EXERCISE #22

PROGRAM INSTRUMENTATION REVIEW

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

Give an example of a program where Steensgard's analysis will indicate a false-positive points-to relationship that Andersen's would avoid

EXERCISE #22 SOLUTION

PROGRAM INSTRUMENTATION REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS

STATIC INSTRUMENTATION

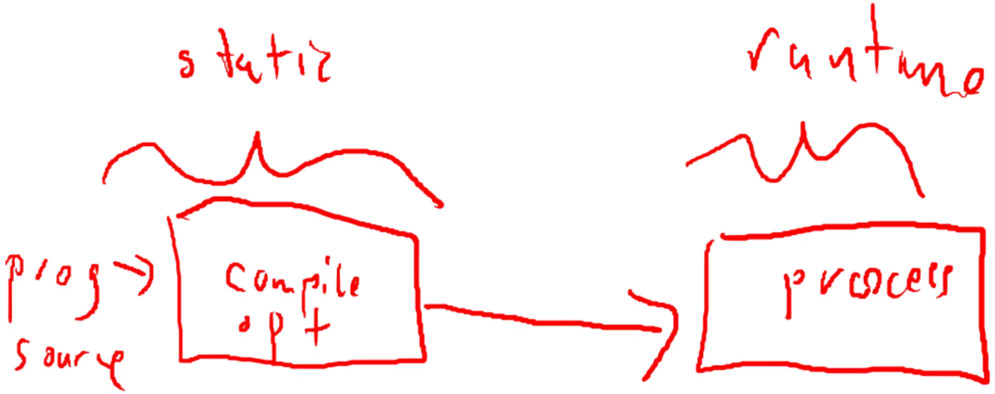
EECS 677: Software Security Evaluation

Drew Davidson

PREVIOUSLY: PROGRAM INSTRUMENTATION

REVIEW: LAST LECTURE

ALTER THE PROGRAM TO GAIN MORE INFORMATION OUT OF DYNAMIC ANALYSIS



TWO FORMS OF INSTRUMENTATION

Static instrumentation – Alter the program statically, to gain information at runtime

Dynamic instrumentation – Alter the program at runtime, potentially leveraging runtime info

THIS LESSON: STATIC INSTRUMENTATION

REVIEW: LAST LECTURE

INSERTING MEASUREMENT PROBES INTO A PROGRAM BEFORE IT IS RUN

More closely associated with proactive software evaluation – (why?)

STATIC INSTRUMENTATION TOOLS

PROGRAM INSTRUMENTATION: APPROACH

OFTEN BUILT RIGHT INTO COMPILER

LLVM Coverage tools

GCC Coverage tools

SOMETIMES BUILT UPON OPTIMIZER

Google's closure compiler

https://github.com/google/closure-compiler

LECTURE OUTLINE

- Example: Test Coverage
- Using LLVM Instrumentation
- Developing LLVM Instrumentation

TEST COVERAGE

EXAMPLE: TEST COVERAGE

HOW DO WE KNOW IF OUR TEST SUITE IS SUFFICIENT?

Line/branch/path coverage – How many lines/branches/paths of the program does suite exercise?

TEST COVERAGE

EXAMPLE: TEST COVERAGE

HOW DO WE KNOW IF OUR TEST SUITE IS SUFFICIENT?

Line/branch/path coverage – How many lines/branches/paths of the program does suite exercise?

```
int f(bool b) {
    Obj * o = null;
    int v = 2;
    if (b) {
        o = new Obj ();
        v = rand_int();
    }
    if (v == 2){
        o->setInvalid();
    }
    return o->property();
}
```

b = true
v = not 2

b=true
v = 2

b = false
v = 2

ASSESSING COVERAGE

PROGRAM INSTRUMENTATION: APPROACH

Big Idea: Inject Counters

Simple: Add a counter at every instruction

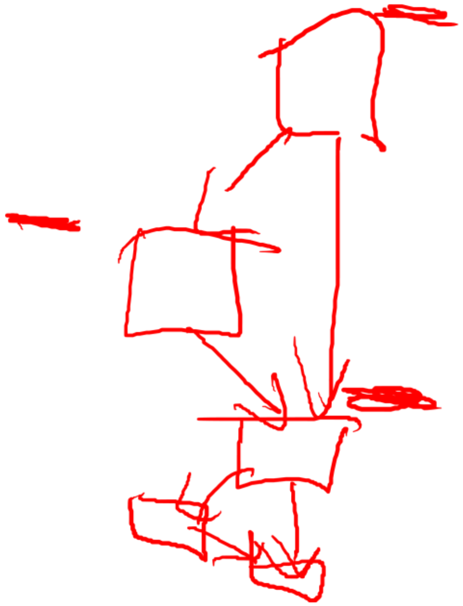
Better: Add a counter at every basic block

What coverage information does this give us?

