CSE467: Computer Security

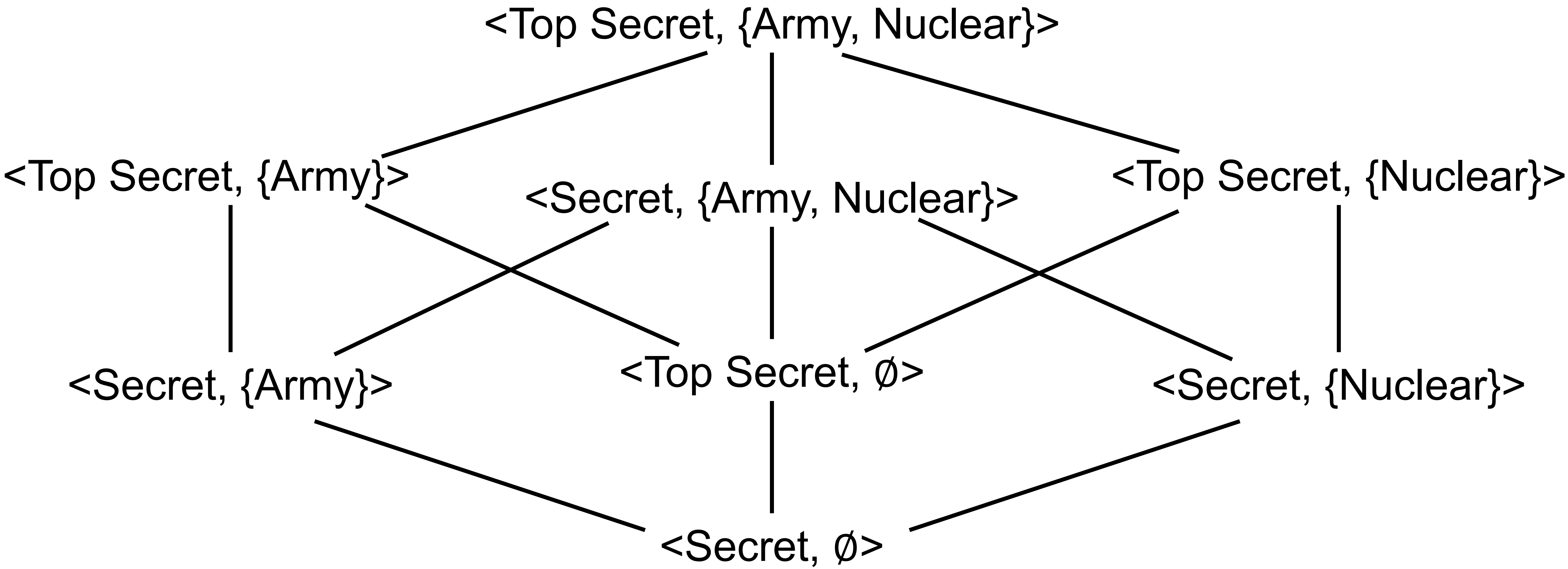
22. Introduction to Program Analysis

Seongil Wi

Department of Computer Science and Engineering

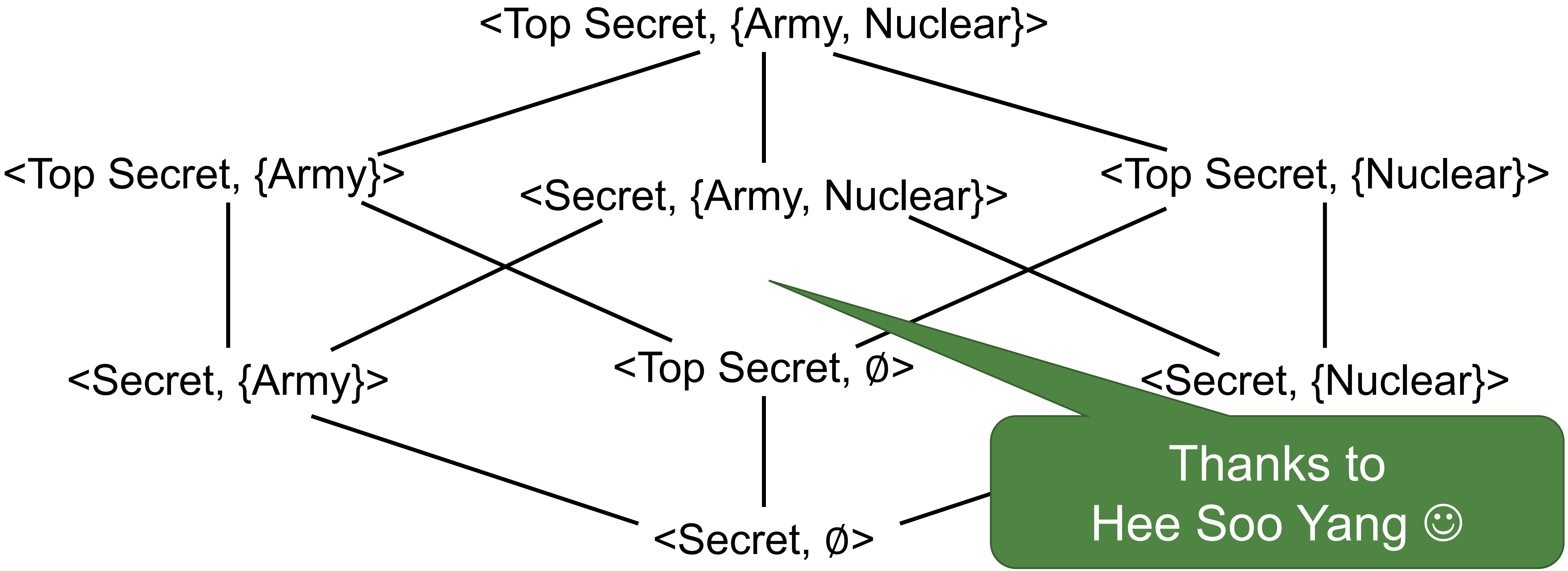
Modification on Previous Lecture Slide (Before)

- The combination of **security level** and **compartment** forms a *lattice*
  - E.g., Security level = {TOP SECRET, SECRET},
    Compartment = {Army, Nuclear}



Modification on Previous Lecture Slide (After)

- The combination of **security level** and **compartment** forms a *lattice*
  - E.g., Security level = {TOP SECRET, SECRET},
    Compartment = {Army, Nuclear}



We will take a Quiz in Next Week

- Date: 11/28 (TUE.), Class time

- Scope:
  - Access Control
  - Authentication

- O/X quiz (3~4 problems) + some computation quiz

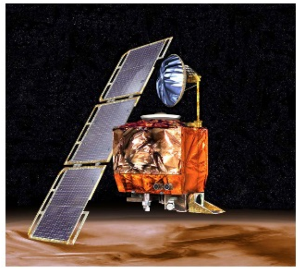
HW2

- I will share the grading results

Impact of Poor Software Quality

The Patriot Missile (1991)
Floating-point roundoff
28 soldiers died

The Ariane-5 Rocket (1996)
Integer Overflow
$100M



NASA's Mars Climate Orbiter (1999)
Meters-Inches Miscalculation
$125M

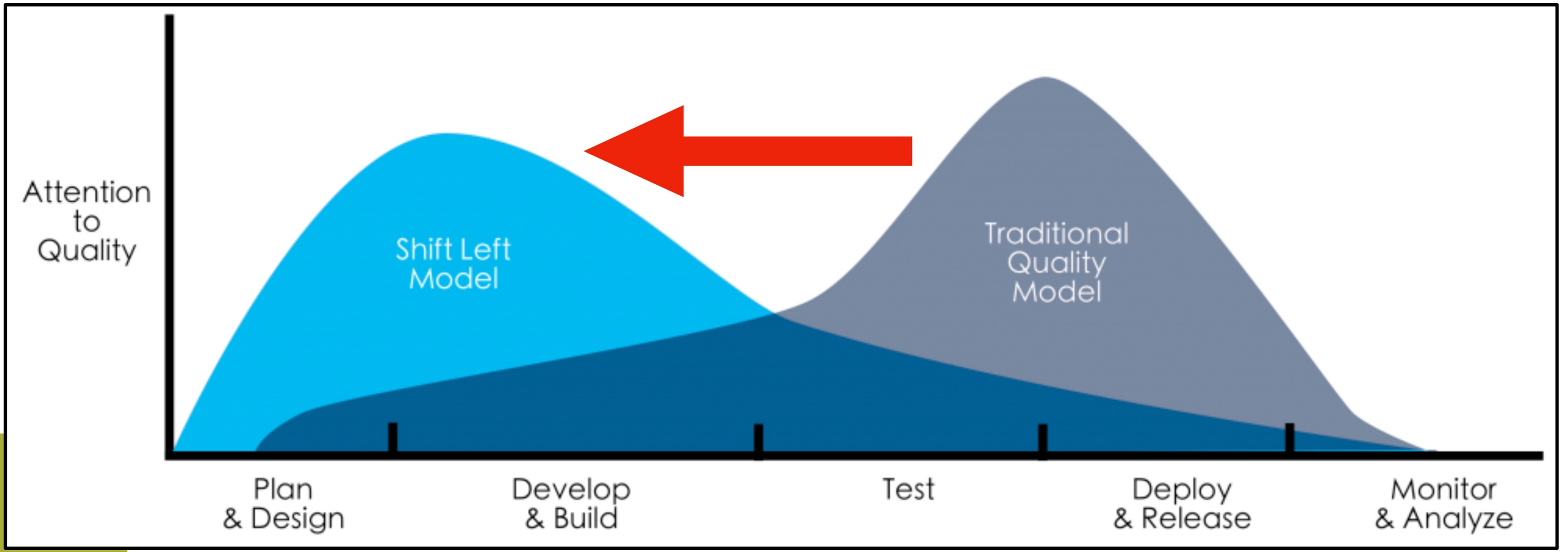
Cost of Software Quality Assurance

*"We have as **many testers** as we have developers. And testers spend **all their time testing**, and developers spend **half their time testing**. We're more of a testing, a quality software organization than we're a software organization"*

- Bill Gates

Discovering Software Bugs

- Very important as software is eating the world!
- Key issue: how to detect software errors as early as possible?



COST OF A SOFTWARE BUG

$100	$1,500	$10,000
If found in Gathering Requirements phase	If found in QA testing phase	If found in Production

- IBM Systems Sciences Institute, 2015

Software Bugs

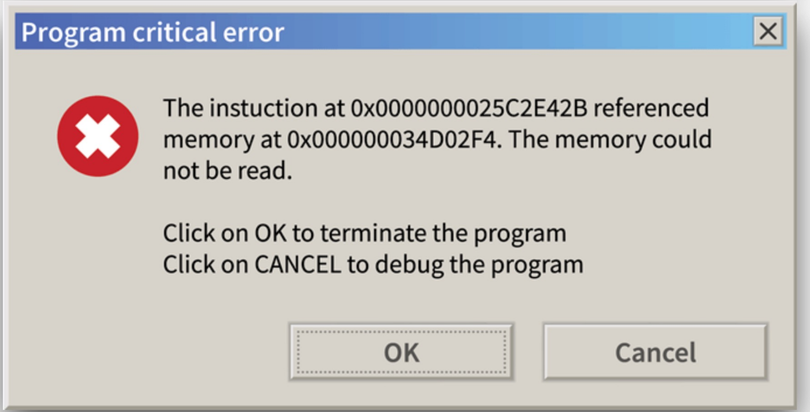
- What?
  - Software runs and produces outputs unexpectedly
- Why?
  - Incorrectly written code by human or AI


WILL KNIGHT BUSINESS SEP 28, 2021 7:00 AM

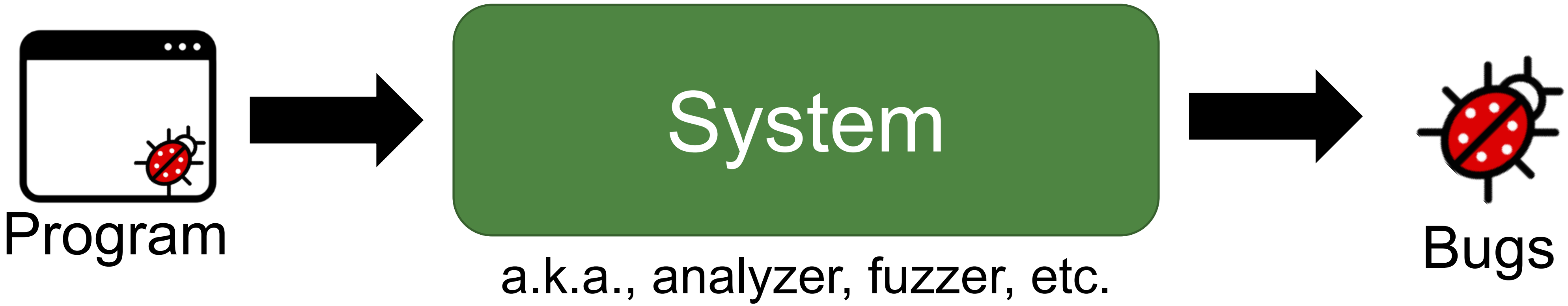
AI Can Write Code Like Humans—Bugs and All

New tools that help developers write software also generate similar mistakes.

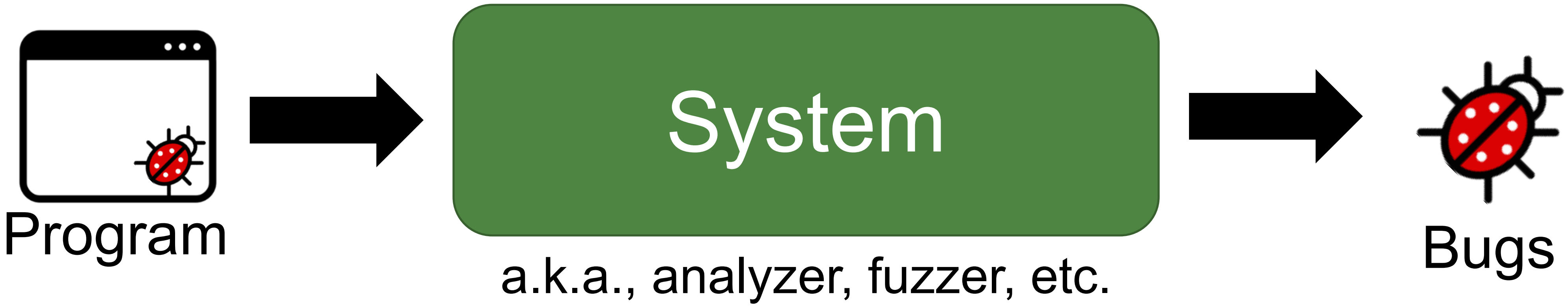
ILLUSTRATION: ELENA LACEY



Build a System that Finds Bugs

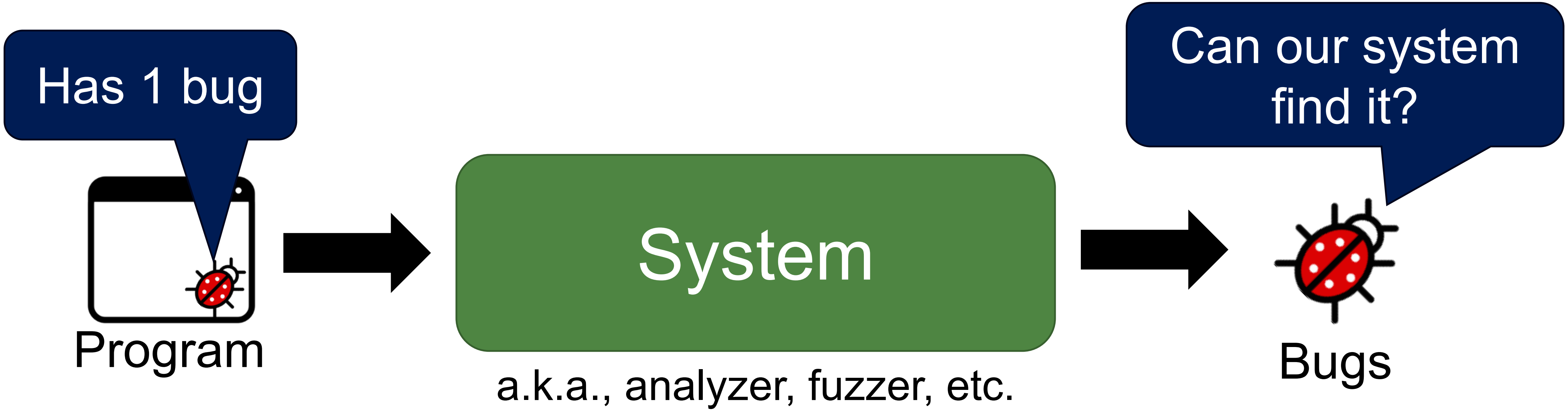


Build a System that Finds Bugs



How *precise* can we make our system?

