

EXERCISE #28

LLVM INSTRUMENTATION REVIEW

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

Describe the difference between the profile-instr-generate and profile-generate options for LLVM instrumentation?

Free exercises from last week

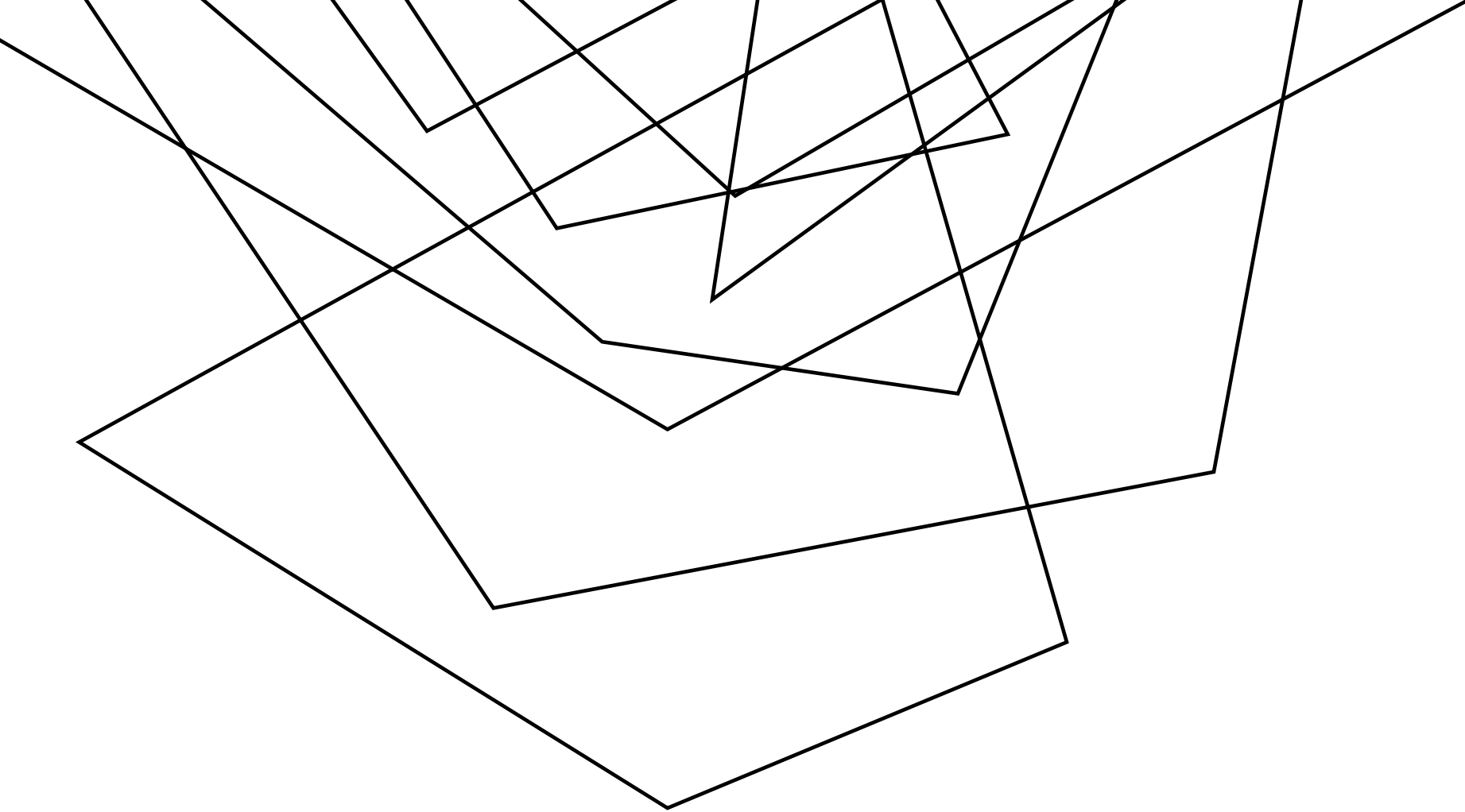
W4 - Review

Ball & Larus

Efficient Path Profiling

Due 11:59:59 PM tonight

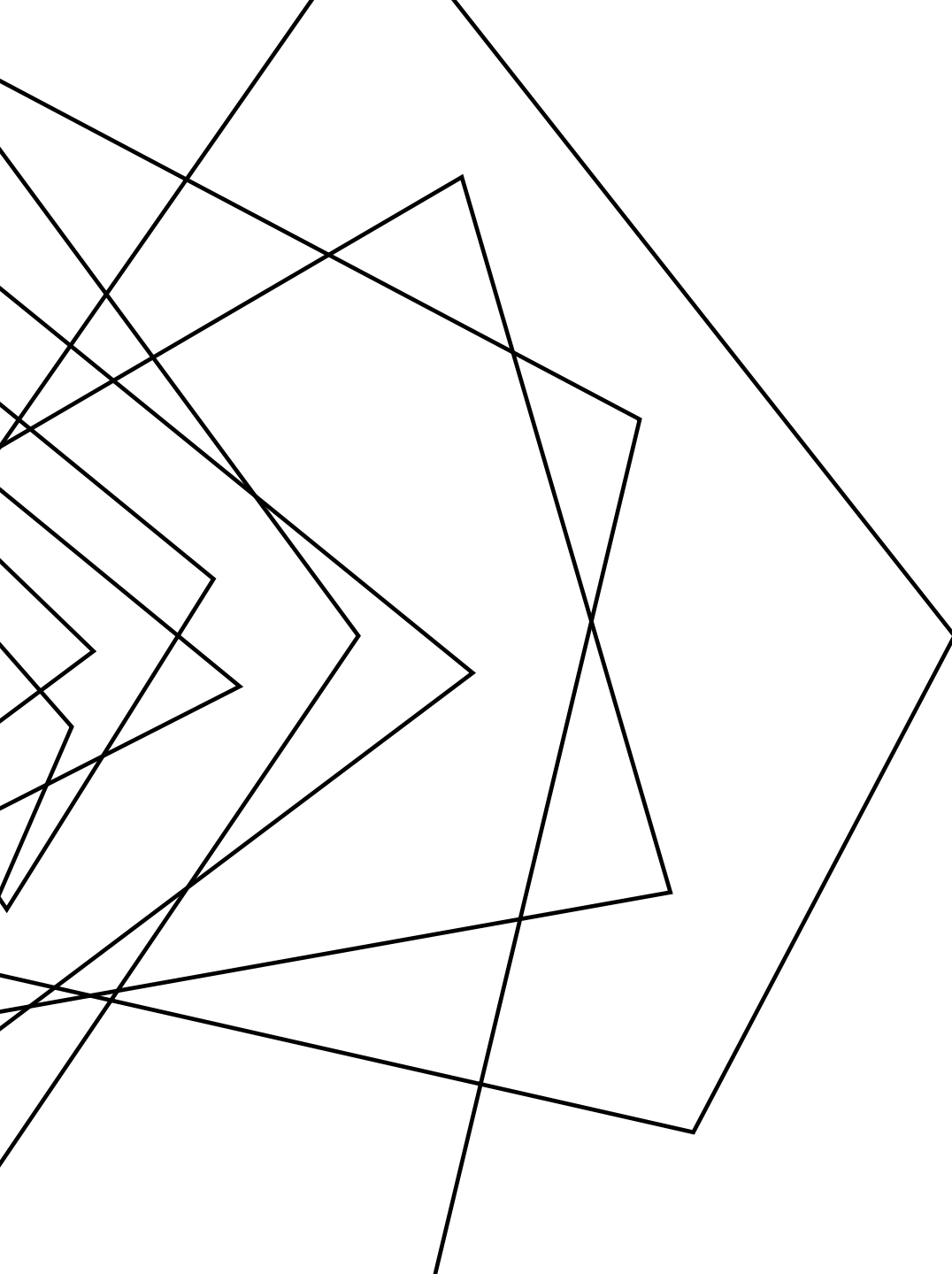
**ADMINISTRIVIA
AND
ANNOUNCEMENTS**



FUZZING

EECS 677: Software Security Evaluation

Drew Davidson



WHERE WE'RE AT

Analysis

DYNAMIC ~~INSTRUMENTATION~~

Use the execution of a program to find
(security) bugs

Necessarily dependent on encountered
execution behavior

PREVIOUSLY: LLVM INSTRUMENTATION

REVIEW: LAST LECTURE

USAGE OF LLVM BUILT-IN INSTRUMENTATION ANALYSIS

Described commands to use PGO for line coverage analysis

SETUP FOR A CUSTOM LLVM ANALYSIS

Described the basic infrastructure necessary to craft a custom instrumentation



