EXERCISE 24

LLVM INSTRUMENTATION REVIEW

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

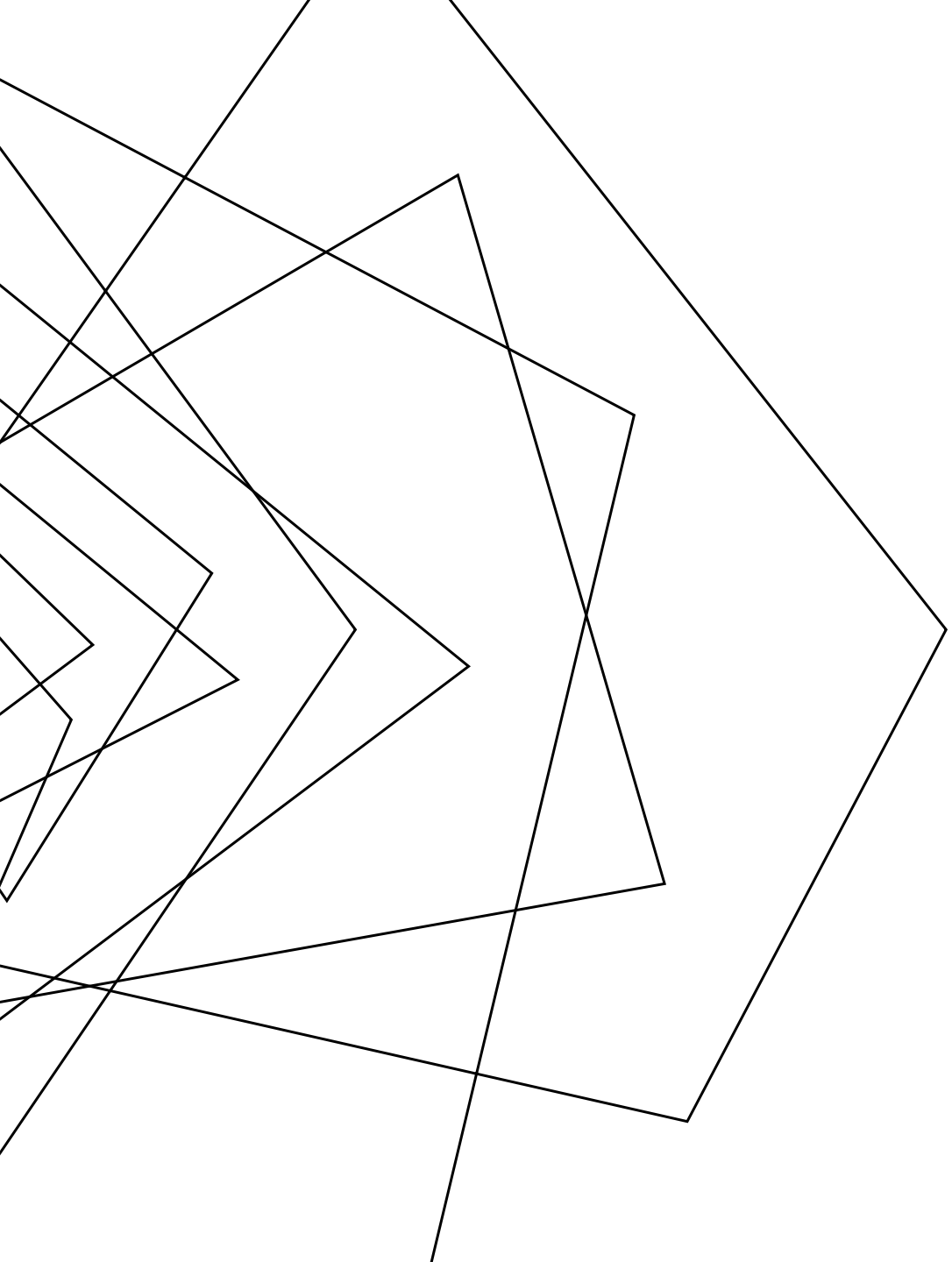
By default, opt creates a binary-coded machine code output (<file>.bc). How is this file translated back to a human-readble file (<file>.ll) ?

