EXERCISE #25

LINTING REVIEW

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

Give an example of a legal program in C that a linter would nevertheless flag

Quiz 2 Grades Forthcoming

Quiz 2 recap lecture

ADMINISTRIVIA AND ANNOUNCEMENTS

PROGRAM INSTRUMENTATION

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: LINTING

REVIEW: LAST LECTURE

Removing the "Stray Fibers" of a program

Analyze common "anti-patterns" that are likely to cause issues (security and otherwise)

Notable analysis tools

Lint – The original analysis tool
Splint – Security analysis tool

CLASS PROGRESS

MOVING ON TO DYNAMIC ANALYSIS

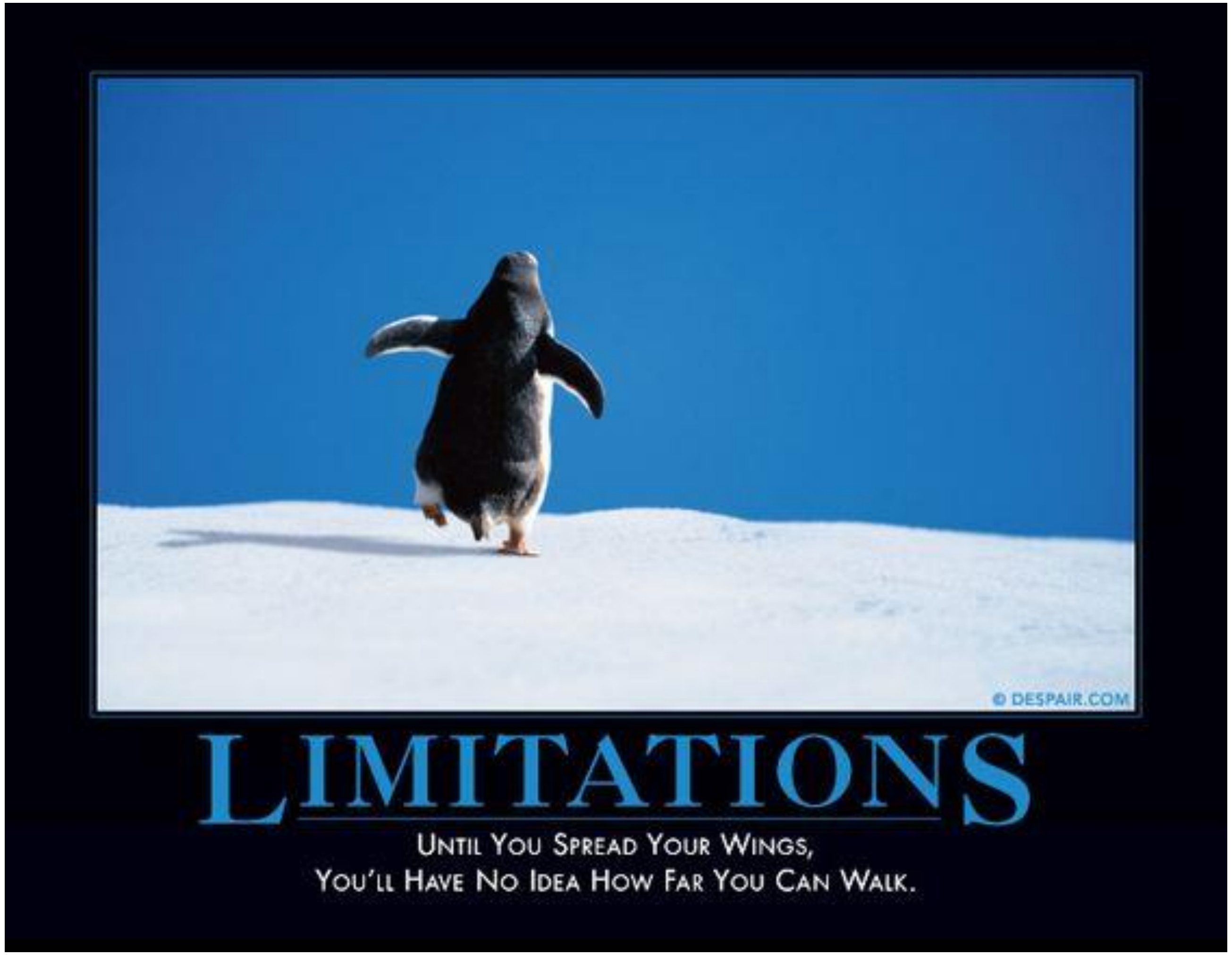
More heuristic by nature

LIMITS OF STATIC ANALYSIS

PROGRAM INSTRUMENTATION: BASIC IDEA

Practical Issues

