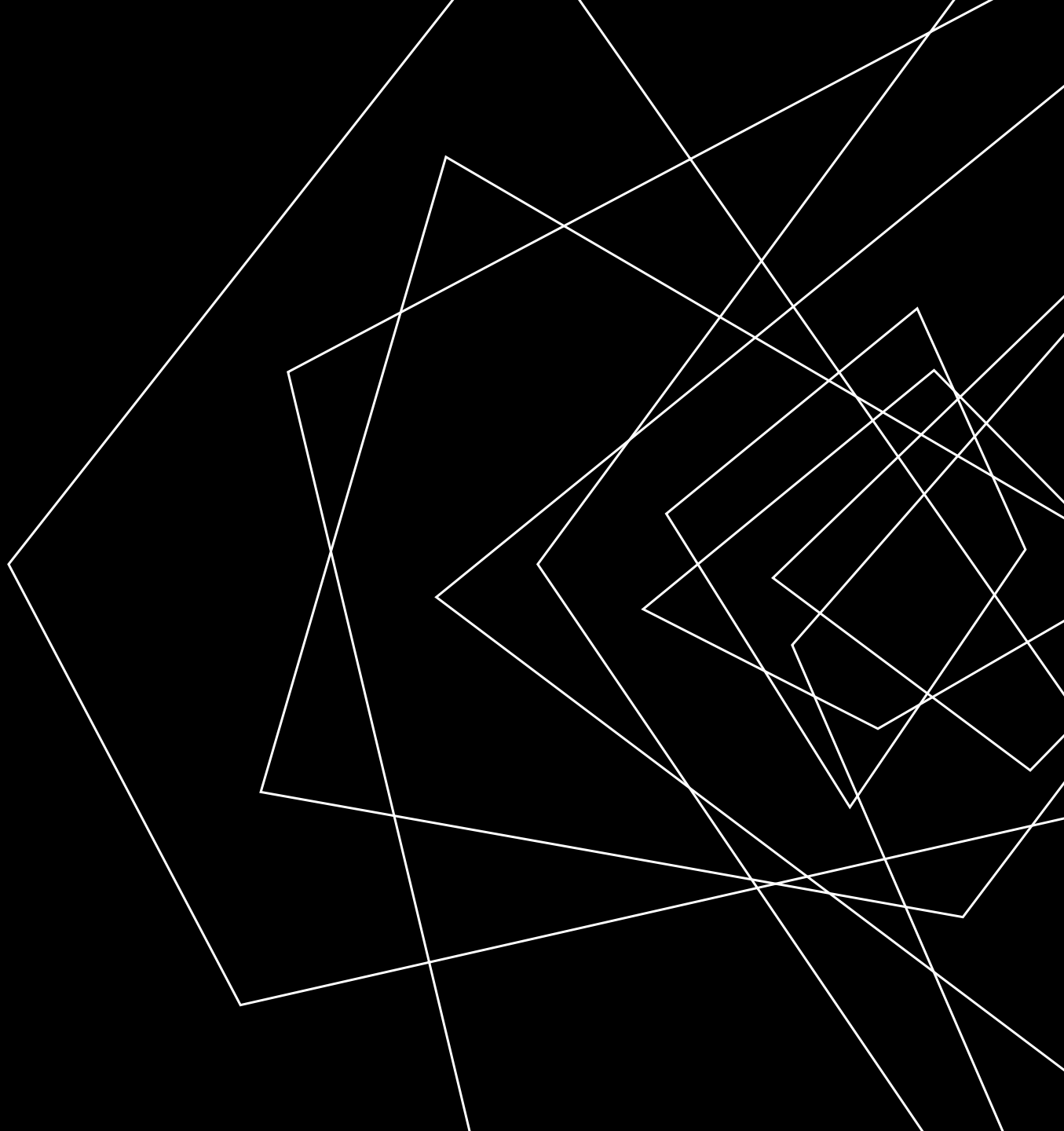
PROGRAM INSTRUMENTATION

EECS 677: Software Security Evaluation

Drew Davidson

LECTURE OUTLINE

- Steensgard's Analysis
- Static Analysis Underview
- Program Instrumentation



OVERHEAD
ANDERSEN'S ANALYSIS

Worst Case: Cubic Time

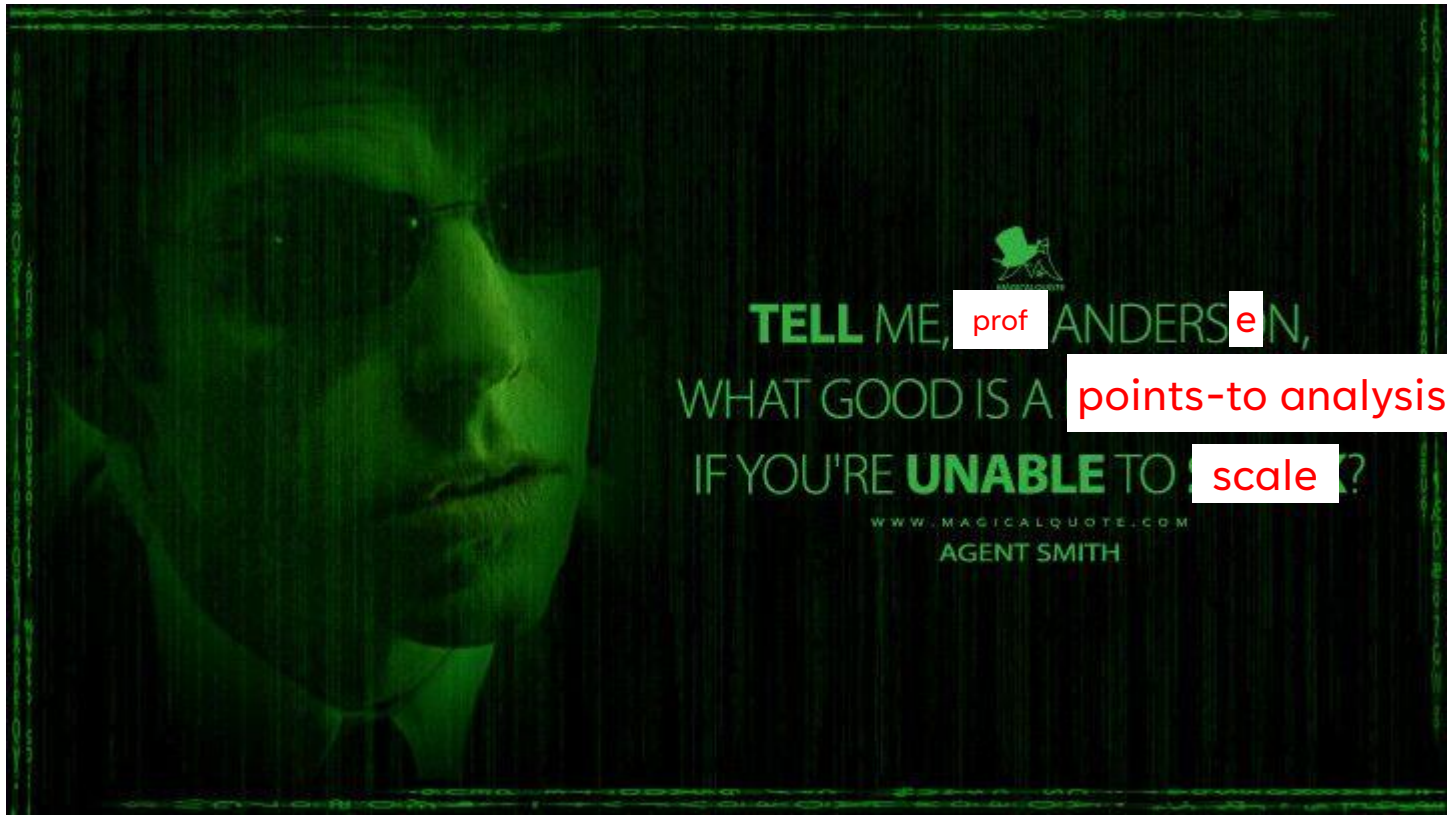
That's not great!

Most of the time is spent in re-analyzing constraints to get to a fixpoint

Optimization: Cycle elimination

Detect and collapse SCCs in the

points-to relation



A MORE-EFFICIENT POINTS-TO
STEENSGARD'S ANALYSIS

RETURN AGAIN TO OUR ANCIENT WISDOM

Simpler abstractions reach fixpoints faster

A MORE-EFFICIENT POINTS-TO

STEENSGARD'S ANALYSIS

RETURN AGAIN TO OUR ANCIENT WISDOM

Simpler abstractions reach fixpoints faster

STEENGARD'S ANALYSIS

Limit the points-to graph nodes to have outdegree <= 1

You can only point to 1 node

If you need to point to > 1 node, merge the "pointees"

Simplifies many points-to constraints from subsets to equalities

Achieves near-linear performance

STEENGARD'S ALGORITHM

AN EFFICIENT OVER-APPROXIMATION

In Practice

Step 1

List pointer-related operations

Step 2

Induce set of ~~subset~~ **equality** constraints

Step 3

Solve system of constraints

Reachability formulation

Step 1

List pointer-related operations

Step 2

Saturate points-to graph **1-out**

Step 3

Compute node reachability

Andersen's

Assignment	Constraint	Meaning
a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
a = b	a ⊇ b	pts(a) ⊇ pts(b)
a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

Steengaard's

Assignment	Constraint	Meaning
a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
a = b	a = b	pts(a) = pts(b)
a = *b	a = *b	∀v∈pts(b). pts(a) = pts(v)
*a = b	*a = b	∀v∈pts(a). pts(v) = pts(b)

EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	a = b	pts(a) = pts(b)
Complex	a = *b	a = *b	$\forall v \in pts(b).\ pts(a) = pts(v)$
Complex	*a = b	*a = b	$\forall v \in pts(a).\ pts(v) = pts(b)$

EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

p = &a

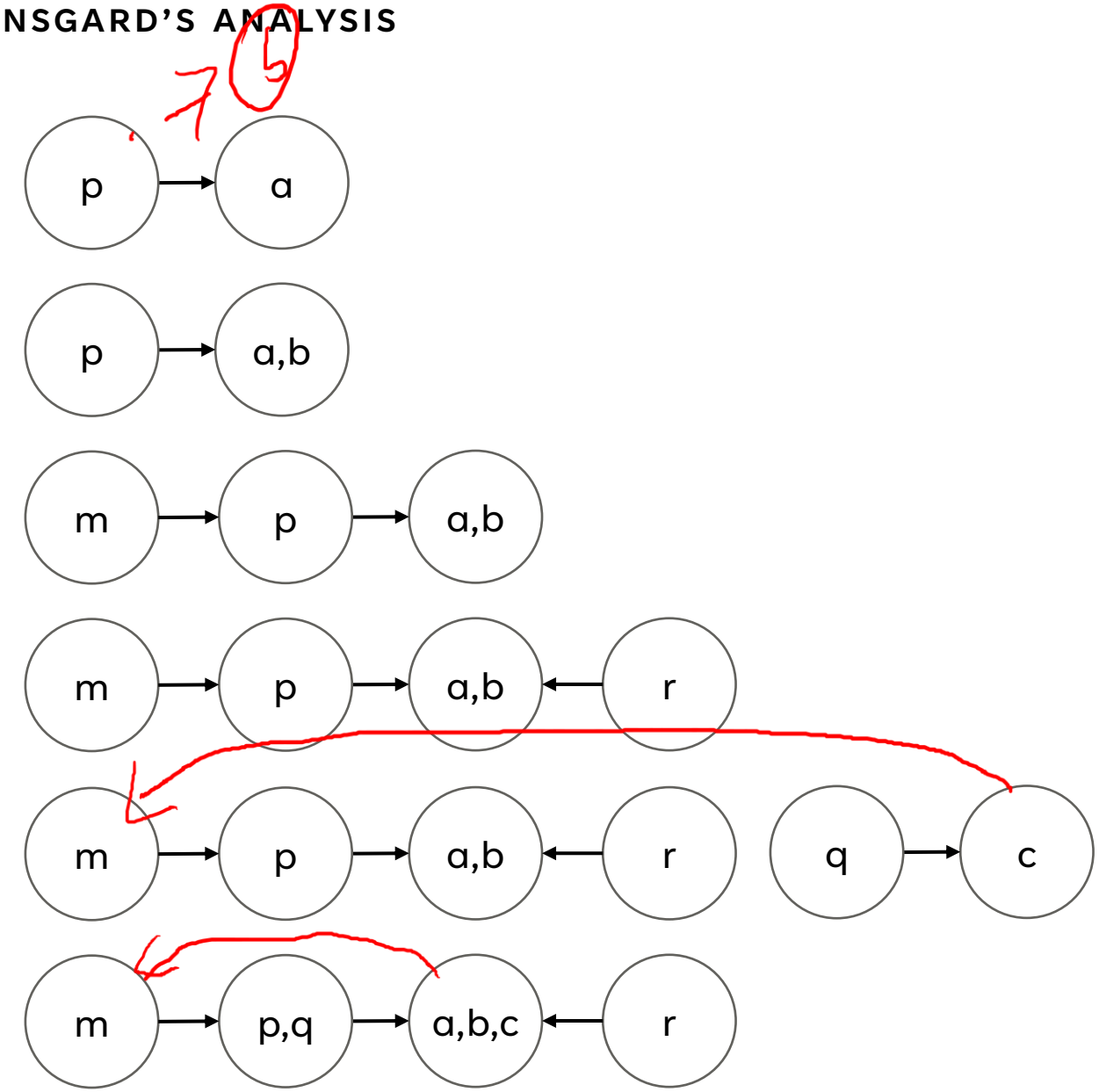
p = &b

m = &p

r = *m

q = &c

m = &q



EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

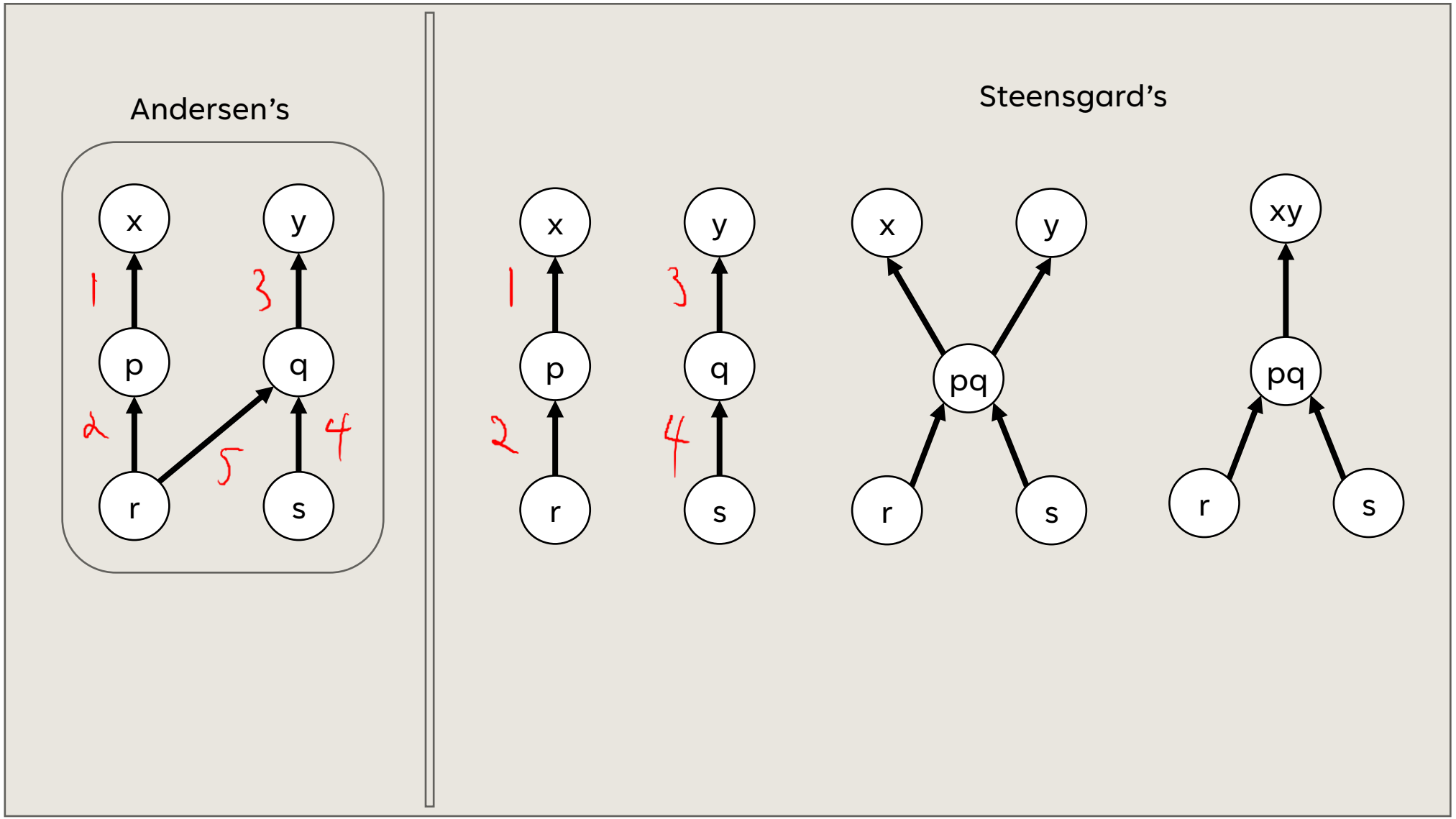
p = &x

r = &p

q = &y

s = &q

r = s



THAT'S POINTS-TO!