EXERCISE 24 SOLUTION

LLVM INSTRUMENTATION REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS

Paper review due Sunday at 11:59 PM



CLASS PROGRESS

SHOWING SOME APPLICATIONS OF STATIC DATAFLOW

- DESCRIBED A PARTICULAR TYPE OF EVASION AGAINST EXPLICIT DATAFLOW: SIDE CHANNELS
- BEGAN TO CONSIDER WHAT WE COULD DO ABOUT IT

LAST TIME: LLVM INSTRUMENTATION

REVIEW: LAST LECTURE

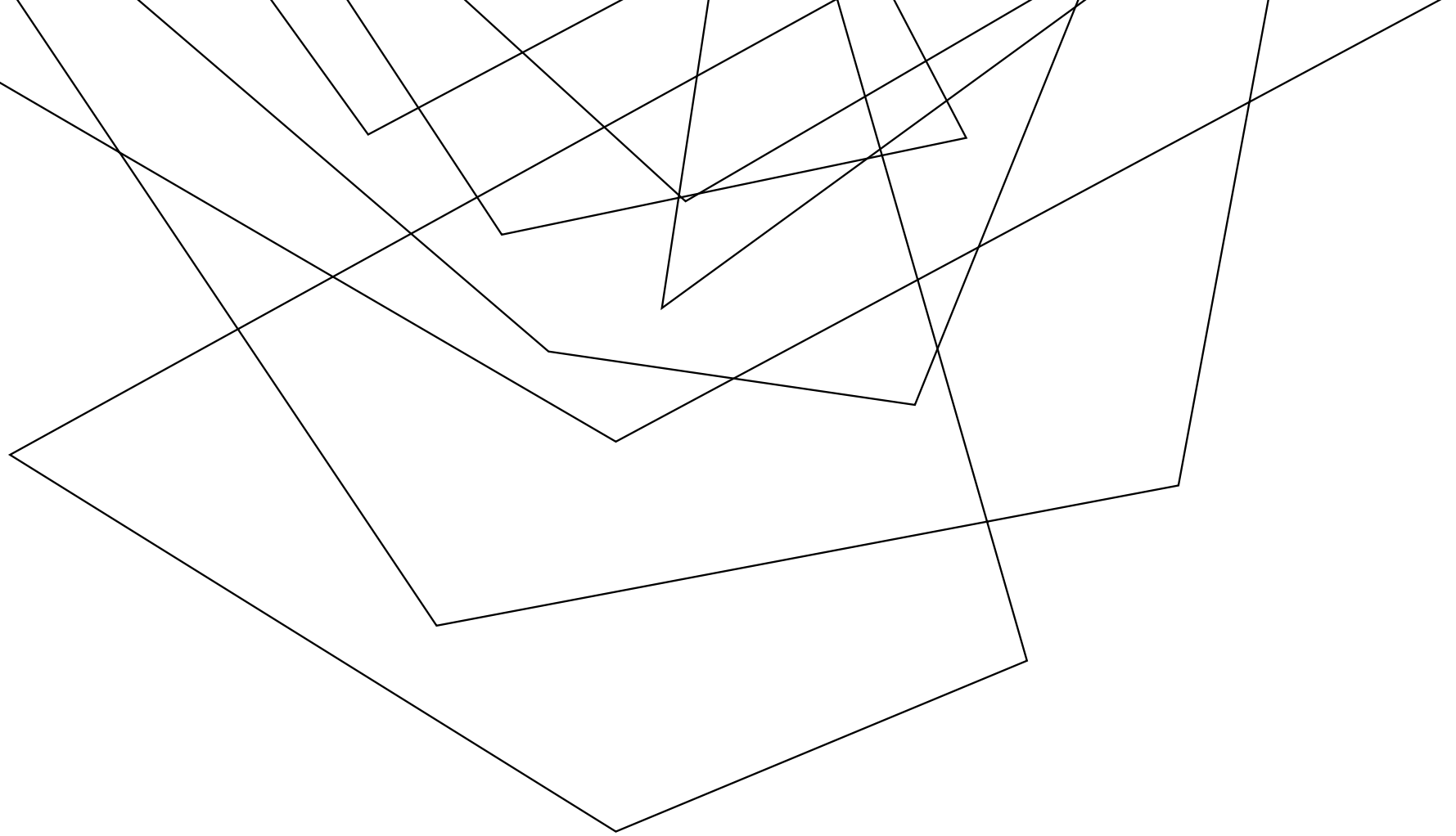
SHOWED THE CONCRETE STEPS TO USING LLVM TO INJECT MEASUREMENT

Example: Inserted printf() calls before every binary operation

Achievable via dynamically loading a .so into llvm...

