EXERCISE 7

MEMORY ATTACK REVIEW

Is return-oriented programming impacted by the W⊗X protection? Why or why not?

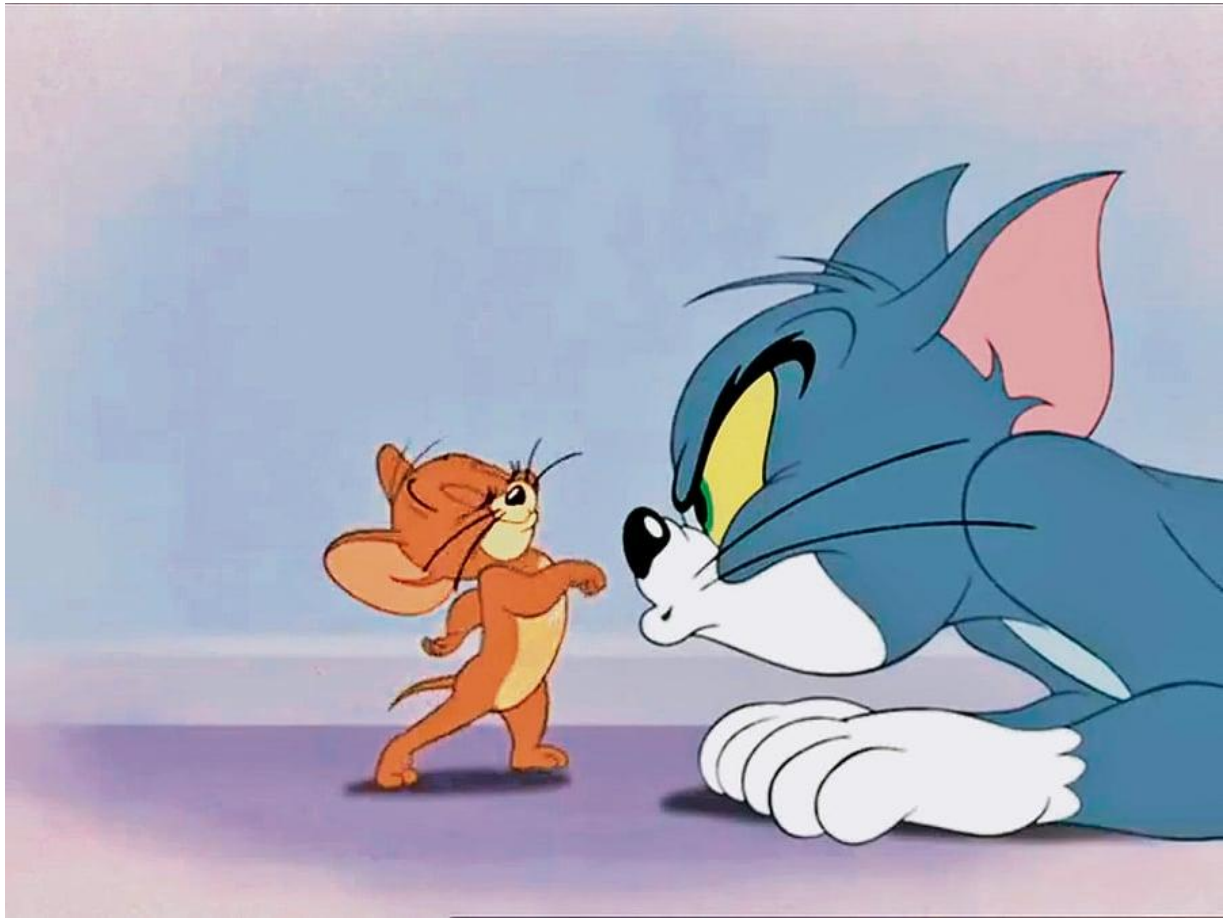
EXERCISE 7 SOLUTION

MEMORY ATTACK REVIEW

LAST TIME

MEMORY REVIEW

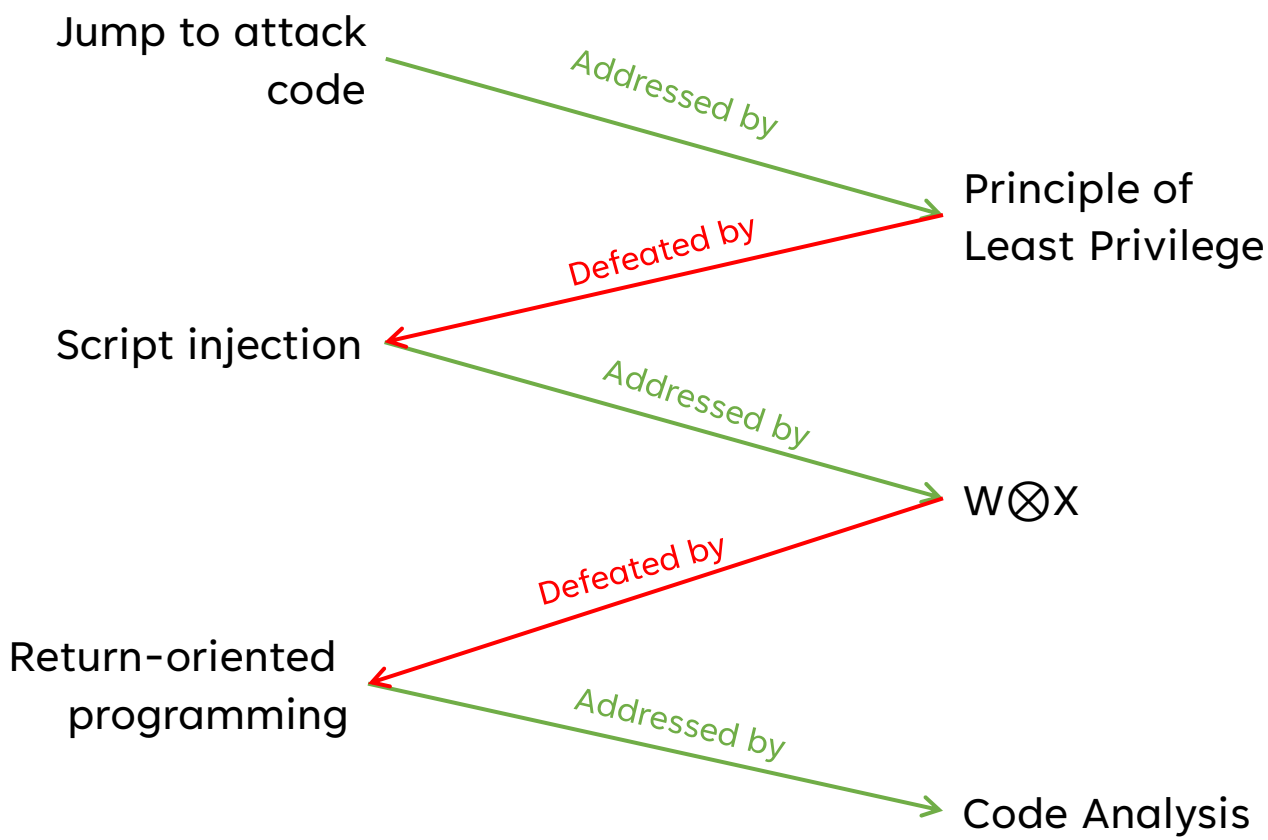
The game of cat-and-mouse for remote code execution