THIS LESSON: FUZZING

OUTLINE / OVERVIEW

```
int main(int argc) {  
    return 1/(1-argc);  
}
```

GENERATING GOOD TEST CASES

Cases that increase coverage of program behaviors

Cases that exercise unexpected behavior

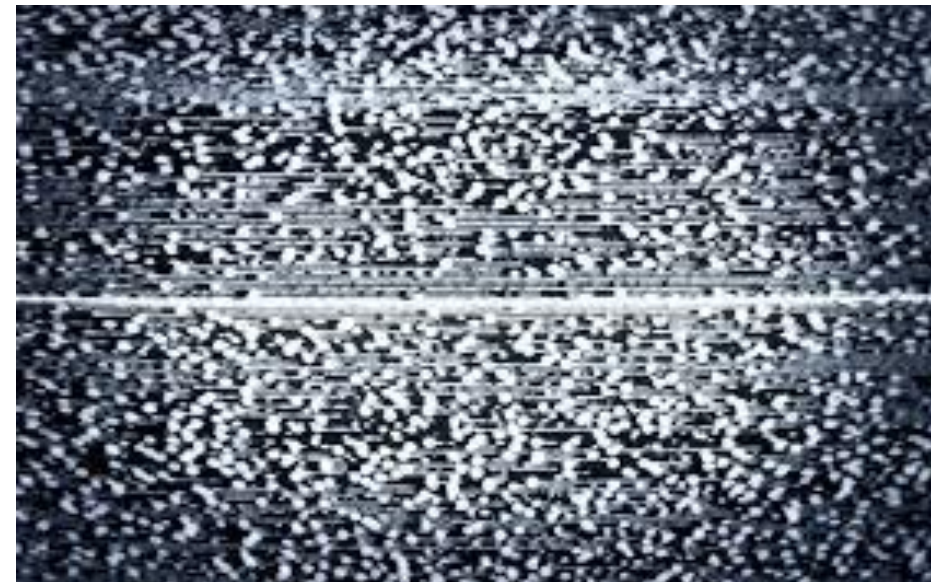
PREVIOUS STABS AT THIS TOPIC

Consider testing as an intrinsic part of the SSDLC methodology

Test-driven development

Post-hoc evaluation via coverage metrics

TODAY: JUST GUESS



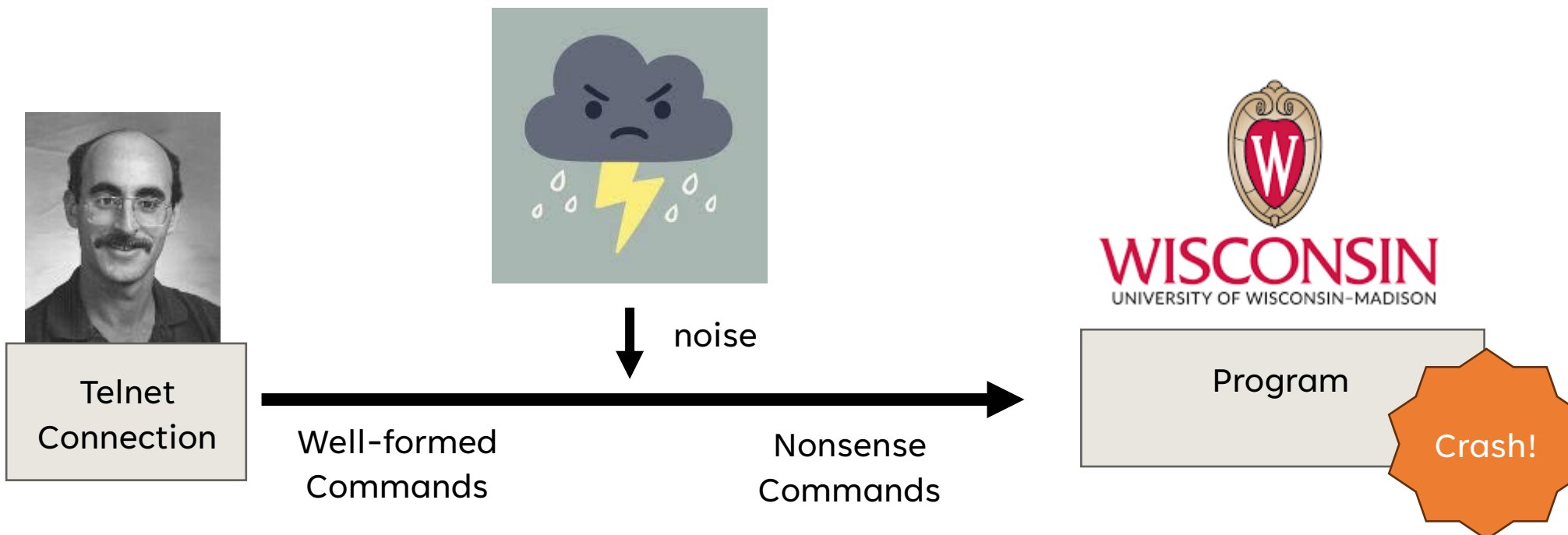
The random “fuzz” of white noise

HISTORY OF FUZZING

OUTLINE / OVERVIEW

1988: IT WAS A DARK AND STORMY NIGHT

Professor Bart Miller attempts to work from home...



BREAKING CIRCULAR LOGIC

OUTLINE / OVERVIEW

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



because circular reasoning works

GRACEFUL FAILURE

OUTLINE / OVERVIEW

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”



THE SIMPLEST FUZZER

FUZZ TESTING