Precision Matters



Given an arbitrary program, can we build a system that decides **whether the program is buggy or not?**

Building a Perfect Analyzer is Impossible

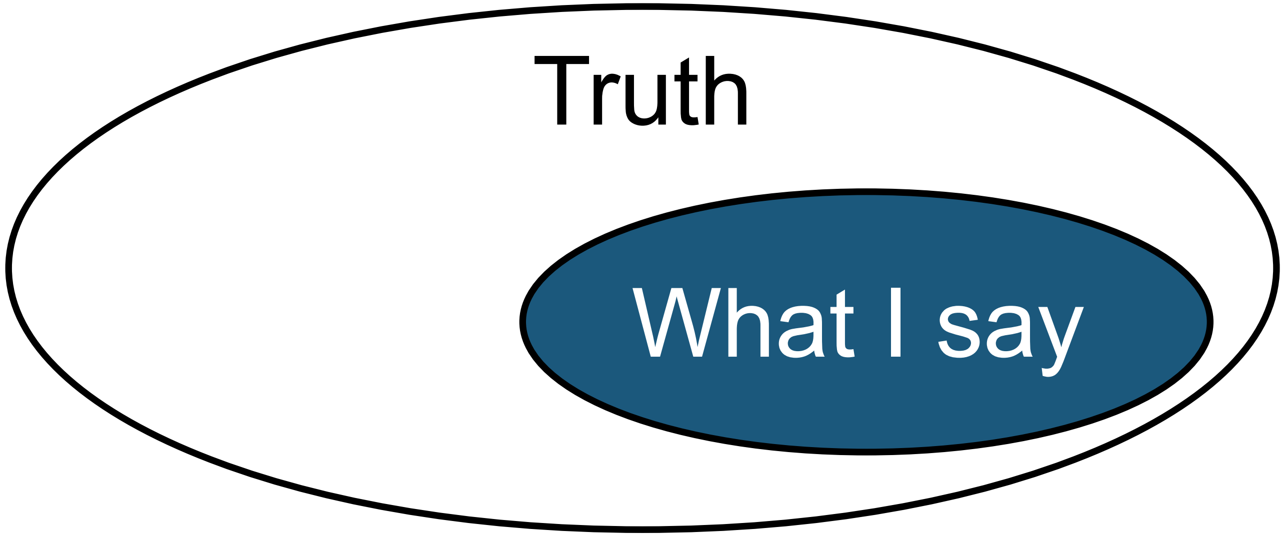
- It only shows the presence of bugs, never their absence!

- But, we can try to find as many bugs as possible

- For example,
  - Bounded model checking
  - Software testing
  - Etc.

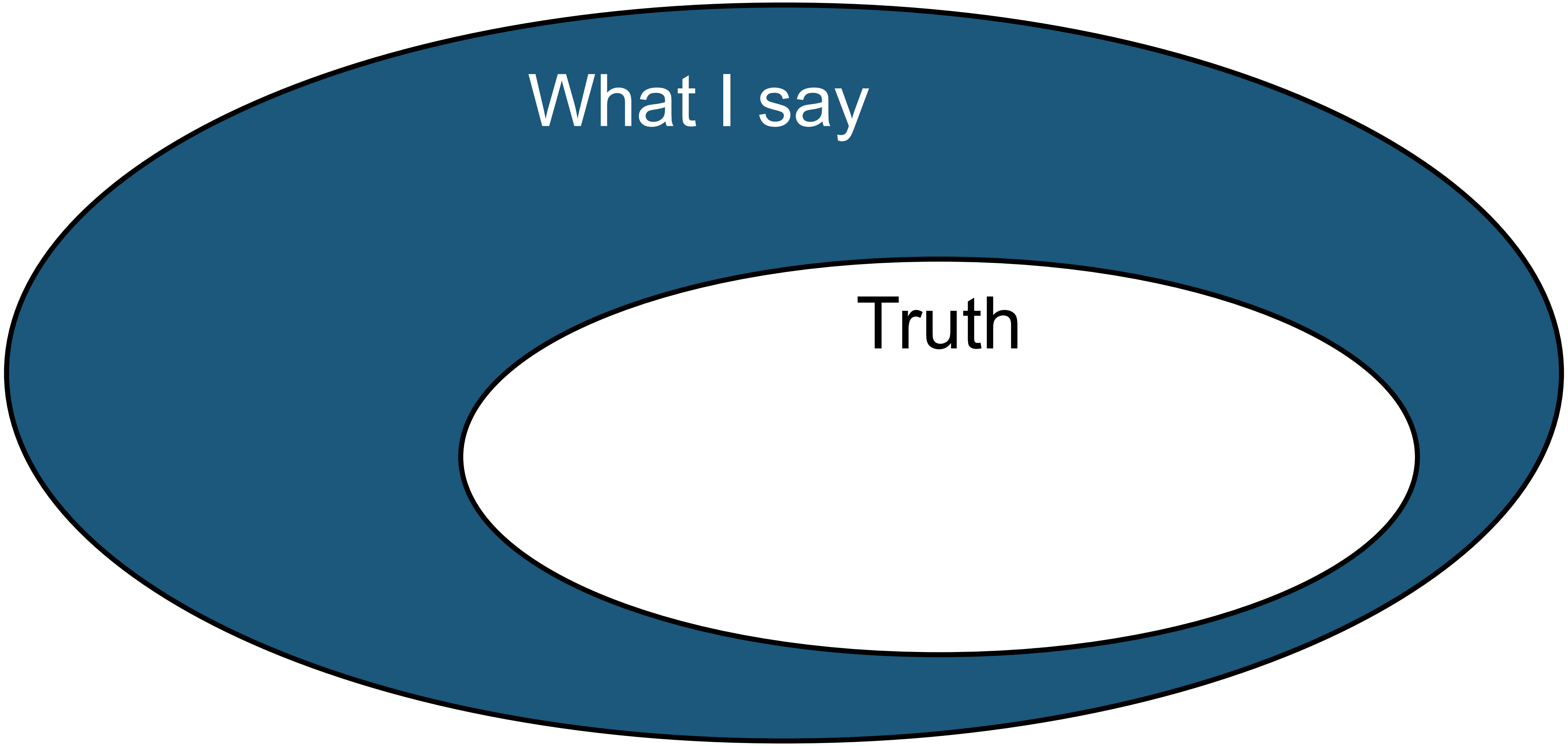
Soundness vs. Completeness

- If an analyzer is ***sound***:



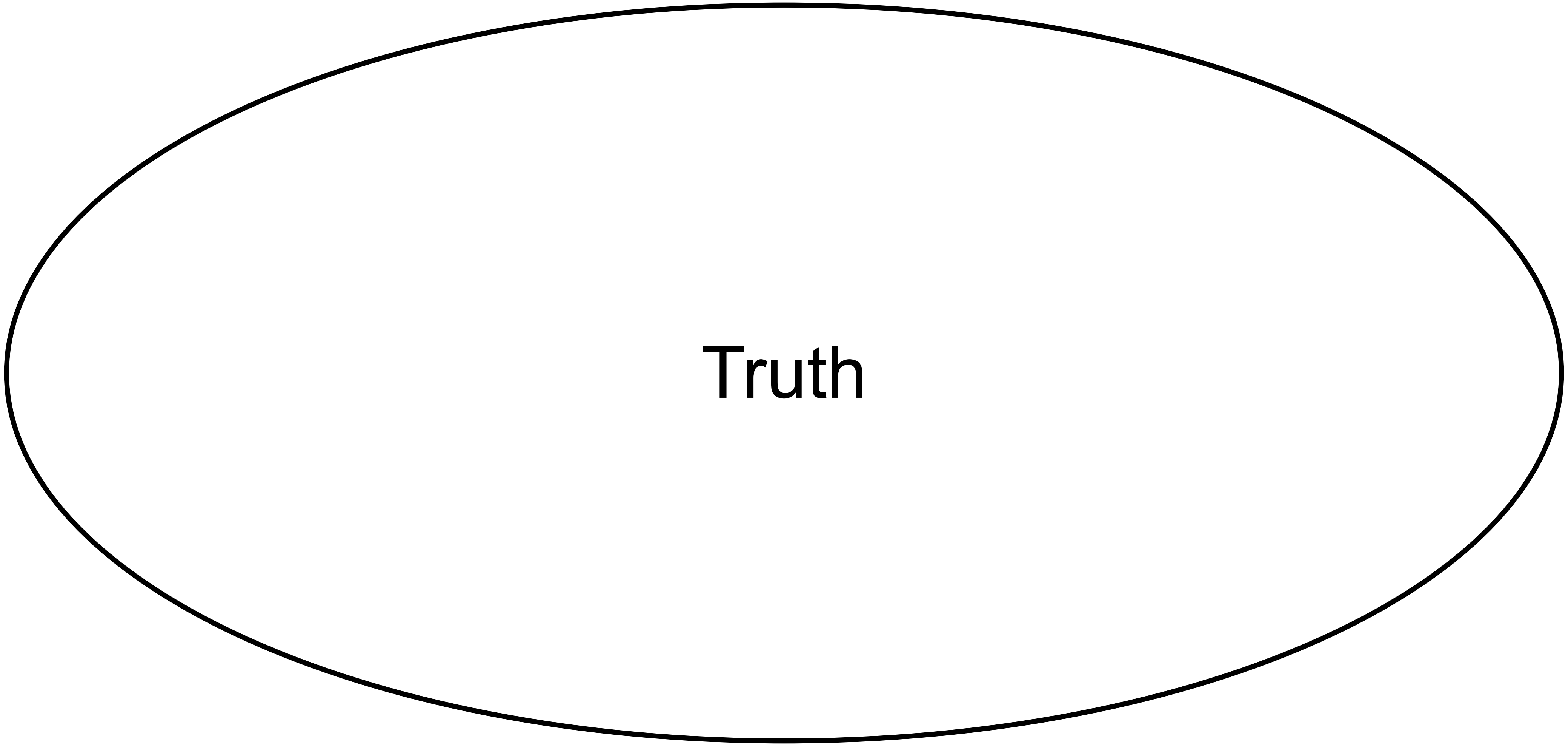
Soundness vs. Completeness

- If an analyzer is ***complete***:



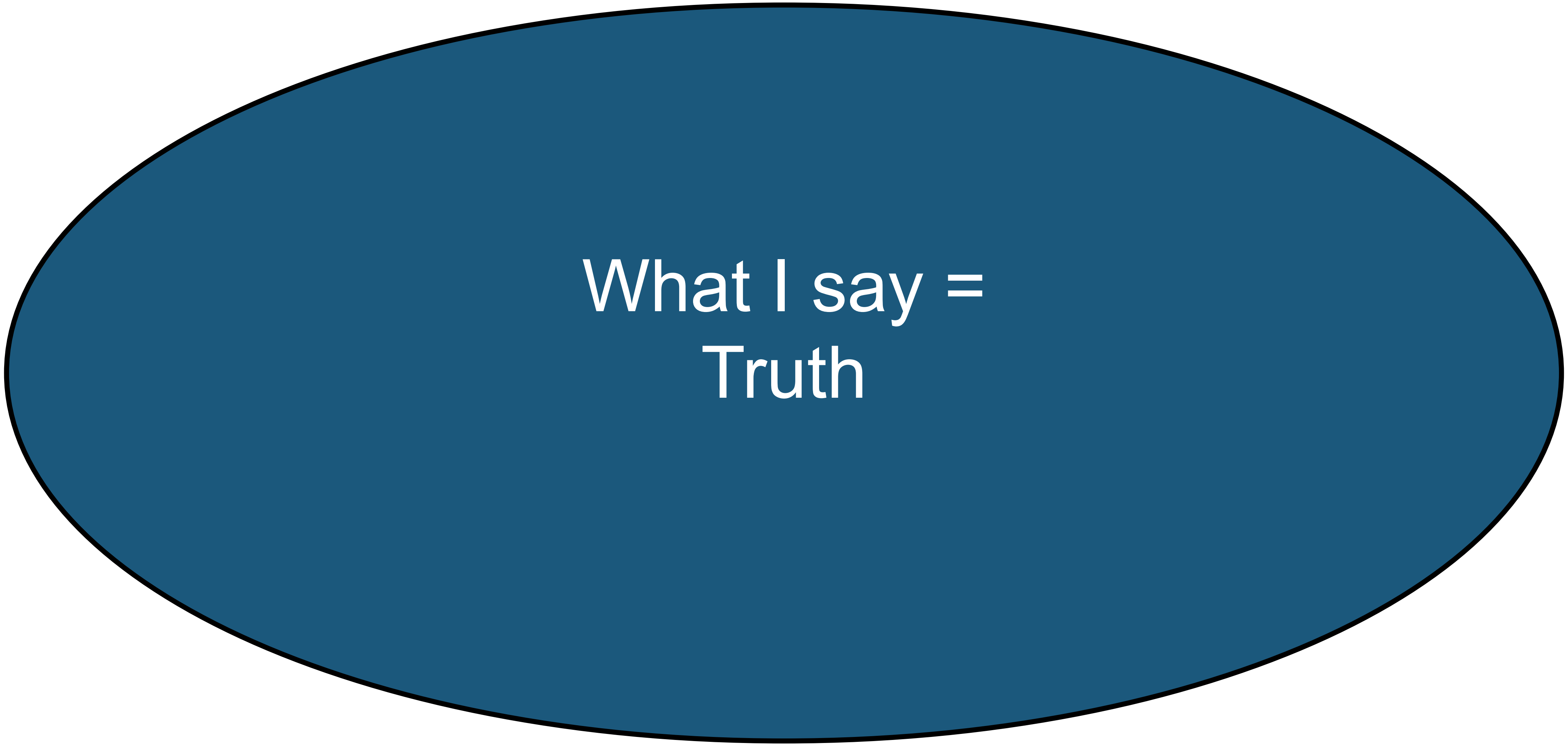
Soundness vs. Completeness

- If an analyzer is ***sound and complete (=perfect)***:

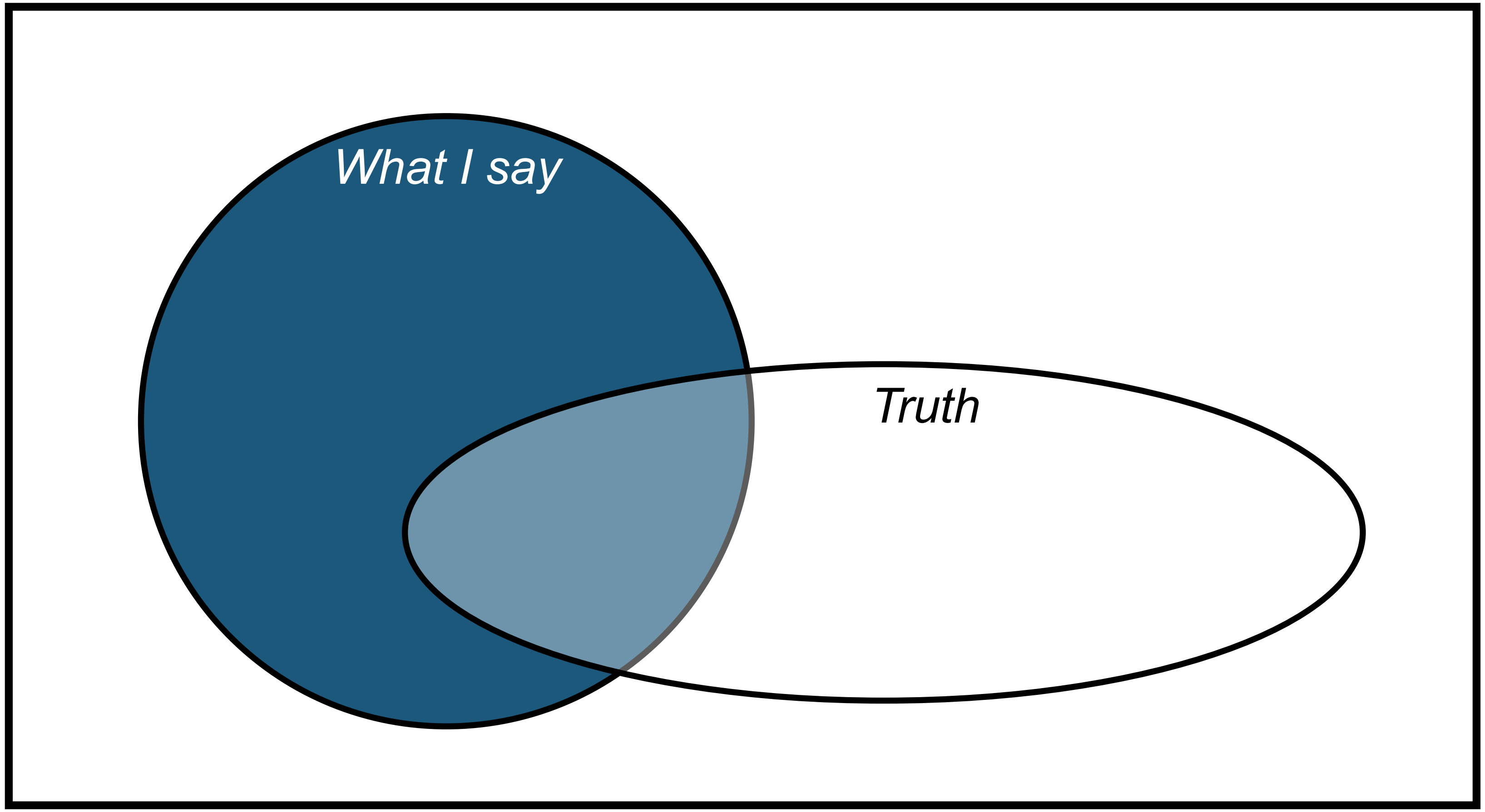


Soundness vs. Completeness

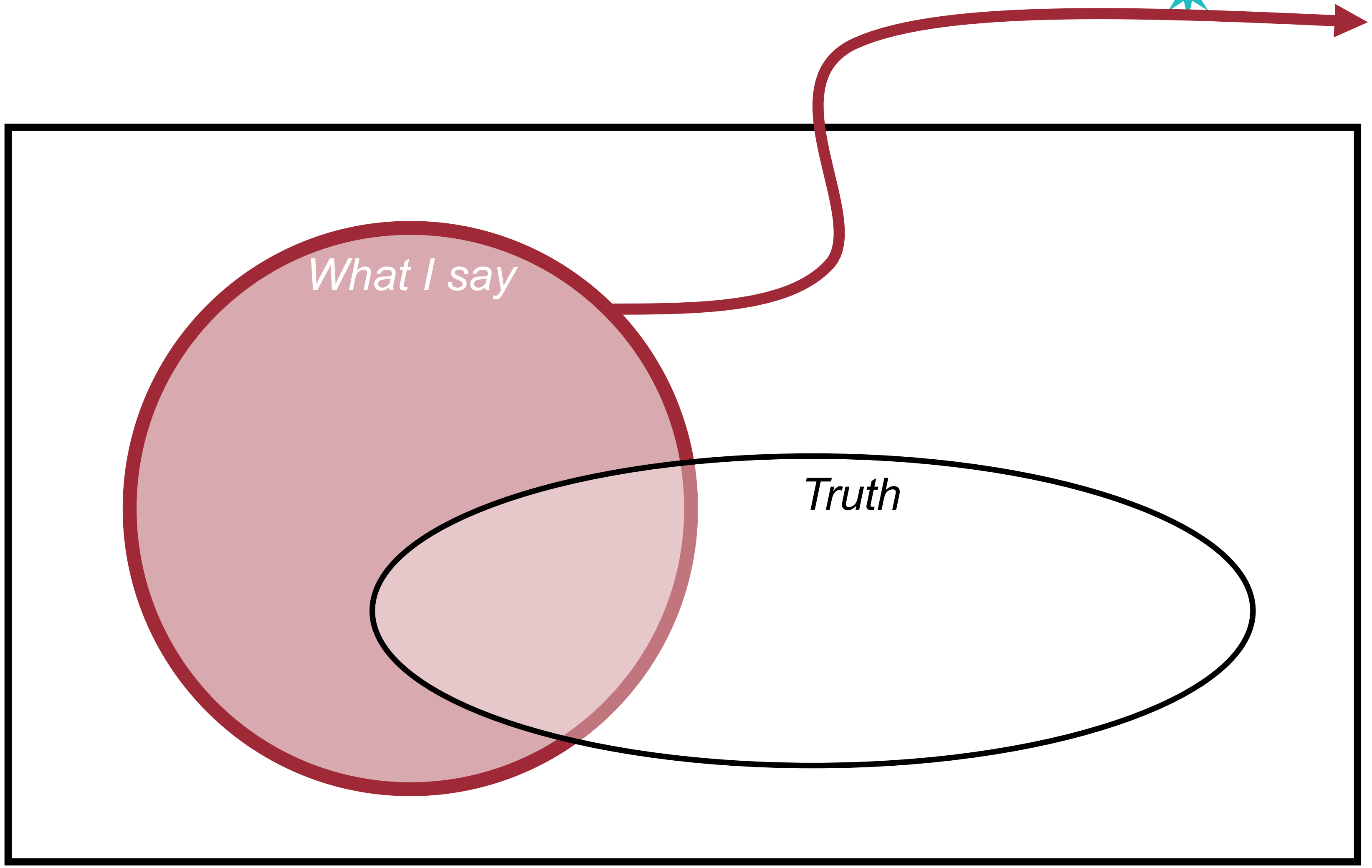
- If an analyzer is ***sound and complete (=perfect)***:



True Positive and False Positive

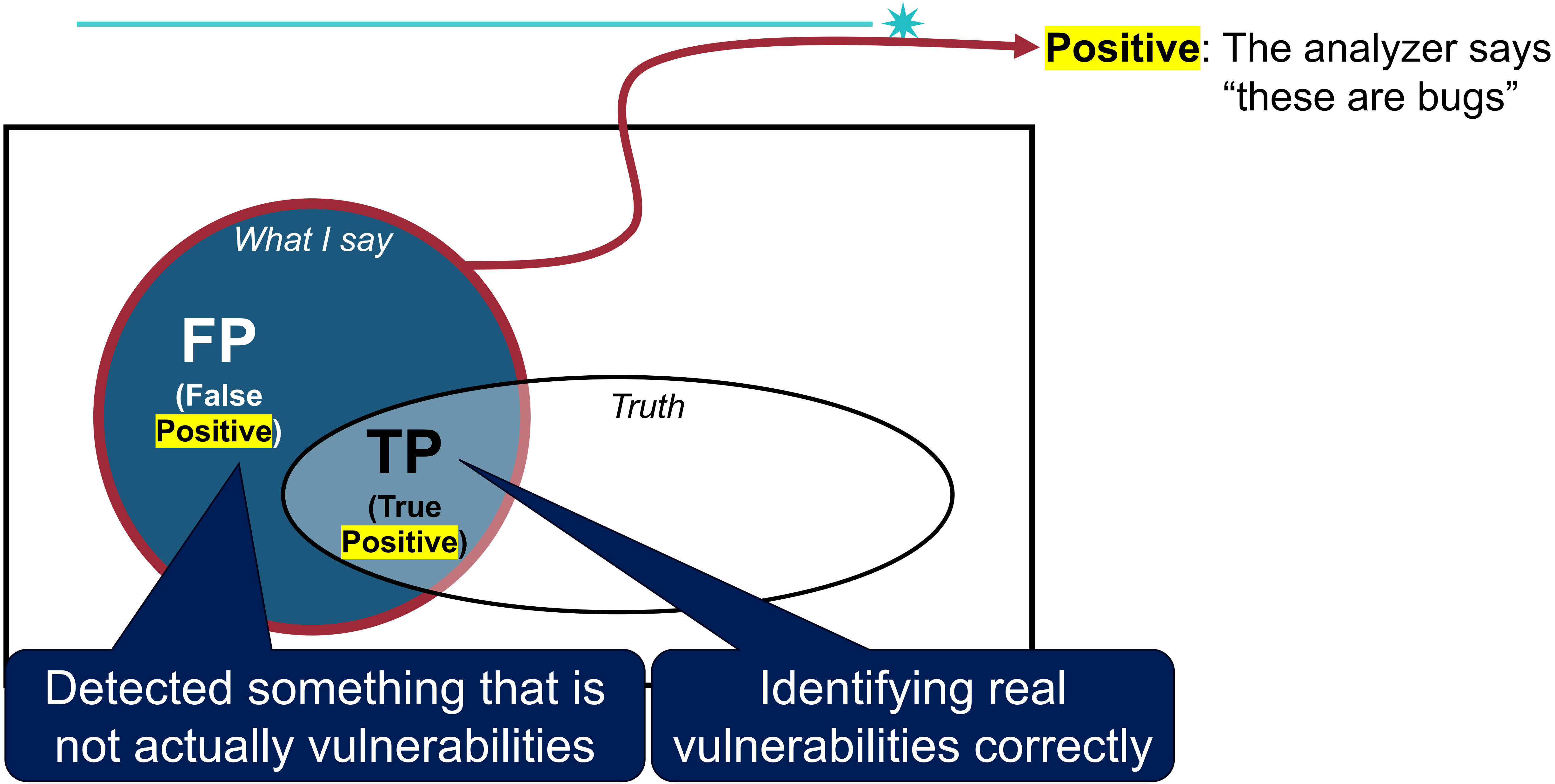


True Positive and False Positive



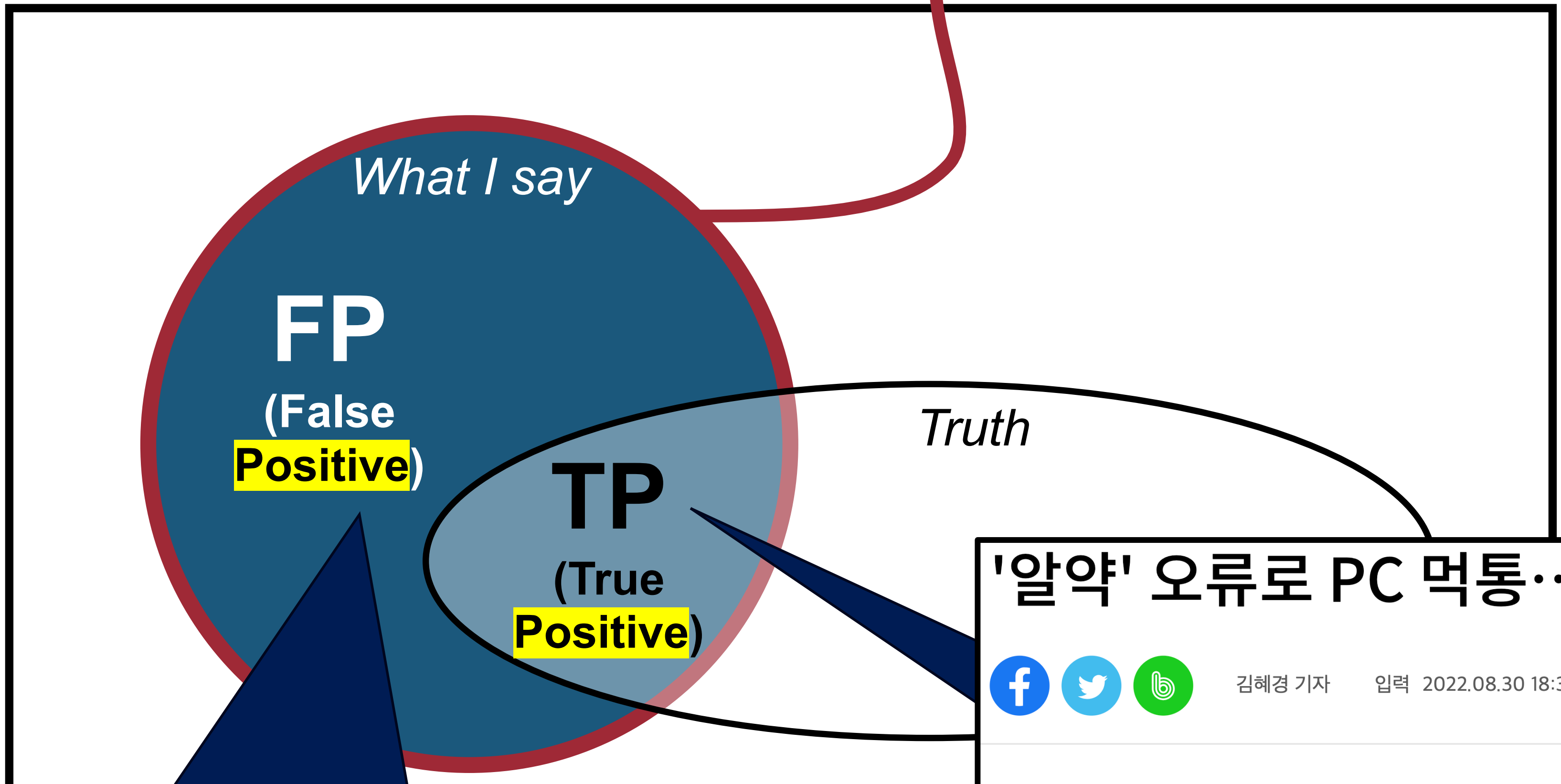
Positive: The analyzer says "these are bugs"