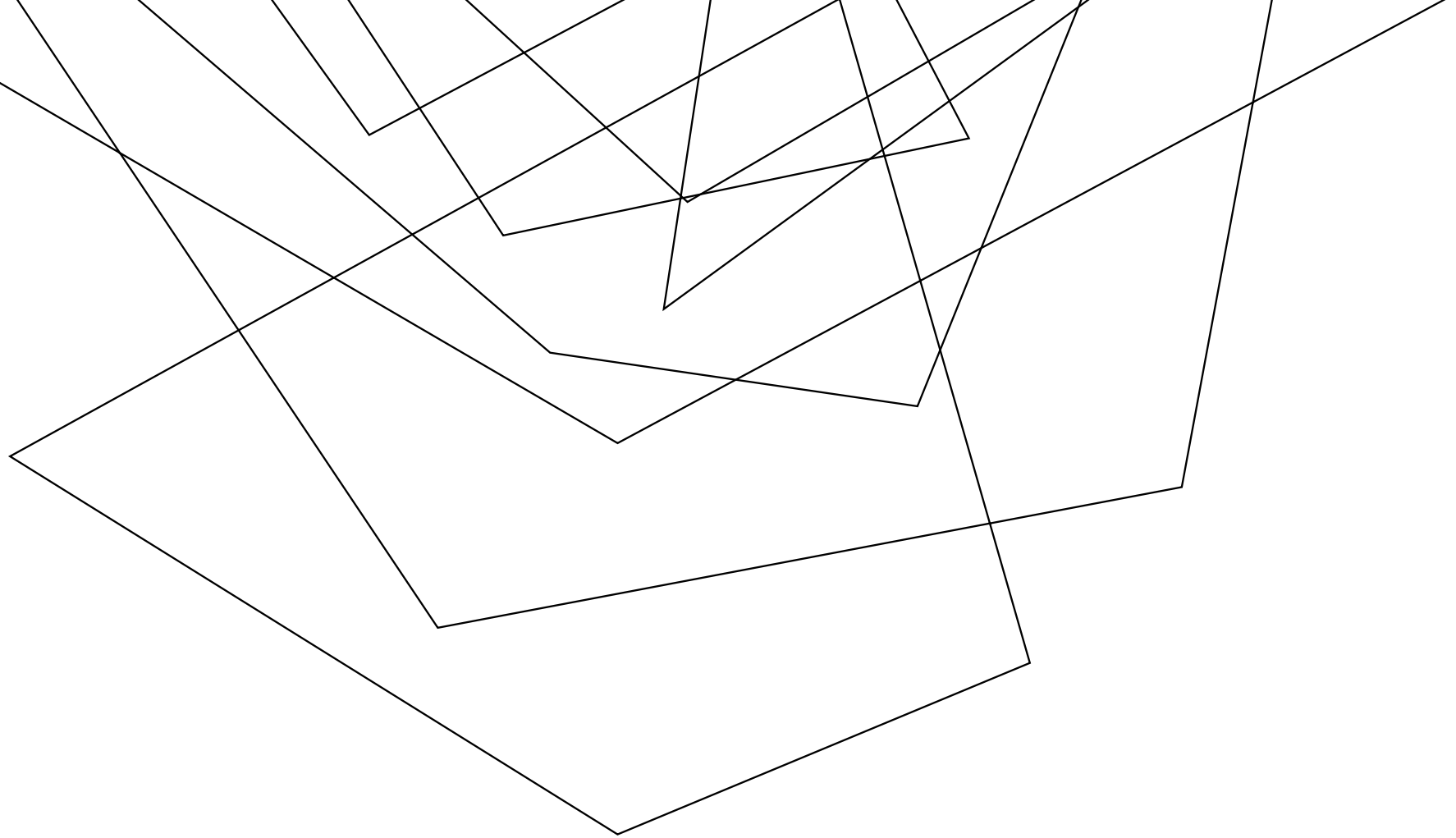
LAST TIME

MEMORY REVIEW

The game of cat-and-mouse for remote code execution



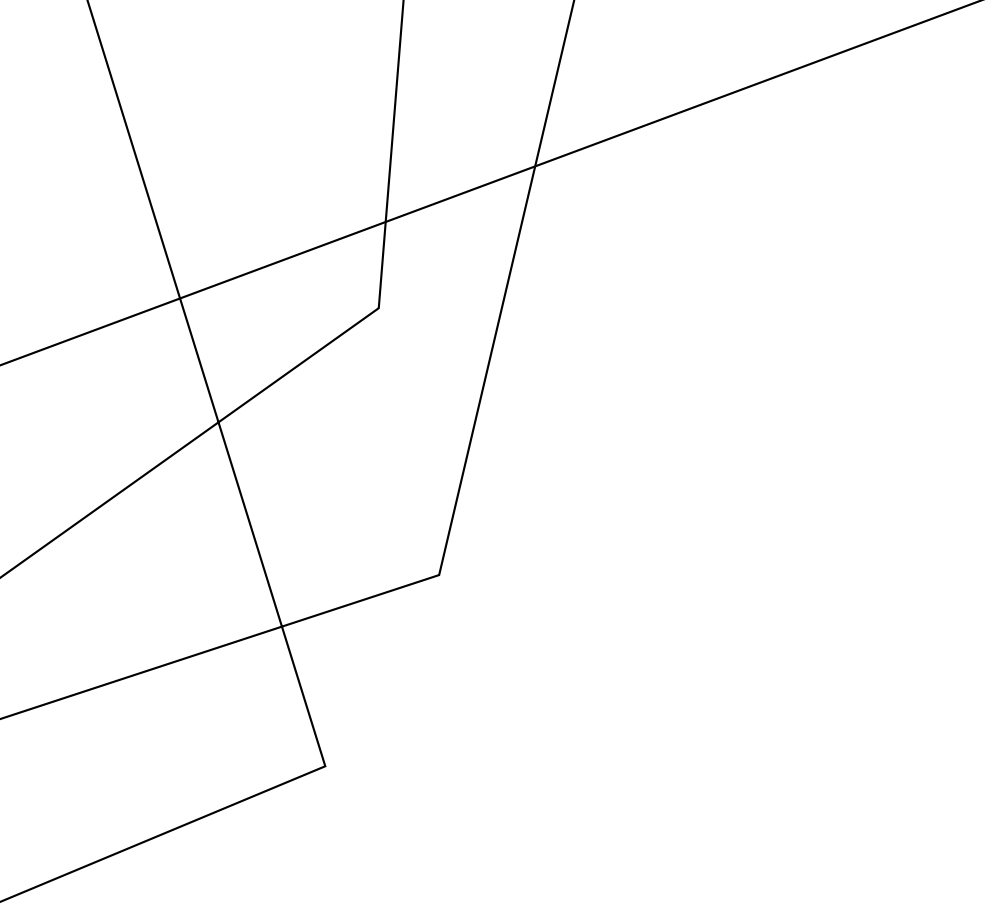
What can we expect out of code analysis?



