

EXERCISE

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:

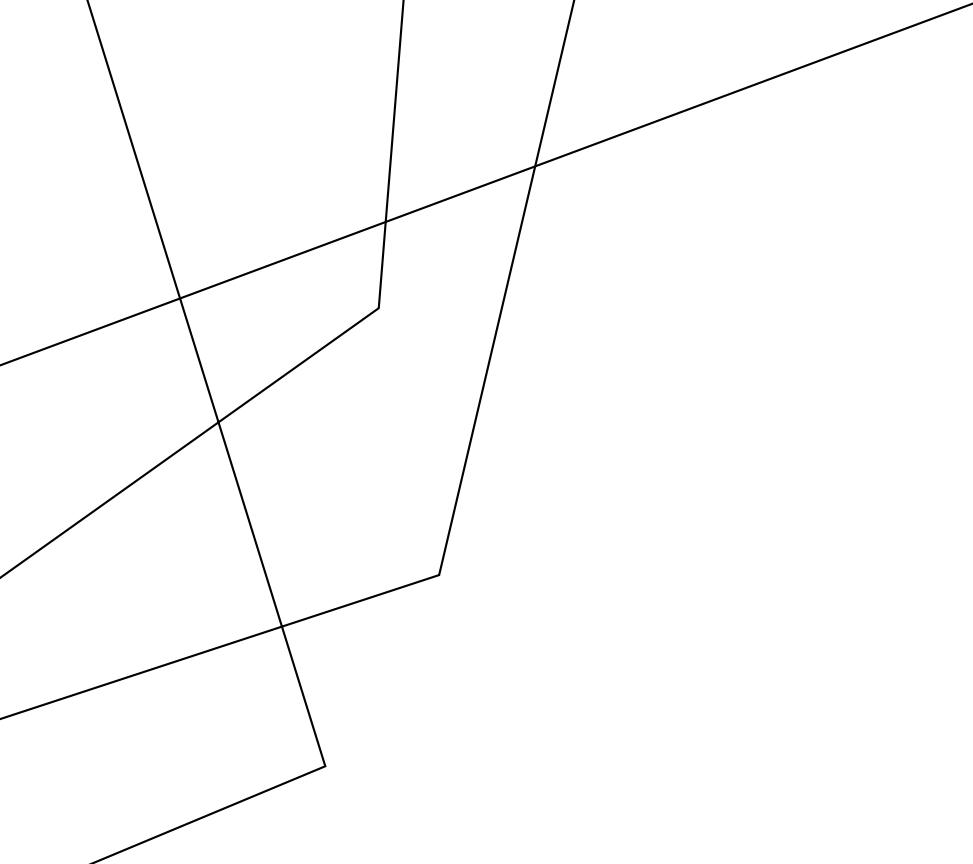
```
1: int main() {  
2:     p = &x;  
3:     r = &p;  
4:     if (true) {  
5:         q = &y;  
6:     }  
7:     s = &q;  
8:     r = s;  
9: }
```

Assignment	Constraint
$a = \&b$	$a \supseteq \{ b \}$
$a = b$	$a \supseteq b$
$a = *b$	$a \supseteq *b$
$*a = b$	$*a \supseteq b$

EXERCISE SOLUTION

POINTS-TO ANALYSIS REVIEW

Assignment	Constraint
$a = \&b$	$a \supseteq \{ b \}$
$a = b$	$a \supseteq b$
$a = *b$	$a \supseteq *b$
$*a = b$	$*a \supseteq b$



ADMINISTRIVIA AND ANNOUNCEMENTS

Quiz 2

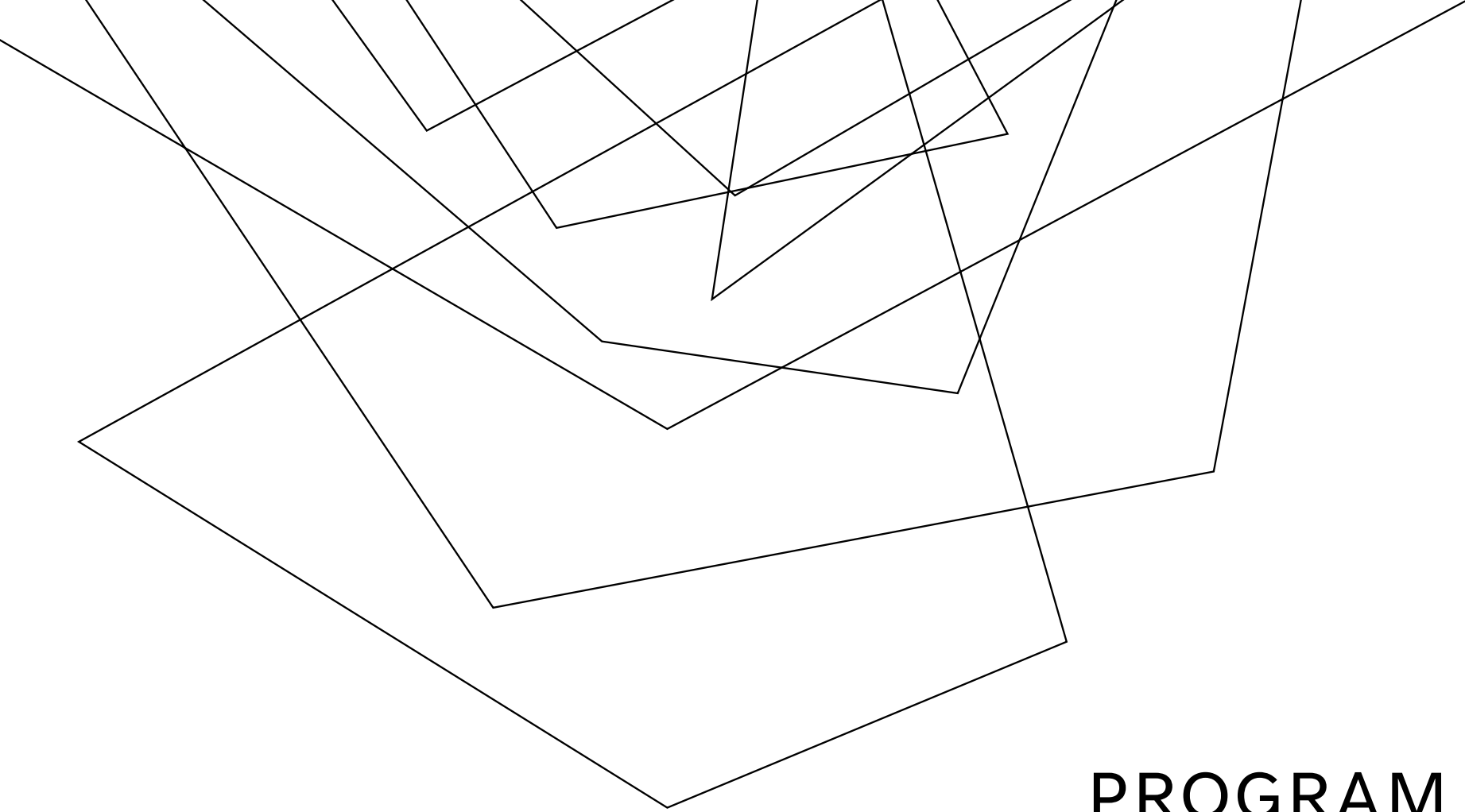
Review session: Thursday at 7:00 PM, Location LEEP2 2425

