CMSC414 Computer and Network Security

Program Analysis for Security

Yizheng Chen | University of Maryland
surrealyz.github.io

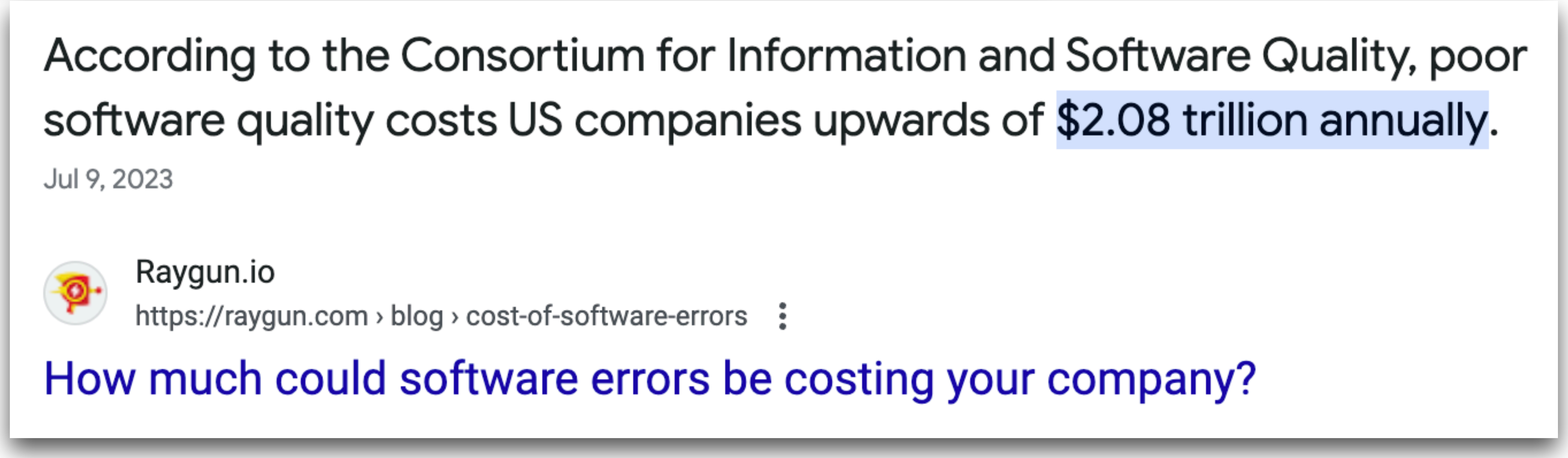
Mar 5, 2024

Software Security is a major problem!

A widely cited 2002 study prepared for NIST reported that even though 50 percent of software development budgets go to testing, flaws in software still cost the U.S. economy $59.5 billion annually. Nov 9, 2010

National Institute of Standards and Technology (.gov)
https://www.nist.gov › news-events › news › 2010/11

Updated NIST Software Uses Combination Testing to Catch ...



According to the Consortium for Information and Software Quality, poor software quality costs US companies upwards of $2.08 trillion annually.

Jul 9, 2023

Raygun.io
https://raygun.com › blog › cost-of-software-errors

How much could software errors be costing your company?

Not all bugs are equal!