COMPUTABILITY

EECS 677: Software Security Evaluation

Drew Davidson



ADMINISTRIVIA AND ANNOUNCEMENTS

P1 graded

- Good job!
- Main issues: Incorrect output values

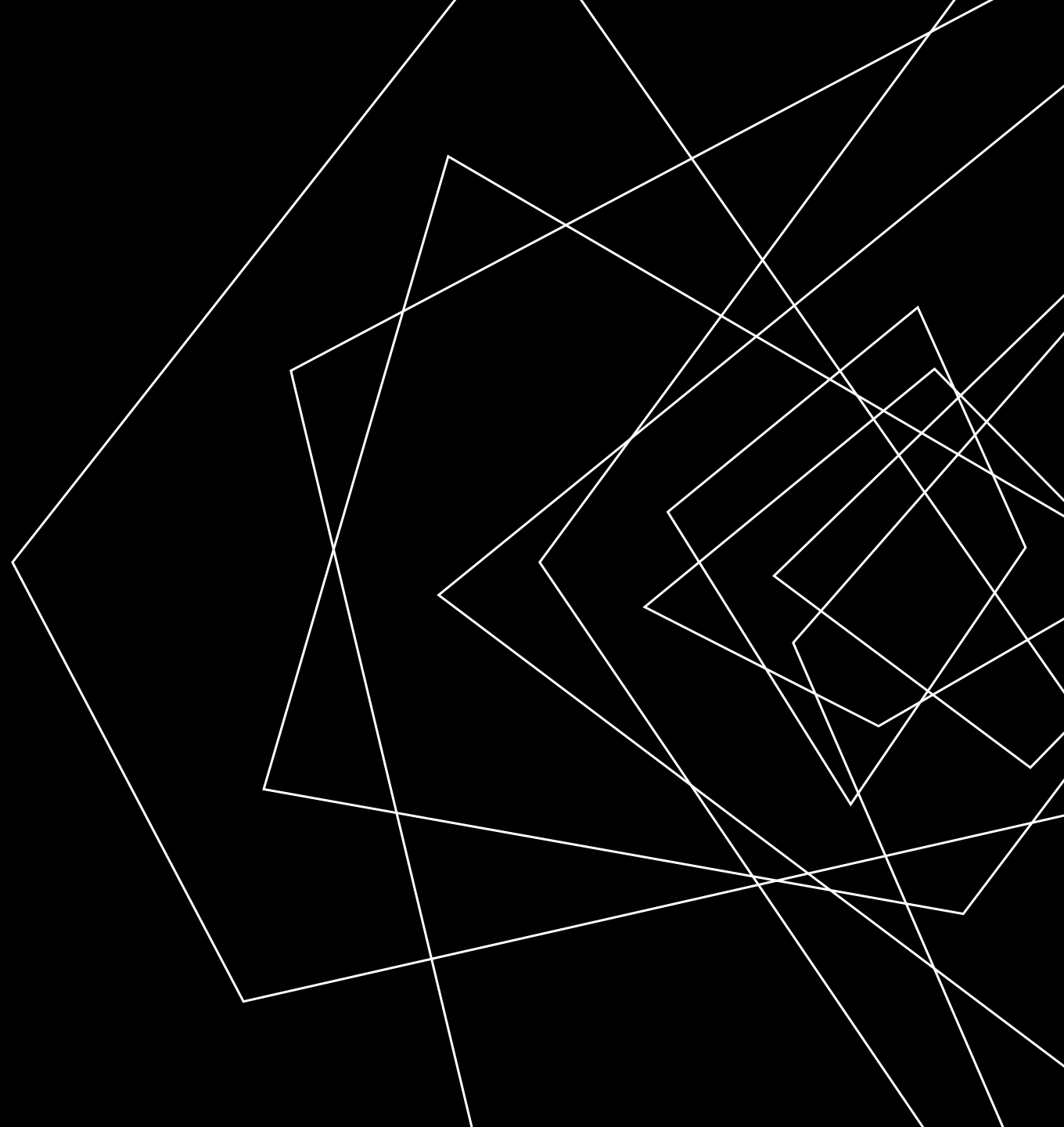
TODAY'S ROADMAP

Decidability

The Halting Problem

Type I/Type II Errors

Soundness / Completeness



THE LIMITS OF COMPUTATION

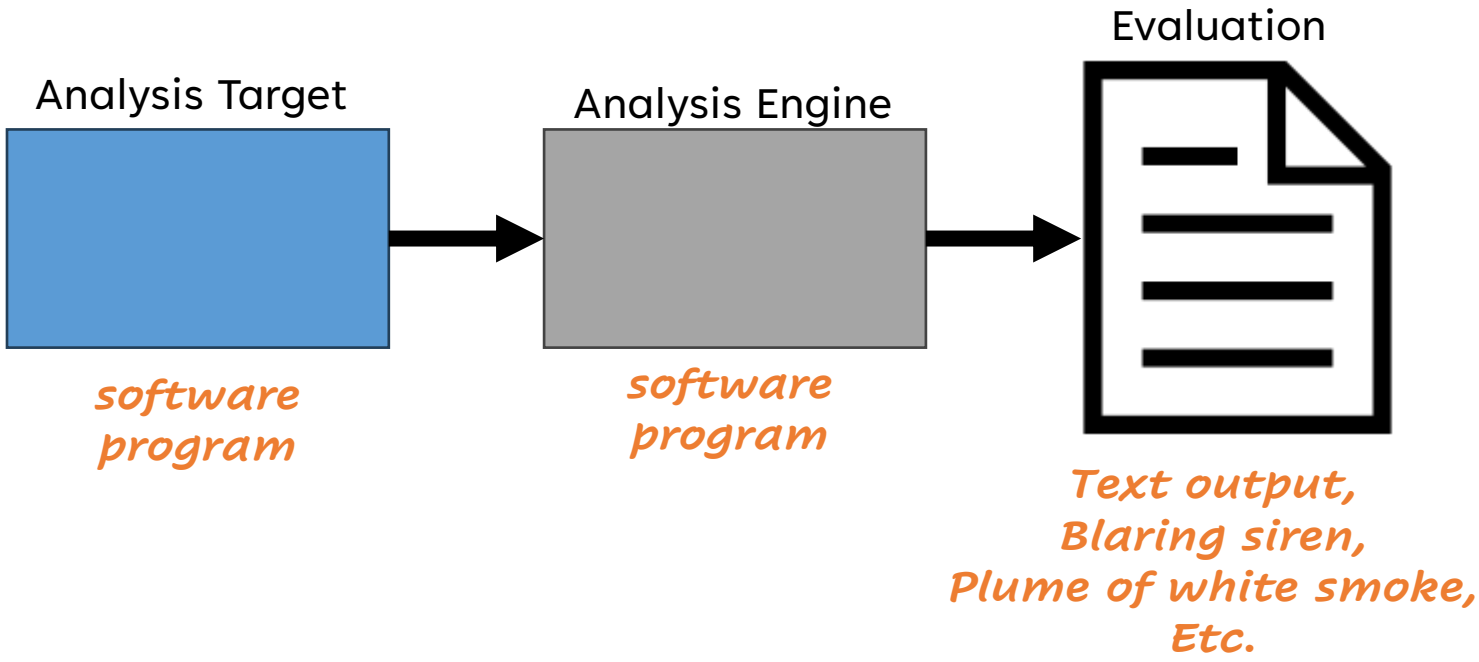
DECIDABILITY

Computers! What <u>can't</u> they do?!

- As we begin our exploration of security evaluation, we care about this question for two reasons:
  - We need to know the capabilities of our analysis target
  - We need to know the capabilities of our analysis engine

BOUNDING OUR WORKFLOW

DECIDABILITY



COMPUTATIONAL POWER

DECIDABILITY

What is a program?

- A set of executable instructions
- Instructions modify memory

There are many formats for programs

i.e. programming languages

- Does the language of the analysis target limit it's computational power?
- Does the language of the analysis engine limit its computational power?

Good(?) news:
It does not!

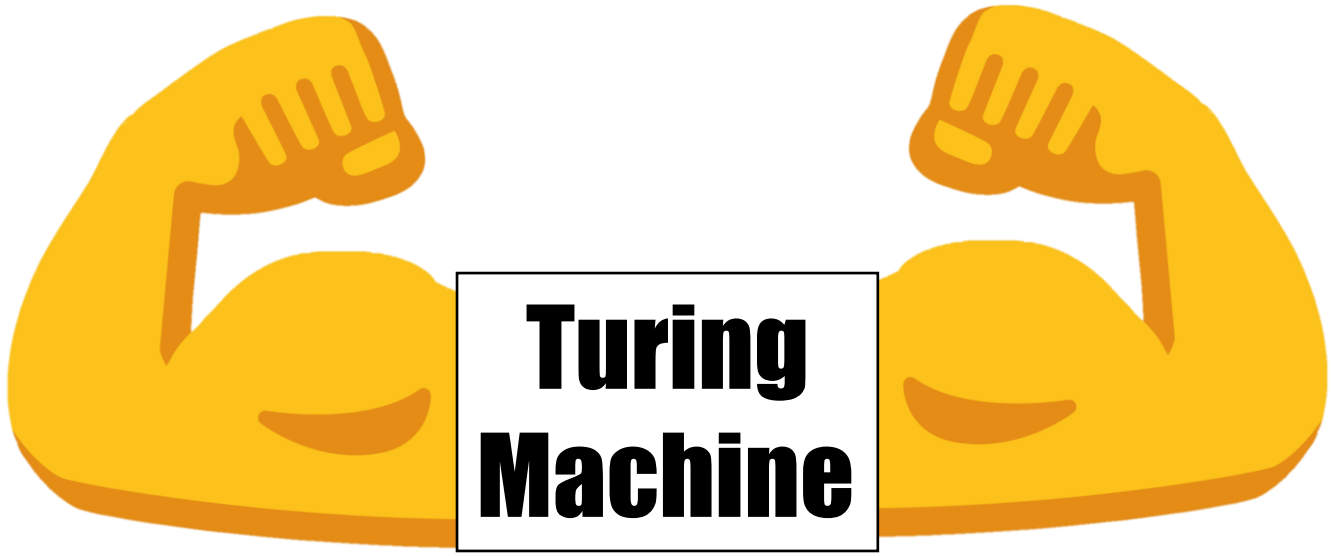
CHURCH-TURING THESIS

DECIDABILITY

Roughly: a function on the natural numbers can be calculated if and only if it is computable by a Turing machine

Practical Upshot: Language choice doesn't matter beyond Turing completeness

(It is sufficient to model the analysis engine and analysis target as Turing machines)



AVERTING DISASTER

DECIDABILITY

Software Security is all about avoidance

- Avert a flaw before it is exploited
- Squash a bug before it bites

C onfidentiality

I ntegrity

A vailability

AVERTING DISASTER

DECIDABILITY

How do we know a disaster is imminent?

Is this code “disastrous”?

```
%ptr = inttoptr i64 0 to ptr
store i32 4, ptr %ptr
```

AVERTING DISASTER

DECIDABILITY

How do we know a disaster is imminent?

Is this code “disastrous”?

```
entry:
  br label %exit
label1:
  %ptr = inttoptr i64 0 to ptr
  store i32 4, ptr %ptr
  br label %exit
exit:
  ret i32 0
```

AVERTING DISASTER

DECIDABILITY

How do we know a disaster is imminent?

Is this code “disastrous”?

```
define i32 @fn() {
true_entry:
  br label label1
entry:
  br label %exit
label1:
  %ptr = inttoptr i64 0 to ptr
  store i32 4, ptr %ptr
  br label %exit
exit:
  ret i32 0
}
```

AVERTING DISASTER

DECIDABILITY

How do we know a disaster is imminent?

Is this code “disastrous”?

```
define i32 @fn() {
true_entry:
  br label label1
entry:
  br label %exit
label1:
  %ptr = inttoptr i64 0 to ptr
  store i32 4, ptr %ptr
  br label %exit
exit:
  ret i32 0
}

define i32 @main(){
  ret i32 0
}
```

DECISION PROCEDURES

DECIDABILITY

A little vocabulary:

A **decision problem** is a computational question that can be solved with either a yes or a no. *Frequently, we consider decision problems as detection of a property in a program*

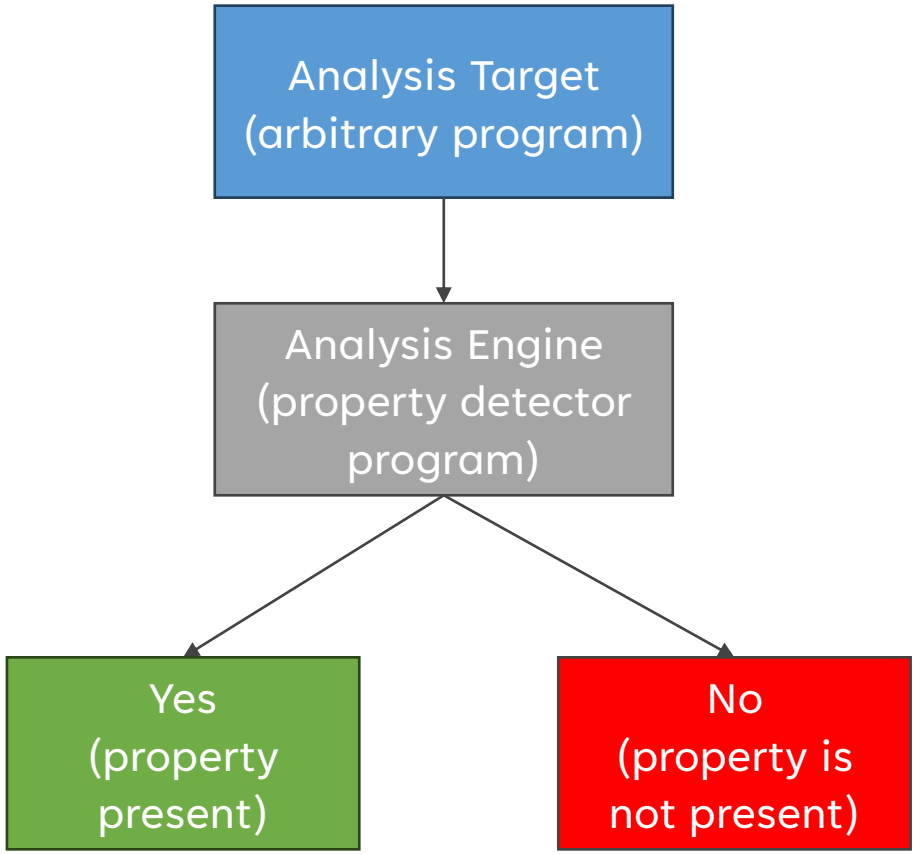
A **decision procedure** is a method for solving a decision problem that always yields the correct answer

If there is no decision procedure for a given decision problem, that decision problem is called **undecidable**

PROGRAM ANALYSIS AS DECISION PROCEDURE

DECIDABILITY

Since a program is just a list of instructions, it is valid input to a decision procedure