STEENSGARD'S ANALYSIS

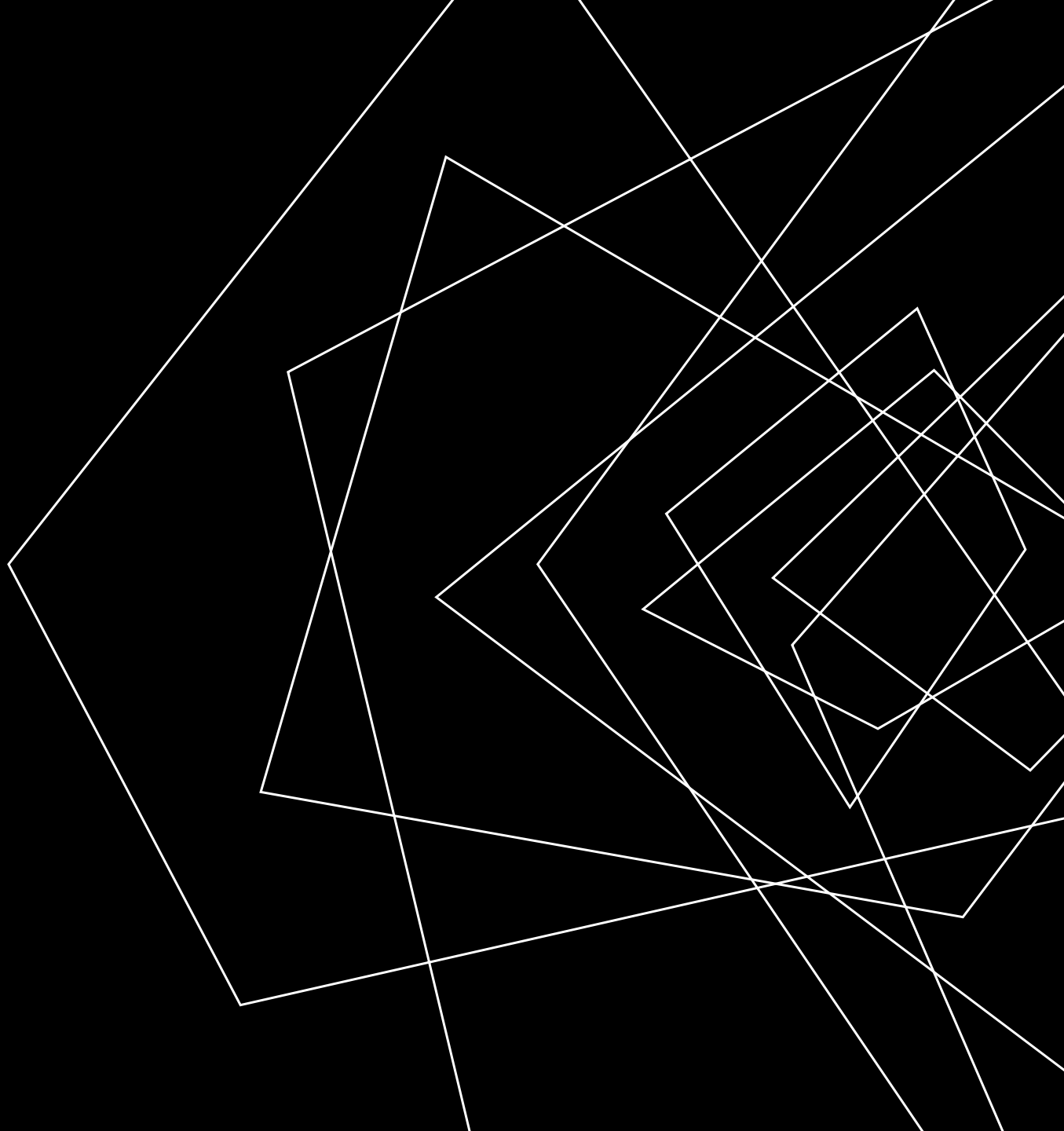
AN ADDITIONAL OVERLAY ON DATAFLOW

Dataflow facts also flow to aliases

When dereferencing a pointer, consider only pointed-to objects

LECTURE OUTLINE

- Steensgard's Analysis
- Static Analysis Underview
- Program Instrumentation



STATIC ANALYSIS READY TO GO!

STATIC ANALYSIS UNDERVIEW

DATAFLOW ANALYSIS CAN BE ADOPTED FOR CHECKING A VARIETY OF SECURITY / CORRECTNESS PROPERTIES

Forms the basis of a lot of static analysis!

Applicable for a variety of analysis goals

- Security leak detection
- Vulnerable program state detection
- Program understanding

STATIC ANALYSIS: BENEFITS

STATIC ANALYSIS UNDERVIEW

"THE ANALYST'S SIEVE"

Focus your attention on potential issues

NON-INTERACTIVE!