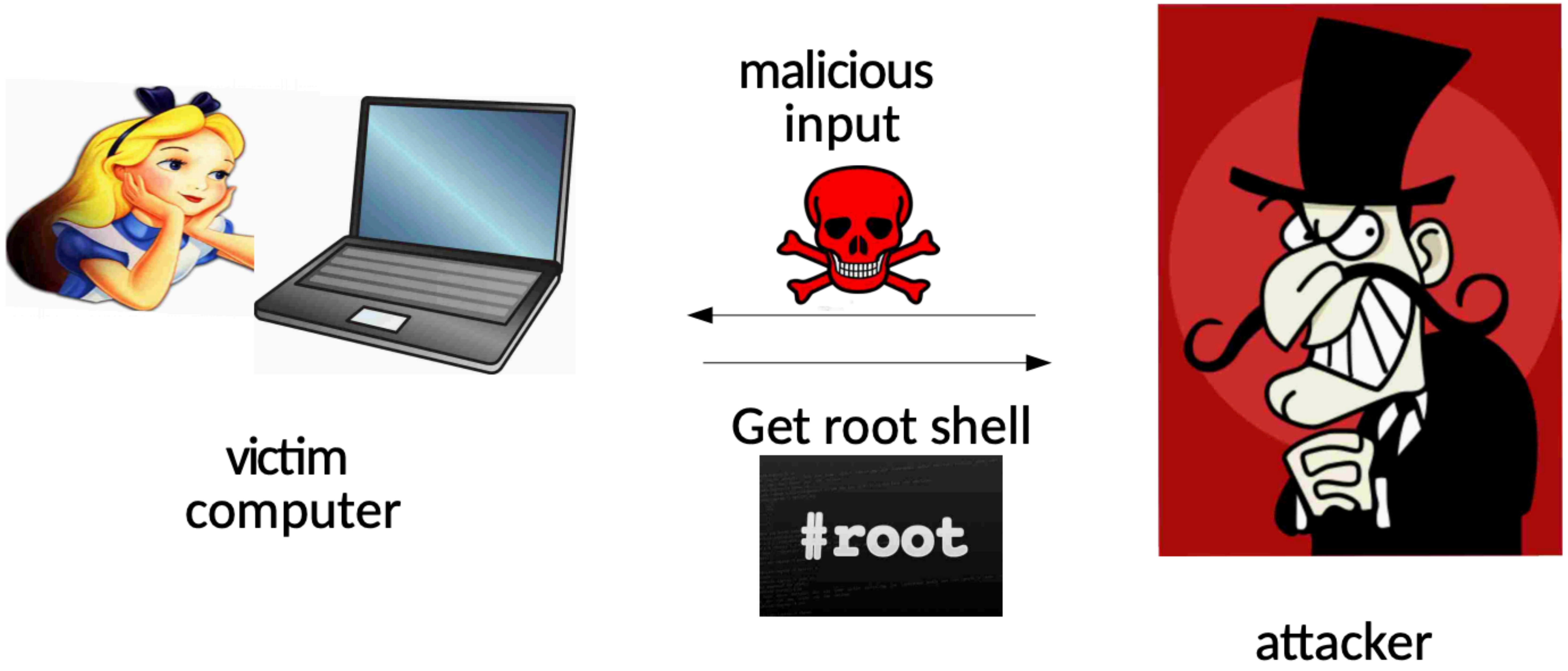
Benign functional bugs

vs.

Security bugs

Why are security bugs more dangerous than other bugs?

Why security bugs are more dangerous?



malicious input

victim computer

Get root shell

attacker

Security bugs allow attackers to cause serious damages: take over machines remotely, steal secrets, etc.

How do we deal with security bugs?

- Monitor a system at runtime to detect and prevent exploits of bugs
    - Reminder: ensure complete mediation
- Accept that programs will have bugs and design the system to minimize damages
    - Example: Sandboxes, privilege separation
- Automatically find and fix bugs

SANDBOXES

Execution environment that restricts what an application running in it can do

Example: Native Client (NaCl)

- Native Client (NaCl) is a **secure sandbox** for running untrusted native machine code in the Chrome browser
- **Special restrictions** on the generated code
- Chrome apps can embed NaCl modules into their pages
  - Chrome apps examples: meeting, chat, kindle reader, writer, Microsoft office online, etc.
  - NaCL module examples: image processing, PDF render

SANDBOXES

Execution environment that restricts what an application running in it can do

NaCl's restrictions

Takes arbitrary x86, runs it in a sandbox in a browser

- Restrict applications to using a narrow API
- Data integrity: No reads/writes outside of sandbox
- No unsafe instructions
- CFI (control flow integrity): insure that all control transfers in the program text target an instruction
- Identified during disassembly