True Positive and False Positive



True Positive and False Positive

Positive: The analyzer says “these are bugs”



Detected something that is not actually vulnerabilities

'알약' 오류로 PC 먹통…정상 프로그램 '랜섬웨어' 오탐

김혜경 기자 입력 2022.08.30 18:38

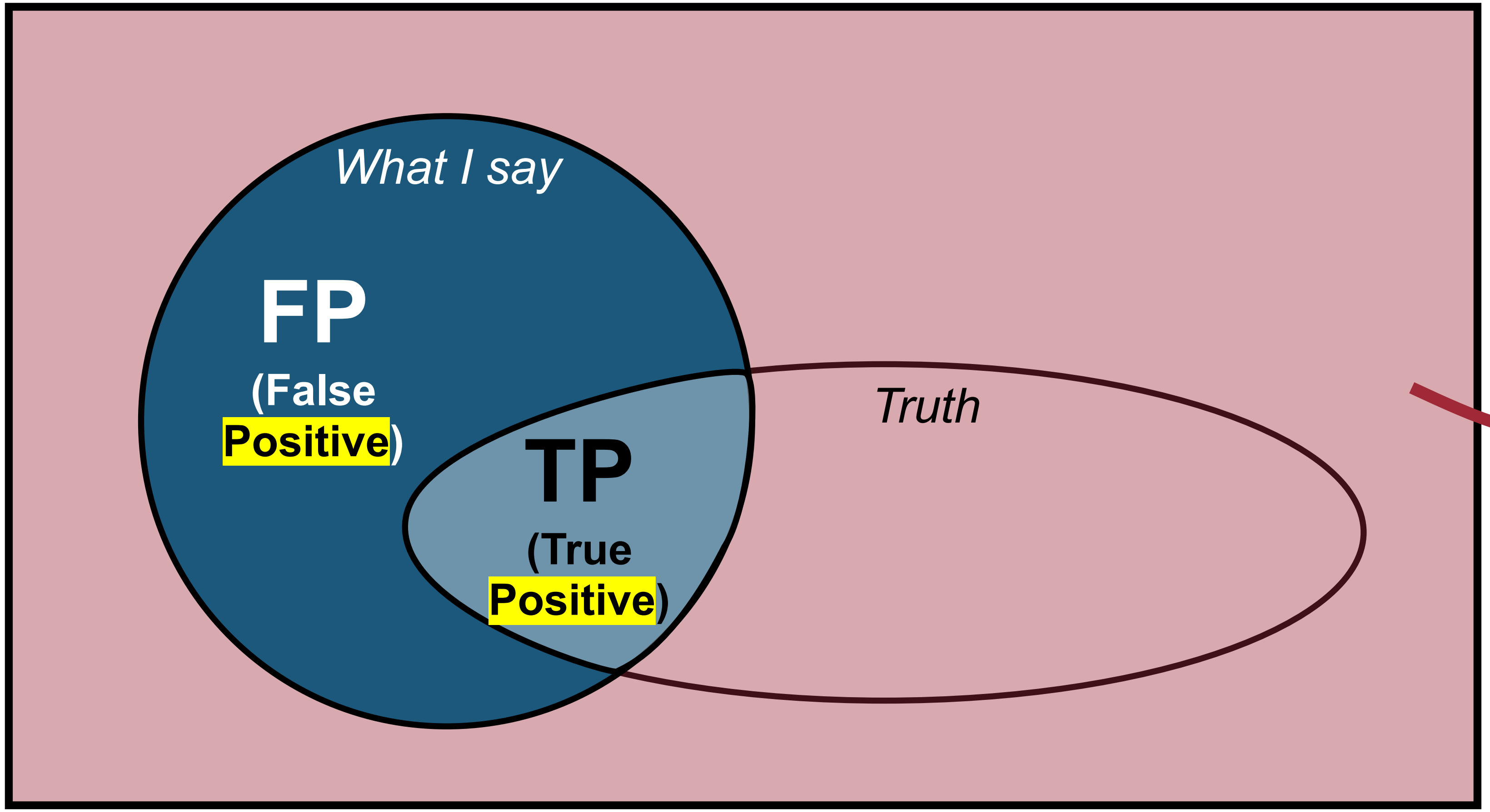
"긴급 대응 중…기업용 제품은 영향 없어"

[아이뉴스24 김혜경 기자] 백신 프로그램 '알약'이 정상 프로그램을 랜섬웨어로 잘못 인식하는 등 오류가 발생하자 개발사 측이 긴급 대응에 나섰다.

False Negatives and True Negatives

Positive: The analyzer says "these are bugs"

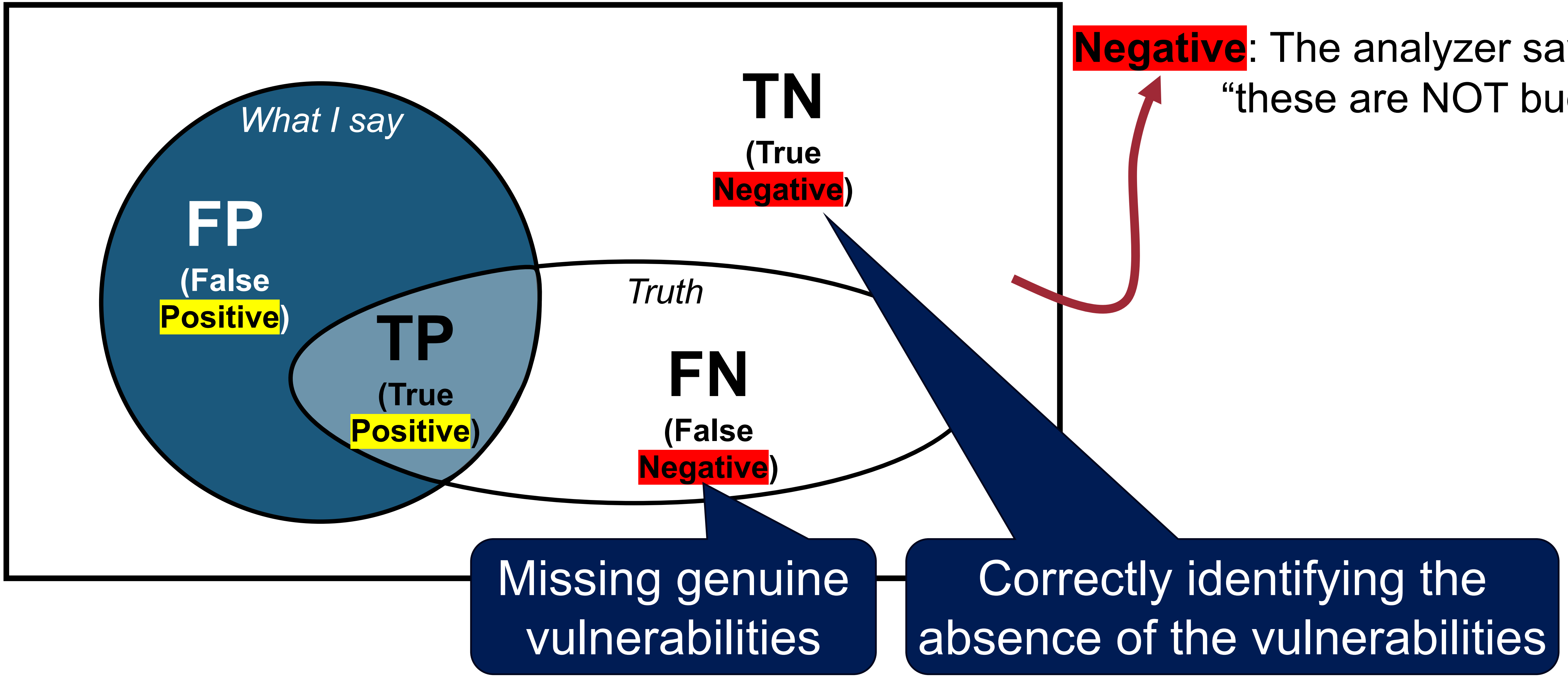
Negative: The analyzer says "these are NOT bugs"



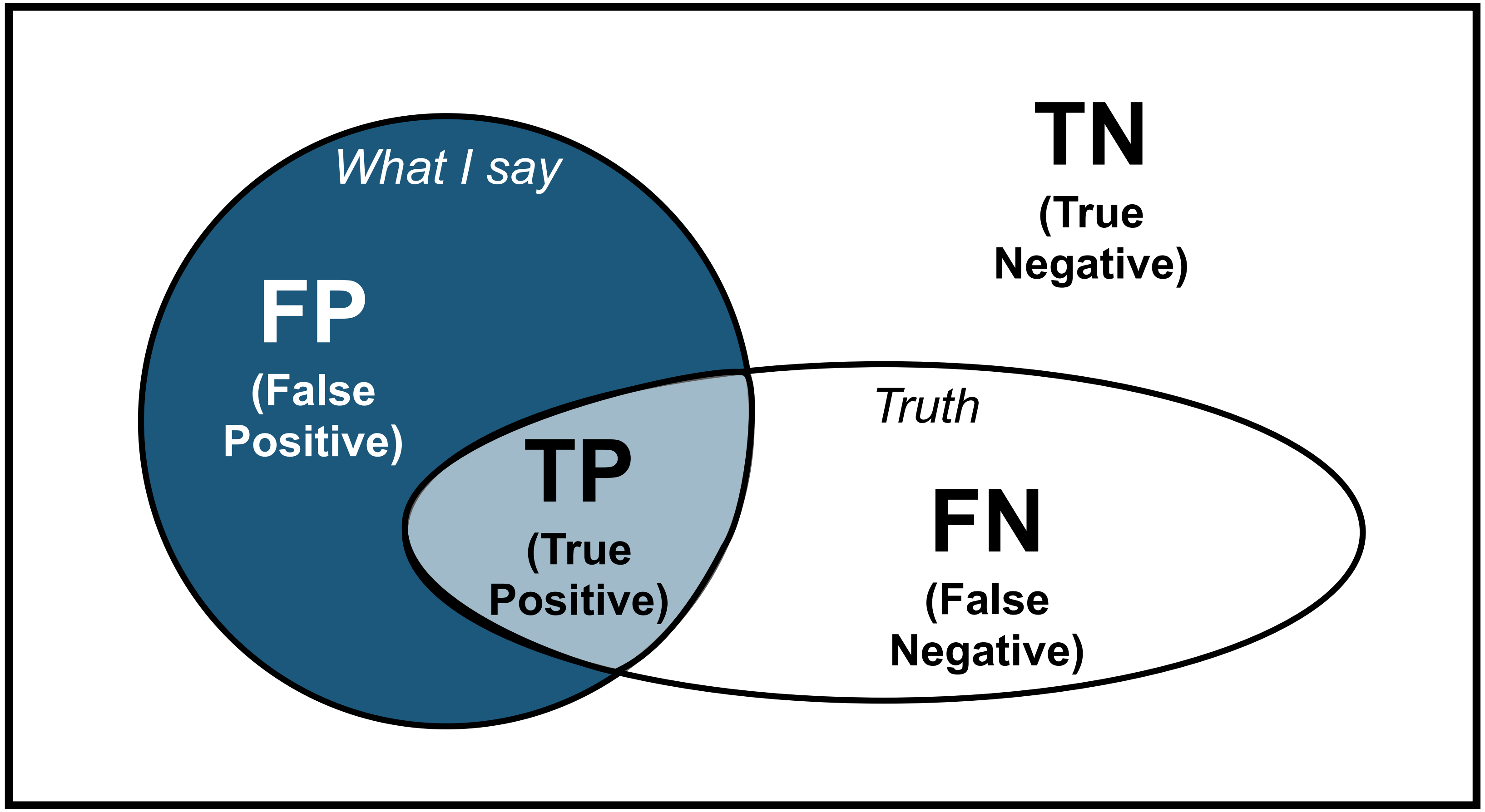
False Negatives and True Negatives

Positive: The analyzer says "these are bugs"

Negative: The analyzer says "these are NOT bugs"

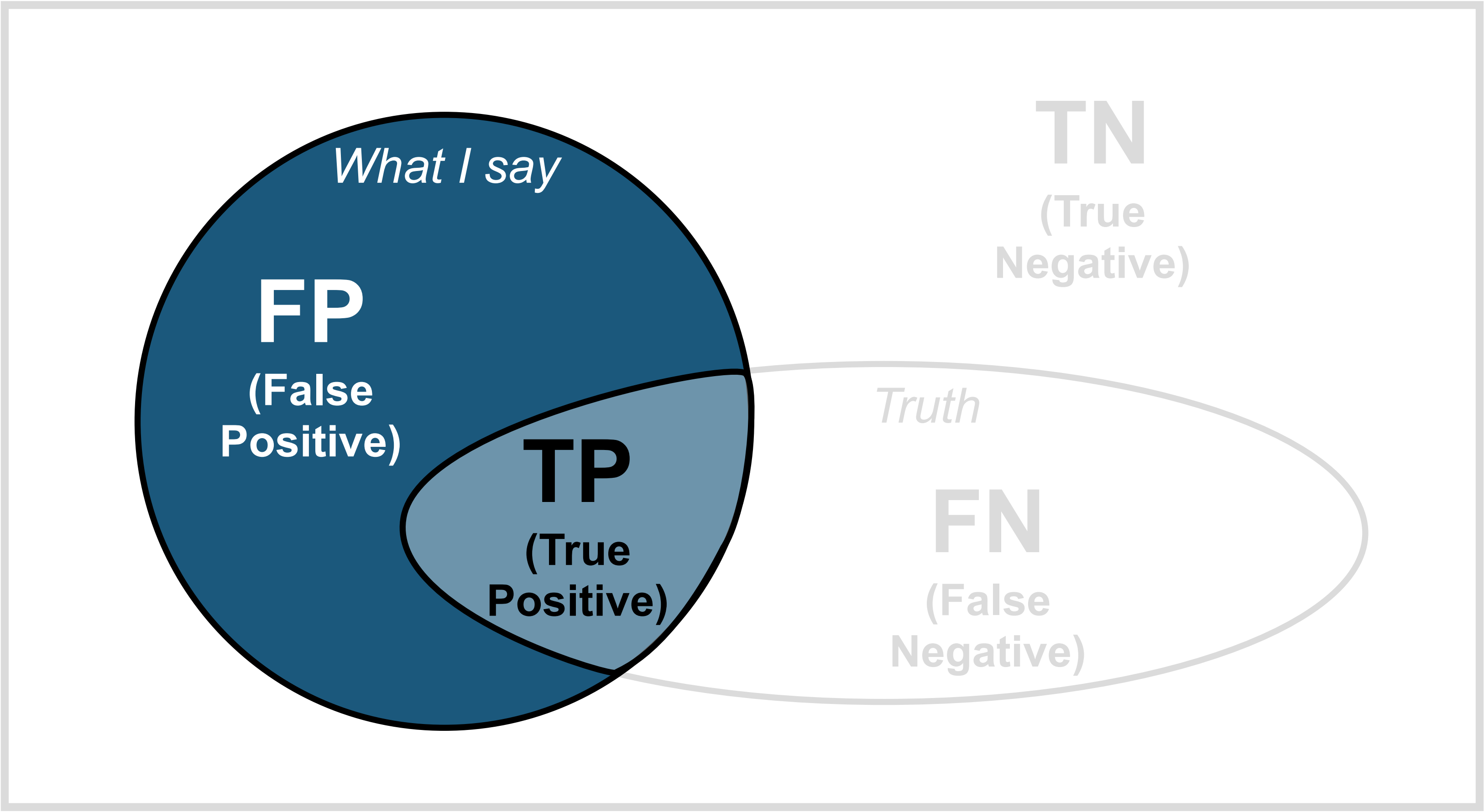


Precision



- Precision
  = TP / (TP + FP)