- Unsoundness of bug finding / incompleteness of program verification
- Scalability
- Significant engineering effort


© DESPAIR.COM
LIMITATIONS
Until You Spread Your Wings,
You'll Have No Idea How Far You Can Walk.

REVISING DYNAMIC ANALYSIS

PROGRAM INSTRUMENTATION: BASIC IDEA

GIVING UP ON GUARANTEES

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

ADVANTAGES

- Simplest form: testing

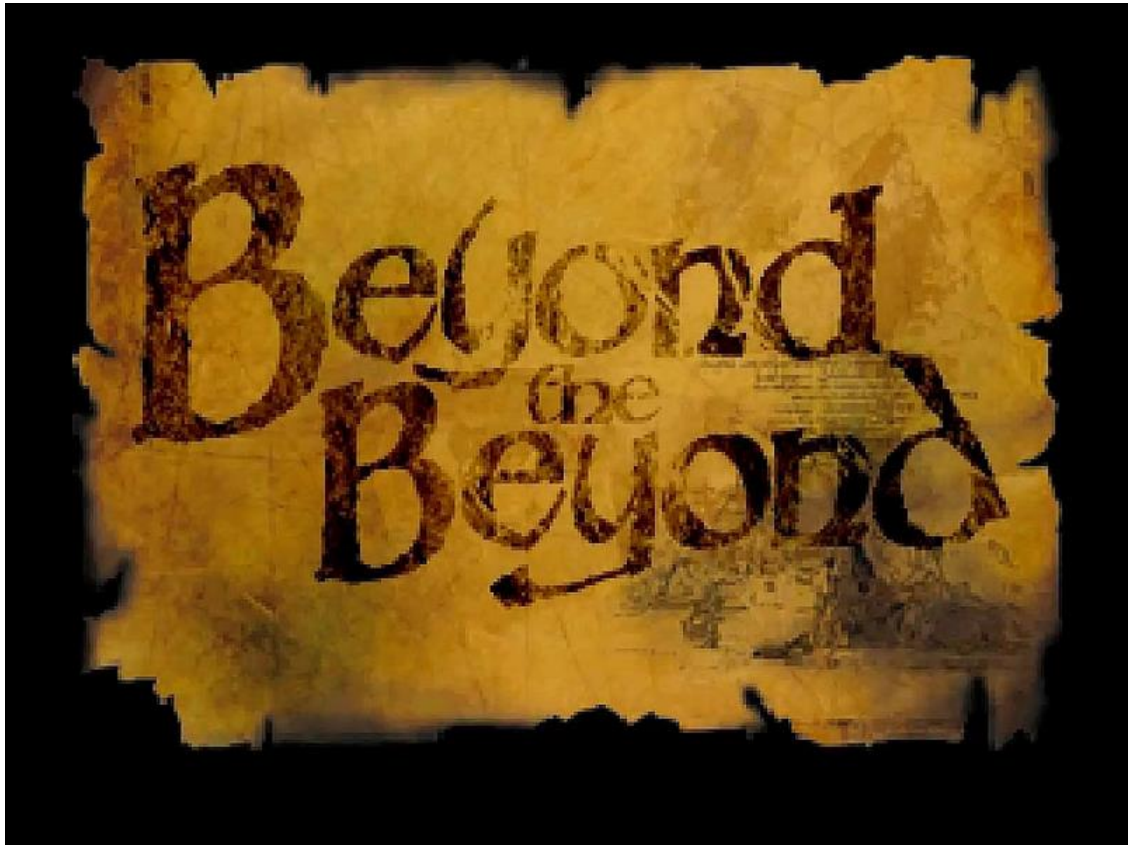
- Scalability & - Partial credit

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



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

Lack of Holistic Information

Somewhat limited by the information the probes can report

what does running a probe gain vs cost

what does calculate smart probe placement gain vs cost?