STRONG GUARANTEES

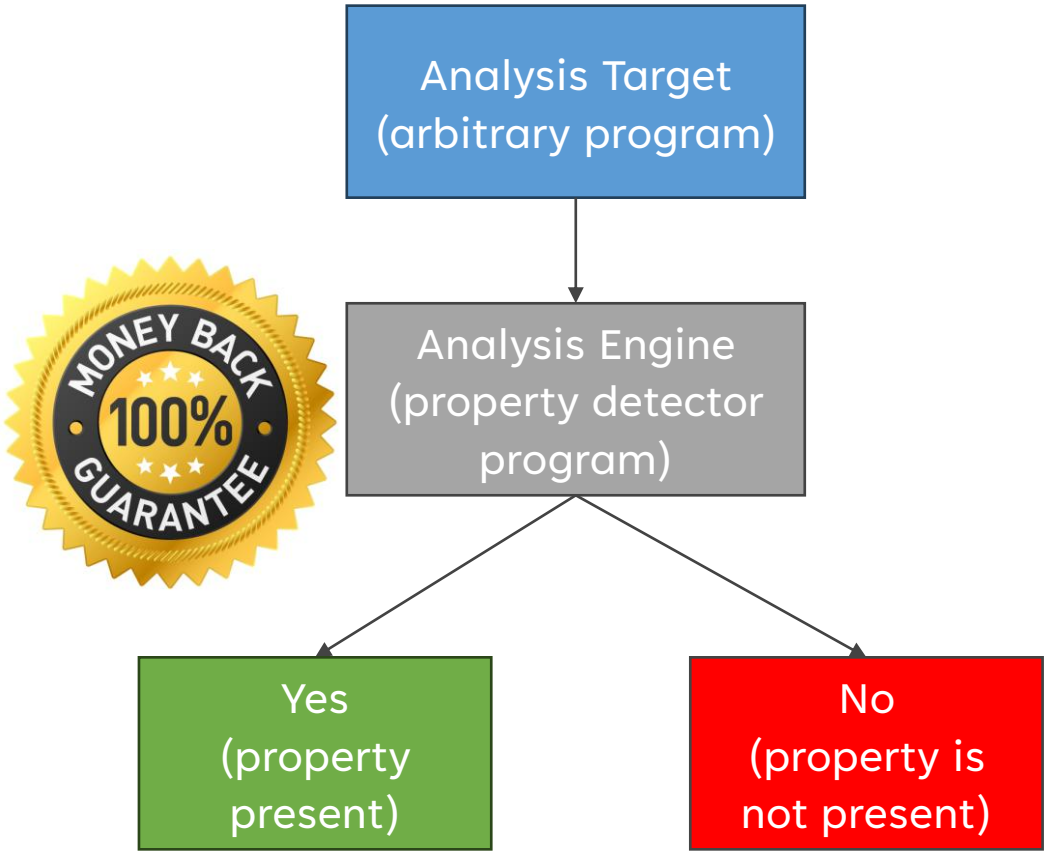
DECIDABILITY

A decision procedure is a high bar

Guarantee that:

- The analysis engine accepts every program
- The analysis engine always returns an answer
- The answer returned is always correct

Rice's Theorem



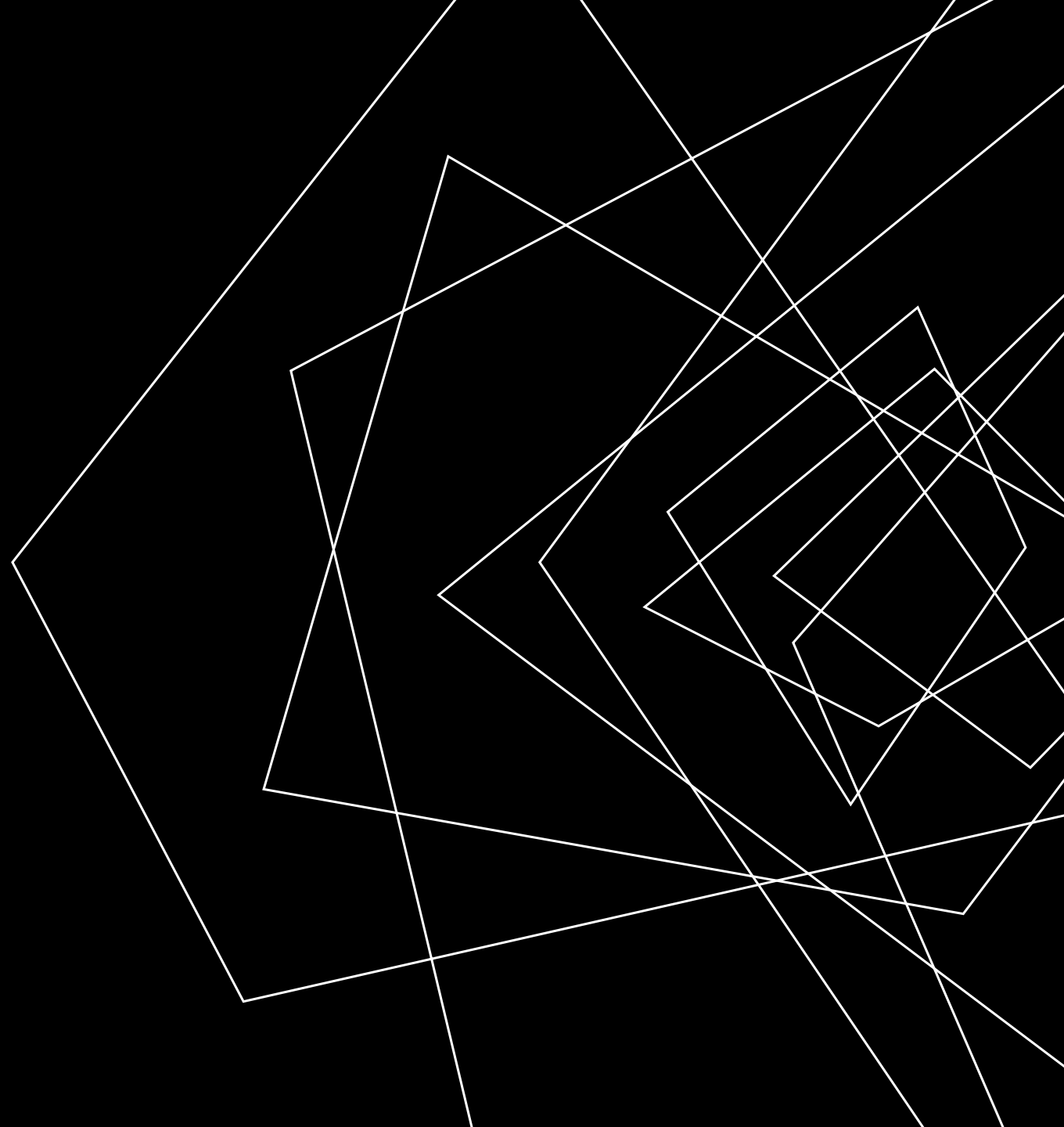
TODAY'S ROADMAP

Decidability

The Halting Problem

Type I/Type II Errors

Soundness / Completeness



STATING THE PROBLEM

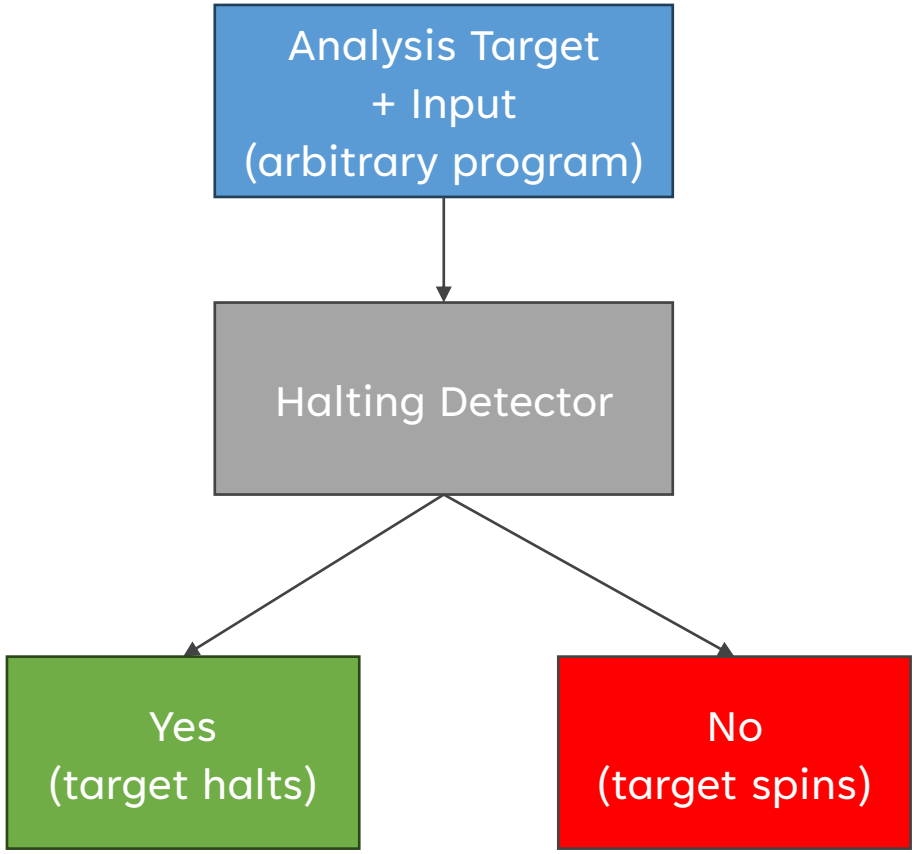
THE HALTING PROBLEM

Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting

A HALTING DETECTOR

THE HALTING PROBLEM

Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting



Is there a decision procedure for the halting problem?