Precision

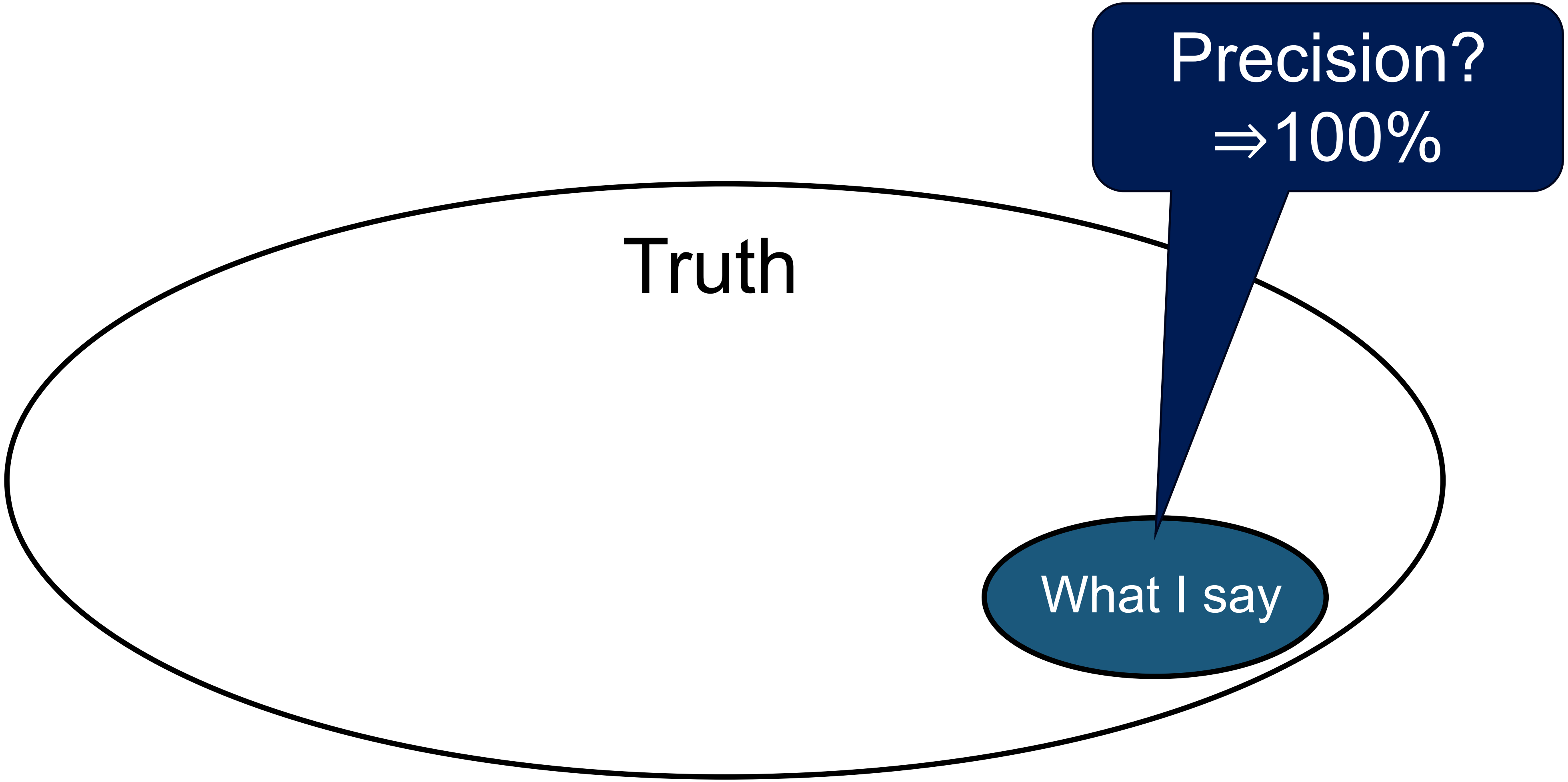


- Precision
  = TP / (TP + FP)

Limitations of Precision Measurement

- If an analyzer is ***sound***:

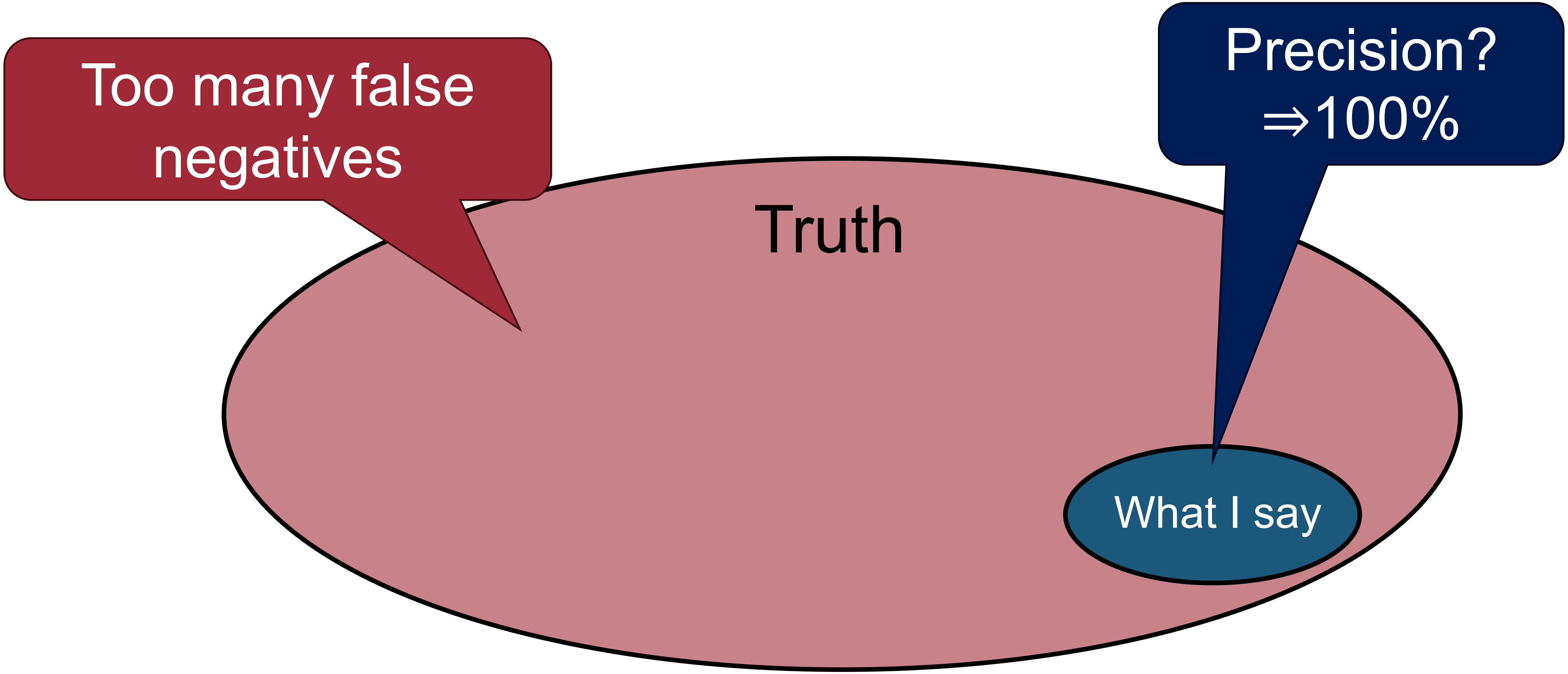
Problems?



Limitations of Precision Measurement

- If an analyzer is ***sound***:

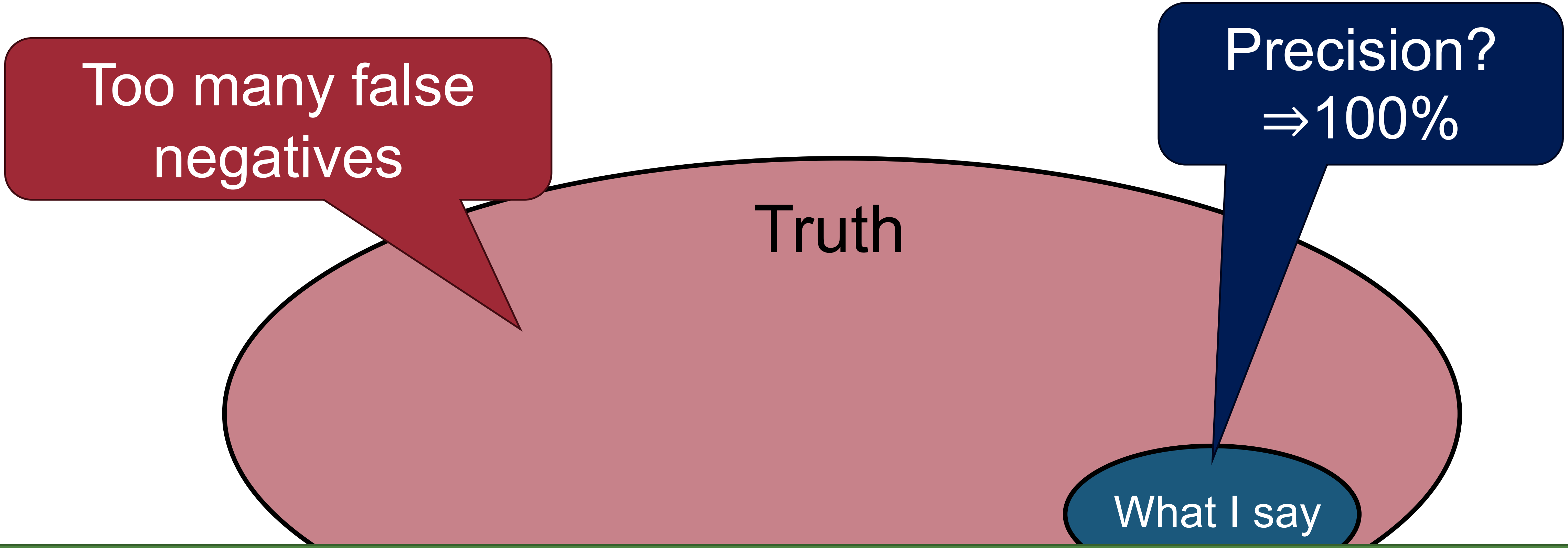
Problems?



Limitations of Precision Measurement

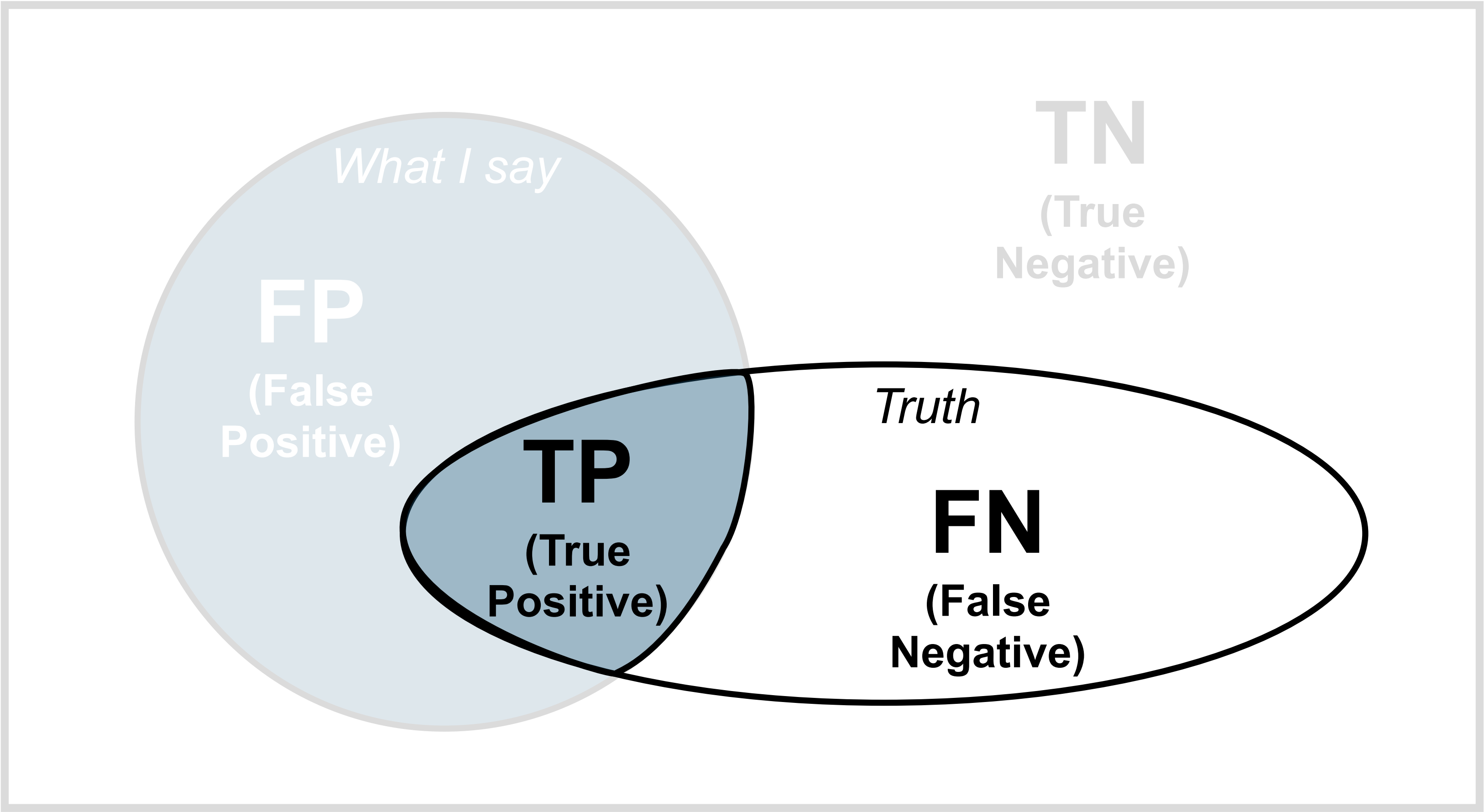
- If an analyzer is ***sound***:

Problems?



When measuring the performance of an analyzer, **the ratio of FN and TP** must also be considered!