EXAMPLE: CONTROL PROFILING

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

Ball and Larus '96

```
V = SECRET
for ( i = 1 .. 1000000 ) {
  i++
}
LEAK(V);
```

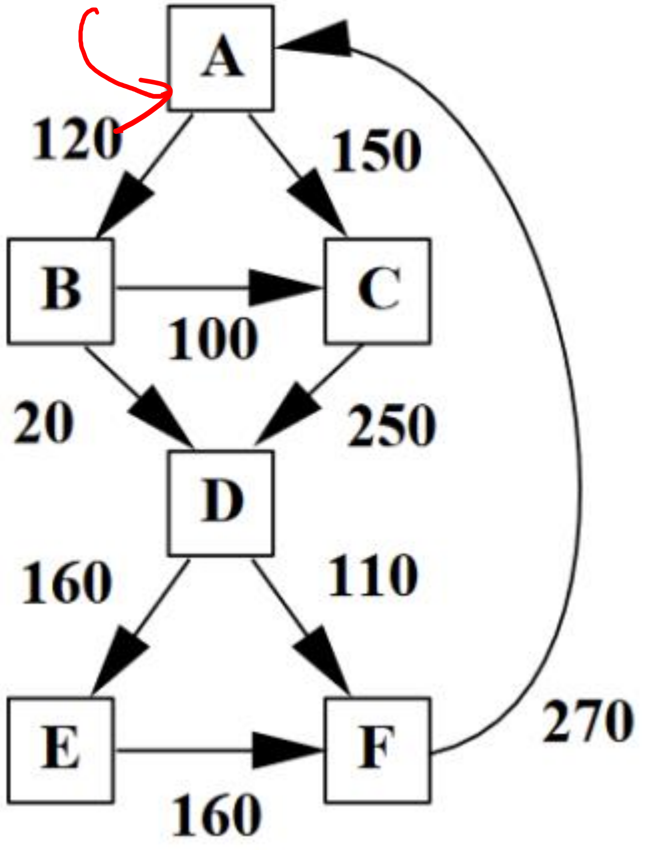
BRANCH FREQUENCY

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)



EXAMPLE: COVERAGE / FREQUENCY

PROGRAM INSTRUMENTATION: APPROACH

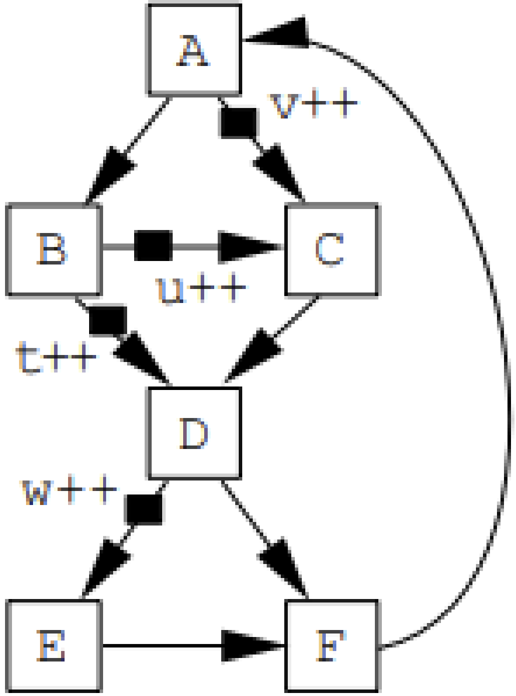
EXAMPLE OF INSTRUMENTATION: COUNTING EXECUTION FREQUENCY

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)

