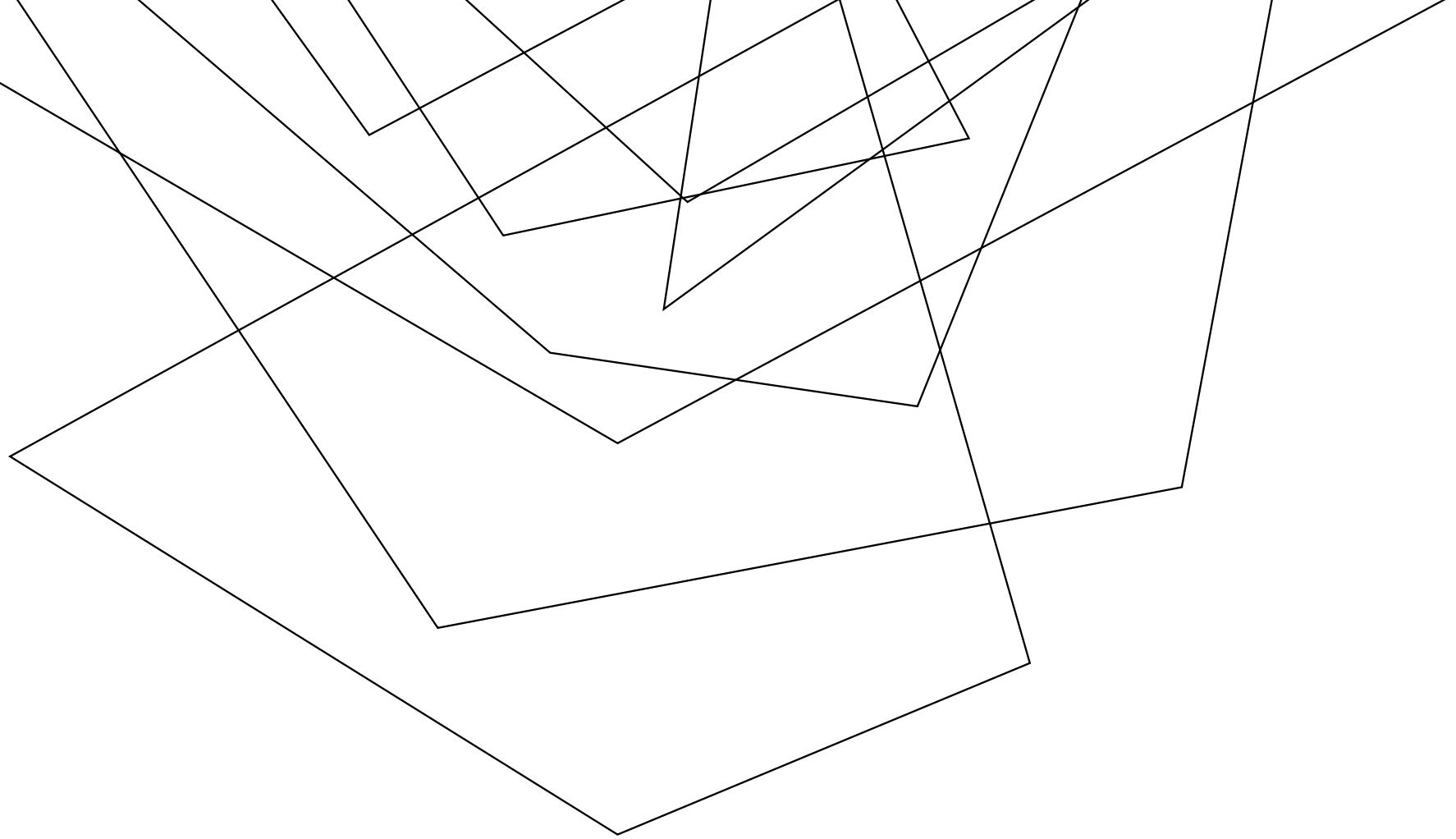
EXERCISE 28

CONTROL FLOW INTEGRITY REVIEW

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

Of the various CFI solutions we explored, none were calling-context sensitive. Why not?



TESTING

EECS 677: Software Security Evaluation

Drew Davidson

ADMINISTRIVIA AND ANNOUNCEMENTS

Midpoint Survey Live: https://analysis.cool/midpoint

LAST TIME: CONTROL-FLOW INTEGRITY

REVIEW: COMPUTABILITY

Instrument the program to prevent “illegal” jumps

- Intel CET
- Microsoft control-flow guard
- Clang instruction injection

TURNING THE PAGE ON THIS CLASS

NEXT TOPIC

Hopefully, you've got a taste of the challenges / benefits of instrumentation and mediation