Can run in the background

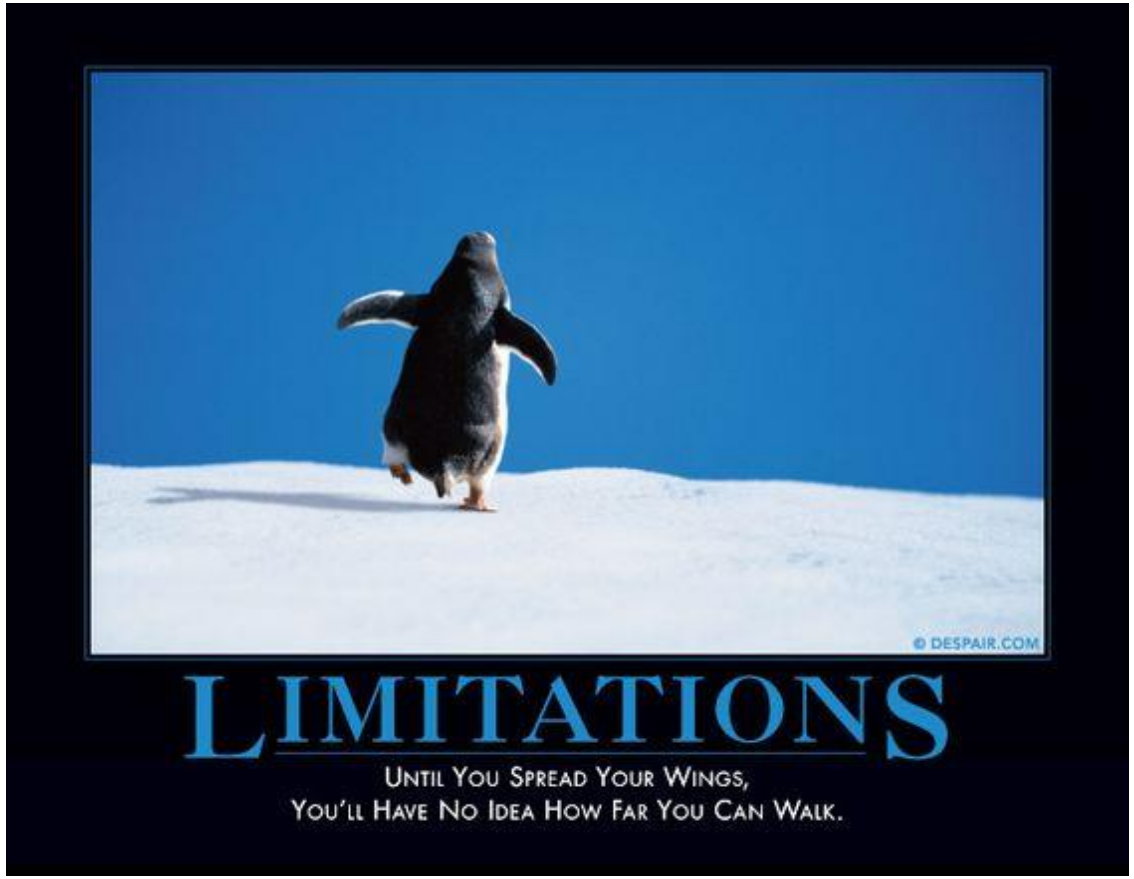
Abstraction obviates need for input

LIMITS OF STATIC ANALYSIS

PROGRAM INSTRUMENTATION: BASIC IDEA

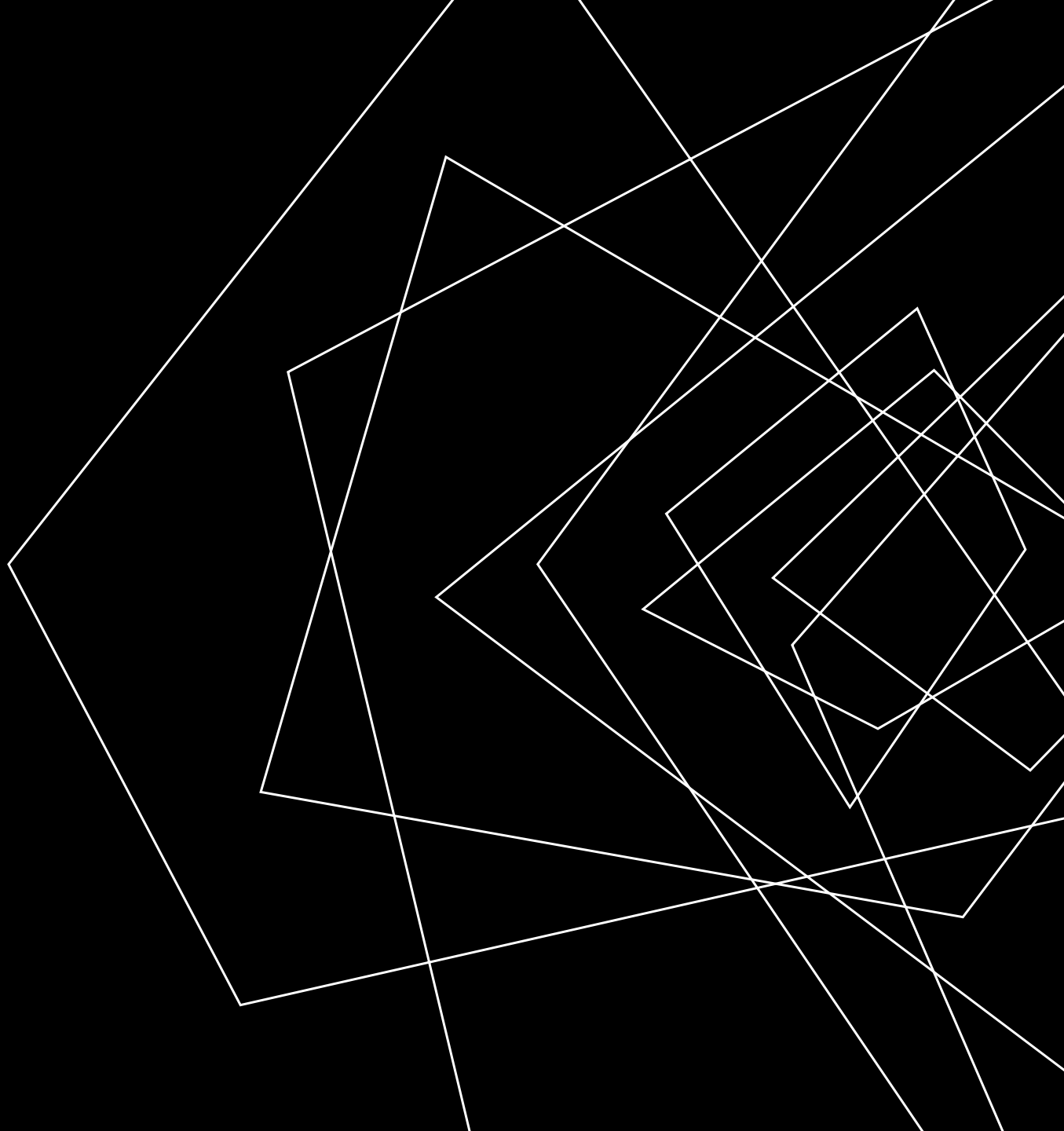
Practical Issues

- Unsoundness of bug finding / incompleteness of program verification
- Scalability
- Significant engineering effort
- Findings may not be super actionable