Instruction: Yes!

Branch: Yes!

Path: No!



```
int f(bool b) {
  Obj * o = null;
  int v = 2;
  if (b) {
    o = new Obj ();
    v = rand_int();
  }
  if (v == 2){
    o->setInvalid();
  }
  return o->property();
}
```

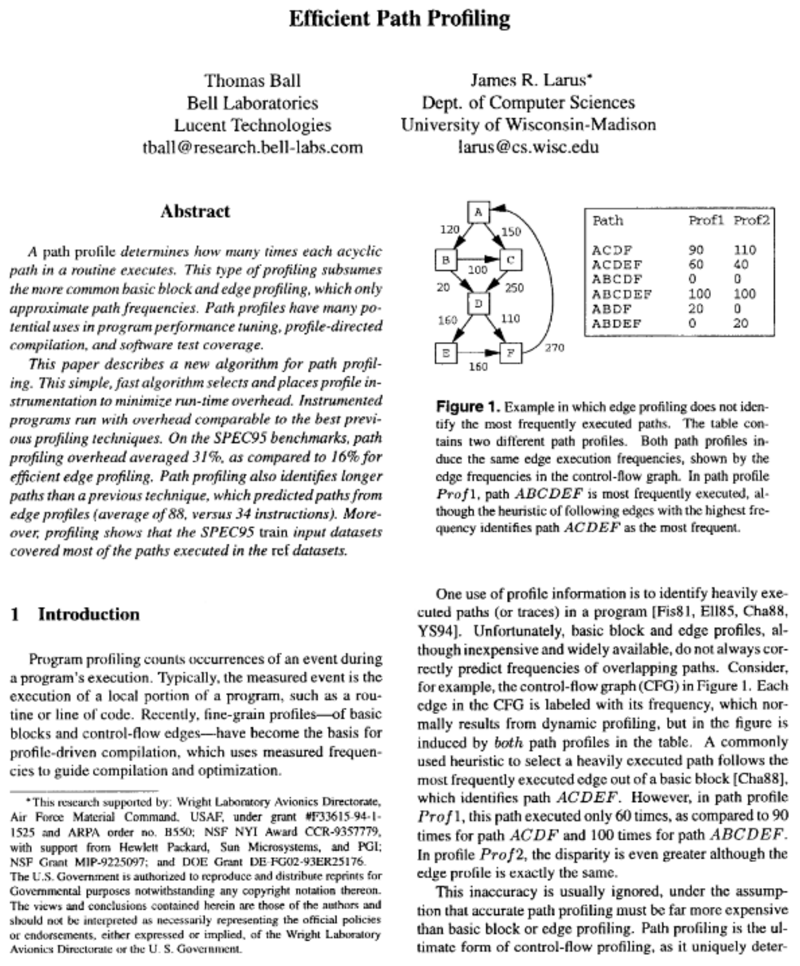
EFFICIENT PATH AND BRANCH COUNTERS

PROGRAM INSTRUMENTATION: APPROACH

BALL AND LARUS '96

Intuition:

- Assign integer values to edges such that no two paths compute the same path sum (Section 3.2).
- Select edges to instrument using a spanning tree



Efficient Path Profiling

Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

James R. Larus*
Dept. of Computer Sciences
University of Wisconsin-Madison
larus@cs.wisc.edu

Abstract

A path profile determines how many times each acyclic path in a routine executes. This type of profiling subsumes the more common basic block and edge profiling, which only approximate path frequencies. Path profiles have many potential uses in program performance tuning, profile-directed compilation, and software test coverage.