- Static cost/benefit
- Runtime cost/benefit

TURNING THE PAGE ON THIS CLASS

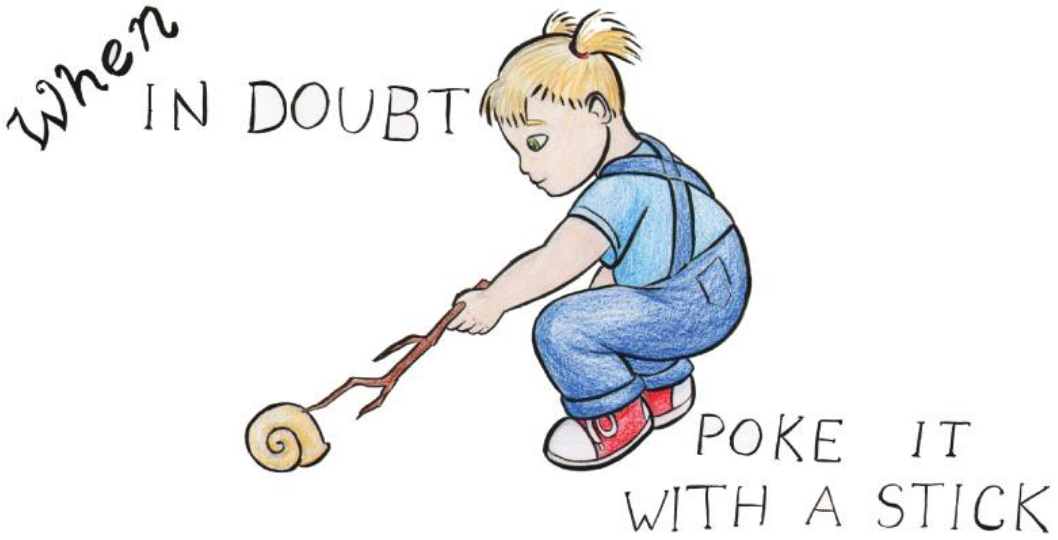
NEXT TOPIC

Dynamic Analysis

- Running the program to see what happens

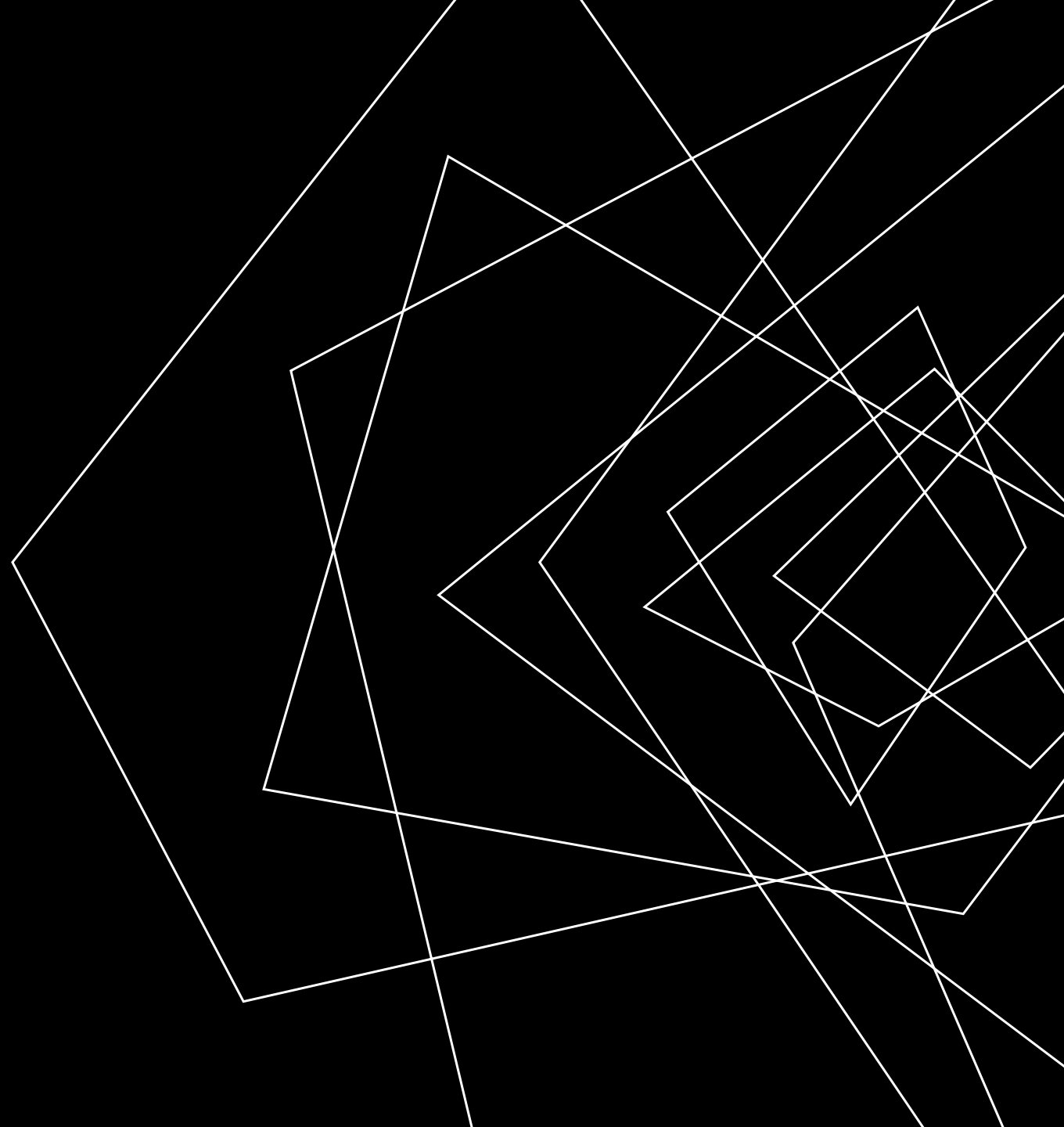
Uses of dynamic analysis

- Program comprehension
- Bug elimination



LECTURE OUTLINE

- The Testing Perspective
- Test Generation
- Fuzzing



A FORMULATION OF DYNAMIC ANALYSIS

TESTING

Input/expected output pairs

- Does the program do what it's supposed to do?