LECTURE OUTLINE

- Steensgard's Analysis
- Static Analysis Underview
- Program Instrumentation



REVISING DYNAMIC ANALYSIS

PROGRAM INSTRUMENTATION: BASIC IDEA

GIVING UP ON COMPLETE BUG-FINDING

- Finding bugs (even “low-hanging fruit”) is useful!

BENEFITS

- Scalability
- Sound bug finding

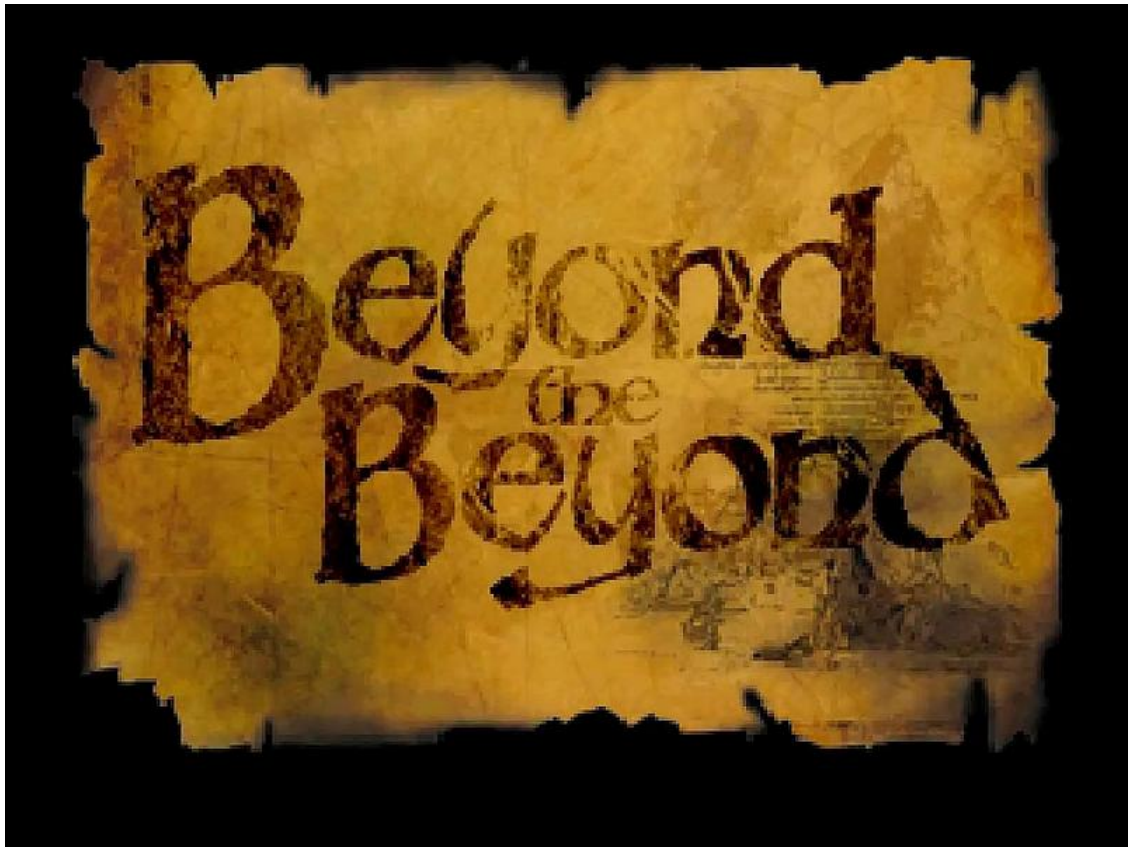
BEYOND TESTING

PROGRAM INSTRUMENTATION: BASIC IDEA

LIMITATIONS OF "PLAIN" TESTING

- Property may not be immediately observable from output alone

- The circumstances under which the issue occurs may not be obvious



PROGRAM INSTRUMENTATION

PROGRAM INSTRUMENTATION: BASIC IDEA

WRITE CODE INTO THE EXECUTABLE TO GATHER INFORMATION

Addresses both of the previous issues – can report upon program state and even program path

EXAMPLE: LLVM INSTRUMENTATION

PROGRAM INSTRUMENTATION: BASIC IDEA

WRITE CODE INTO THE EXECUTABLE TO GATHER INFORMATION

Addresses both of the previous issues – can report upon program state and even program path

INSERTING PROGRAM PROBES

PROGRAM INSTRUMENTATION: BASIC IDEA

Insert checks / Reports into the analysis target

Addresses both of the previous issues – can report upon program state and even program path