This paper describes a new algorithm for path profiling. This simple, fast algorithm selects and places profile instrumentation to minimize run-time overhead. Instrumented programs run with overhead comparable to the best previous profiling techniques. On the SPEC95 benchmarks, path profiling overhead averaged 31%, as compared to 16% for efficient edge profiling. Path profiling also identifies longer paths than a previous technique, which predicted paths from edge profiles (average of 88, versus 34 instructions). Moreover, profiling shows that the SPEC95 train *input datasets covered most of the paths executed in the* ref *datasets.*

1 Introduction

Program profiling counts occurrences of an event during a program's execution. Typically, the measured event is the execution of a local portion of a program, such as a routine or line of code. Recently, fine-grain profiles—of basic blocks and control-flow edges—have become the basis for profile-driven compilation, which uses measured frequencies to guide compilation and optimization.

Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Figure 1. Example in which edge profiling does not identify the most frequently executed paths. The table contains two different path profiles. Both path profiles induce the same edge execution frequencies, shown by the edge frequencies in the control-flow graph. In path profile $Prof1$, path $ABCDEF$ is most frequently executed, although the heuristic of following edges with the highest frequency identifies path $ACDEF$ as the most frequent.

One use of profile information is to identify heavily executed paths (or traces) in a program [Fis81, Ell85, Cha88, YS94]. Unfortunately, basic block and edge profiles, although inexpensive and widely available, do not always correctly predict frequencies of overlapping paths. Consider, for example, the control-flow graph (CFG) in Figure 1. Each edge in the CFG is labeled with its frequency, which normally results from dynamic profiling, but in the figure is induced by *both* path profiles in the table. A commonly used heuristic to select a heavily executed path follows the most frequently executed edge out of a basic block [Cha88], which identifies path $ACDEF$. However, in path profile $Prof1$, this path executed only 60 times, as compared to 90 times for path $ACDF$ and 100 times for path $ABCDEF$. In profile $Prof2$, the disparity is even greater although the edge profile is exactly the same.

This inaccuracy is usually ignored, under the assumption that accurate path profiling must be far more expensive than basic block or edge profiling. Path profiling is the ultimate form of control-flow profiling, as it uniquely deter-

---

[*] This research supported by: Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550; NSF NYI Award CCR-9357779, with support from Hewlett Packard, Sun Microsystems, and PGI; NSF Grant MIP-9225097; and DOE Grant DE-FG02-93ER25176. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U. S. Government.

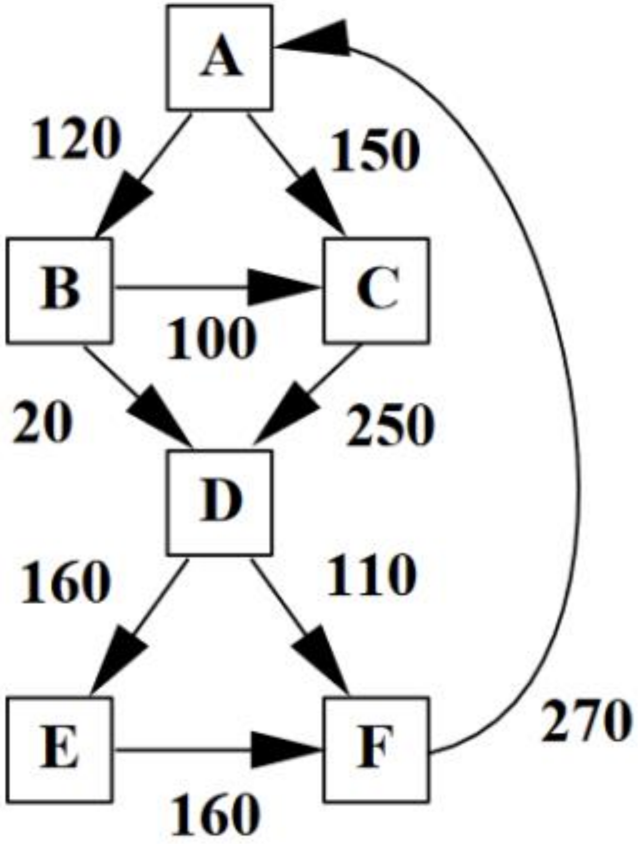
BRANCH FREQUENCY

PROGRAM INSTRUMENTATION: APPROACH

NAÏVE APPROACH: INSTRUMENT PROBES AT EACH EDGE

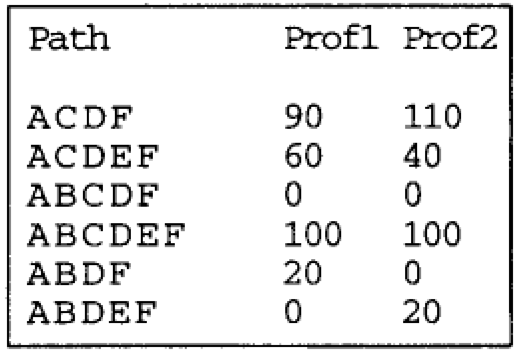
Inefficient!