Exam: in-class on Friday ...or is it?!?!?

Bonus Office Hours Tomorrow

Zoom: <https://drew.davidson.cool/zoom>

10:00 – 1:00

Abstract geometric lines in the top-left corner of the slide, consisting of several thin black lines forming overlapping, irregular polygons and triangles.

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
$a = \&b$	$a \supseteq \{b\}$	$\text{loc}(b) \in \text{pts}(a)$
$a = b$	$a \supseteq b$	$\text{pts}(a) \supseteq \text{pts}(b)$
$a = *b$	$a \supseteq *b$	$\forall v \in \text{pts}(b). \text{pts}(a) \supseteq \text{pts}(v)$
$*a = b$	$*a \supseteq b$	$\forall v \in \text{pts}(a). \text{pts}(v) \supseteq \text{pts}(b)$

ANDERSEN'S ALGORITHM: REACHABILITY

REVIEW: LAST LECTURE

REACHABILITY FORMULATION

Step 1: Extract pointer-related operations

Step 2: Saturate points-to graph

Step 3: Compute node reachability

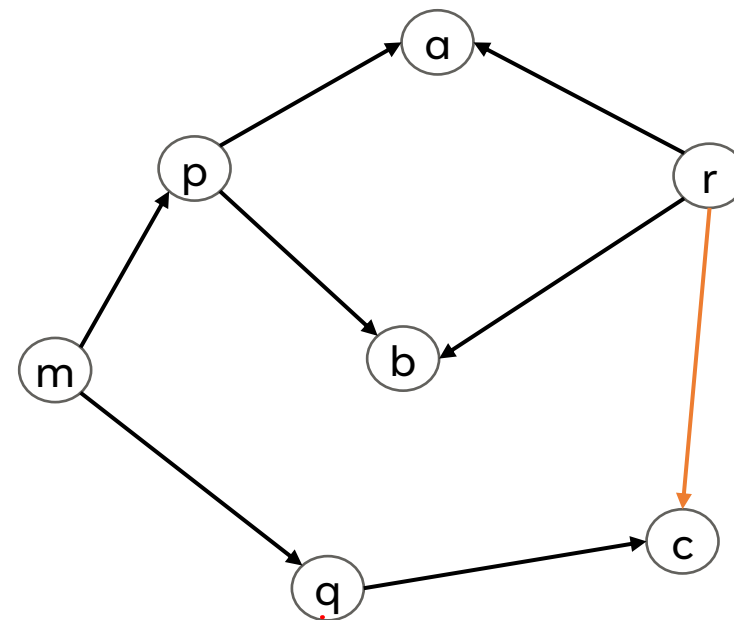
Assignment	Constraint	Meaning
$a = \&b$	$a \supseteq \{b\}$	$\text{loc}(b) \in \text{pts}(a)$
$a = b$	$a \supseteq b$	$\text{pts}(a) \supseteq \text{pts}(b)$
$a = *b$	$a \supseteq *b$	$\forall v \in \text{pts}(b). \text{pts}(a) \supseteq \text{pts}(v)$
$*a = b$	$*a \supseteq b$	$\forall v \in \text{pts}(a).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

Program

$p = \&a$
 $p = \&b$
 $m = \&p;$
 $r = *m;$
 $q = \&c;$
 $m = \&q$

Constraints

$p \supseteq \{a\}$
 $p \supseteq \{b\}$
 $m \supseteq \{p\}$
 $r \supseteq *m$
 $q \supseteq \{c\}$
 $m \supseteq \{q\}$