- via the optimizer (opt –load-pass-plugin)
- via the compiler frontend (clang –fpass-plugin)

A new way of interacting with LLVM: as a library/framework



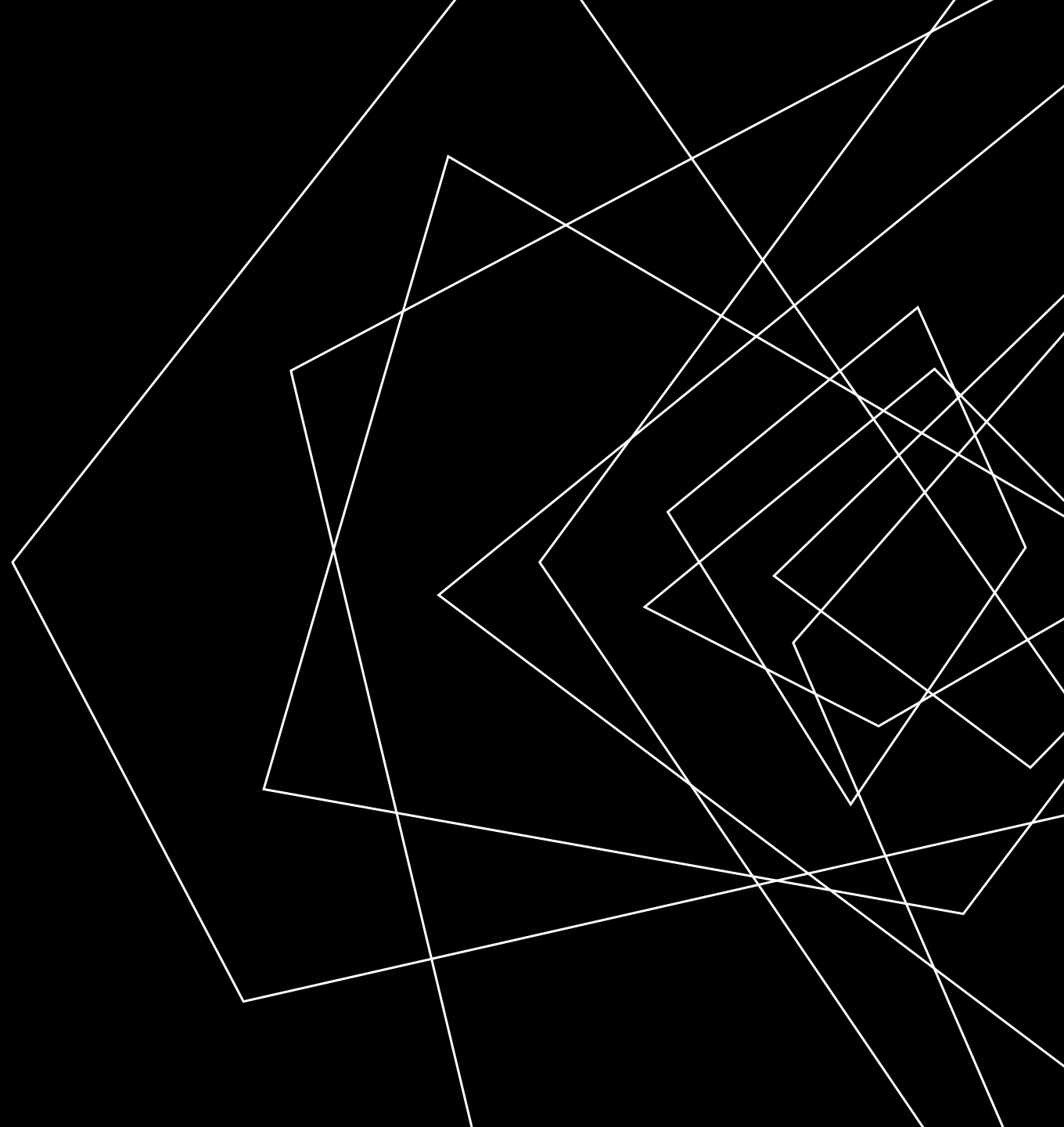
REFERENCE MONITORS

EECS 677: Software Security Evaluation

Drew Davidson

LECTURE OUTLINE

- Overview
- Details
- Instances



BEYOND PASSIVE ANALYSIS

REFERENCE MONITORS: OVERVIEW

So far, our focus has been largely on detecting undesirable behavior

- That's valuable!
  - Ask developers to correct their own mistakes
  - Empower users to forgo running bad software

LIMITATIONS OF ANALYSIS

REFERENCE MONITORS: OVERVIEW

Detection might not be enough

- May be in a position where we can't run the analysis

Static Analysis

- False positives
- Scalability issues

Dynamic Analysis

- False negatives
- Run time issues