TEST SUITES

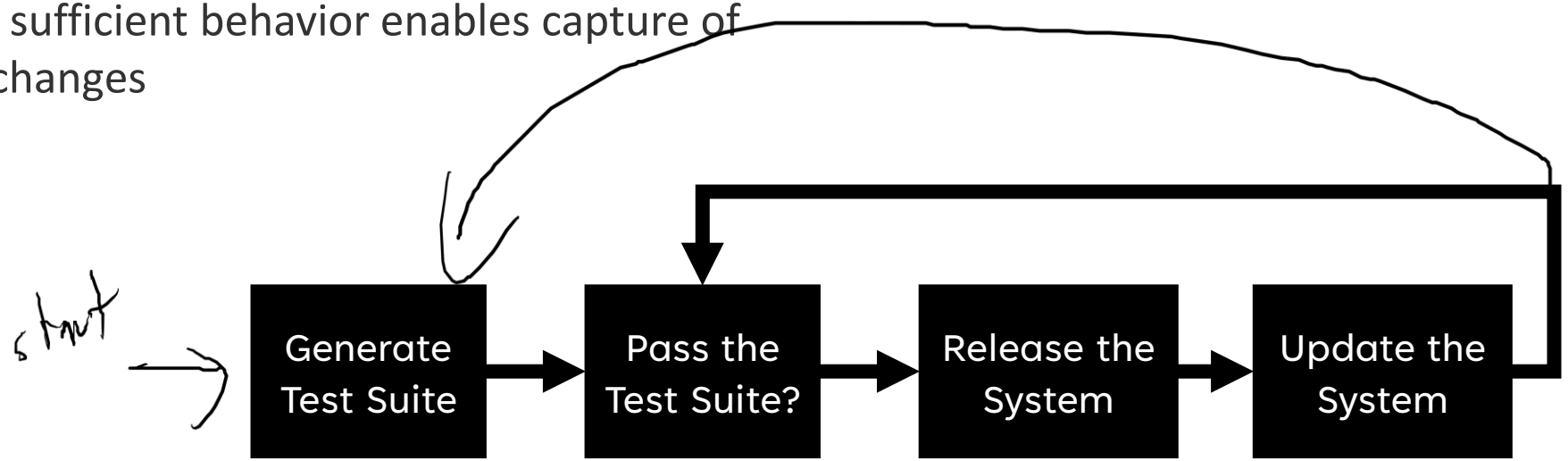
TESTING

Ideally, we'll capture a variety of behaviors

- We'll refer to the collection of test cases as our test suite

Regression suite

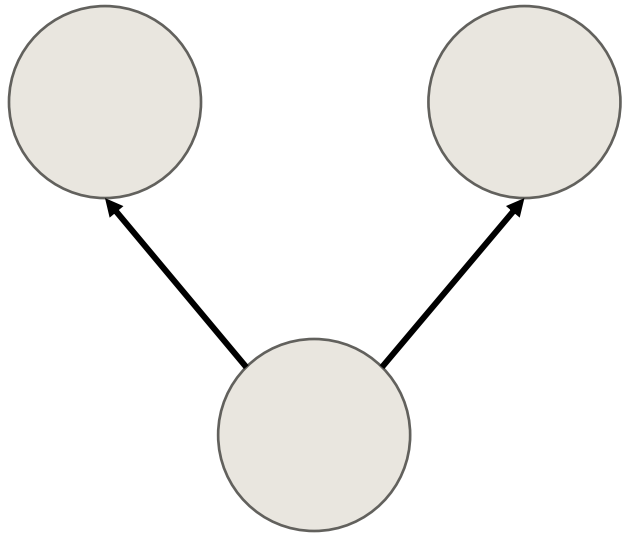
- Capturing sufficient behavior enables capture of breaking changes



NON-DETERMINISM
TESTING

Factor out external details into the environment

- Time is an input
- Random seed is an input
- Network response is an input



TESTING SCOPE

Unit testing

- Testing at the submodule level (e.g. function i/o)

Integration testing

- Testing at the boundary between modules (e.g. library interfaces)