- We'll sketch the proof outline that there is NOT
- Relies on a proof by contradiction

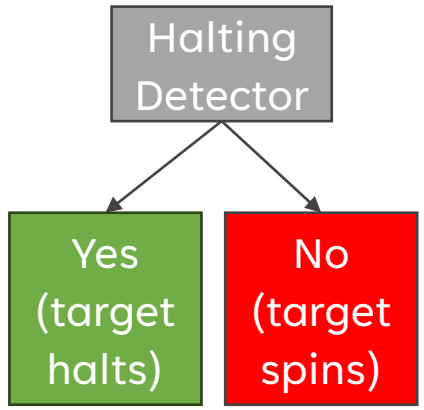
PROOF BY CONTRADICTION

THE HALTING PROBLEM

Reductio ad absurdum – Assuming the premise has obviously incorrect consequences

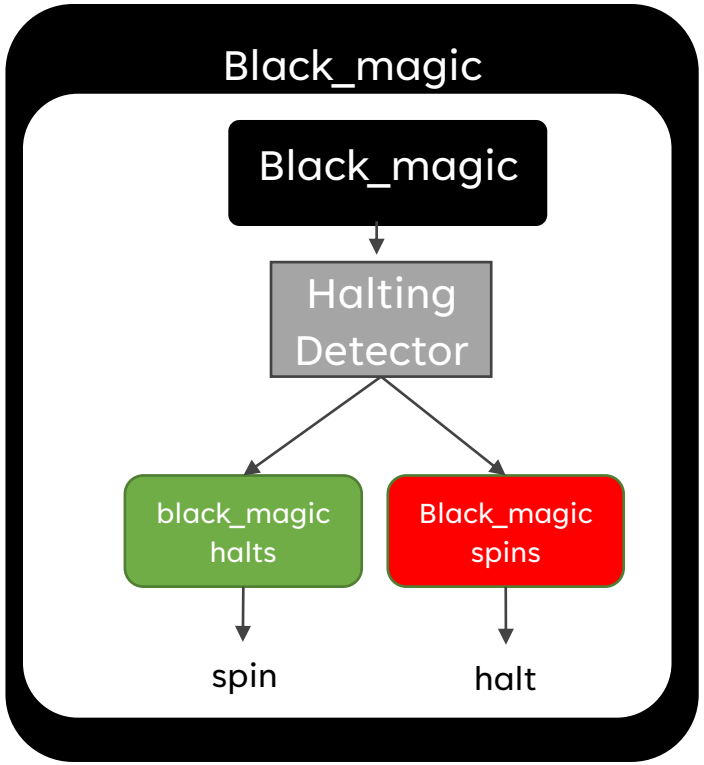
Here: assume there is a halting detector

Assumption



`halts(ANY ARBITRARY PROGRAM)`

```
black_magic(){
    if (halts(black_magic){
      while(true){} //Spin
    }
    //Halt
}
```



WHO CARES?

THE HALTING PROBLEM

No halting decision procedure means no reachability decision procedure

```
1. int main(){
2.     if ( black_magic() ){
3.         int * a = nullptr;
4.         *a = 1;
5.     }
6. }
```

This program crashes if and only if it reaches line 4, which depends on the result of a function call being true

RICE'S THEOREM

THE HALTING PROBLEM

No halting decision procedure means no reachability decision procedure

```
1. int main(){
2.     if ( black_magic() ){
3.         int * a = nullptr;
4.         *a = 1;
5.     }
6. }
```

behavior you care about

Exhibits the behavior you care about

This program ~~crashes~~ if and only if it reaches line 4, which depends on the result of a function call being true

RICE'S THEOREM

THE HALTING PROBLEM

"All non-trivial semantic properties of programs are undecidable"

LIMITATIONS OF RICE'S THEOREM

THE HALTING PROBLEM

Rice’s Theorem is less catastrophic than you might expect for security:

- A decision procedure is a pretty high bar
- A Turing machine is actually not a perfect approximation of the computers we use!

Despite these limitations, it is widely accepted that program analysis is **always** approximate

- We can’t be right all of the time
- We can choose what types of errors we make

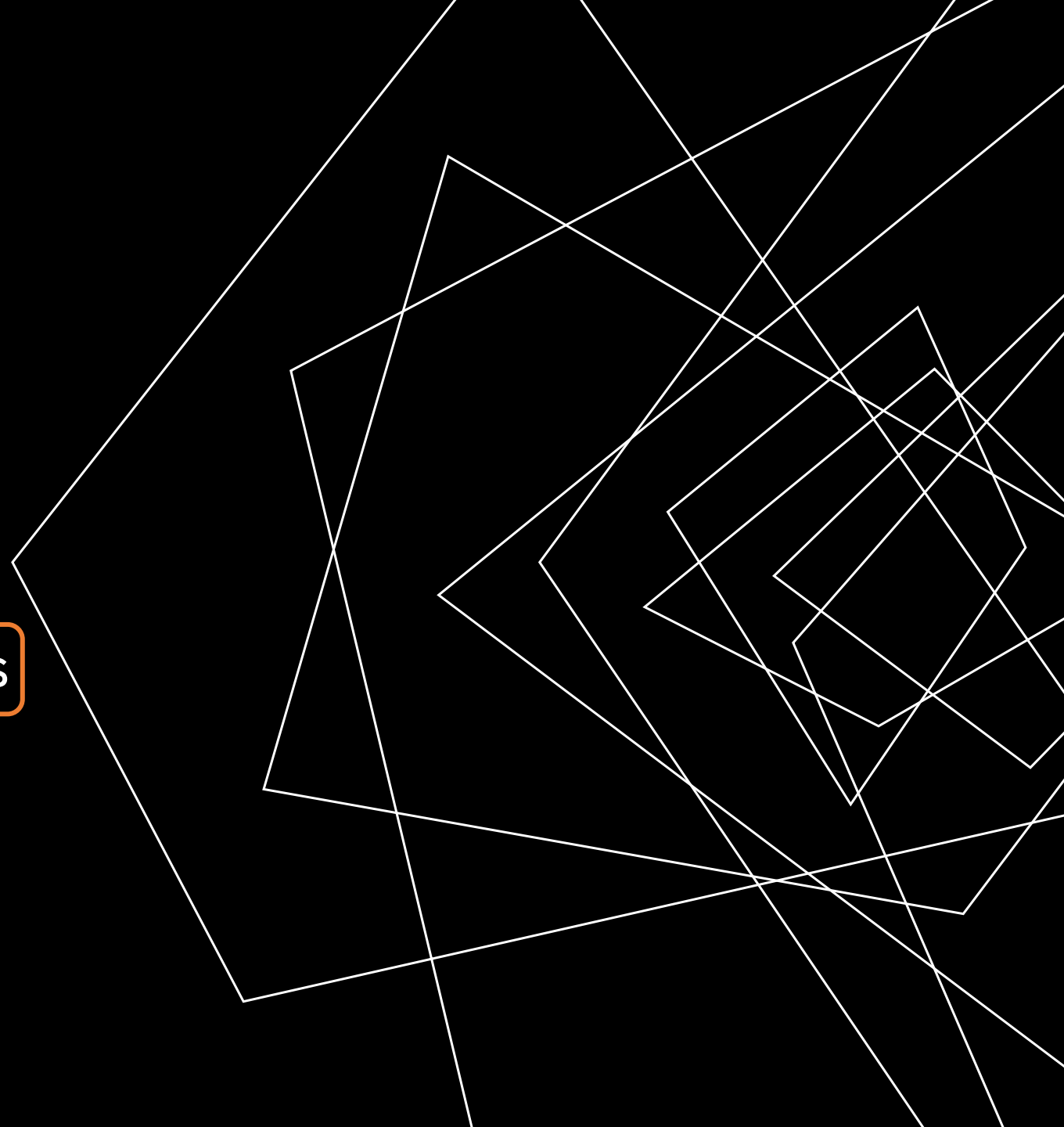
TODAY'S ROADMAP

Decidability

The Halting Problem

Categorizing Program Analyses

Soundness / Completeness



CLASSIFYING DETECTORS
CATEGORIZING PROGRAM ANALYSES

Abstractly: an analysis is a system to detect a phenomenon

A hand detector: when hand detected, emit soap

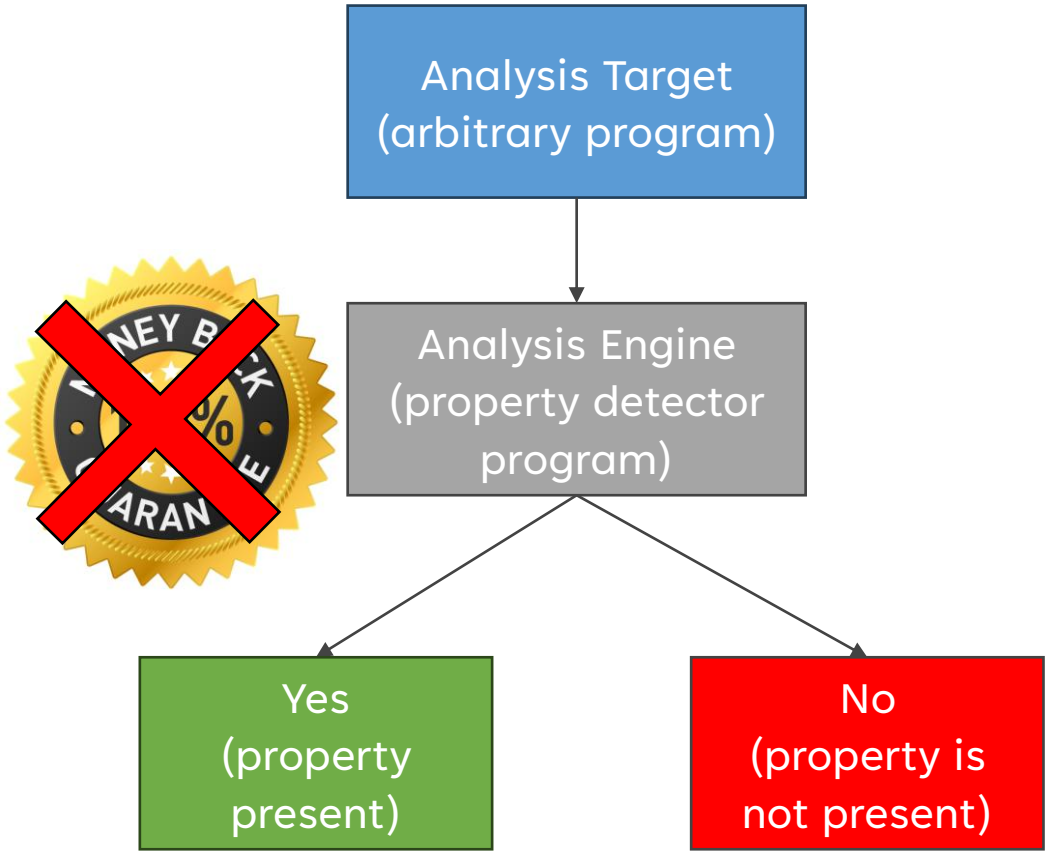
CLASSIFYING DETECTORS

CATEGORIZING PROGRAM ANALYSES

	TRUE correct	**False** wrong
POSITIVE “present”		
NEGATIVE “absent”		