A HANDS-ON ALTERNATIVE

REFERENCE MONITORS: OVERVIEW

KEEP BAD THINGS FROM HAPPENING DURING SYSTEM EXECUTION

- Requires some sort of specification for “bad things”
- Requires some sort of preventative capabilities

PREVENTATIVE CAPABILITIES

REFERENCE MONITORS: OVERVIEW

Simple form

Kill the program

Dataflow Form

Sanitize the data

THE BIG IDEA

REFERENCE MONITORS: OVERVIEW

KEEP PROGRAMS ON THE "STRAIGHT AND NARROW"

- Articulate a policy for allowed behavior
- Keep a running record of security-relevant behavior
- Prevent a violation of the policy

SAFETY POLICIES

REFERENCE MONITORS: INSTANCES

Execution of a process as a sequence of states

Policy is a predicate on sequence prefix

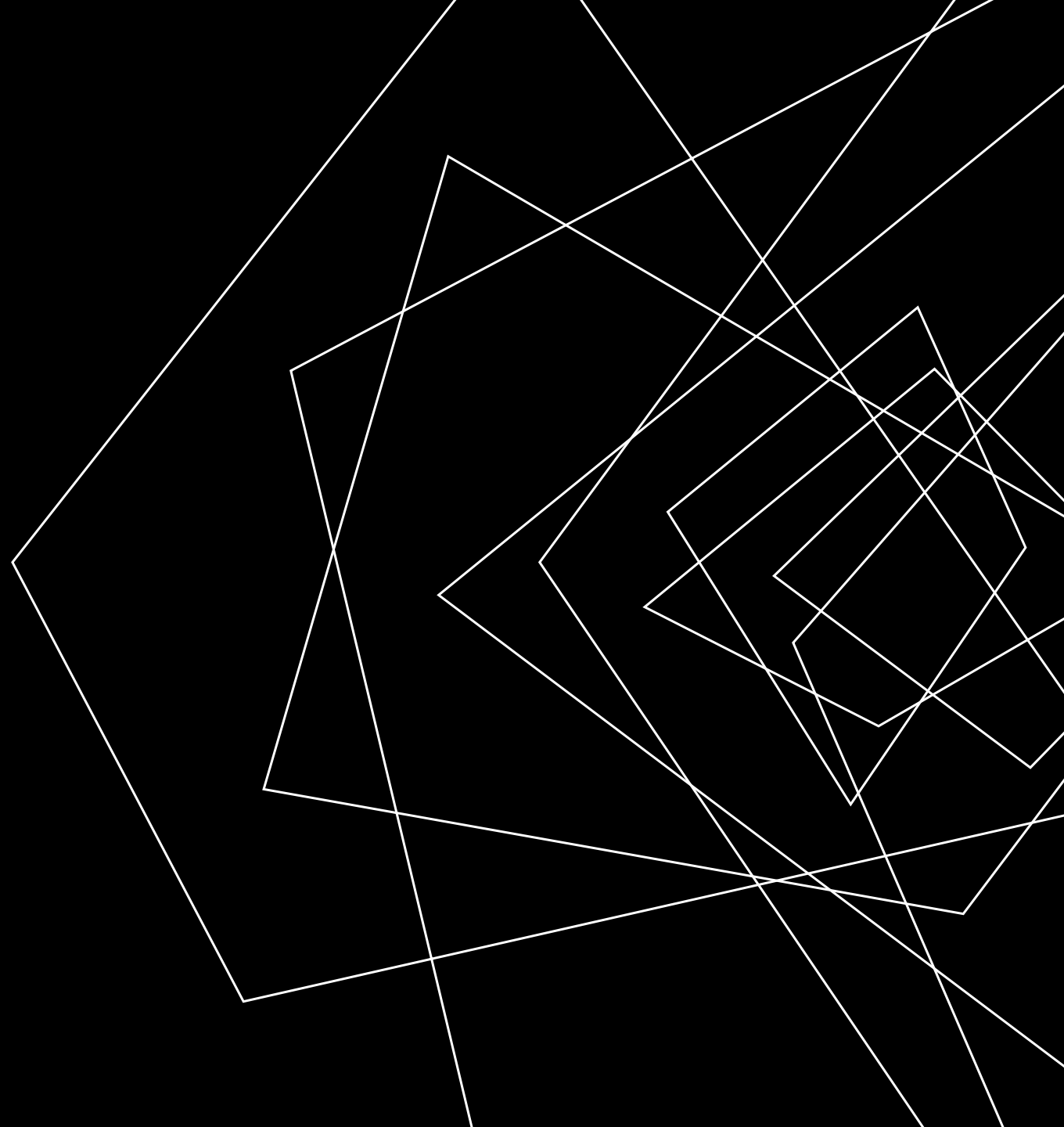
Policy depends only on the past of a particular execution – once violated, never “unviolates”