We don’t really need an A -> B counter
(it’s the sum of the B-> C and B -> D counters)

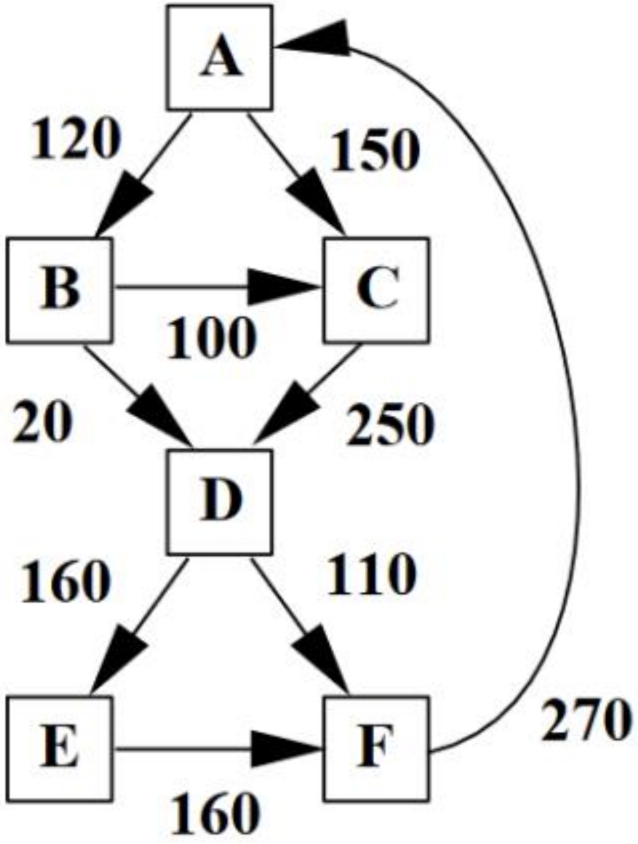


PATH FREQUENCY

REVIEW: THE PROBLEM



Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20



EXAMPLE: COVERAGE / FREQUENCY

PROGRAM INSTRUMENTATION: APPROACH

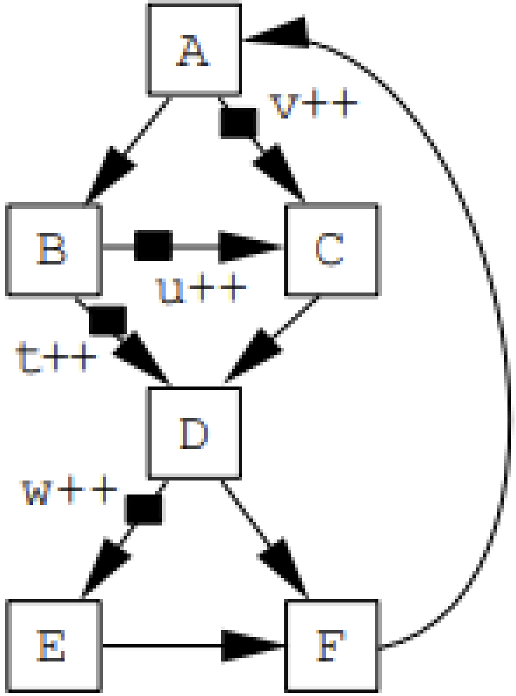
EXAMPLE OF INSTRUMENTATION: Counting Execution Frequency

Why is this useful? (Placing sanitizers)

Let’s first consider inserting edge counters

Inefficient!

We don’t really need an A -> B counter
(it’s the sum of the B-> C and B -> D counters)



C->D = u + v
D->F = t + u + v - w
E->F = w
A->B = t + u
F->A = t + u + v

LECTURE OUTLINE

- Example: Test Coverage
- Using LLVM Instrumentation
- Developing LLVM Instrumentation

SETUP / ASSUMPTIONS

LLVM BUILT-IN INSTRUMENTATION

THIS PORTION OF THE LECTURE USES A CLANG++ INSTALLATION.