A new concern – The efficiency of the (Instrumented) program

Potential slowdown on each program path

Old Concern – The efficiency of Placement analysis

Somewhat limited by the information the probes can report

EXAMPLE: CODE COVERAGE

PROGRAM INSTRUMENTATION: BASIC IDEA

```
b << q to; (c/4);
a / b;
```

EXAMPLE: CODE COVERAGE

PROGRAM INSTRUMENTATION: BASIC IDEA

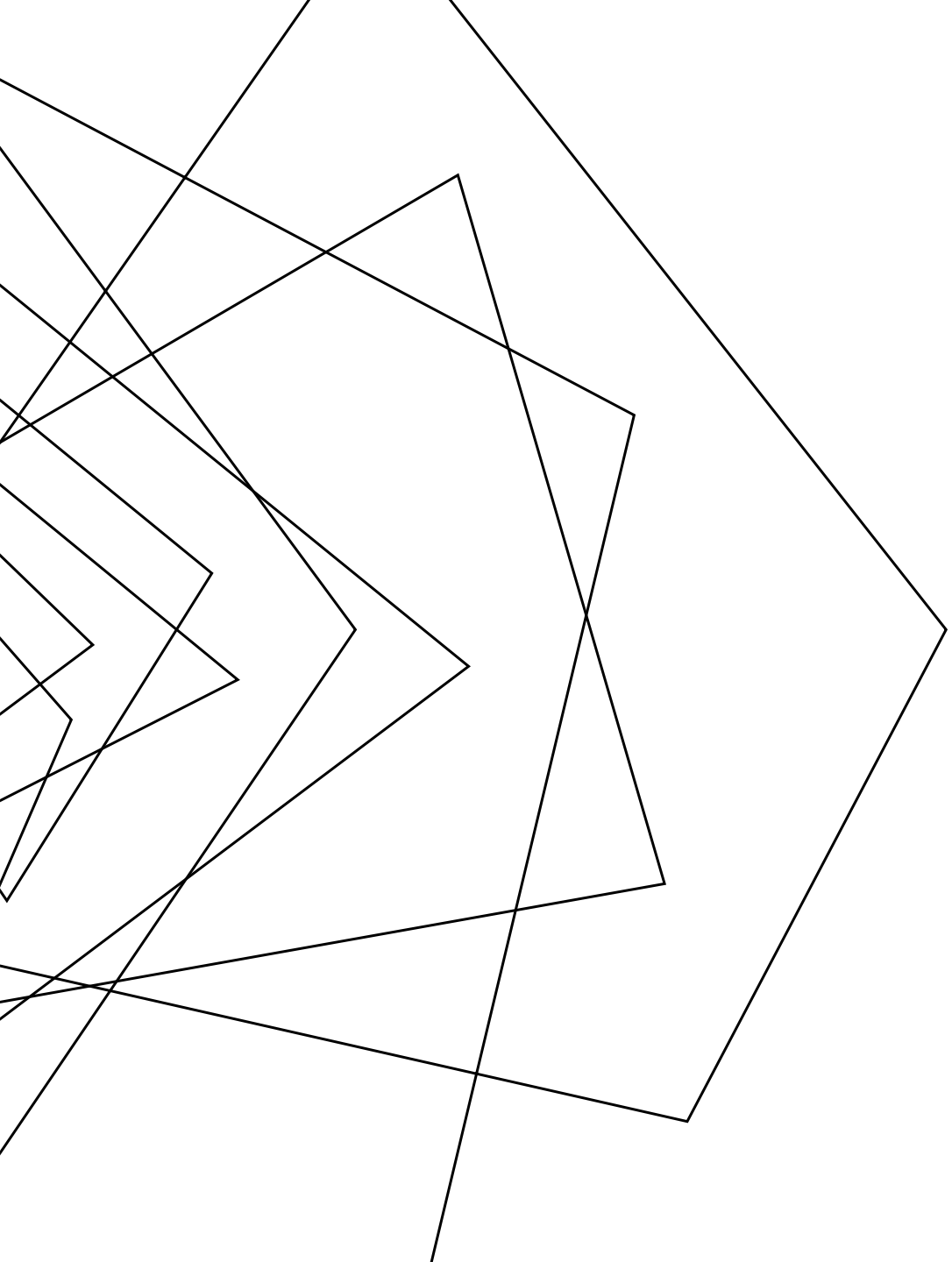
COUNTING HOW MANY TIMES CERTAIN BEHAVIORS OF THE PROGRAM ARE EXERCISED

Why is this useful? (Placing sanitizers)

THIS ACTUALLY TURNS OUT TO BE A LITTLE BIT TRICKY!

Actually turns out to be a little bit tricky!

We'll describe some of the issues / solution as per Ball and Larus, '96



WRAP-UP

WE'VE BEGUN TO CONSIDER A WAY TO MOVE BEYOND STATIC ANALYSIS WHILE USING OUR EXISTING TOOLS: PROGRAM INSTRUMENTATION