Incapable of handling liveness policies

“If this server accepts a SYN, it will eventually send a response”

LECTURE OUTLINE

- Overview
- Details
- Instances



CONSIDER THE REACTIVE ADVERSARY

REFERENCE MONITORS: OVERVIEW

DEFINITION

Reactive Adversary: An adversary with the capability to understand the defense mechanism and an opportunity to avoid it

IF A DEFENSE CAN BE AVOIDED, IT HARDLY MATTERS WHAT THE ENFORCEMENT DOES

Recall the history of the Maginot Line

SECURITY VS PRECISION

REFERENCE MONITORS: OVERVIEW

PROGRAM PROXIMITY

Inline reference monitor

External reference monitor

REFERENCE MONITOR DESIGN

REFERENCE MONITORS: INSTANCES

KERNELIZED

Baked into the kernel
Coarse-grained
Secure / hard to subvert

WRAPPER

Specialized execution environment

INLINE

Rewrite the program / hook syscalls
Precise
No special privileges (easier to subvert)

PROPERTIES WE CARE ABOUT

REFERENCE MONITORS: INSTANCES

Memory Safety

e.g. Programs respect aggregate type sizes, process boundaries, code v data

Type Safety

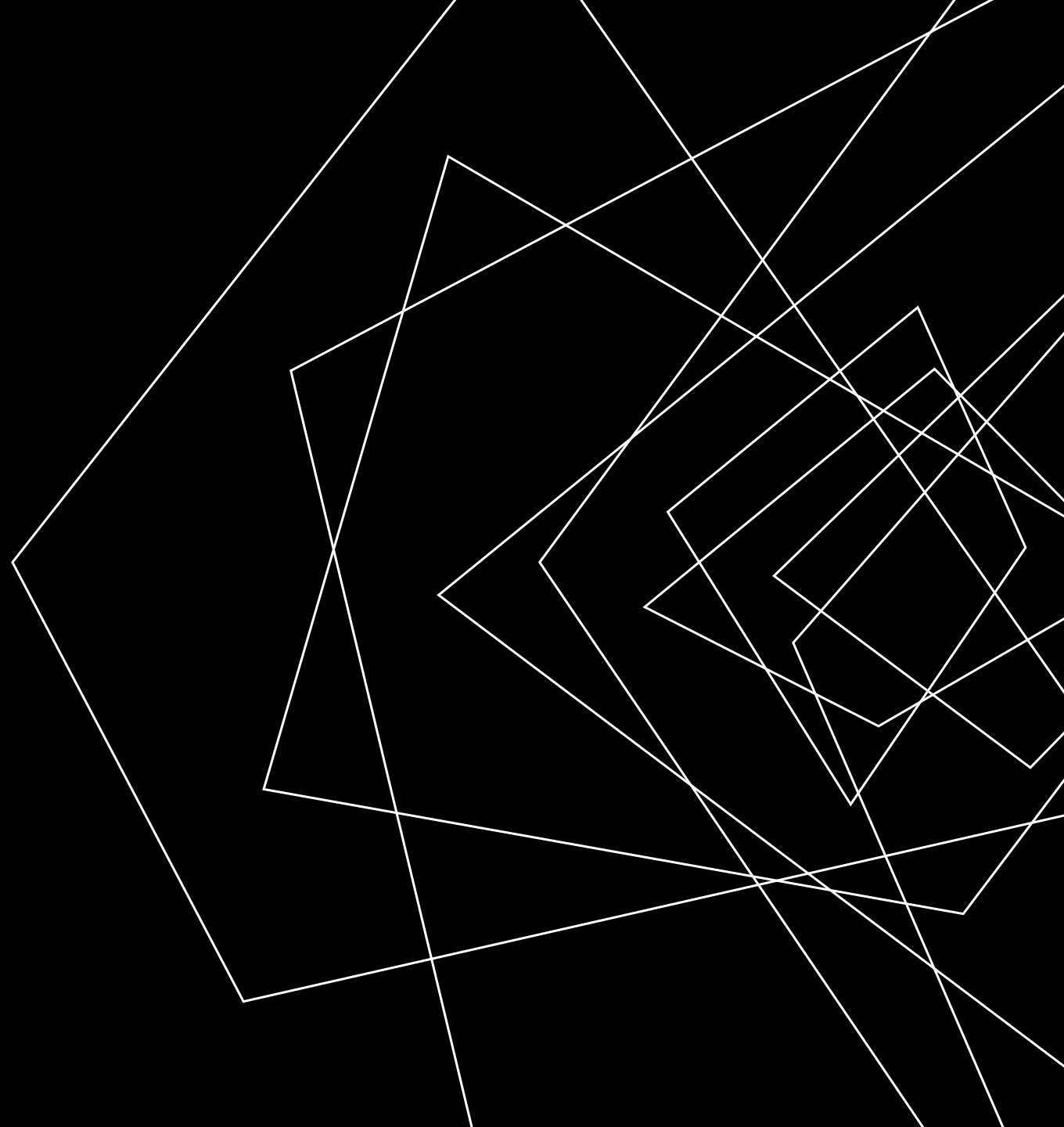
e.g. Functions and intrinsic operations have arguments that adhere to the type system