Should work for many versions of LLVM (tested on clang++-18)
Works on clang++14 (which is installed on the cycle servers)

INSTALLATION (ON A LOCAL MACHINE)

```
sudo apt install clang++ llvm-dev
```

LLVM COVERAGE INSTRUMENTATION

LLVM BUILT-IN INSTRUMENTATION

GOAL: ASSESS THE COVERAGE OF A TEST SUITE

APPROACH: USE LLVM'S BUILT-IN INSTRUCTION INSTRUMENTATION

Piggyback on LLVM's PGO facilities

Profile-guided optimization

1) Insert PGO probes

2) Interpret probes as coverage measurements

LLVM: INSERTING PGO PROBES

LLVM BUILT-IN INSTRUMENTATION

`-fcoverage-mapping`

Map instrumentation results to source code

`-fprofile-instr-generate`

Generate profile information at the source instruction level

`-fprofile-generate`

Generate profile information at the IR level

LLVM: INSERTING PGO PROBES

LLVM BUILT-IN INSTRUMENTATION

Let’s write a simple LLVM program, then observe the probes…

```
clang++ prog.c –o prog -fprofile-instr-generate –emit-llvm
-fcoverage-mapping
```

This will cause the program to output an additional coverage file in the location of the environment variable LLVM_PROFILE_FILE

```
export LLVM_PROFILE_FILE=test1.prof
```

LLVM COVERAGE INSTRUMENTATION

LLVM BUILT-IN INSTRUMENTATION

GOAL: ASSESS THE COVERAGE OF A TEST SUITE

APPROACH: USE LLVM'S BUILT-IN INSTRUCTION INSTRUMENTATION

Piggyback on LLVM’s PGO facilities

1) Insert PGO probes

2) Interpret probes as coverage measurements

LLVM: COVERAGE REPORT

LLVM BUILT-IN INSTRUMENTATION

The profile file is useful for a variety of things (i.e. PGO). As such, it is not (immediately) human-readable

We’ll use some extra tools to generate a readable report

```
llvm-profdata merge -sparse test1.prof -o final.profdata
```

`llvm-cov`~~`-14`~~ `show` ~~`badcalc`~~ `-instr-profile=final.profdata >& profile.report`

prog

PUTTING IT ALL TOGETHER

REVIEW: LAST LECTURE

THESE COMMANDS WORK FINE FOR 1 TEST RUN, BUT WE CARE ABOUT TEST SUITES

```
clang++-18 prog.ll –o prog -fprofile-instr-generate -fcoverage-mapping

export LLVM_PROFILE_FILE=test1.prof
./prog

export LLVM_PROFILE_FILE=test2.prof
./prog

llvm-profdata merge -sparse test*.prof -o final.profdata

llvm-cov-18 show prog -instr-profile=final.profdata >& profile.report
```