Initial

$\text{pts}(a) = \{ \}$
 $\text{pts}(b) = \{ \}$
 $\text{pts}(m) = \{ \}$
 $\text{pts}(p) = \{ \}$
 $\text{pts}(q) = \{ \}$
 $\text{pts}(r) = \{ \}$

Final

$\text{pts}(a) = \{ \}$
 $\text{pts}(b) = \{ \}$
 $\text{pts}(m) = \{ p, q \}$
 $\text{pts}(p) = \{ a, b \}$
 $\text{pts}(q) = \{ c \}$
 $\text{pts}(r) = \{ a, b, c \}$

POINTS TO AND TYPE SAFETY

REVIEW: LAST LECTURE

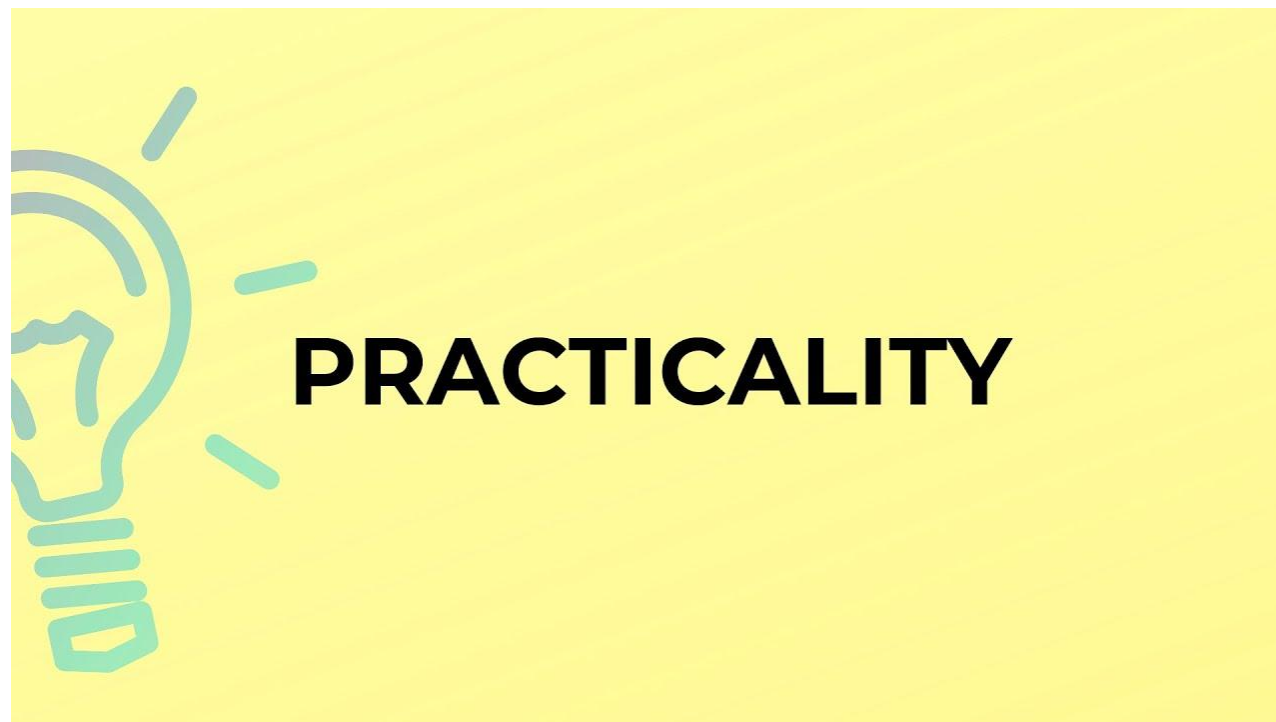
A “FEATURE” OF THE ANALYSIS

Our points-to relationships are somewhat contrived

Would a program ever actually have both of these statements?

`*a = b;`

`*a = *b;`



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



ANCIENT PROVERBS
FROM AROUND THE WORLD

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

Steengard's

IN PRACTICE ~~Andersen's~~

Step 1

List pointer-related operations

Step 2

equality

Induce set of ~~subset~~ constraints

Step 3

Solve system of constraints

REACHABILITY FORMULATION

Step 1

List pointer-related operations

Step 2

1-out

Saturate points-to graph

Step 3

Compute node reachability

Andersen's

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

Steengard's

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

EQUALITY CONSTRAINTS

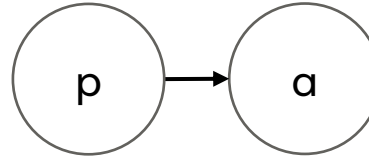
STEENSGARD'S ANALYSIS

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

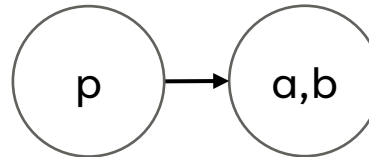
EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