Control Flow Safety

e.g. All control transfers are envisioned by the original program

LECTURE OUTLINE

- Overview
- Details
- Instances



KERNALIZED REFERENCE MONITOR

REFERENCE MONITORS: INSTANCES

SEMANTIC ABSTRACTION:

Collection of running processes and files

Processes are associated with users

Files have ACLs

OS ENFORCES VARIOUS SAFETY POLICIES

- File access
- Process space write

Simplest case: same policy for all processes of the same user

EXAMPLE OS-LEVEL REFERENCE MONITORS

REFERENCE MONITORS: INSTANCES

AppArmor

Capability-based, per-program policies
Restricts file access and system calls

SELinux

EXAMPLE

```
deny @{HOME}/Documents/ rw,
deny @{HOME}/Private/ rw,
deny @{HOME}/Pictures/ rw,
deny @{HOME}/Videos/ rw,
deny @{HOME}/fake/ rw,
deny @{HOME}/.config/ rw,
deny @{HOME}/.ssh/ rw,
deny @{HOME}/.bashrc rw,
```

WRAPPER-LEVEL REFERENCE MONITOR

REFERENCE MONITORS: INSTANCES

JAVA SECURITY MANAGER

Each process is a logical fault domain

Ensure all memory references and jump is within the process fault domain

```
java Program -Djava.security.manager -Djava.security.policy==~/Program.policy
```