TYPES OF ANALYSIS
CATEGORIZING PROGRAM ANALYSES

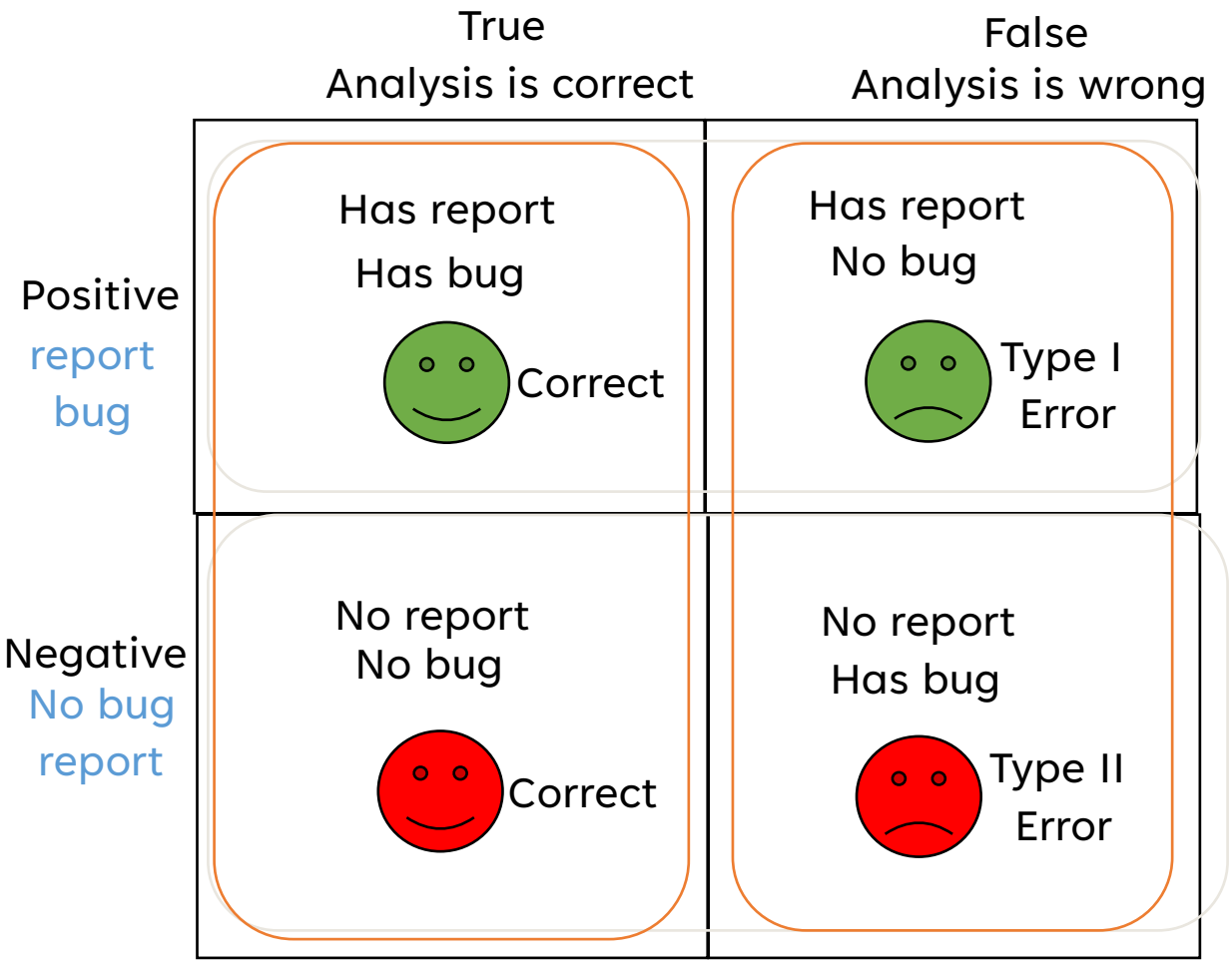
In order to determine the properties of a given program analysis, let's frame it as a detector

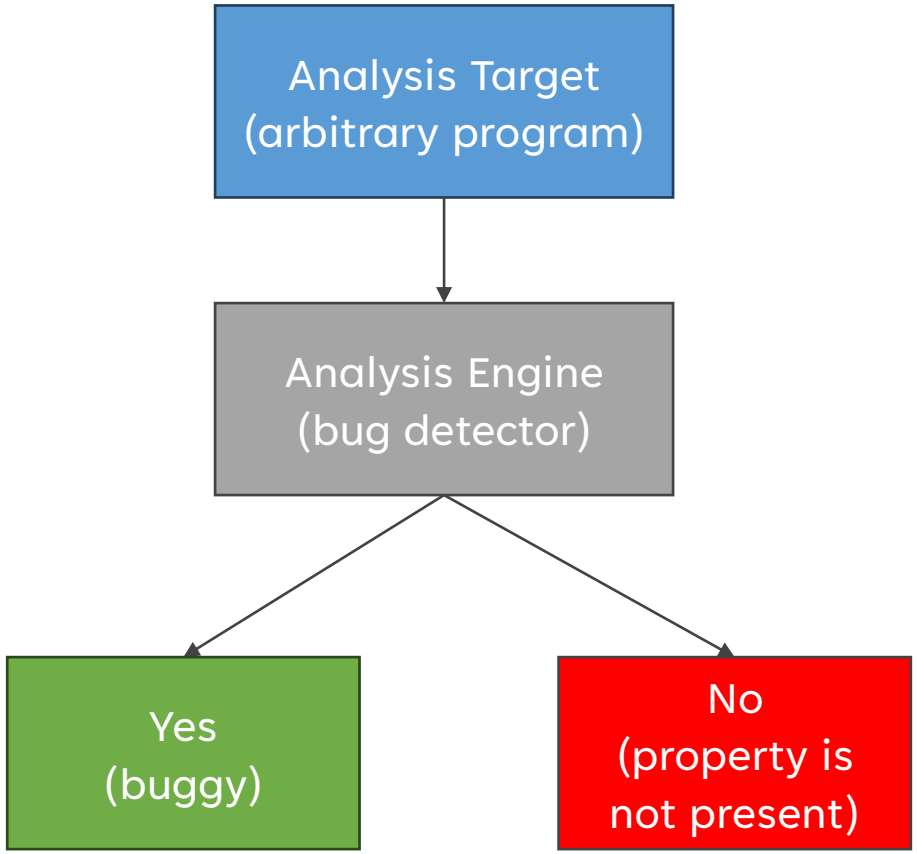


Note: we can detect bad behavior or good behavior

CLASSIFYING ERRORS

CATEGORIZING PROGRAM ANALYSES





TODAY'S ROADMAP

Decidability

The Halting Problem

Categorizing Program Analyses

Soundness / Completeness

GUARANTEES OF IMPERFECT ANALYSES

SOUNDNESS / COMPLETENESS

Consistency / Reliability super important for users

We'd like to limit the kinds of errors we report

We can choose which type of bug report error to avoid

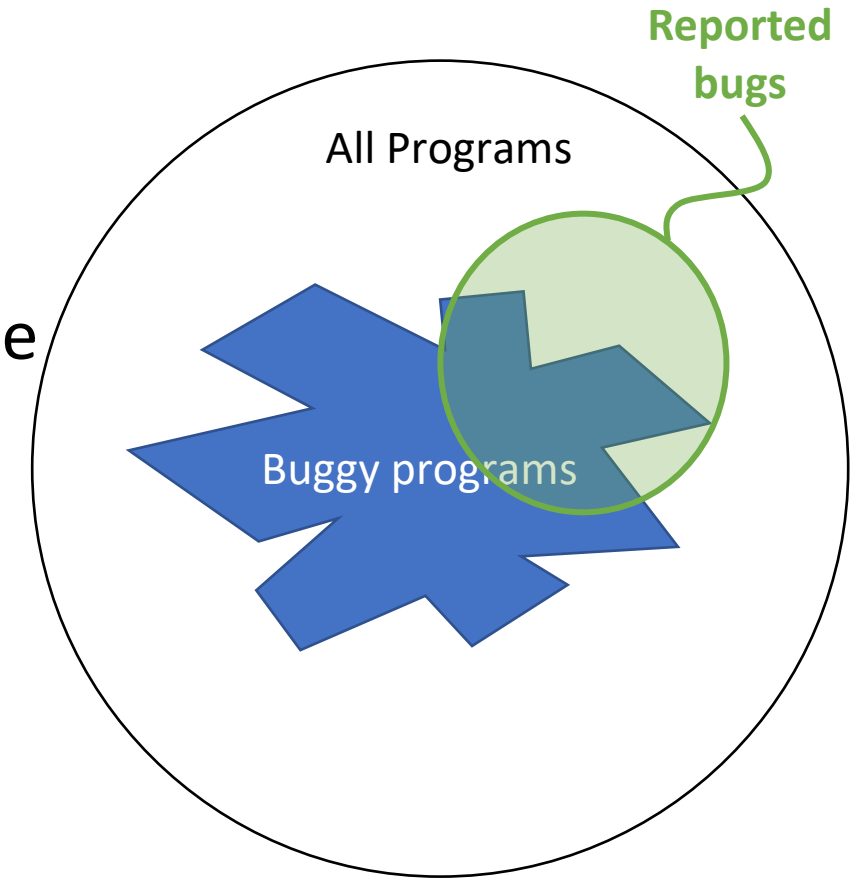
- Soundness: No false positives
- Completeness: No false negatives