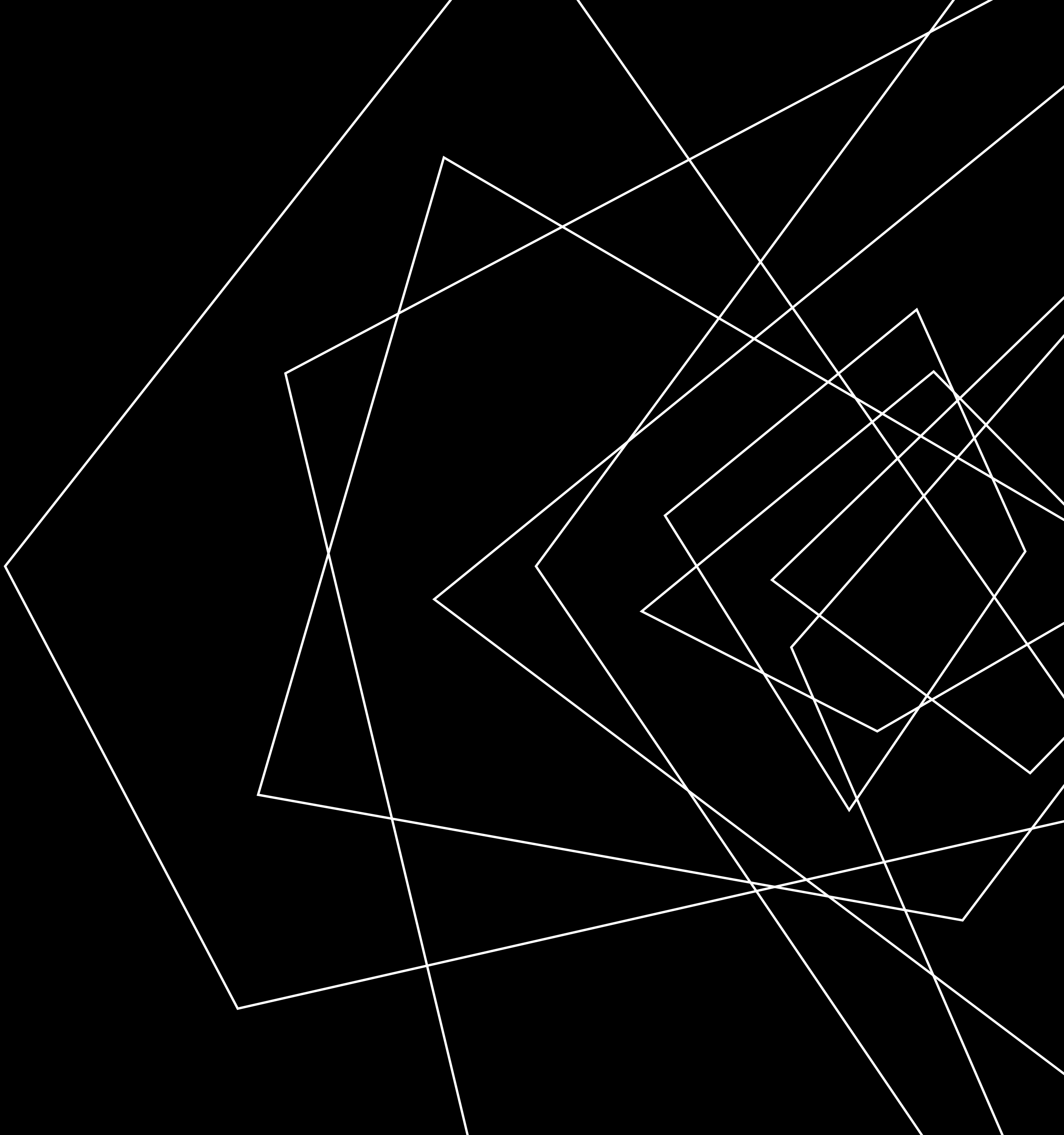
EXAMPLE: LLVM COVERAGE INSTRUMENTATION

PROGRAM INSTRUMENTATION: APPROACH

LET’S TAKE IT TO THE TERMINAL!

LECTURE OUTLINE

- Example: Test Coverage
- Using LLVM Instrumentation
- Developing LLVM Instrumentation



CUSTOM INSTRUMENTATION

PROGRAM INSTRUMENTATION: APPROACH

THE PREVIOUS EXAMPLE TOOK ADVANTAGE OF PRE-EXISTING INSTRUMENTATION

What if we wanted to make our own custom instrumentation?

CUSTOM INSTRUMENTATION

PROGRAM INSTRUMENTATION: APPROACH

GETTING STARTED

1) Reference the LLVM API
2) Build our own (trivial) analysis pass
3) Hook into the LLVM opt infrastructure
4) Run our analysis pass

GOING FURTHER

Insert more full-featured functionality (https://llvm.org/doxygen/classllvm_1_1IRBuilder.html)

EXAMPLE: LLVM CUSTOM INSTRUMENTATION

PROGRAM INSTRUMENTATION: APPROACH

LET'S REMOVE AND ADD SOME INSTRUCTIONS!