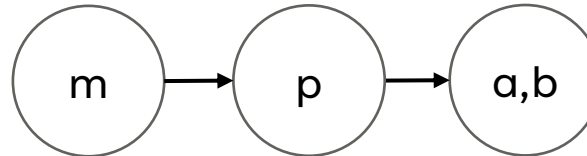
$p = \&a$



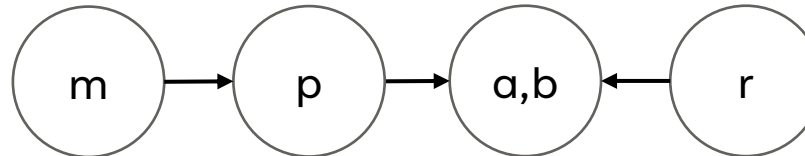
$p = \&b$



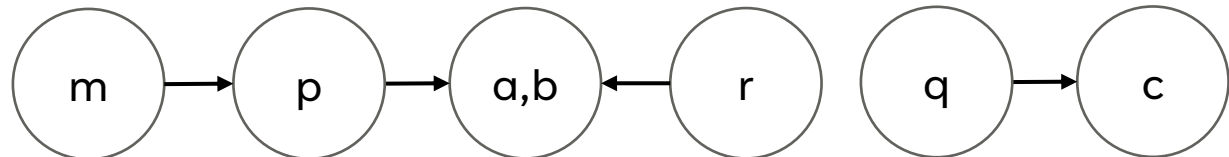
$m = \&p$



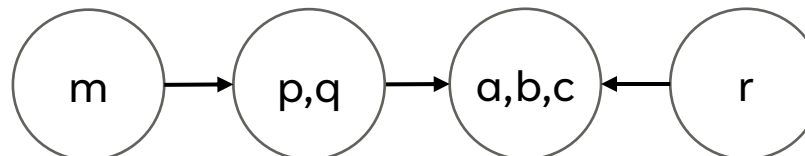
$r = *m$



$q = \&c$



$m = \&q$

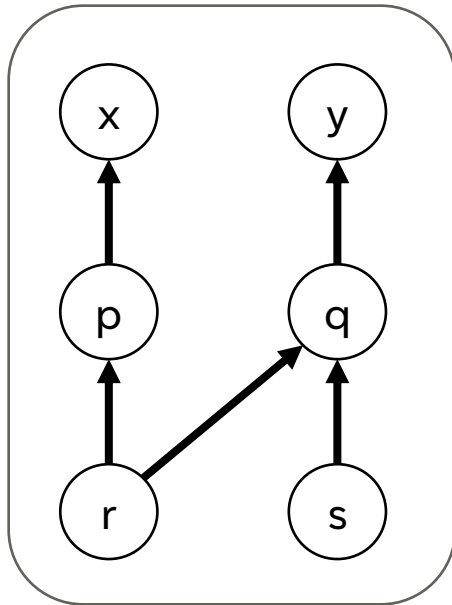


EQUALITY CONSTRAINTS

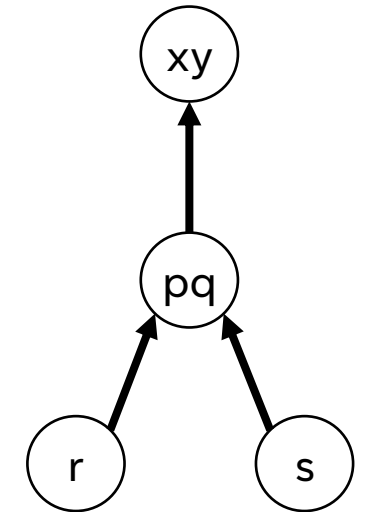
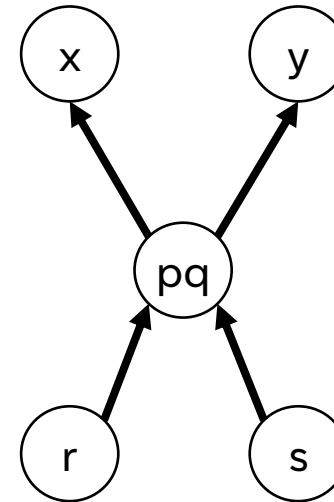
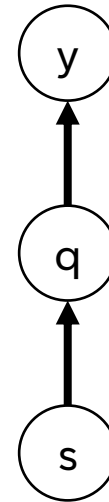
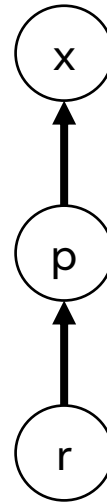
STEENSGARD'S ANALYSIS