VISUAL ANALOGY

SOUNDNESS / COMPLETENESS

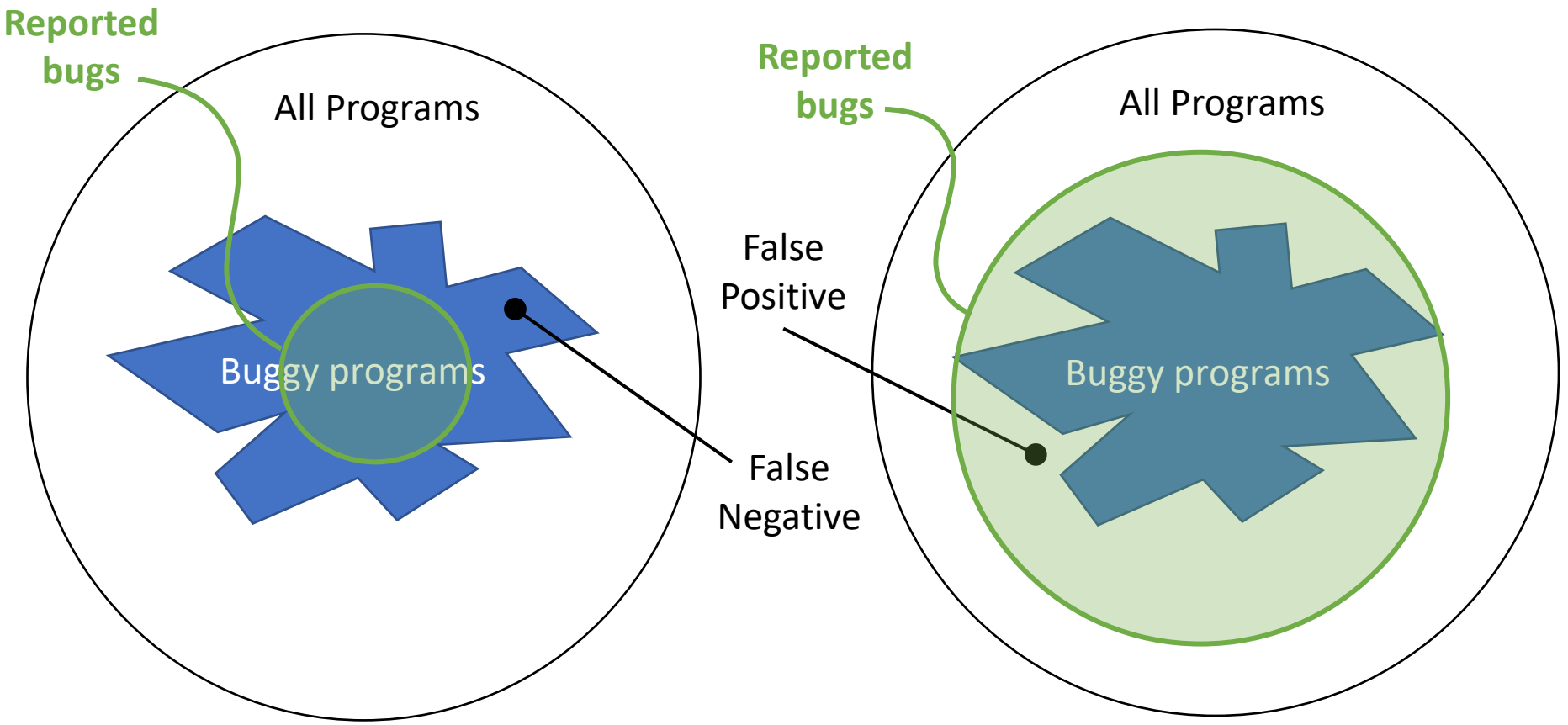
Imagine the universe of all programs is contained in a circle

- You can draw a circle around the programs you report as buggy
- The actual buggy programs occupy a jagged region



VISUAL ANALOGY

SOUNDNESS / COMPLETENESS



Sound bug detection

All correct programs pass through
(No false positive problem)

Some buggy programs pass through
(has false negative problem)

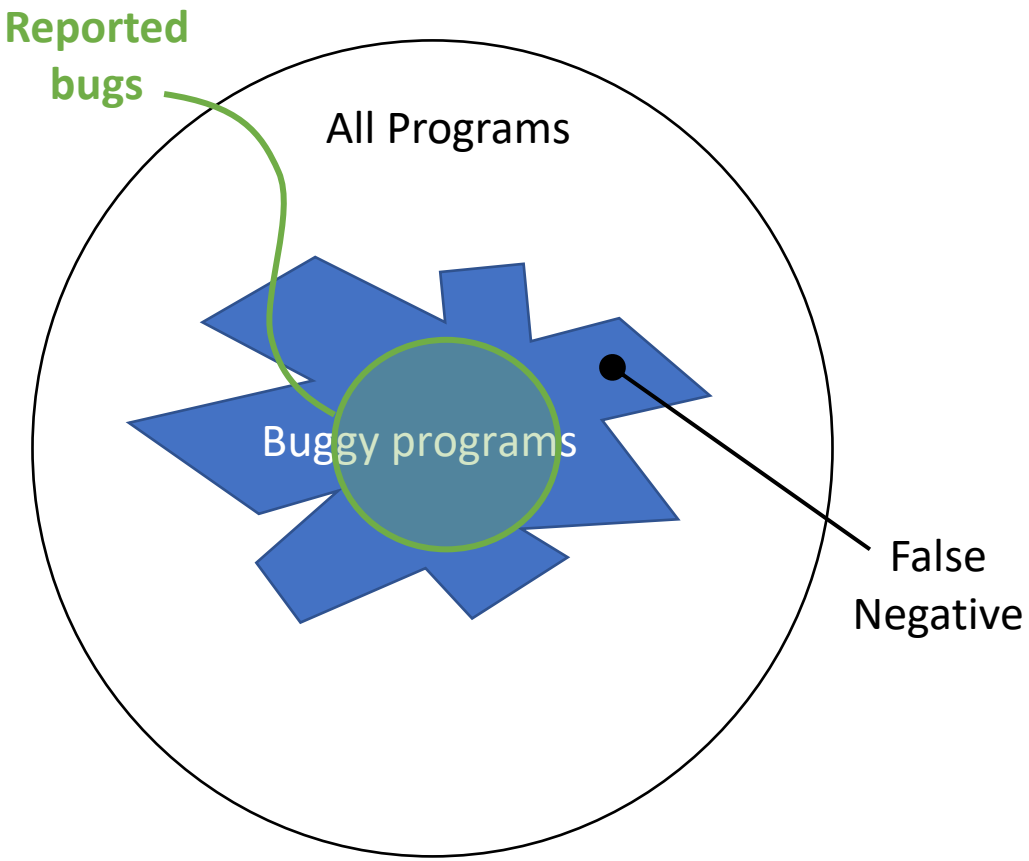
Complete bug detection

All buggy programs get flagged
(No false negative problem)

Some correct programs get flagged
(has false positive problem)

TRIVIAL SOUNDNESS

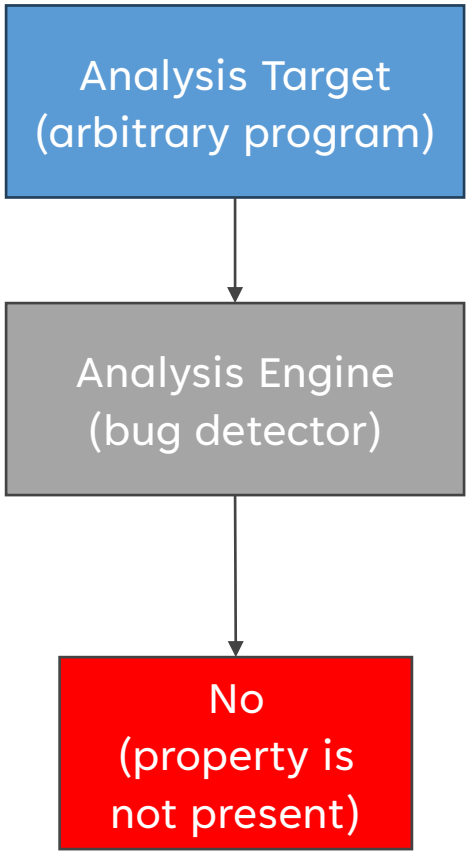
SOUNDNESS / COMPLETENESS



Sound bug detection

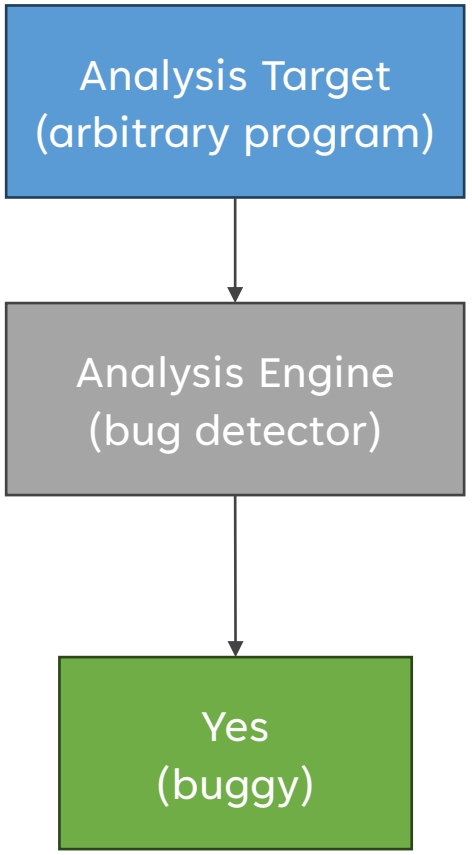
All correct programs pass through
(No false positive problem)