Application testing

- Testing at the whole-program level

PROGRAM VISIBILITY

CATEGORIZING ANALYSIS

White box

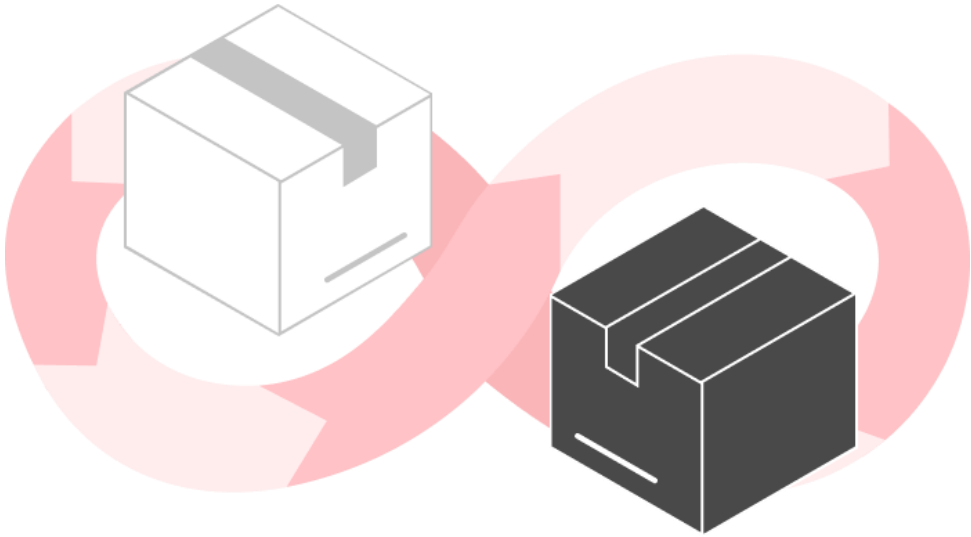
- Testing with “complete” information about the analysis target (typically means source code)

Black box

- Testing with “no” information about how the analysis target is architected (typically means binary only)

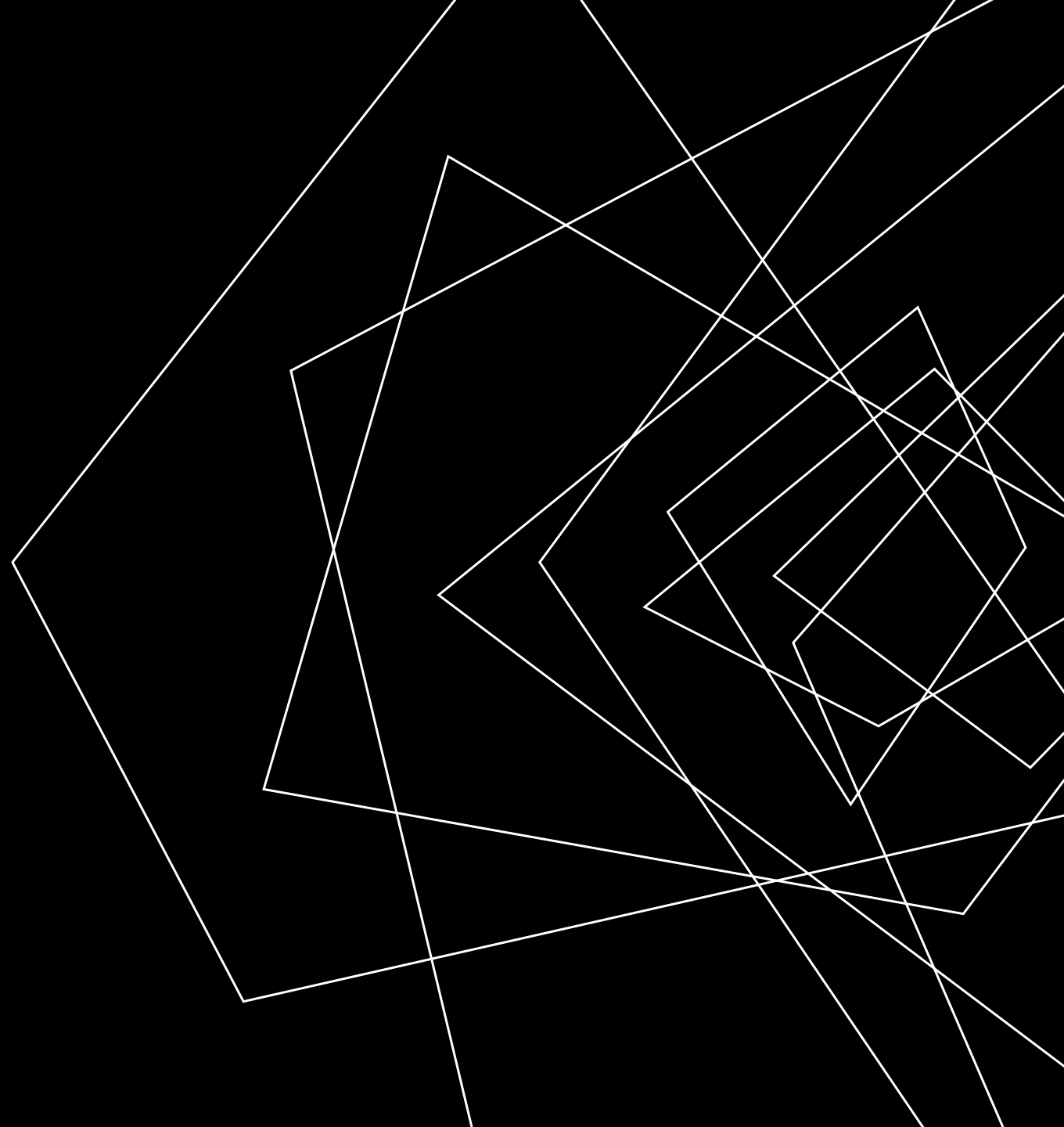
Grey box

- Testing with “some” information about how the analysis target is architected (binary + some static analysis / probing)