INLINE REFERENCE MONITORS: SASI

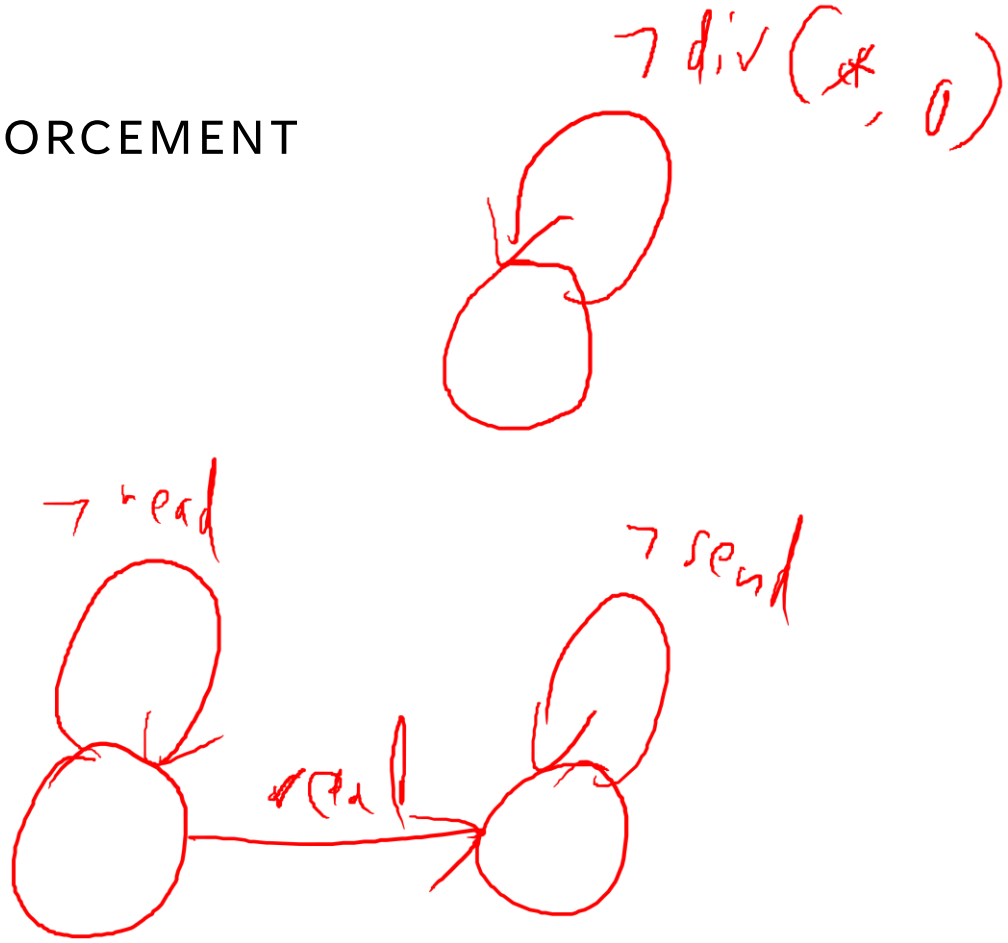
REFERENCE MONITORS: INSTANCES

CORNELL PROJECT FOR INLINE POLICY ENFORCEMENT

Change the program to enforce “any” safety policy

Express allowed behavior as an FSM

Examples:
- No division by zero
- No network send after file read



SASI: COST

REFERENCE MONITORS: INSTANCES

ATTEMPTS TO MINIMIZE THE NUMBER OF CHECKS

Looking at every instruction is incredibly expensive

Example: only need to check divide-by-zero
before DIV instructions

CONSTRUCTING AN IRM

REFERENCE MONITORS: INSTANCES

LLVM-BASED INSTRUMENTATION

Assume source code (or at least IR availability)

Inject enforcement instructions at appropriate points

LEVERAGING STATIC ANALYSIS

Only inject checks where there is the possibility of failure

```
1: int main(int argc){
2:     if (argc > 0){
3:         return 5 / argc;
4:     }
5: }
```

SUMMARY
REFERENCE MONITORS

Reference monitor Intuition (from our perspective)

Dynamic program analyses that take action to alter the semantics of the program due to a safety policy violation

Explores the semantic gap tradeoff: being close to the target may add specificity, but may make the enforcement attackable

NEXT TIME: CFI

REFERENCE MONITORS: INSTANCES

USE IRM TO DETERMINE IF CODE VIOLATES ITS SUPERGRAPH

Why would we need to do this?

WRAP-UP