$p = \&x$
 $r = \&p$
 $q = \&y$
 $s = \&q$
 $r = s$

Andersen's



Steensgard's



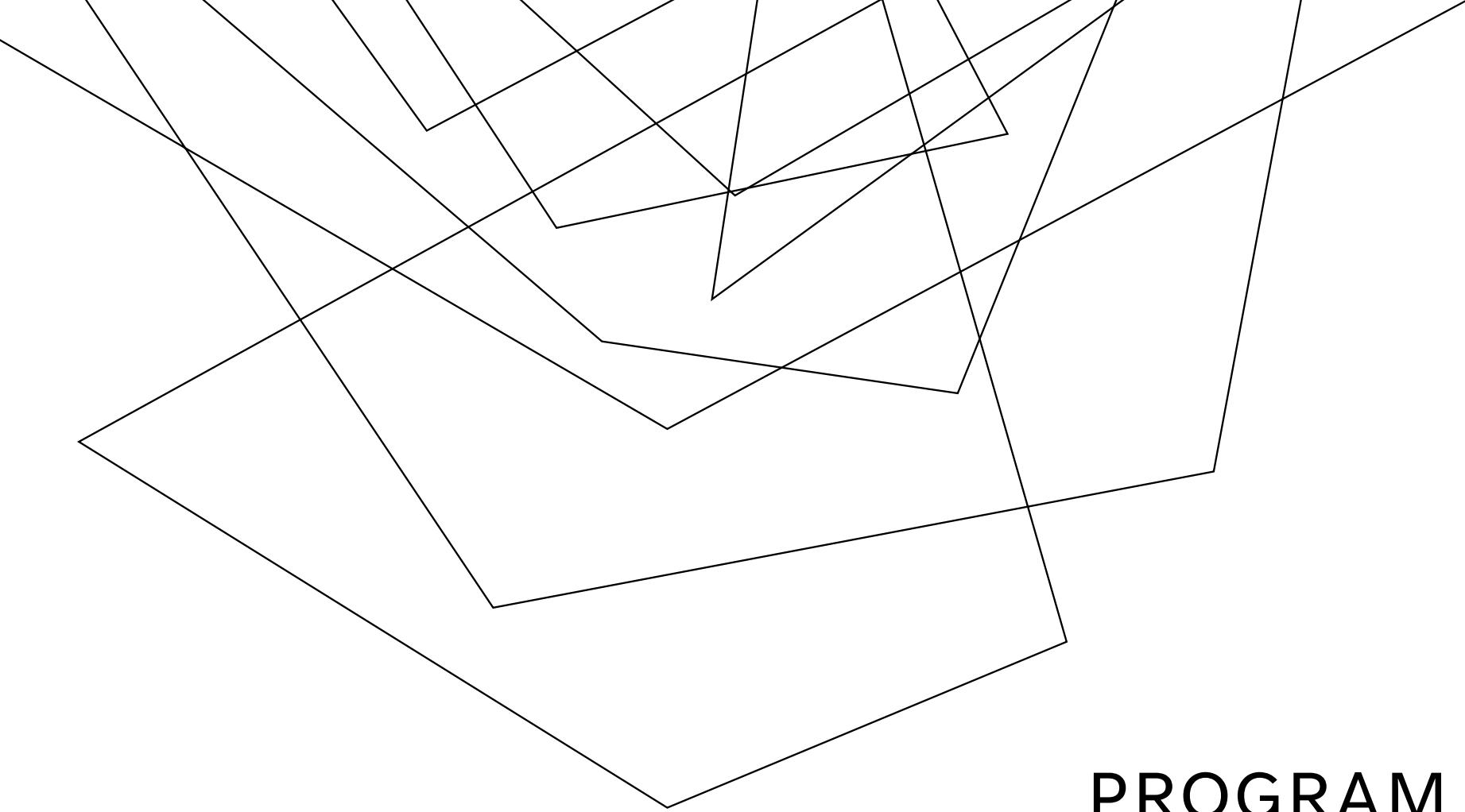
THAT'S POINTS-TO!

STEENSGARD'S ANALYSIS

AN ADDITIONAL OVERLAY ON DATAFLOW

Dataflow facts also flow to aliases

When dereferencing a pointer, consider only pointed-to objects

Abstract geometric lines in the top-left corner of the slide, consisting of several overlapping, irregular polygons and lines that create a complex, layered effect.