LECTURE OUTLINE

- The Testing Perspective
- Test Generation
- Fuzzing



CLASSIC LIMITATIONS OF TESTING

TEST GENERATION

It’s hard to predict what might go wrong (presumably you’d have fixed it in this first place)

“FIXING” TESTING

TEST GENERATION

It’s hard to predict what might go wrong (presumably you’d have fixed it in this first place)

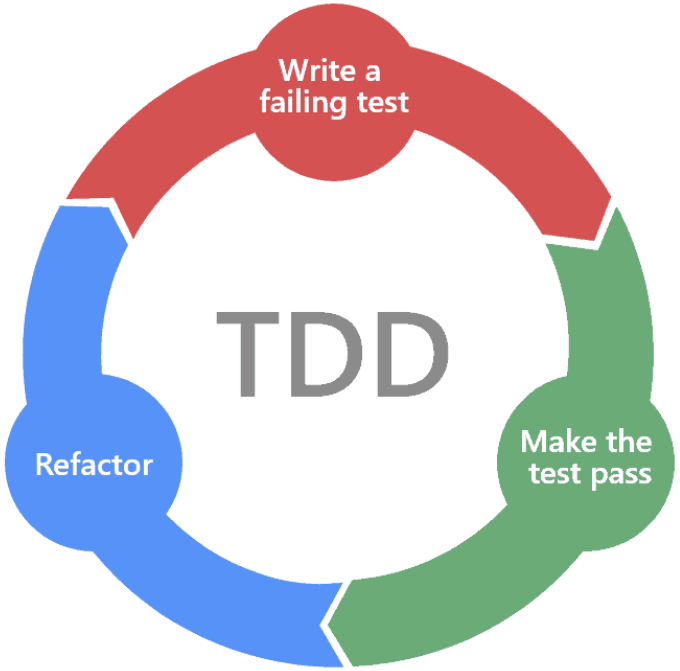
- Could try to make a more intentional correspondence (TDD)
- Could try to leverage tools (Fuzzing)

TEST-DRIVEN DEVELOPMENT

CATEGORIZING ANALYSIS

1. Write a test case (expecting it to fail)
2. Implement enough functionality to pass the test case
3. Fix up the program

(repeat)



TESTING VS STATIC ANALYSIS

TEST GENERATION

A fight (I guess?) in the software engineering community

The clean code blog
"The Dark Path", 1/2017
"Tools are not the Answer", 10/2017

I think that good software tools make it easier to write good software. However, tools are not the answer to the "Apocalypse".

Nowhere in the article did the author examine the possibility that programmers are generally undisciplined.

Ask yourself why we are trying to plug defects with language features. The answer ought to be obvious. We are trying to plug these defects because these defects happen too often.

Now, ask yourself why these defects happen too often. If your answer is that our languages don't prevent them, then I strongly suggest that you quit your job and never think about being a programmer again; because defects are *never* the fault of our languages. Defects are the fault of *programmers*. It is *programmers* who create defects – not languages.

And what is it that programmers are supposed to do to prevent defects? I'll give you one guess. Here are some hints. It's a verb. It starts with a "T". Yeah. You got it. *TEST!*