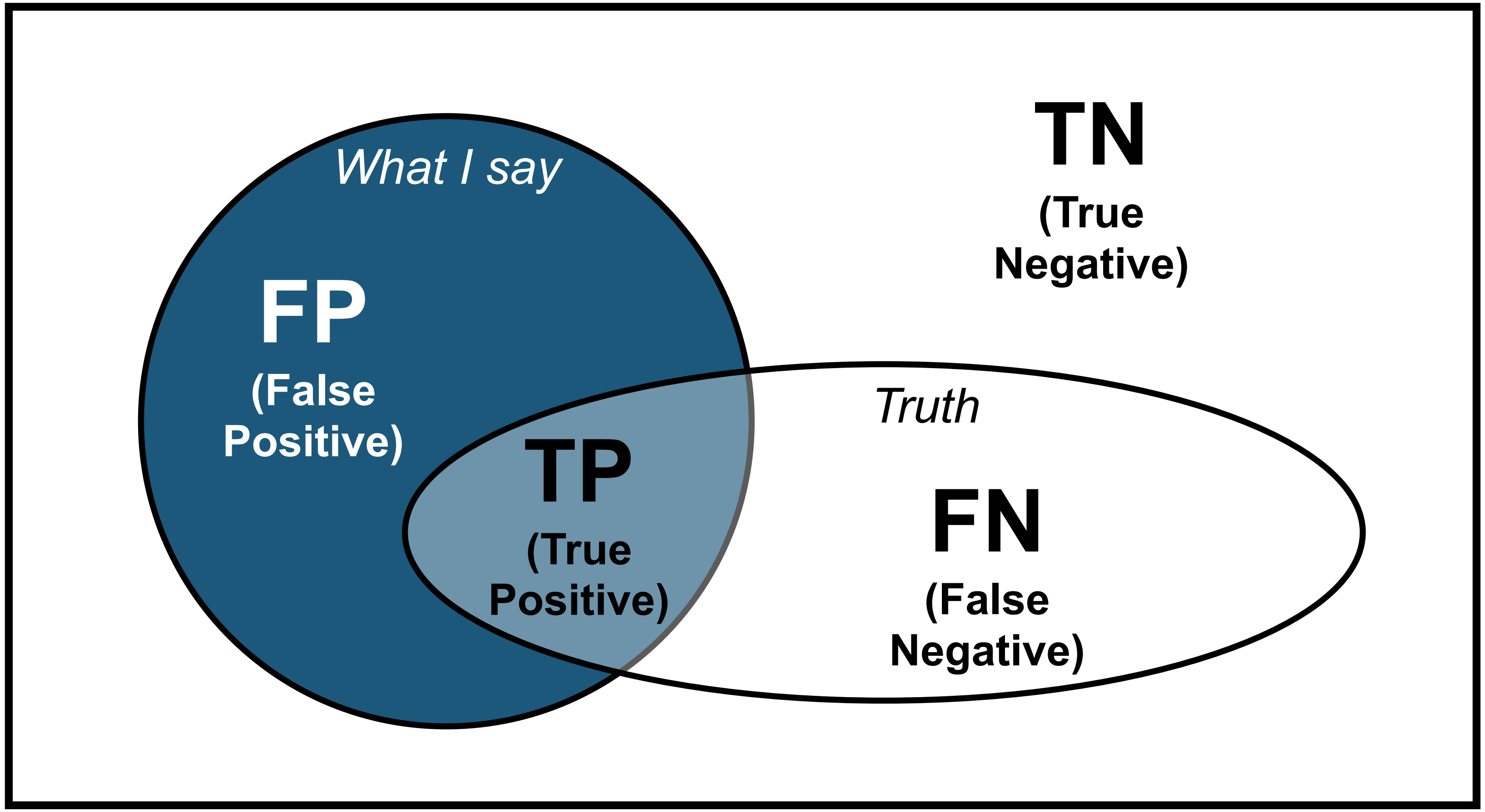
Recall



- Precision
  = TP / (TP + FP)

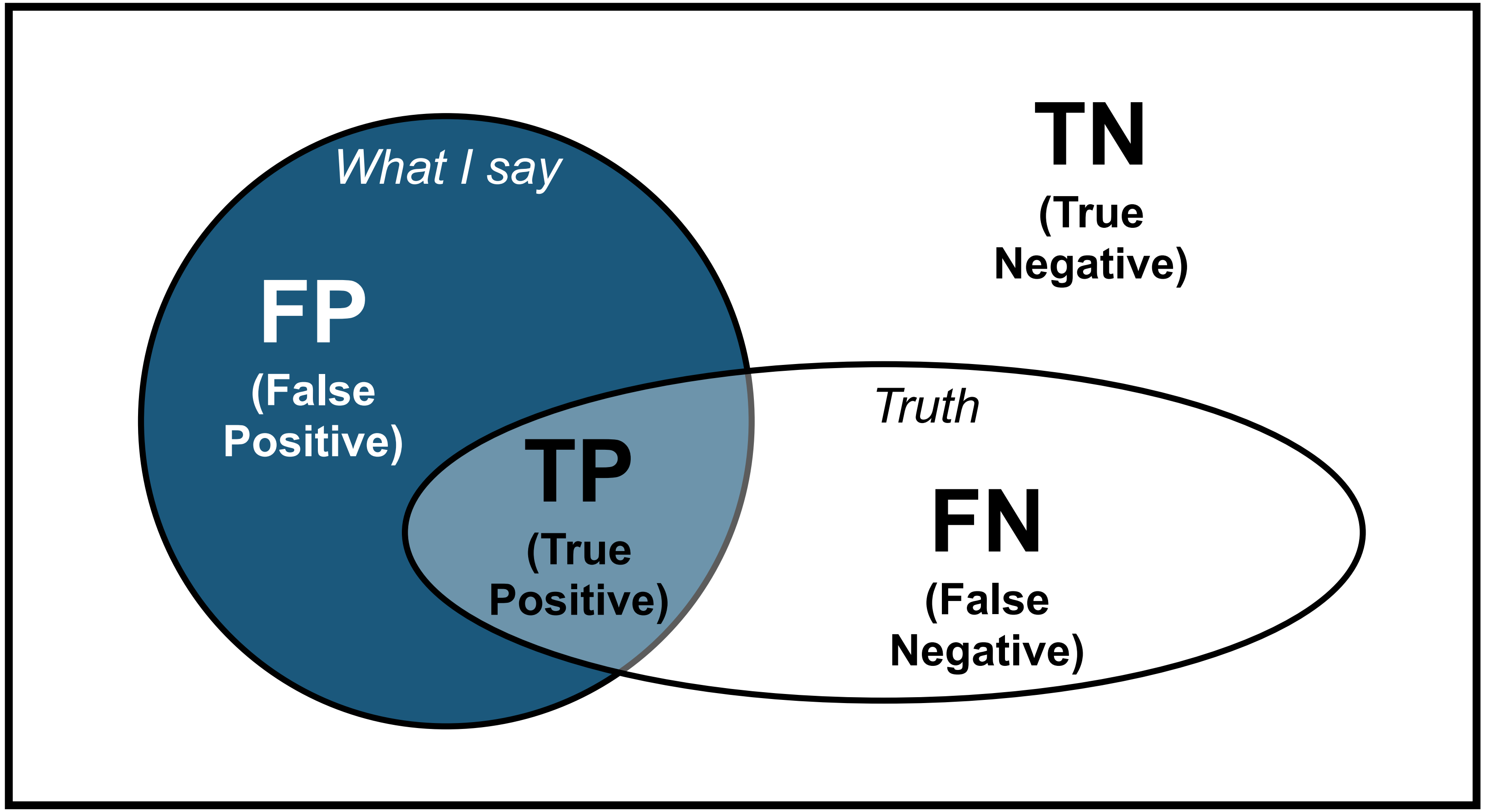
- Recall
  = TP / (FN + TP)

Accuracy



- Precision
  = TP / (TP + FP)
- Recall
  = TP / (FN + TP)
- Accuracy
  = (TP+TN)/
  (TP + FP + FN + TN)

False Positive Rate vs. False Negative Rate



- FP Rate
  = FP / (TP + FP)
- FN Rate
  = FN / (FN + TN)

Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Manual Testing

- “Debug by `printf`”

1. Read documentation and understand functionality
2. Get familiar with the code structure and components
3. Draft test cases that cover requirements from document
4. Review and discuss test cases
5. Execute the test cases
6. Report bugs
7. After bugs are fixed, execute test cases again!

Manual Testing

- Pros
  - Simple to setup for running target programs
  - Gives good feedback if test cases are carefully designed

- Cons
  - Requires manual effort to create each test
  - Tests must be kept up to date as specification evolves

Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

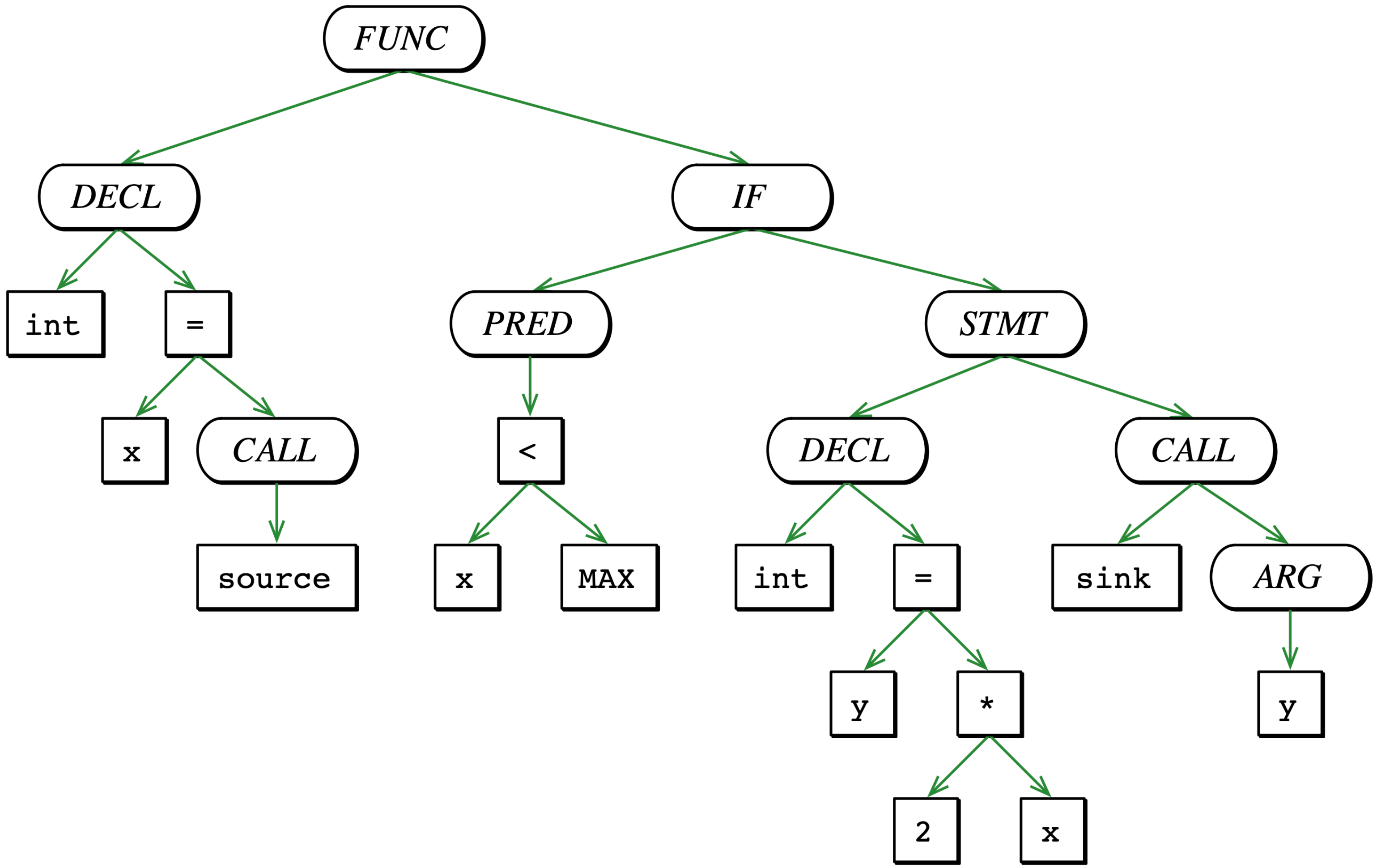
Static Analysis

- Analyze the program ***without executing it*** to detect potential security bugs
- *Abstract (over-approximate)* across ***all possible executions***

- Keywords: (static) taint analysis, (static) symbolic execution, abstract interpretation, abstract syntax tree, control flow graph, data flow graph

Example: Abstract Syntax Tree (AST)