Consider a simple "add2" program:

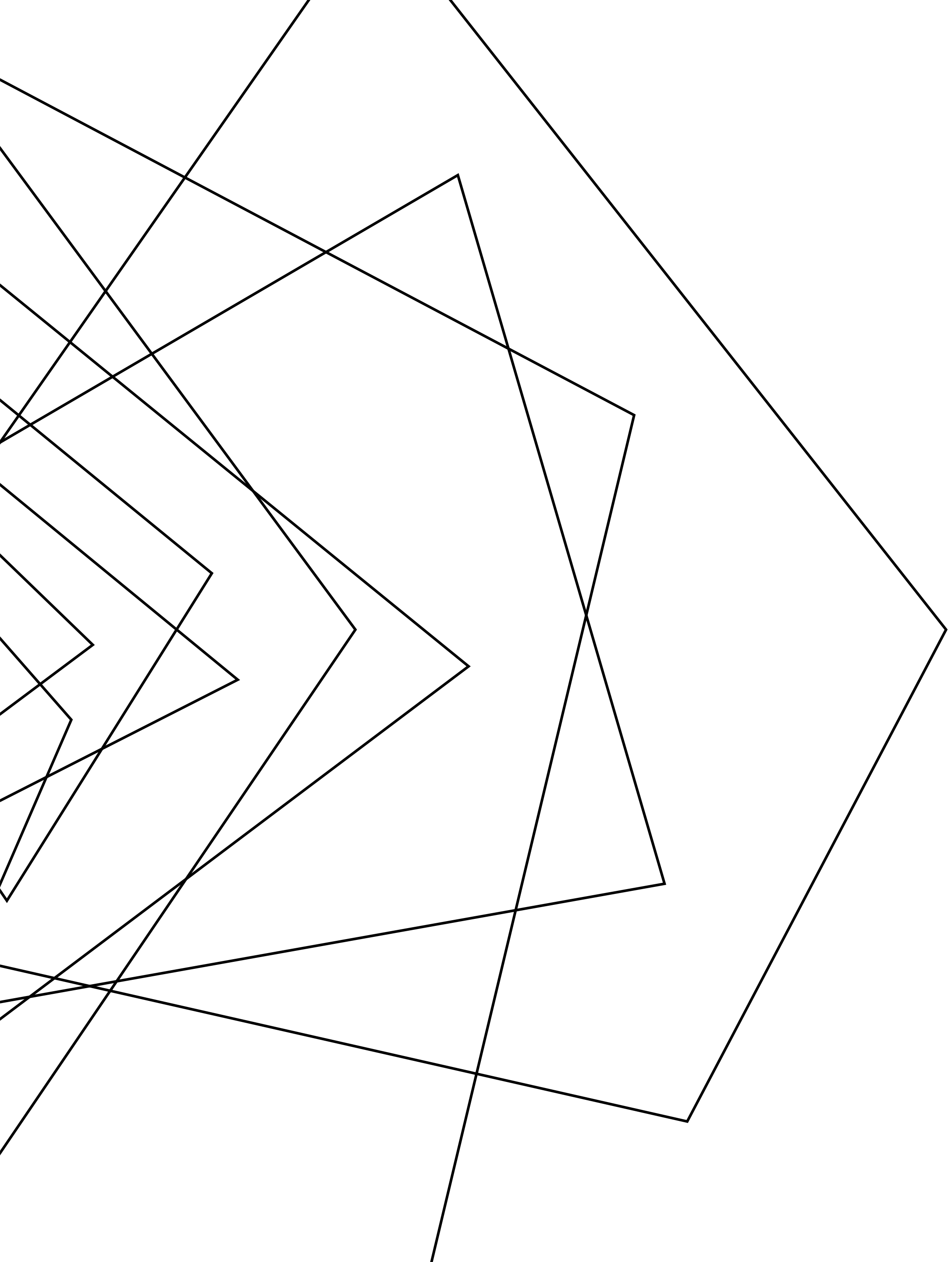
```
#include <stdio.h>
int main(int argc, const char**
argv) {
    int num;
    scanf("%i", &num);
    printf("%i\n", num + 2);
    return 0;
}
```