SANDBOXES

Execution environment that restricts what an application running in it can do

NaCl's restrictions

Takes arbitrary x86, runs it in a sandbox in a browser

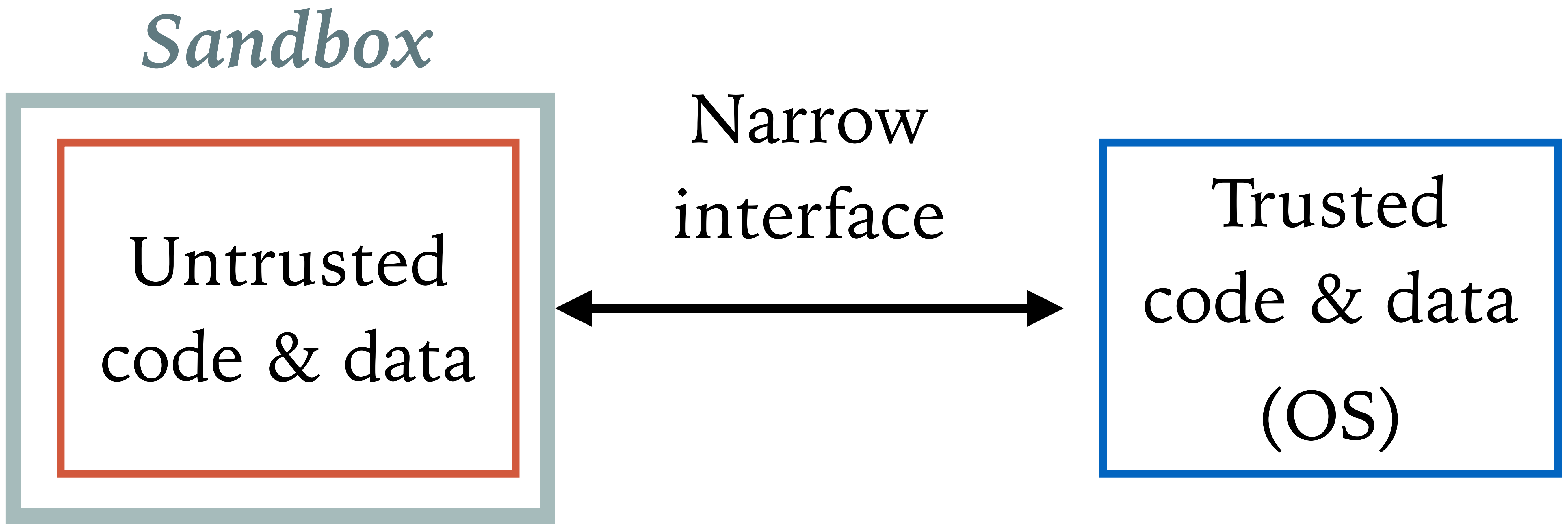
- Restrict applications to using a narrow API
- Data integrity: No reads/writes outside of sandbox
- No unsafe instructions
- CFI

Chromium's restrictions

Runs each webpage's rendering engine in a sandbox

- Restrict rendering engines to a narrow "kernel" API
- Data integrity: No reads/writes outside of sandbox (incl. the desktop and clipboard)

Sandbox mental model



Sandbox

Untrusted code & data

Narrow interface

Trusted code & data (OS)