THE MOST BASIC FORM OF FUZZING

1 run of the fuzzer

`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

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



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

FUZZING

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

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

OUTLINE / OVERVIEW

EXAMPLE COMMAND

“TRADITIONAL FUZZING”

```
mkdir in_dir
```

```
echo "hello" > in_dir/hello
```

```
afl-fuzz -n -i in_dir -o out_dir cat
```

*↑
traditional
fuzzing*

```
american fuzzy lop ++4.00c {} (cat) [fast]
lq process timing overall results
x run time : 0 days, 0 hrs, 1 min, 39 sec      cycles done : 4
x last new find : n/a (non-instrumented mode)   corpus count : 1
x last saved crash : none seen yet             saved crashes : 0
x last saved hang : none seen yet              saved hangs : 0
tq cycle progress map coverage
x now processing : 0*13 (0.0%)                  map density : 0.00% / 0.00%
x runs timed out : 0 (0.00%)                   count coverage : 0.00 bits/tuple
tq stage progress findings in depth
x now trying : havoc                            favored items : 0 (0.00%)
x stage execs : 78/512 (15.23%)                 new edges on : 0 (0.00%)
x total execs : 6360                             total crashes : 0 (0 saved)
x exec speed : 62.84/sec (slow!)                total tmouts : 1 (1 saved)
tq fuzzing strategy yields item geometry
x bit flips : disabled (default, enable with -D) levels : 1
x byte flips : disabled (default, enable with -D) pending : 0
x arithmetics : disabled (default, enable with -D) pend fav : 0
x known ints : disabled (default, enable with -D) own finds : 0
x dictionary : n/a                               imported : n/a
x havoc/splice : 0/6272, 0/0                      stability : n/a
x py/custom/rq : unused, unused, unused, unused  [cpu001: 6%]
x trim/eff : n/a, disabled
mq
```