SOME DIFFICULTIES OF UNIT TESTING

TEST GENERATION

Integrating testing into a workflow

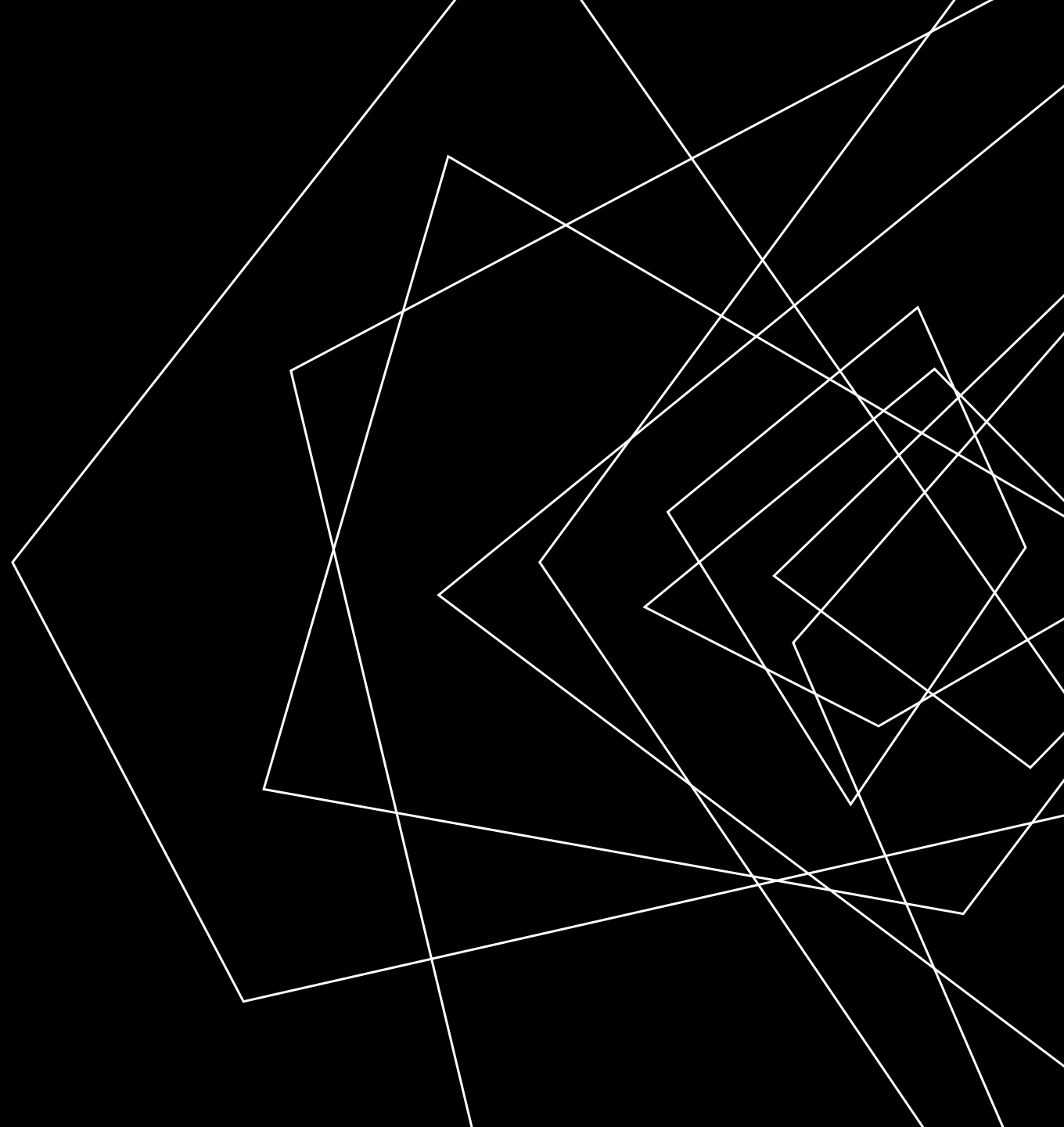
googletest

apt install googletest
apt install libgtest-dev

What to do about a function's "environment"?