Let's change this to a multiply

EXAMPLE: LLVM CUSTOM INSTRUMENTATION

PROGRAM INSTRUMENTATION: APPROACH

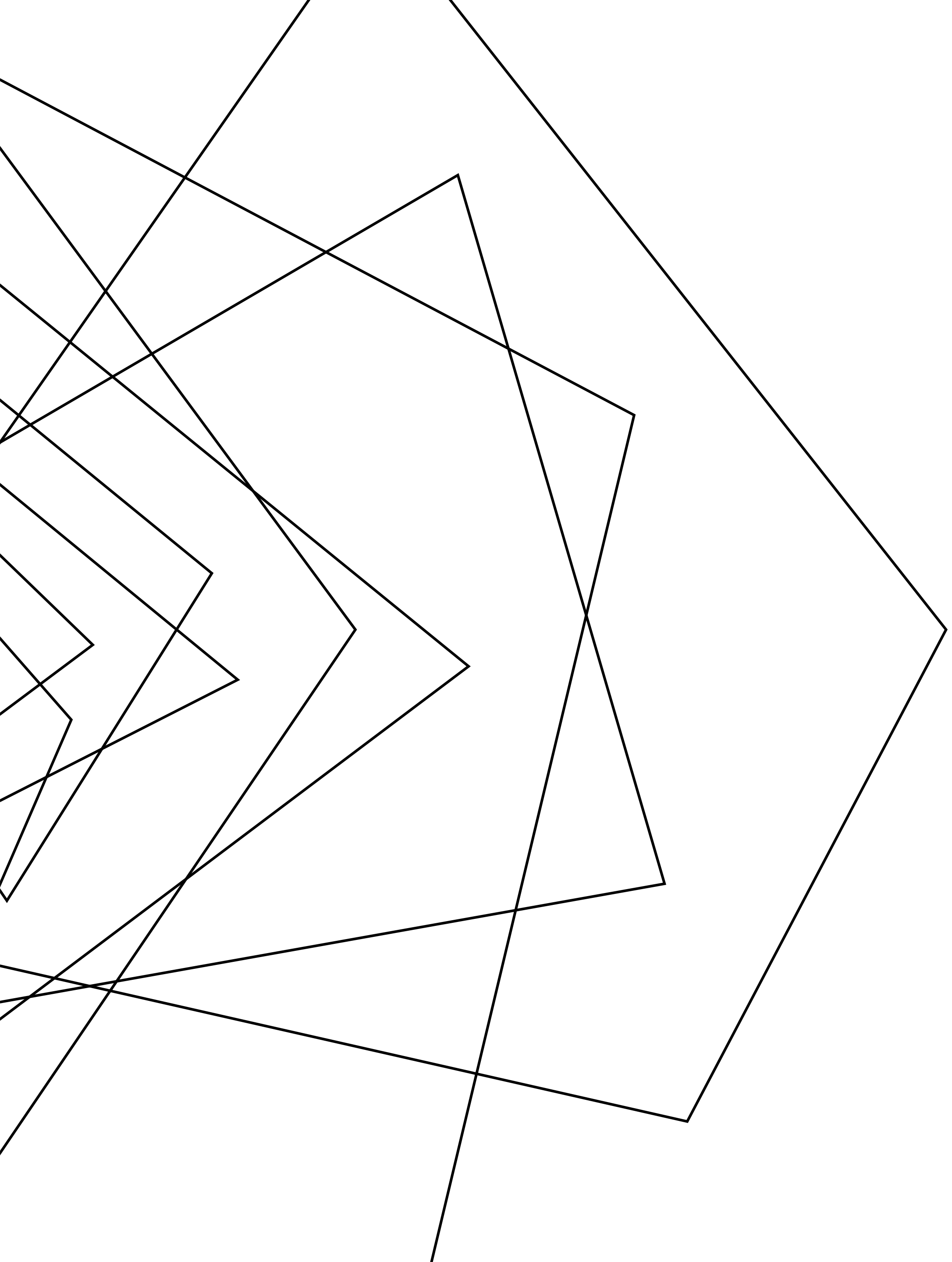
LET'S TAKE IT TO THE TERMINAL!



WRAP-UP

WE'VE DESCRIBED THE THEORY AND PRACTICE OF PROGRAM INSTRUMENTATION

Next time: Consider how we generate test cases



WRAP-UP

WE'VE DESCRIBED 2 FORMS OF ALTERING THE PROGRAM

More heuristic by nature

LLVM: COVERAGE MAPPING

LLVM BUILT-IN INSTRUMENTATION

For understanding line coverage, we need to map changes to source code

```
clang++ prog.c –o prog -fprofile-instr-generate –emit-llvm
-fcoverage-mapping
```

This will cause the program to output an additional coverage file in the location of the environment variable LLVM_PROFILE_FILE

```
export LLVM_PROFILE_FILE=test1.prof
```