All data and syscalls must be accessed via the narrow i/f

Can access data
Can make syscalls

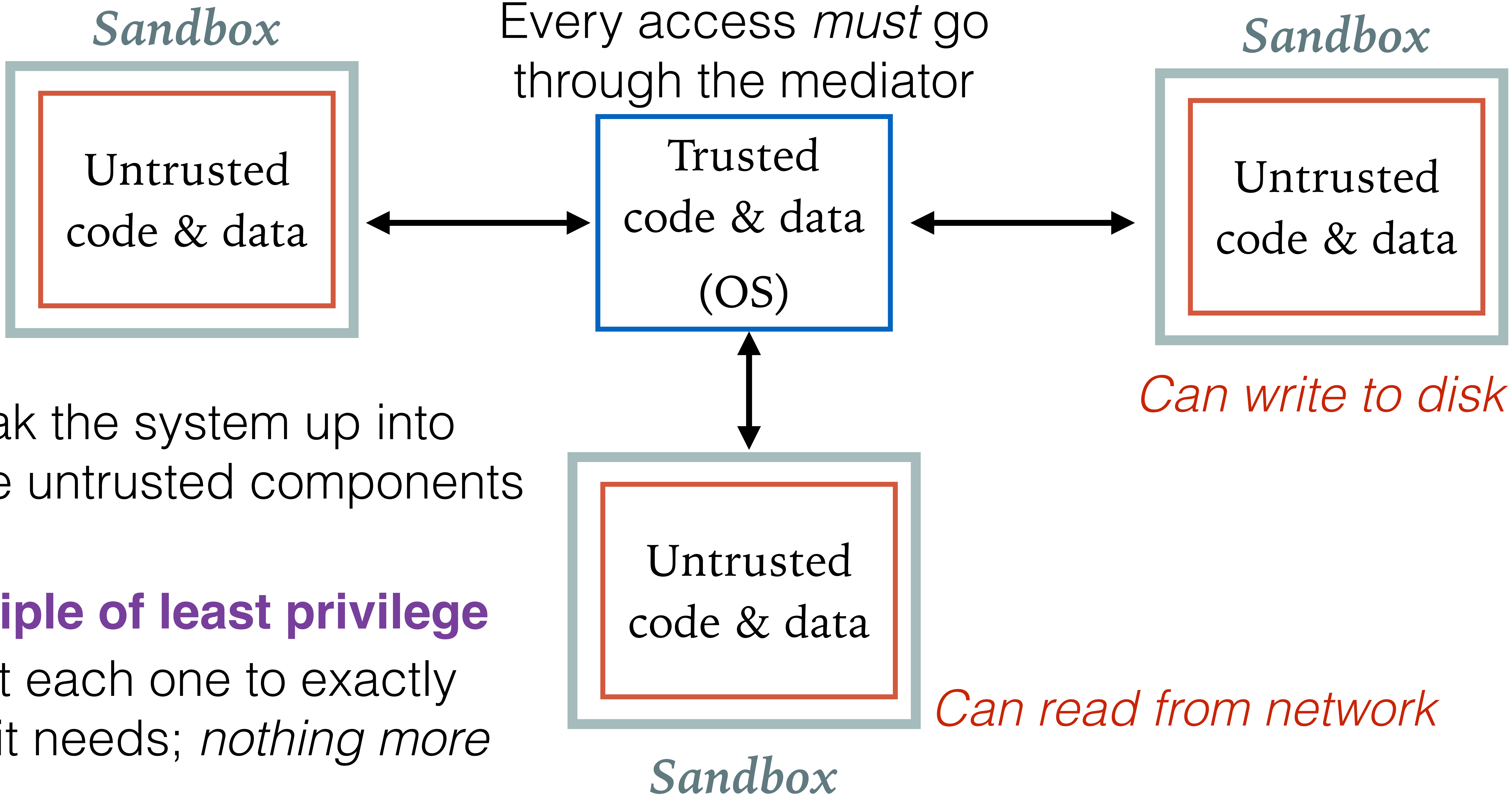
- Even the untrusted code needs input and output

- The goal of the sandbox is to constrain what the untrusted program can do:
  - What it can execute
  - What data it can access
  - What system calls it can make, etc.