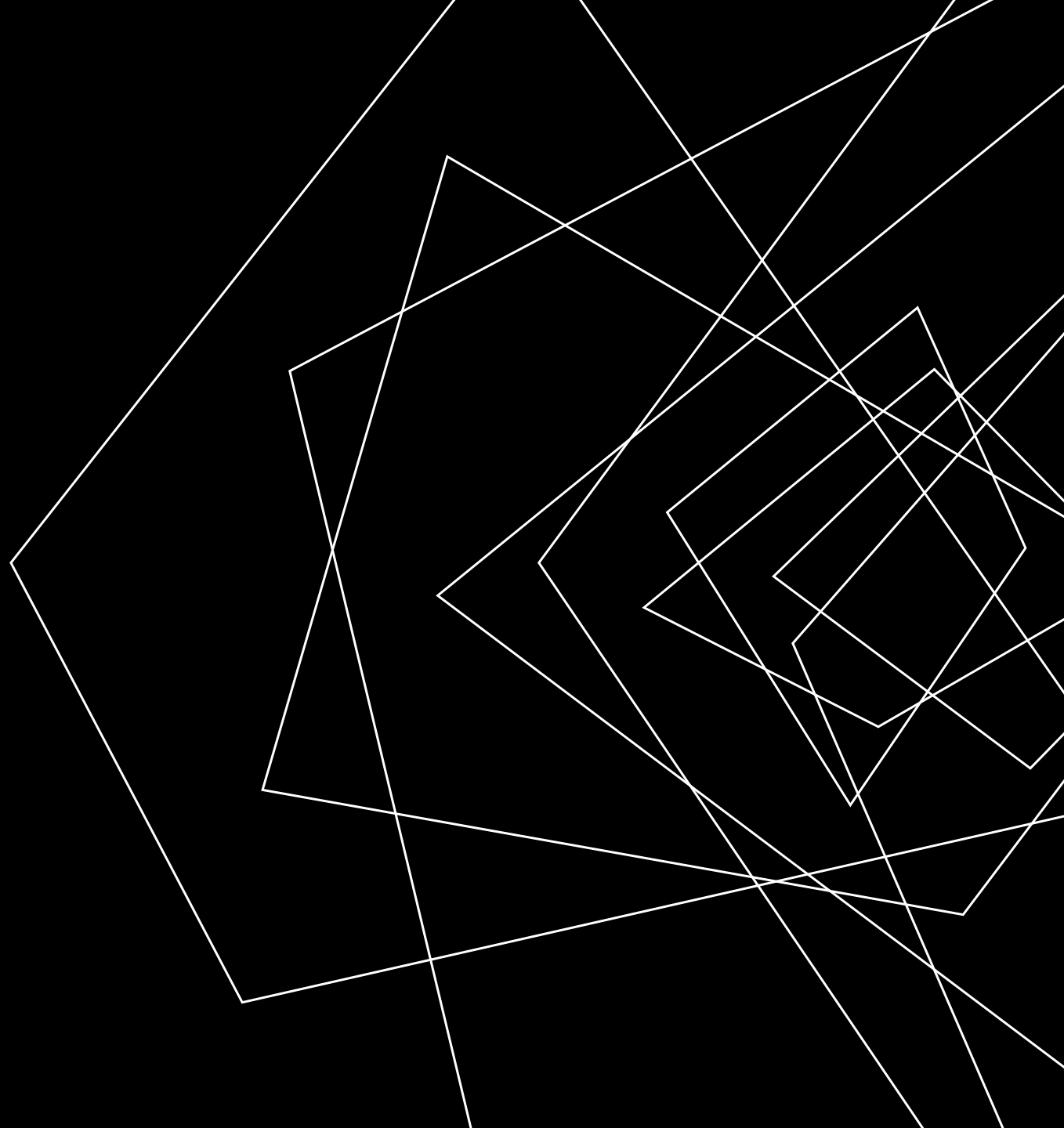
PROGRAM INSTRUMENTATION

EECS 677: Software Security Evaluation

Drew Davidson

LECTURE OUTLINE

- 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

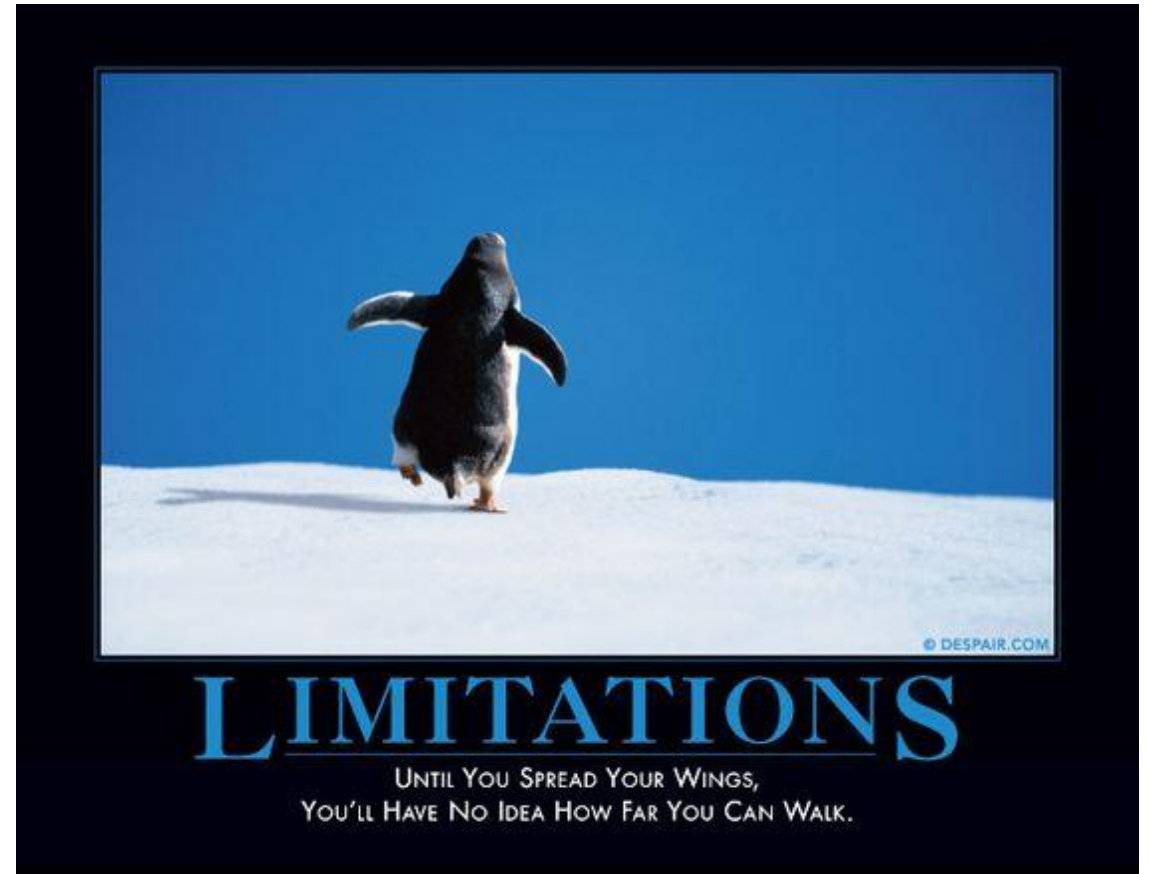


LIMITS OF STATIC ANALYSIS

PROGRAM INSTRUMENTATION: BASIC IDEA

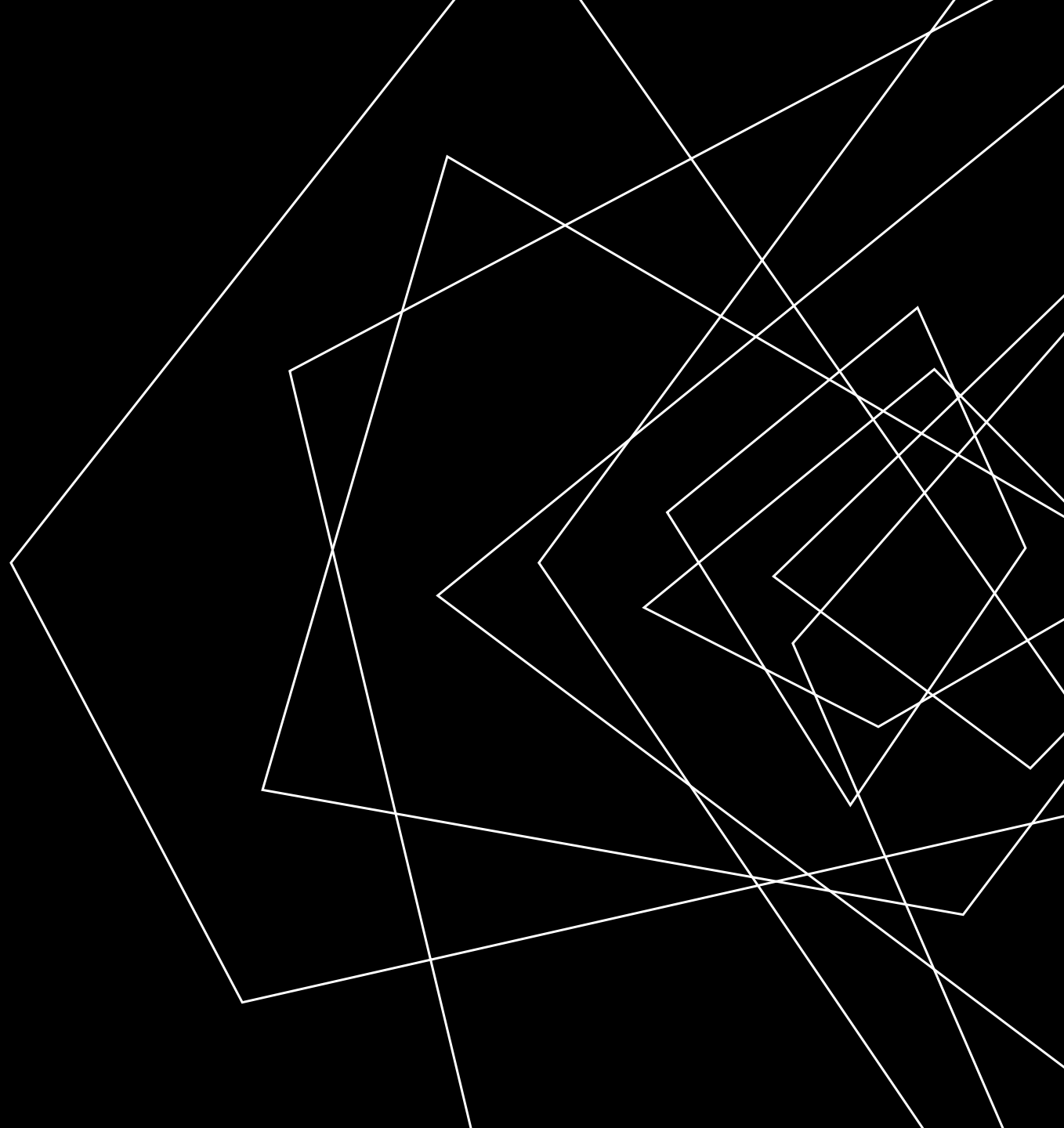
PRACTICAL ISSUES

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



LECTURE OUTLINE

- 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