LECTURE OUTLINE

- The Testing Perspective
- Test Generation
- Fuzzing



FUZZING

CATEGORIZING ANALYSIS

GENERATING GOOD TEST CASES

Cases that increase coverage of program behaviors

Cases that exercise unexpected behavior

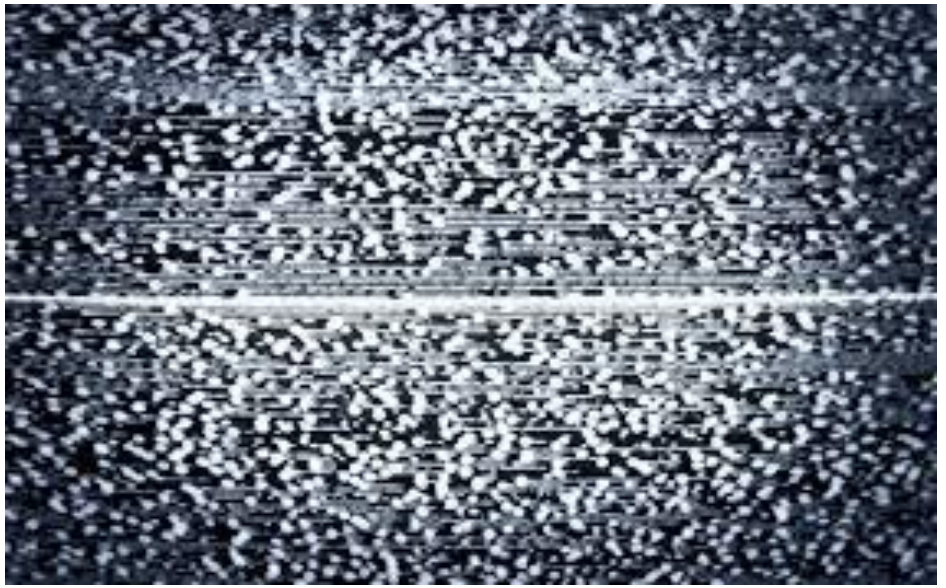
PREVIOUS STABS AT THIS TOPIC

Consider testing as an intrinsic part of the SSDLC methodology

Test-driven development

Post-hoc evaluation via coverage metrics

TODAY: JUST GUESS



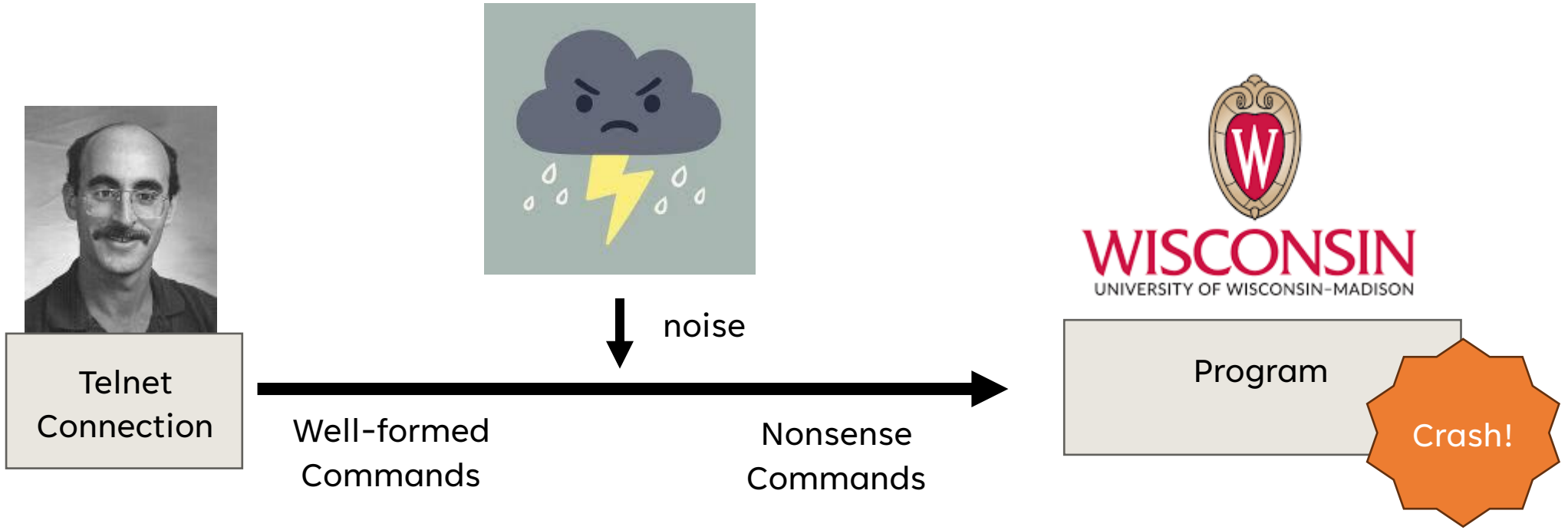
The random “fuzz” of white noise

HISTORY OF FUZZING

FUZZING

1988: It was a Dark and Stormy Night

Professor Bart Miller attempts to work from home...



BREAKING CIRCULAR LOGIC

FUZZING

AUTOMATED TEST CASE GENERATION RESOLVES A FUNDAMENTAL CONFLICT IN TESTING...

Tautologically impossible to predict unpredictable behavior

Apply a technique that obviated the need for expectations

GRACEFUL FAILURE

FUZZING

Any error should be anticipated and handled by the system, with an informative error message should recovery become impossible

A KEY PRINCIPLE IN THE VALIDITY OF FUZZING

“The user should never see a seg fault”

EXPLORING UNEXPECTED BEHAVIOR

FUZZING

Random input is surprisingly effective

Numerous bugs found in practice via fuzzing...

Busybox utilities

Windows bugs

Linux Kernel bugs

Benefits of fuzzing

Very easy to run

Instant results

Highly scalable

THE SIMPLEST FUZZER

FUZZ TESTING

The Most Basic Form of Fuzzing

```
cat /dev/random | program
```

A study in the 90s basically did this, finding bugs in...
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

PRIORITIZING INPUT

FUZZING

THE CHALLENGE OF FUZZERS IS (USUALLY) GETTING PAST THE FIRST VALIDATION CHECK

```
if (!sane_input()){
  exit 1;
}
//The rest of the program
```

SIMPLE TESTING STRATEGY

FUZZING

CONSIDER "INTERESTING" INPUT

Values close to the maximum, minimum, middle, etc

CASE STUDY: CARD READER INPUT: [FRISBY ET AL., 2012]

MUTATION-BASED FUZZERS

EXPLORE DEVIATIONS FROM KNOWN INPUT

Example mutations:

Binary input

– Bit flips

- Byte flips
- Change random bytes
- Insert random byte chunks
- Delete random byte chunks
- Set randomly chosen byte chunks to interesting values e.g. INT_MAX, INT_MIN, 0, 1, -1, ... §

Text input

- Insert random symbols or keywords from a dictionary

BLACK-BOX FUZZING

THROW RANDOM INPUT AT THE APPLICATION INTERFACE

REPRESENTATIVE TOOL: ZZUF

BLACK-BOX FUZZING

THE "MULTIPURPOSE FUZZER"

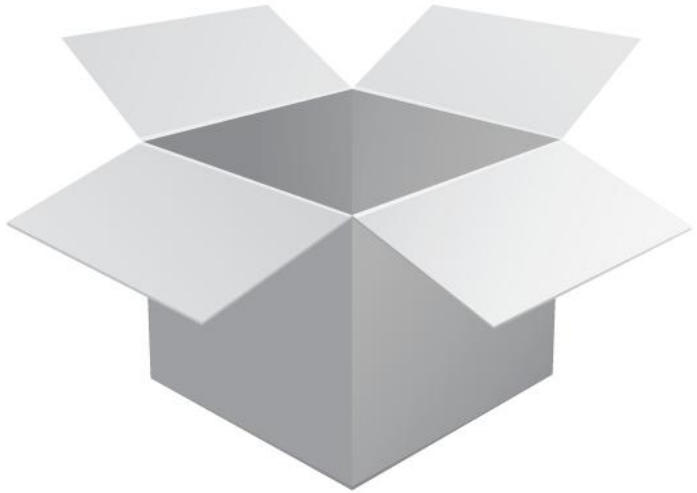
zzuf cat /dev/zero | hd -vn 32

zzuf -r 0.05 hd -vn 32 /dev/zero

WHITE-BOX FUZZING

FUZZING

THROW RANDOM INPUT AT THE UNIT INTERFACE



REPRESENTATIVE TOOL: AFL

FUZZING

AFL (American Fuzzy Lop)