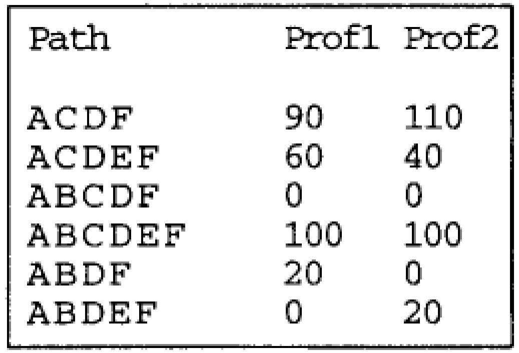


C->D = u + v
D->F = t + u + v - w
E->F = w
A->B = t + u
F->A = t + u + v

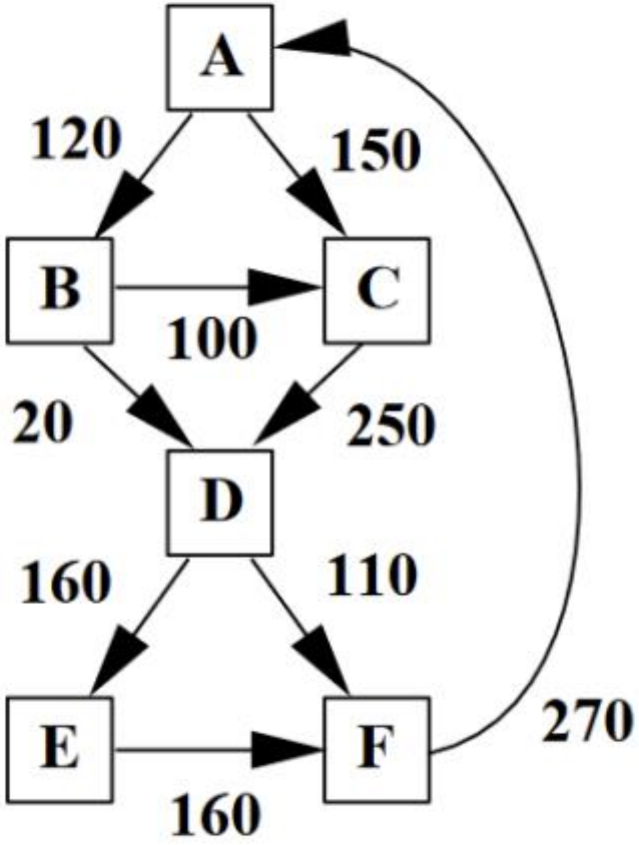
PATH FREQUENCY

PROGRAM INSTRUMENTATION: APPROACH

NAÏVE IMPLEMENTATION: SUM UP EDGE COUNTERS



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