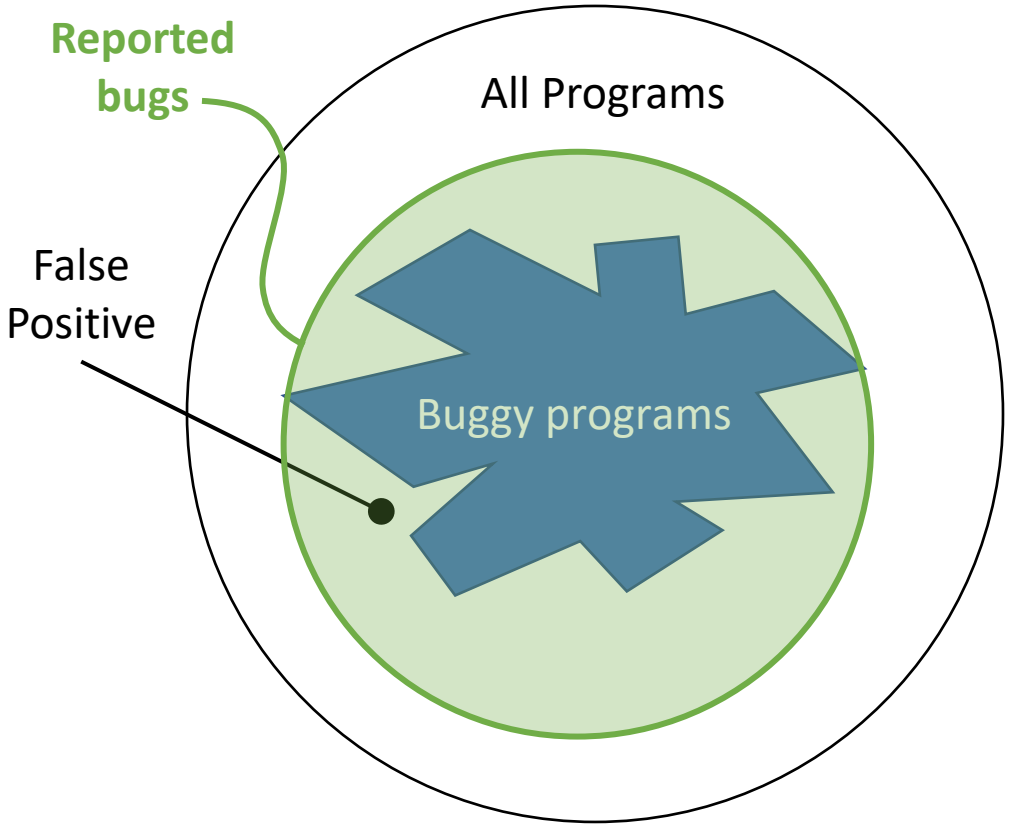
Some buggy programs pass through
(has false negative problem)



TRIVIAL COMPLETENESS

SOUNDNESS / COMPLETENESS





Complete bug detection
All buggy programs get flagged
(No false negative problem)

Some correct programs get flagged
(has false positive problem)

THE IMPORTANCE OF GUARANTEES

SOUNDNESS / COMPLETENESS

Classic security posture: Ensure safety

A complete bugfinder:

Reports bugs in programs

Equivalently, a sound verifier

Reports bug-free programs

ANALYSIS METHOD VS ERRORS

SOUNDNESS / COMPLETENESS

It's natural to consider the types of compromises of each analysis method

- Static analysis
  - Often builds a model of the program, makes inferences on that model
  - Tends to make completeness easier
  - Scalability concerns for large programs
- Dynamic analysis
  - Often performs the analysis by straight up running the program, observing behavior
  - Tends to make soundness easier
  - Coverage problems

BEYOND ALL-OR-NOTHING

SOUNDNESS / COMPLETENESS

As you can imagine, soundness and completeness are not the full story

- Guarantees are nice, but we want legitimately useful analyses!
- Many practical analyses are neither sound nor complete

SUMMARY

Explored the limits of program analysis
Theoretical abstractions can be sound
or complete, but not both

Guarantees are still possible, e.g.:

guarantee all bugs (of some sort) reported are real, but we might miss some bugs (sound, incomplete bugfinder)

guarantee total absence of bugs (of some sort), but may report fake bugs (complete, unsound bugfinder)

SUMMARY

Explored the limits of program analysis

Theoretical abstractions can be sound or complete, but not both

Guarantees are still possible, e.g.:

guarantee all bugs (of some sort) reported are real, but we might miss some bugs (sound, incomplete bugfinder)

guarantee total absence of bugs (of some sort), but may report fake bugs (complete, unsound bugfinder)

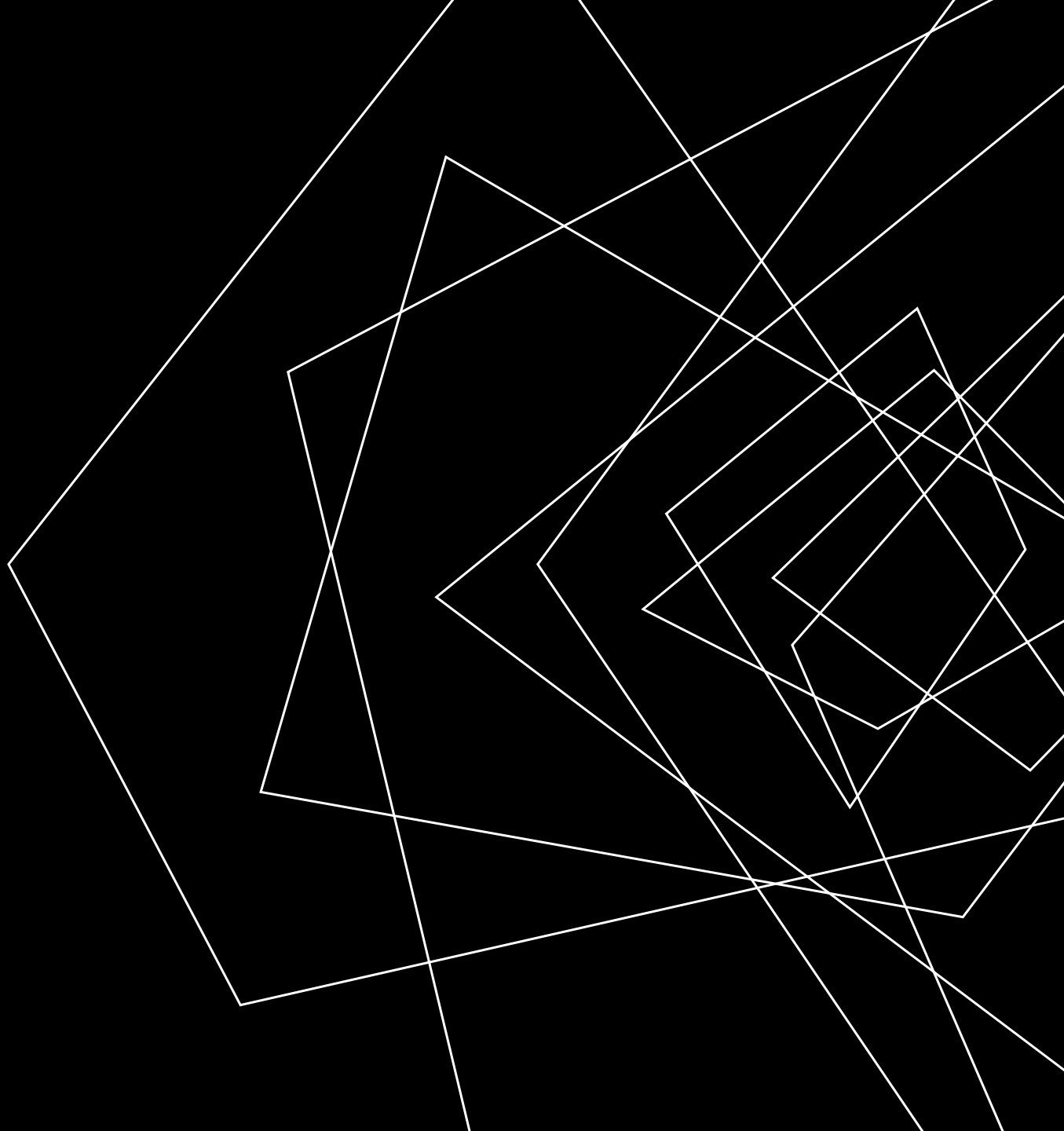
NEXT TIME

Look at how we build a useful complete bugfinder (static analysis)

LECTURE END!

TODO:

- E7
- Look out for P2



TURING MACHINES V REALITY

DECIDABILITY

Are Turing machines good models of computation?

- i.e. a memory-safe language is not subject to memory attacks

Perhaps computational power is a bad proxy for *target* capabilities

- I actually agree with this perspective

  BUT! Rice's Theorem gives us a sense of what we're up against

  In practice, simultaneous soundness and completeness will hit scalability challenges first

TURING MACHINES V REALITY

DECIDABILITY

Are Turing machines good models of computation?

- i.e. a memory-safe language is not subject to memory attacks

Perhaps computational power is a bad proxy for *target* capabilities

- I actually agree with this perspective

  BUT! Rice's Theorem gives us a sense of what we're up against

  > In practice, simultaneous soundness and completeness will hit scalability challenges first