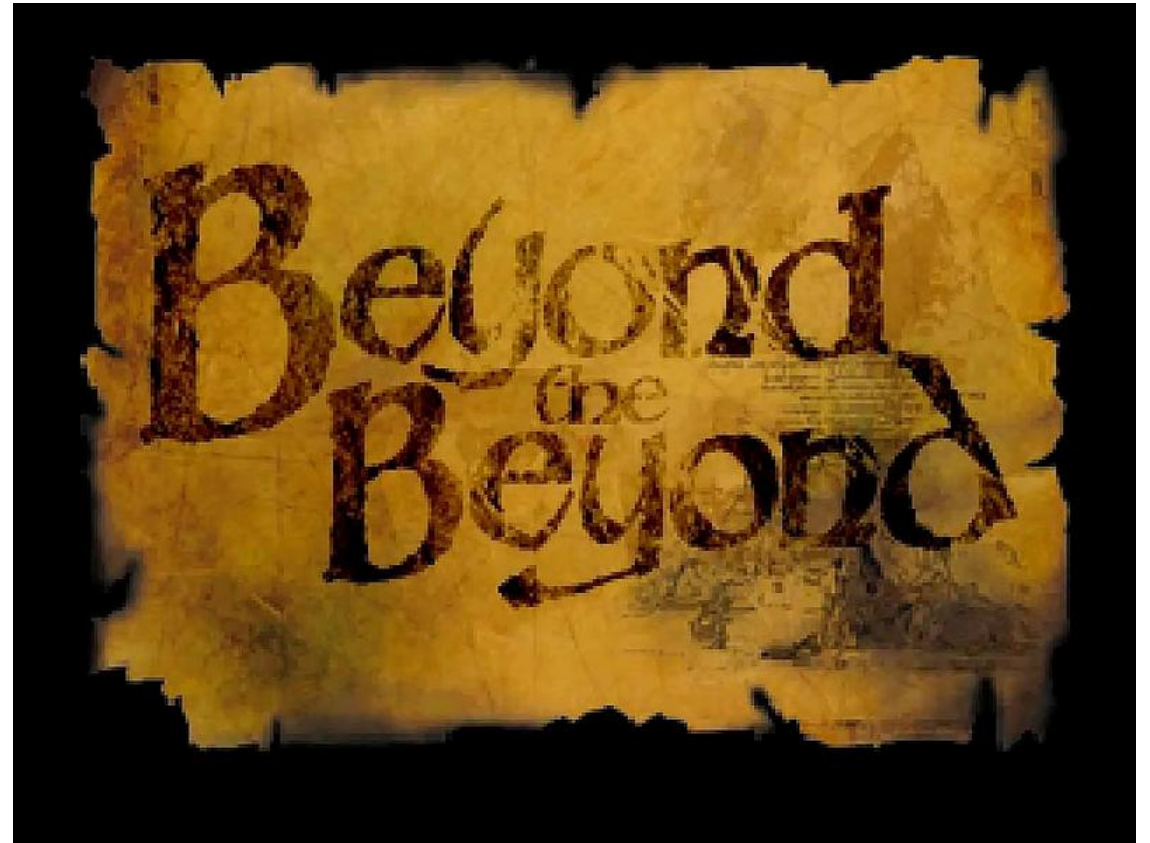


BEYOND TESTING

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

```
1: int main(){  
2:     int b = atoi(cin);  
3:     cout << a / b;  
4: }
```

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

```
1: int main(){  
2:     int b = atoi(cin);  
3:     cout << a / b;  
4: }
```

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



```
1: int main() {  
2:     int b = atoi(cin);  
3:     cout << a / b;  
4: }
```

Coverage metrics