Maintained by Google

State of the art

Generally considered the best, state-of-the-art fuzzer

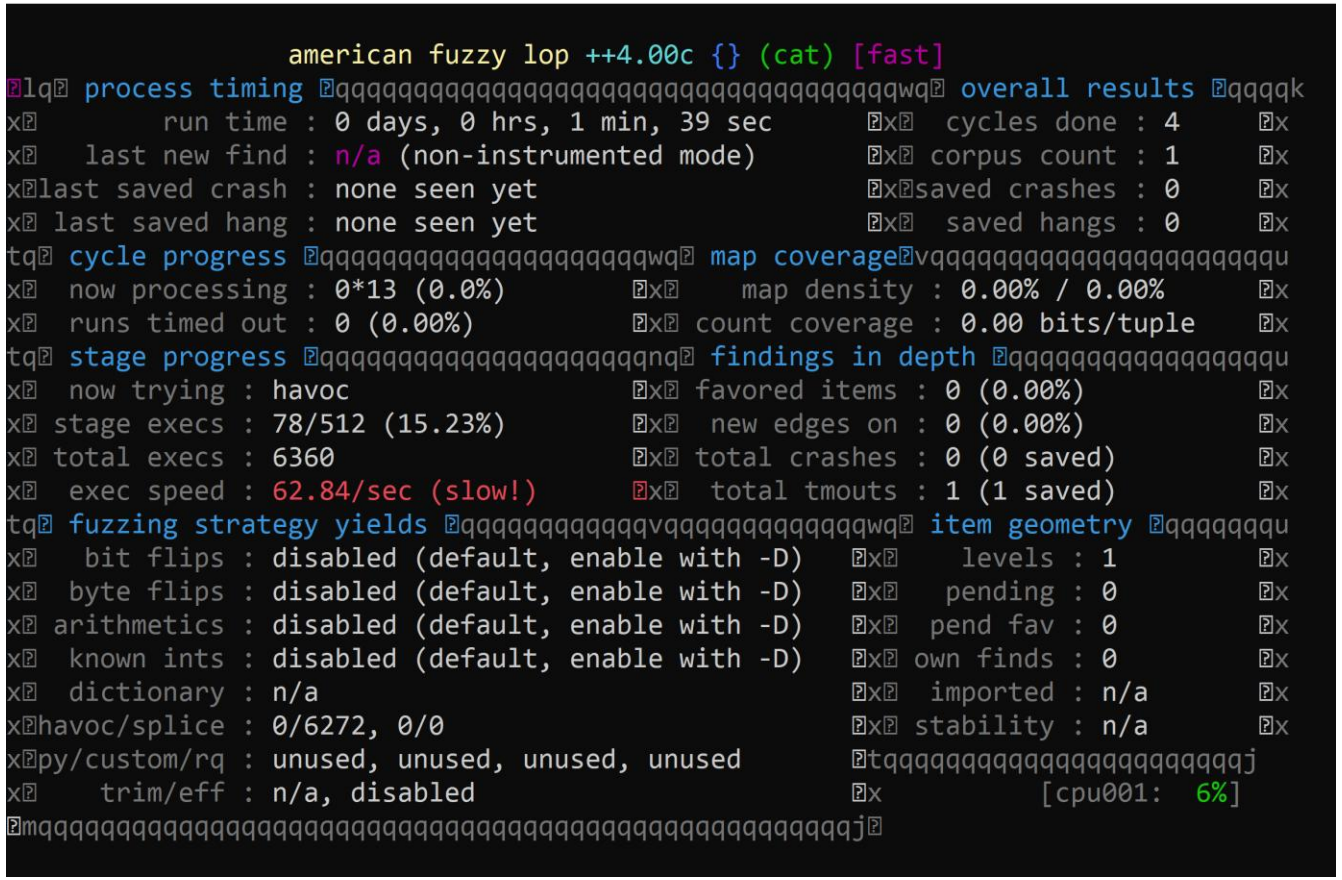
REPRESENTATIVE TOOL: AFL

FUZZING

EXAMPLE COMMAND

"TRADITIONAL FUZZING"

```
mkdir in_dir
echo "hello" > in_dir/hello
afl-fuzz -n -i in_dir -o out_dir cat
```



REPRESENTATIVE TOOL: AFL

FUZZING

INSTRUMENTATION MODE

1) Compile the program with coverage probes
2) Attempt to prioritize / mutate test cases that extend coverage

```
afl-clang++ <build command>
```

GENERATION-BASED FUZZING

Attempt to discern patterns / Alternatives in Input

Potentially with the help of some guide of guide / input grammar

FUZZING ORACLES

Beyond graceful Failure

In C/C++ there are a lot of violations of proper behavior that are invisible

“Seems fine until it’s a huge problem”

Sanitizers

UBSan – Undefined behavior sanitizer
ASan – Address sanitizer
TSan – Thread sanitizer

RESEARCH DIRECTION: “GUNKING”

FUZZING

Fuzzing as adversarial recon

Fuzzing is so good at finding bugs that even the bad guys do it

Perhaps a program should deploy Anti-Fuzzing tech

What would that look like?