- Syntax information: models a hierarchical decomposition of each statement

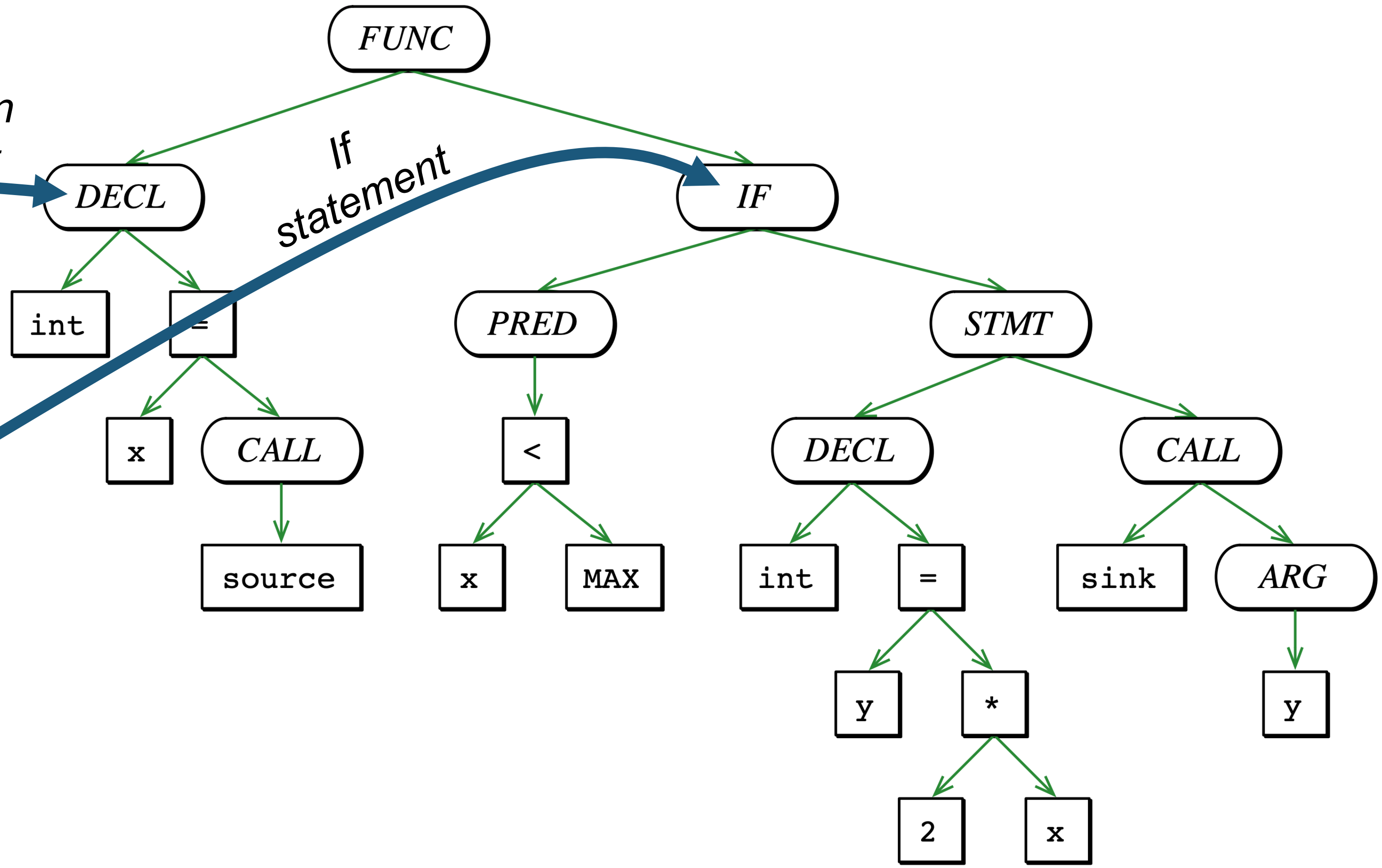



```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```

Example: Abstract Syntax Tree (AST)

- Syntax information: models a hierarchical decomposition of each statement

Declaration statement

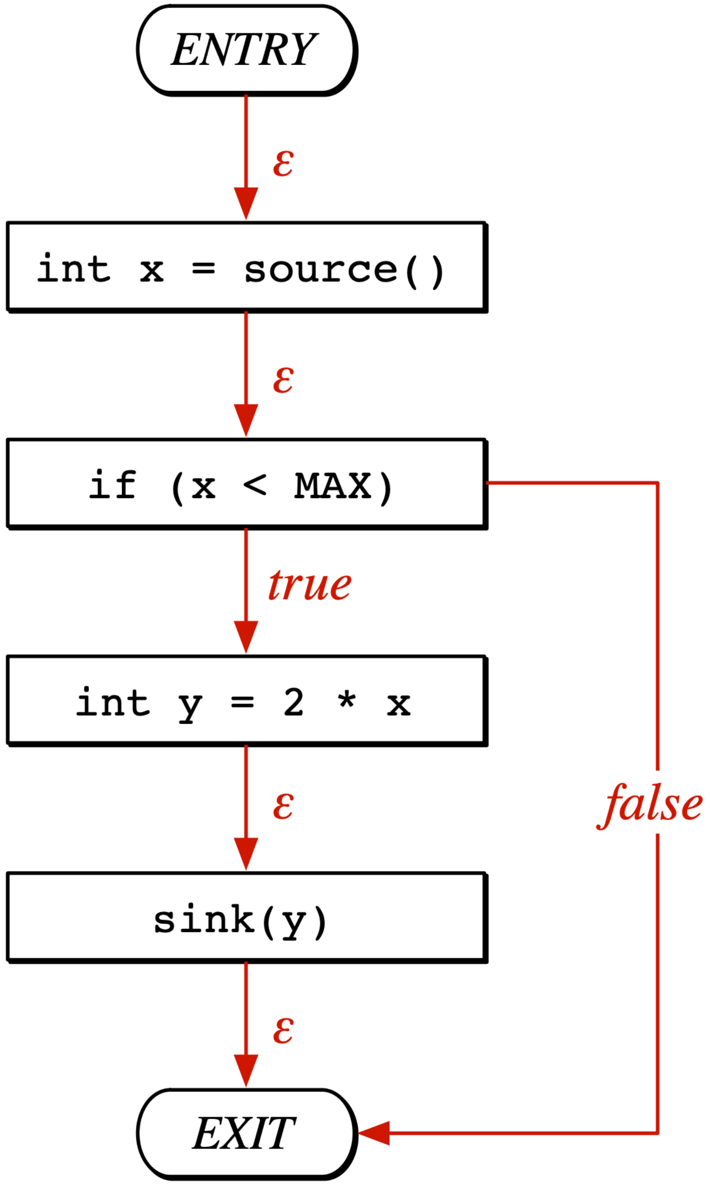



```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```

Example: Control Flow Graph (CFG)

- Semantic information: a program’s control flow among statement

```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```



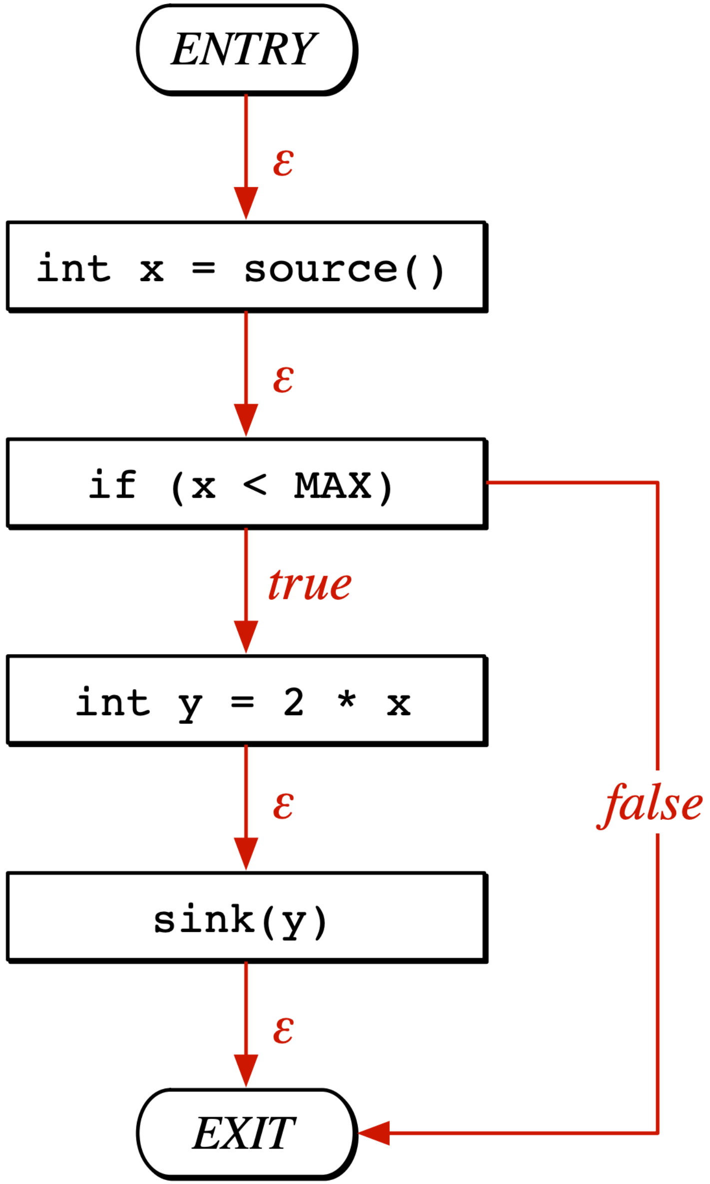
Example: Control Flow Graph (CFG)

- Semantic information: a program's control flow among statement

```
void foo() {
    int x = source();
    if (x < MAX) {
      int y = 2 * x;
      sink(y);
    }
}
```

true

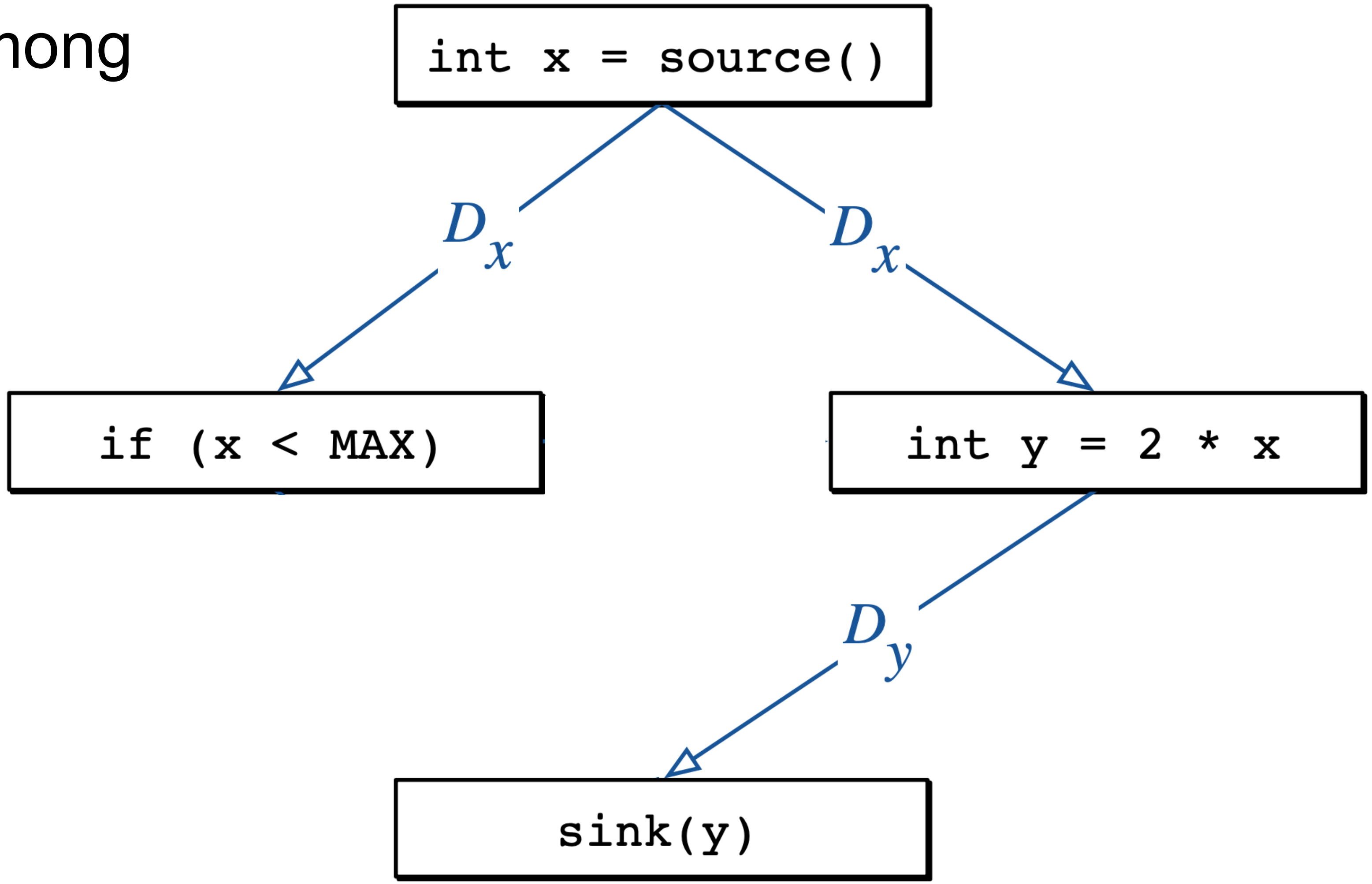
false



Example: Data Flow Graph (DFG)

- Semantic information: a program's data flow among statement

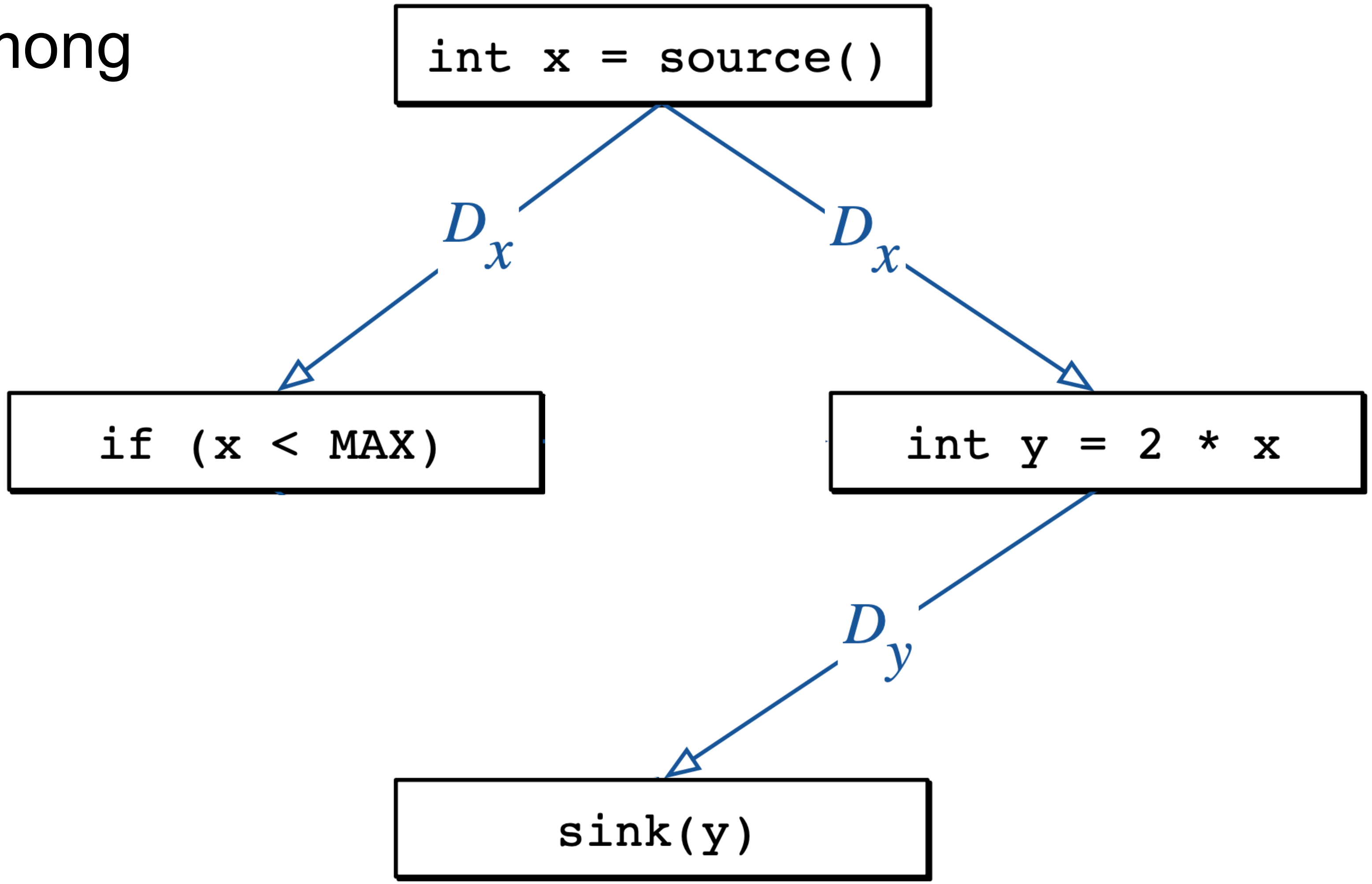
```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```



Example: Data Flow Graph (DFG)