line coverage
branch coverage
path coverage

if(x){

}
if(y){

}

EXAMPLE: CODE COVERAGE

PROGRAM INSTRUMENTATION: BASIC IDEA

COUNTING HOW MANY TIMES CERTAIN
BEHAVIORS OF THE PROGRAM ARE
EXERCISED

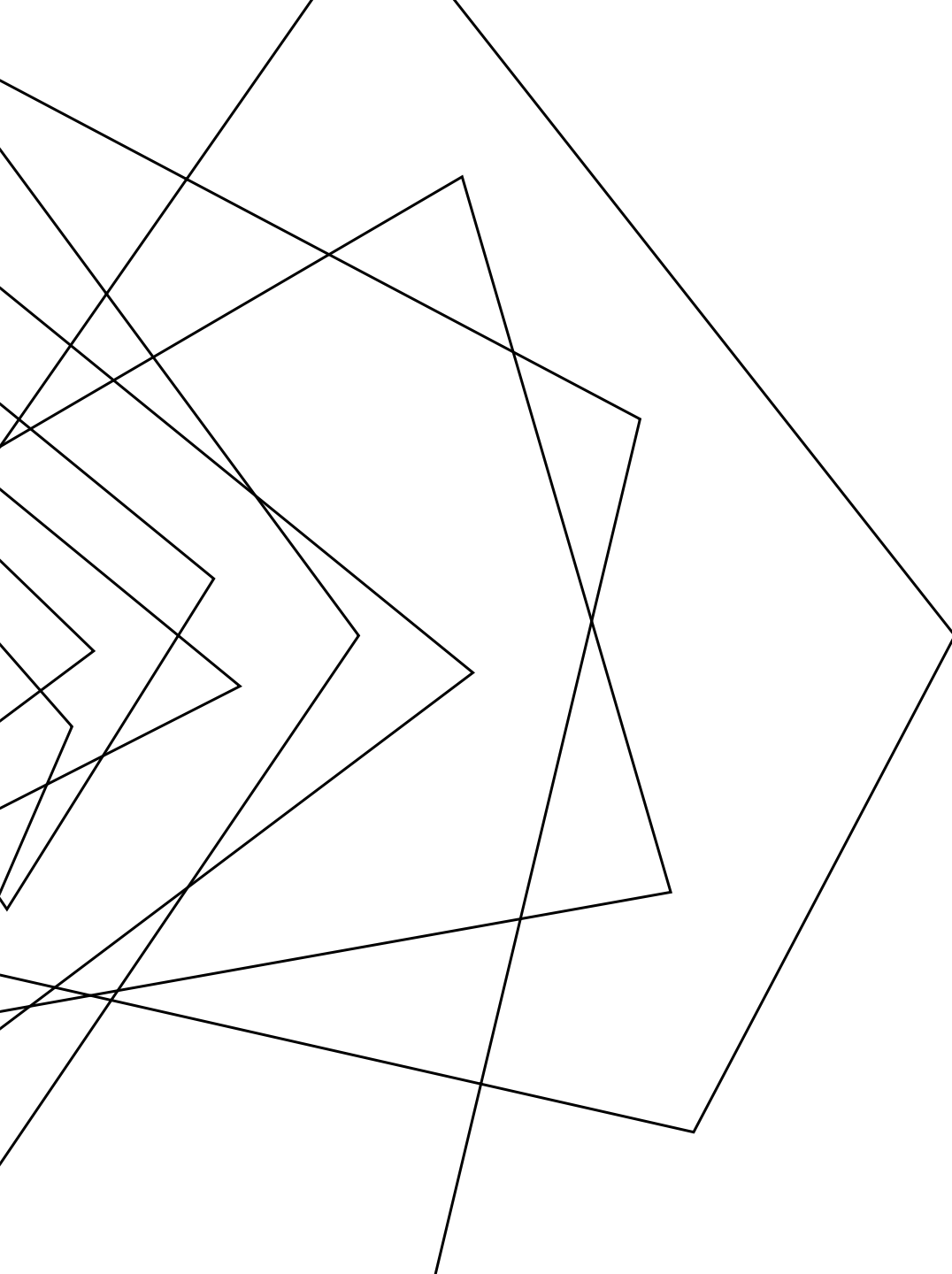
Why is this useful? (Placing sanitizers)

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

```
1: int main() {  
2:     int b = atoi(cin);  
3:     cout << a / b;  
4: }
```



WRAP-UP

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