Sandbox mental model

Ensure complete mediation

Every access *must* go through the mediator



Can write to disk

Can read from network

Break the system up into multiple untrusted components

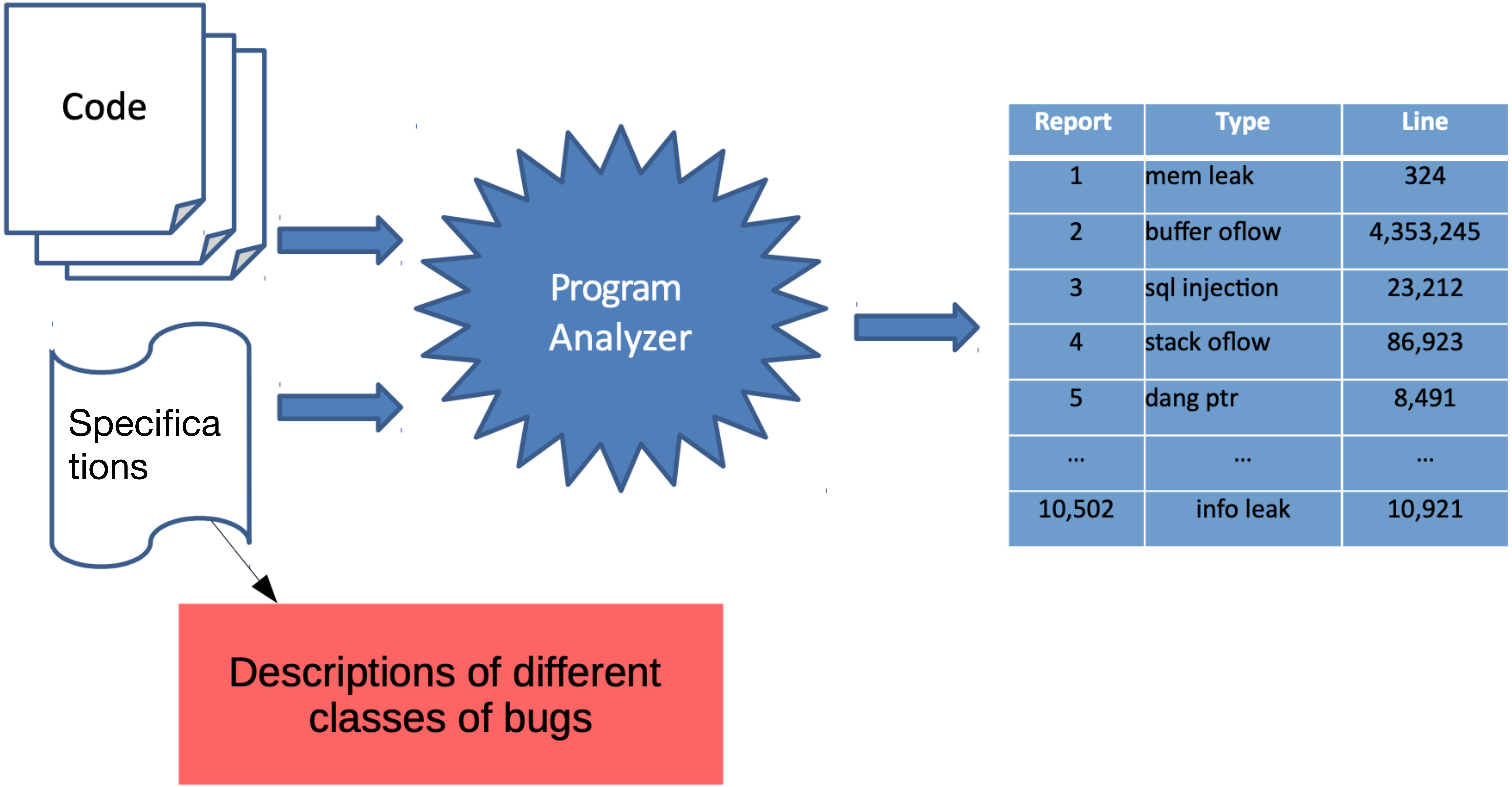
Principle of least privilege

Limit each one to exactly what it needs; *nothing more*

How do we deal with security bugs?

- Monitor a system at runtime to detect and prevent exploits of bugs
  - Reminder: ensure complete mediation
- Accept that programs will have bugs and design the system to minimize damages
  - Example: Sandboxes, privilege separation
- **Automatically find and fix bugs**