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>
```

↑ command-line-compatible with clang++

FUZZING ORACLES

FUZZING

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”

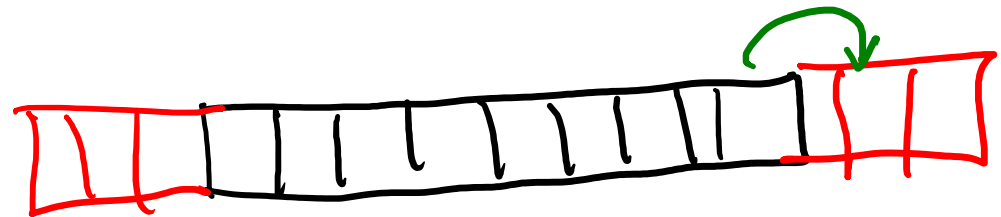
SANITIZERS

UBSan – Undefined behavior sanitizer

ASan – Address sanitizer

TSan – Thread sanitizer

- `fsanitize = address`
- `fsanitize = bounds-strict`



RESEARCH DIRECTION: “GUNKING”

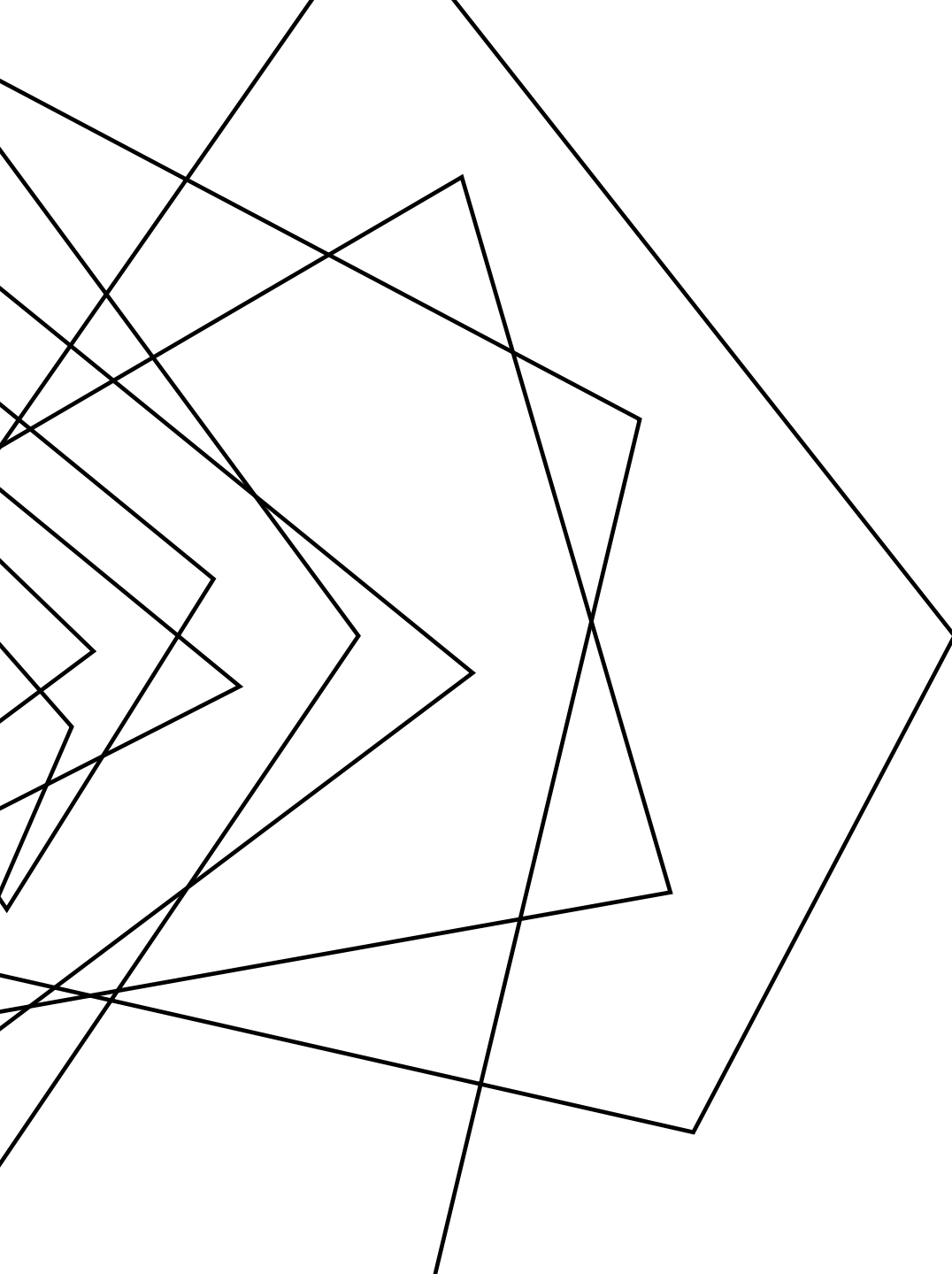
FUZZING

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?



WRAP-UP

INTRODUCED THE CONCEPT AND THE
“INDUSTRY STANDARD” TOOL OF FUZZING

A simple, elegant idea