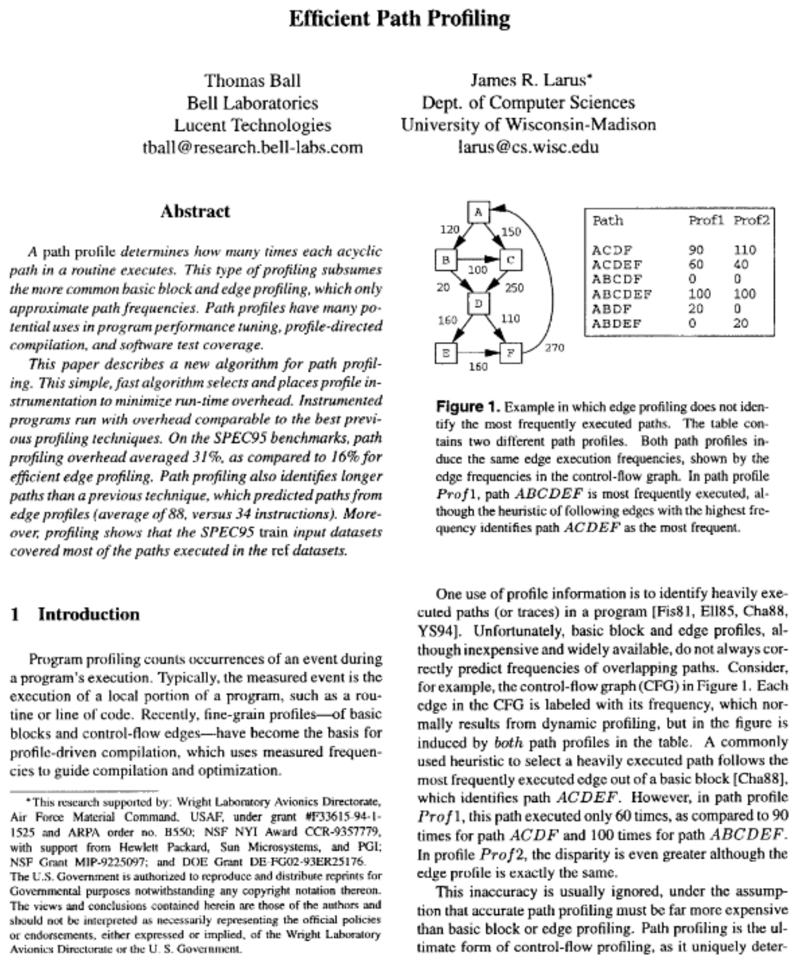


EFFICIENT PATH AND BRANCH COUNTERS

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.

INSTRUMENTATION APPROACHES

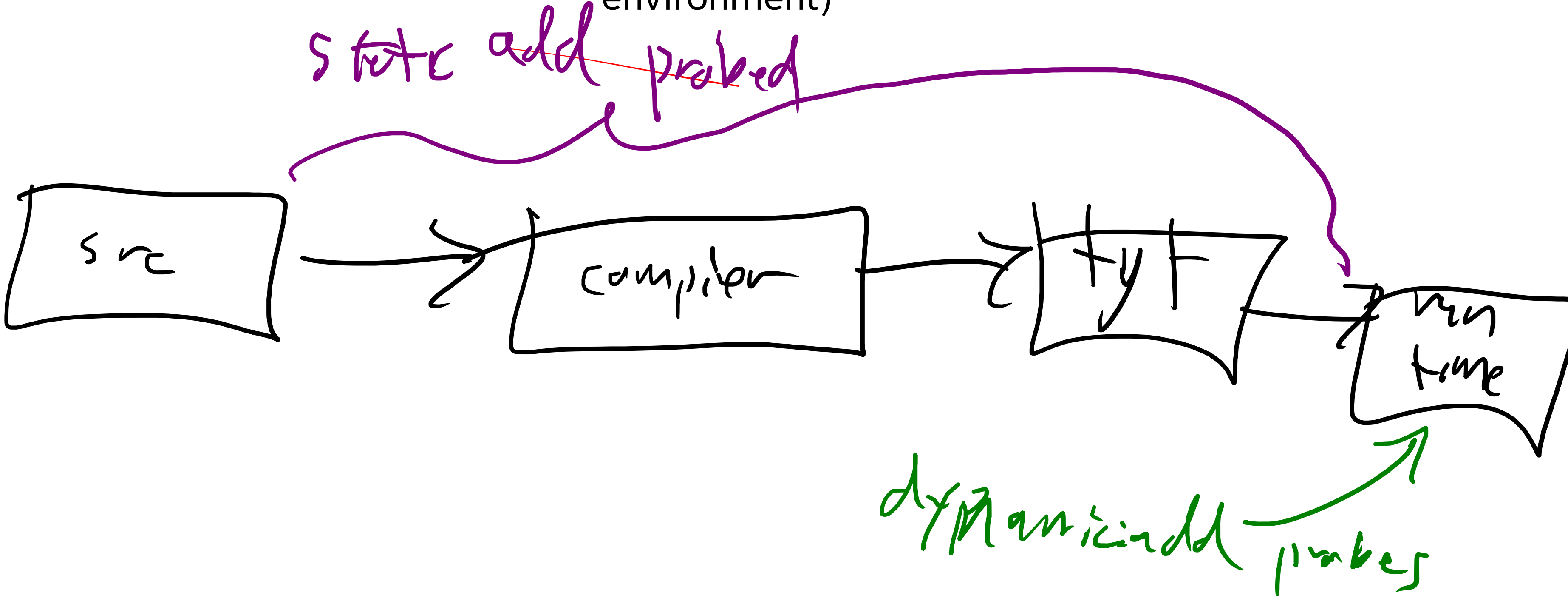
PROGRAM INSTRUMENTATION: APPROACH

Static Instrumentation

Add probes before the program is run (i.e. rewrite the program executable)