- Semantic information: a program's data flow among statement

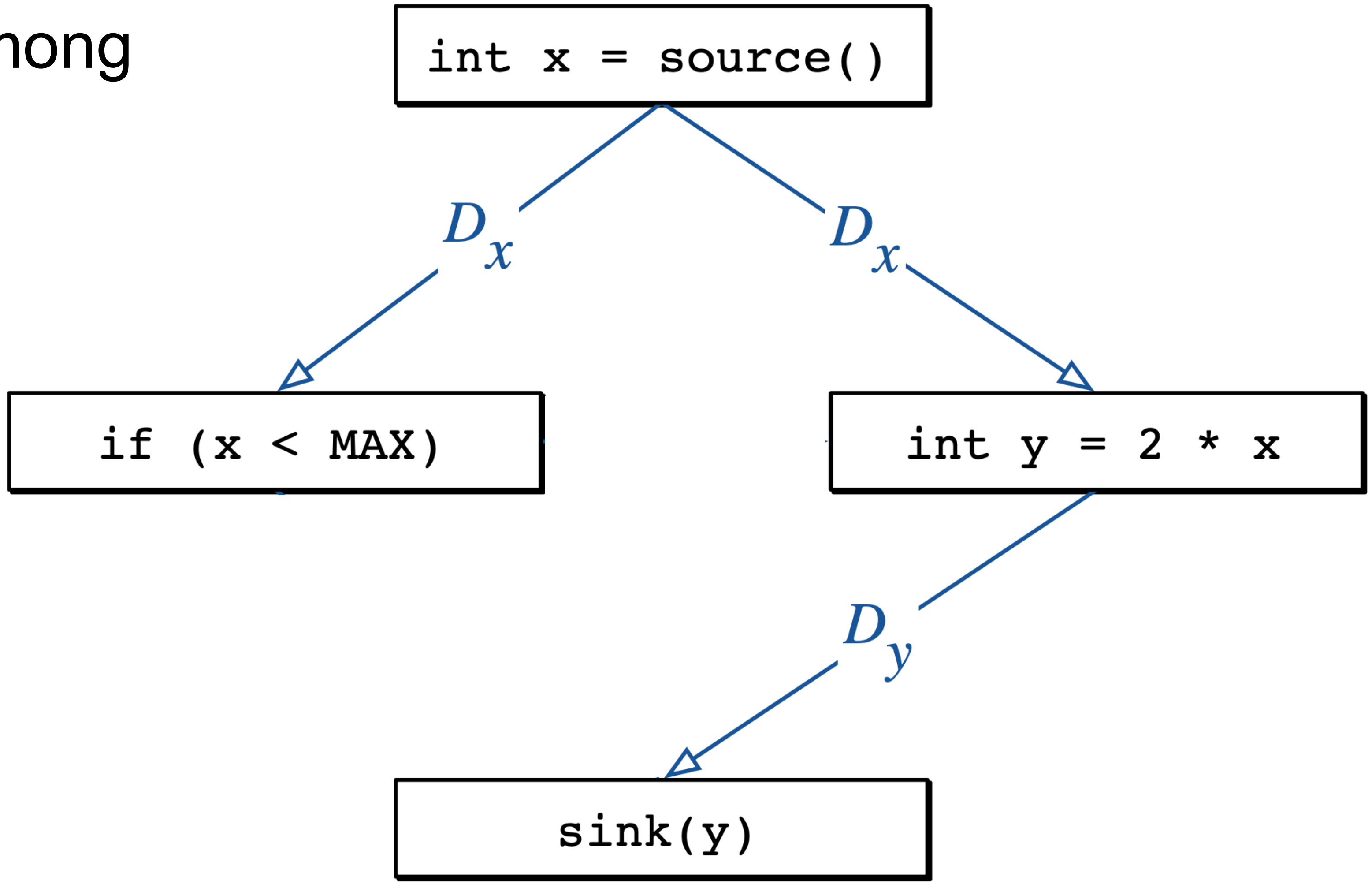
```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```



Example: Data Flow Graph (DFG)

- Semantic information: a program's data flow among statement

```
void foo() {
    int x = source();
    if (x < MAX) {
        int y = 2 * x;
        sink(y);
    }
}
```



Static Analysis

- Pros
  - Save time and resources (we do not need to execute the program)
  - A highly scalable method (it can run on multiple code bases)
  - Aiming for completeness
    - Has a global view of the program
- Cons
  - Requires manual configuration of rules or standards
    - E.g., graph traversal rules for each vulnerability type
  - May have large amounts of false positives

False Positives

- May have spurious alarms because of over-approximation
  - Can be improved by more advanced design

```
void foo() {
    int x = source();
    if (unknown(x)) {
        int y = 2 * x;
        sink(y);
    }
}
```

Dynamically resolved code:

```
if x includes exploit:
    sanitize(x)
```

False Positives

- May have spurious alarms because of over-approximation
  - Can be improved by more advanced design

```
void foo() {
    int x = source();
    if (unknown(x)) {
        int y = 2 * x;
        sink(y);
    }
}
```

Dynamically resolved code:
```
if x includes exploit:
    sanitize(x)
```

The analyzer has no knowledge of the **runtime information**

⇒ Just check the data flow

The analyzer will say that "this is a potential bug"

Three Forms of Testing

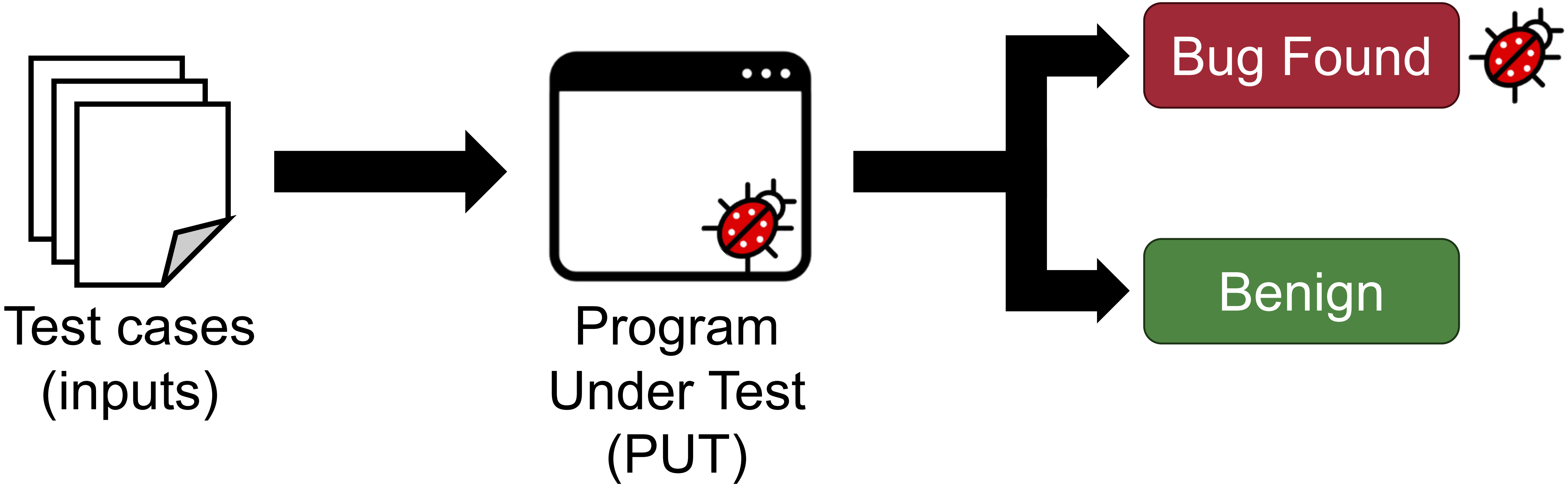
- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

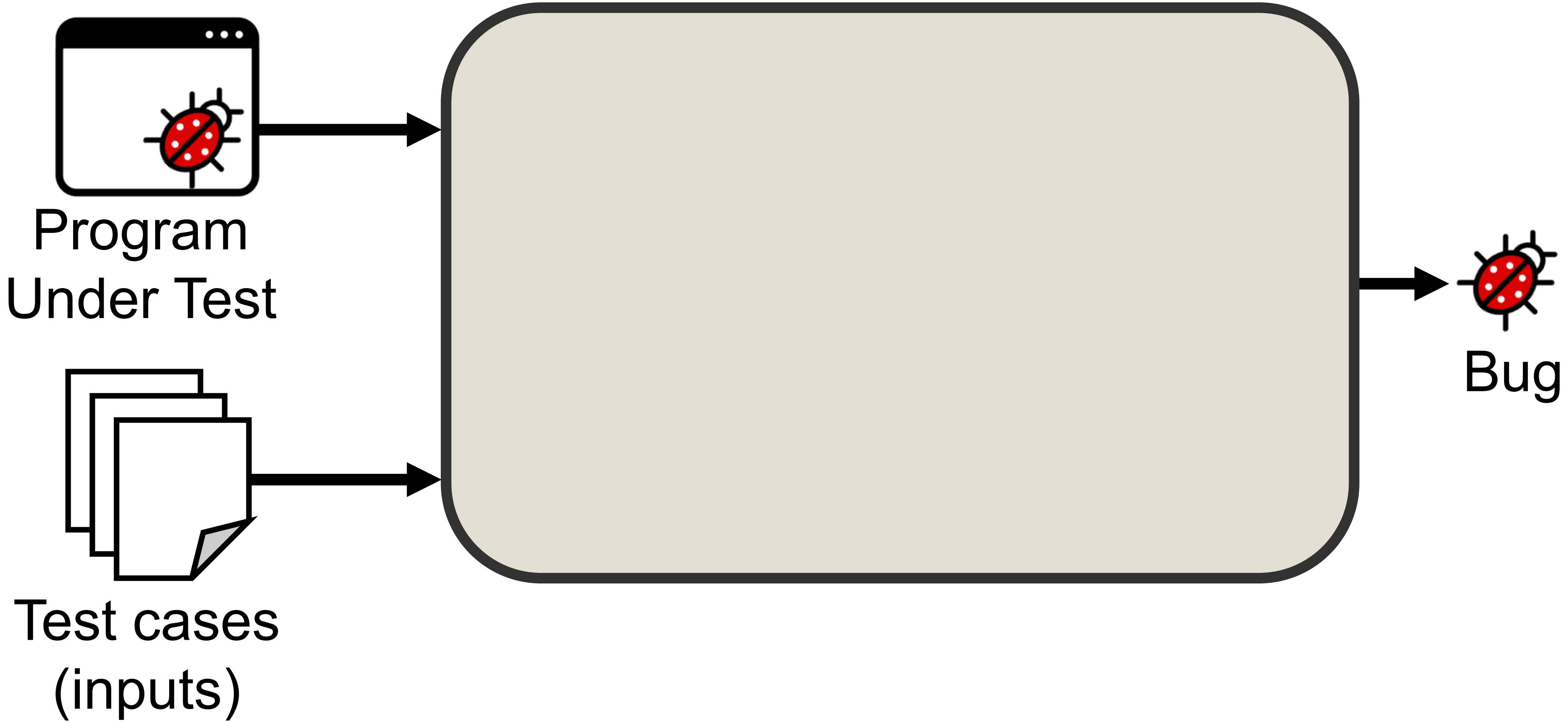
Dynamic Analysis

- Analyze the program **during an execution** with the concrete input
  - Focuses on a single concrete run
- Keywords: **fuzzing**, penetration testing, scanner, concolic execution, dynamic taint analysis



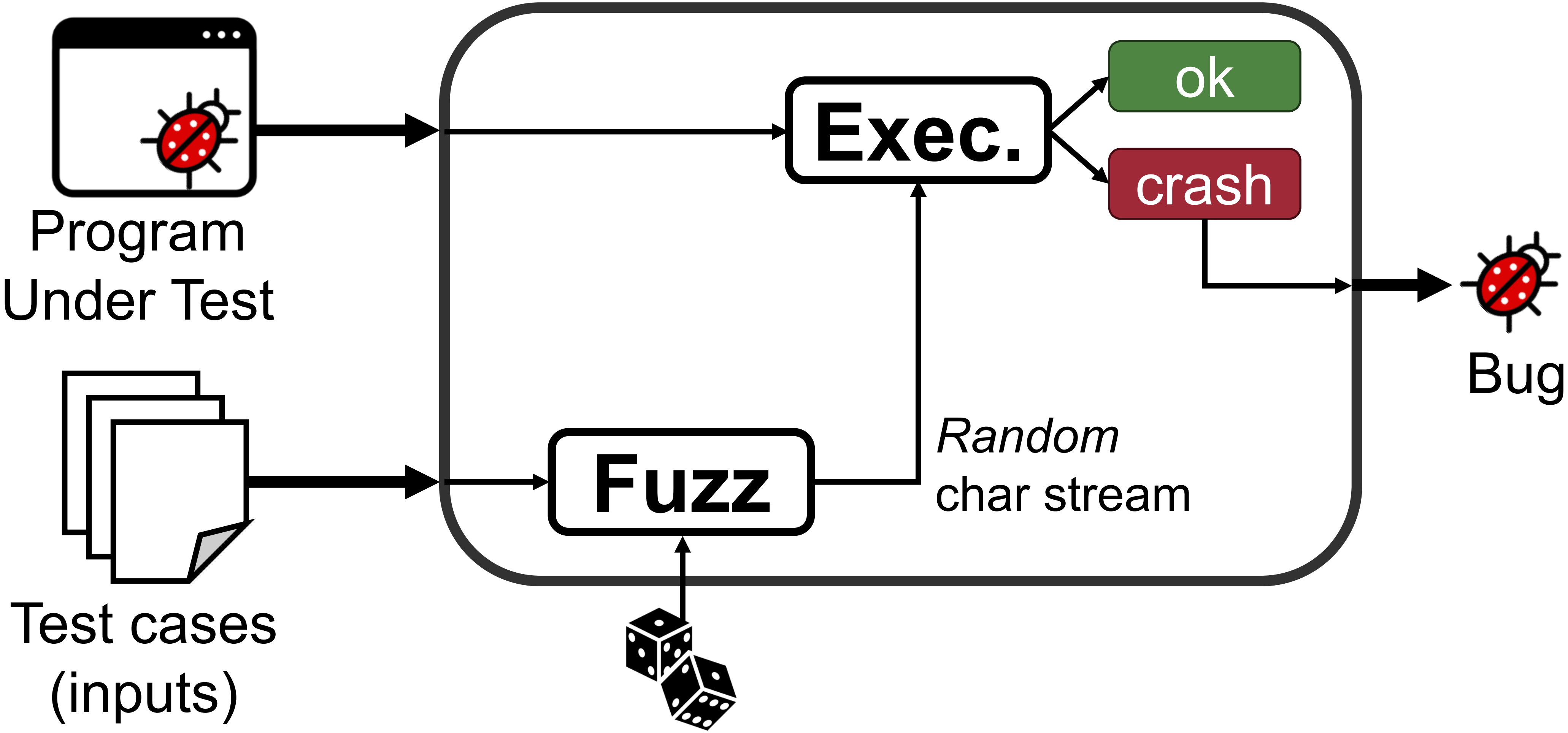
Example: Fuzzing

- Initially, developed by Barton Miller in 1990



Example: Fuzzing

- Initially, developed by Barton Miller in 1990



Fuzzing is …

- Simple, and popular way to find security bugs
- Used by security practitioners
- Research questions:
  - Why fuzzing works so well in practice?
  - Are we maximizing the ability of fuzzing?

Dynamic Analysis

- Pros
  - False positives are rare
    - Because it considers dynamically resolved information
- Cons
  - Not scalable
  - Testing is incomplete ⇒ produces many false negatives
    - The limited focus on a given (generated/mutated) inputs

Conclusion

- Software testing finds bugs before an attacker can exploit them!

- Building a perfect analyzer is impossible

- Manual testing
- Static analysis – Next Lecture!
- Dynamic analysis – Next and Next Lecture!