Dynamic Instrumentation

Probe while the program is run (i.e. insert probes just-in-time or as part of the environment)



DYNAMIC INSTRUMENTATION TOOLS

PROGRAM INSTRUMENTATION: APPROACH

Frequently Involve a custom runtime

Add probes before the program is run (i.e. rewrite the program executable)

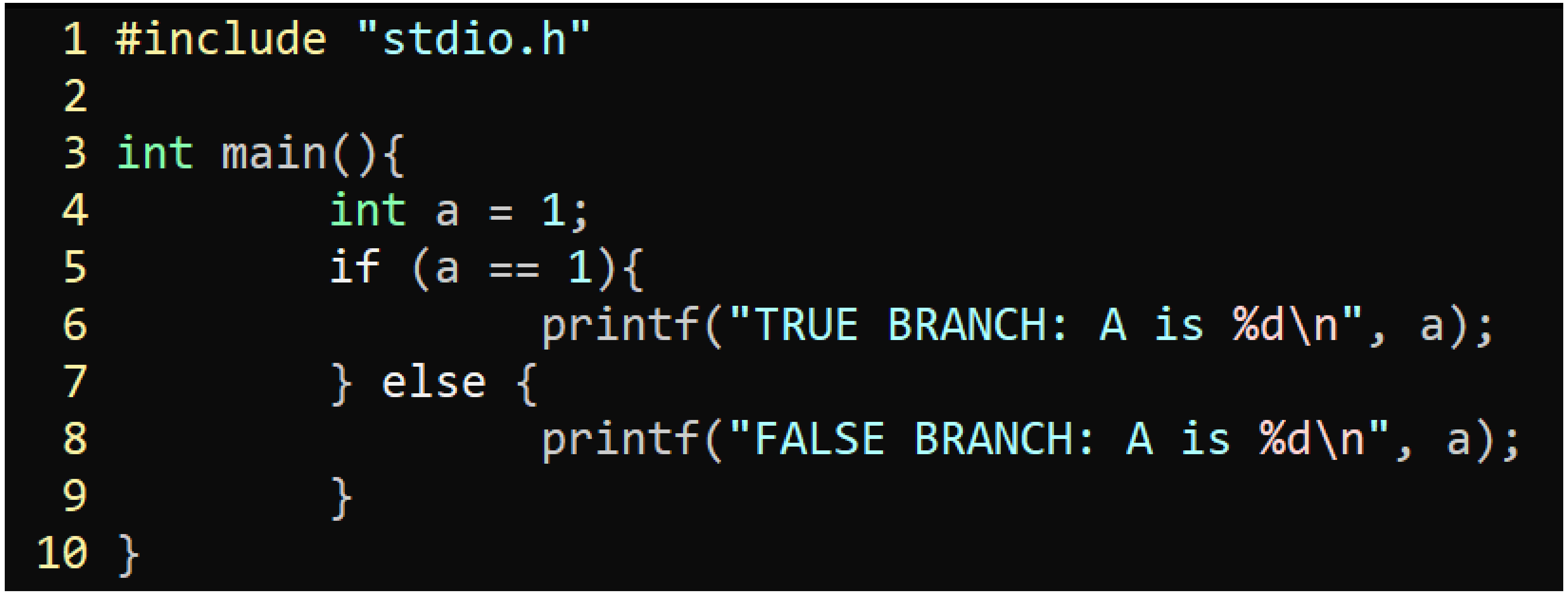
Examples

Intel PIN

GDB

DYNAMIC INSTRUMENTATION EXAMPLE: GDB

PROGRAM INSTRUMENTATION: APPROACH

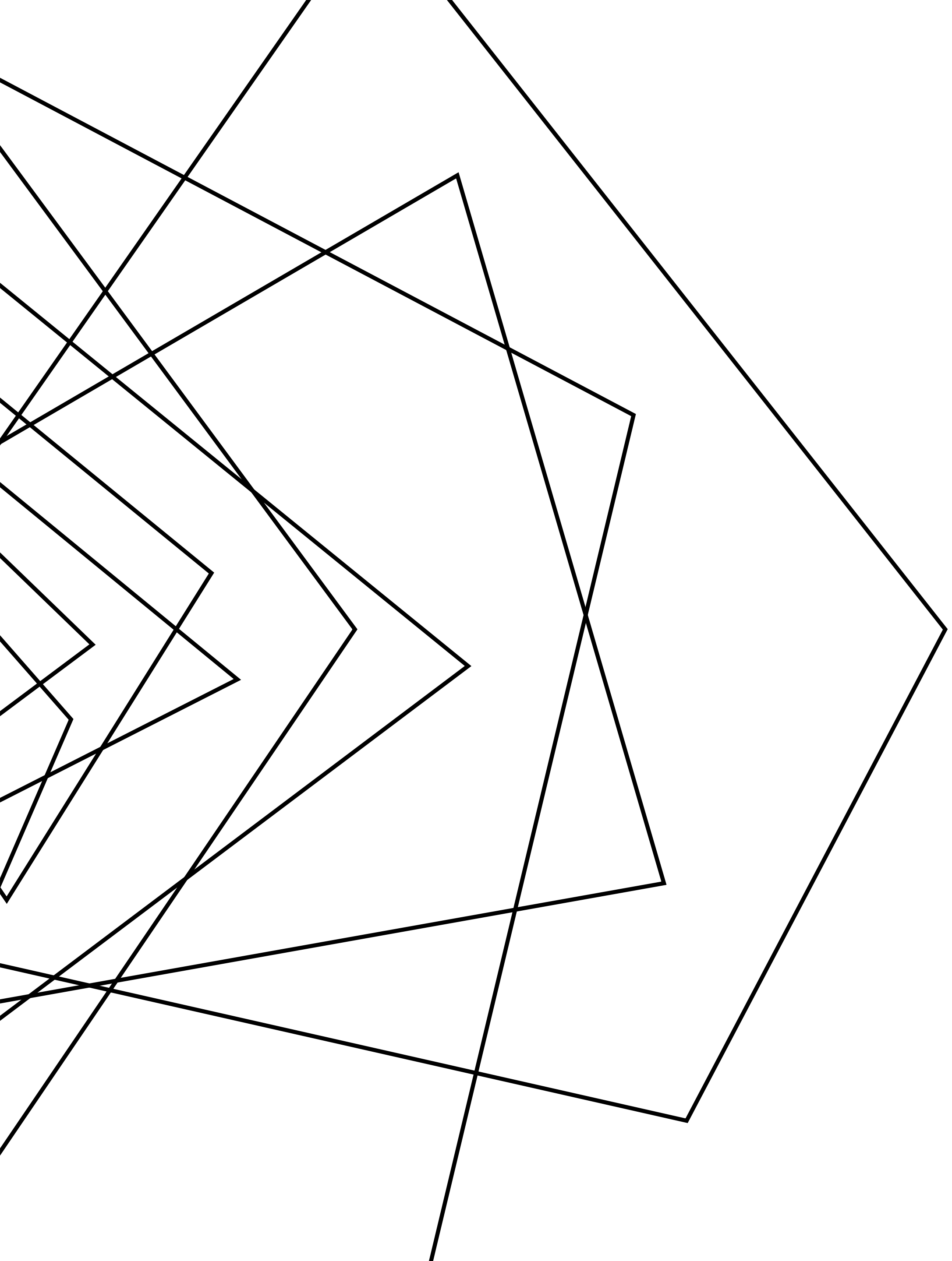


```
 1 #include "stdio.h"
 2
 3 int main(){
 4         int a = 1;
 5         if (a == 1){
 6                 printf("TRUE BRANCH: A is %d\n", a);
 7         } else {
 8                 printf("FALSE BRANCH: A is %d\n", a);
 9         }
10 }
```

```
gcc -O0 –g prog.c –o prog

gdb prog

b 5
set variable a = 3
n
```



WRAP-UP

WE'VE DESCRIBED 2 FORMS OF ALTERING THE PROGRAM

More heuristic by nature