Finding bugs with Program analyzers



Code

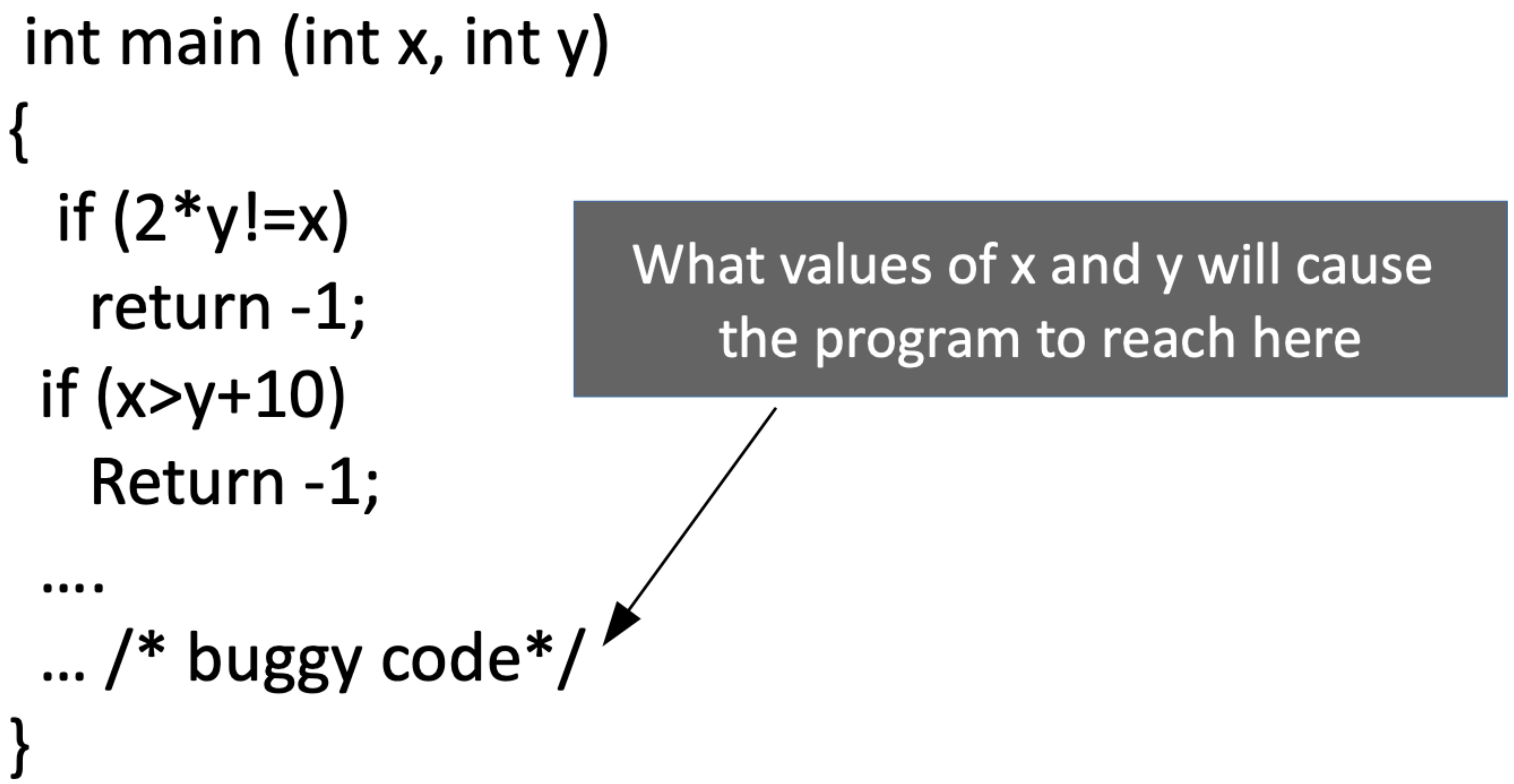
Specifications

Program Analyzer

Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
…	…	…
10,502	info leak	10,921

Descriptions of different classes of bugs

Automated bug detection: main challenges



```
 int main (int x, int y)
{
  if (2*y!=x)
    return -1;
 if (x>y+10)
    Return -1;
 ….
 … /* buggy code*/
}
```

What values of x and y will cause the program to reach here

- Too many paths (may be infinite)
- How will program analyzer find inputs that will reach different parts of code to be tested?

Automated bug detection: two options

- Static analysis
  - Inspect code or run automated method to
    - 1) find errors
    - or 2) gain confidence about their absence
  - Try to aggregate the program behavior over a large number of paths without enumerating them explicitly
- Dynamic analysis
  - Run code, possibly under instrumented conditions, to see if there are likely problems in code
  - Enumerate paths but avoid redundant ones

Static vs dynamic analysis

- Static
  - Can consider all possible inputs
  - Find bugs and vulnerabilities
  - Can prove absence of bugs, in some cases
- Dynamic
  - Need to choose sample test input
  - Can find bugs and vulnerabilities
  - Cannot prove their absence

Soundness & Completeness

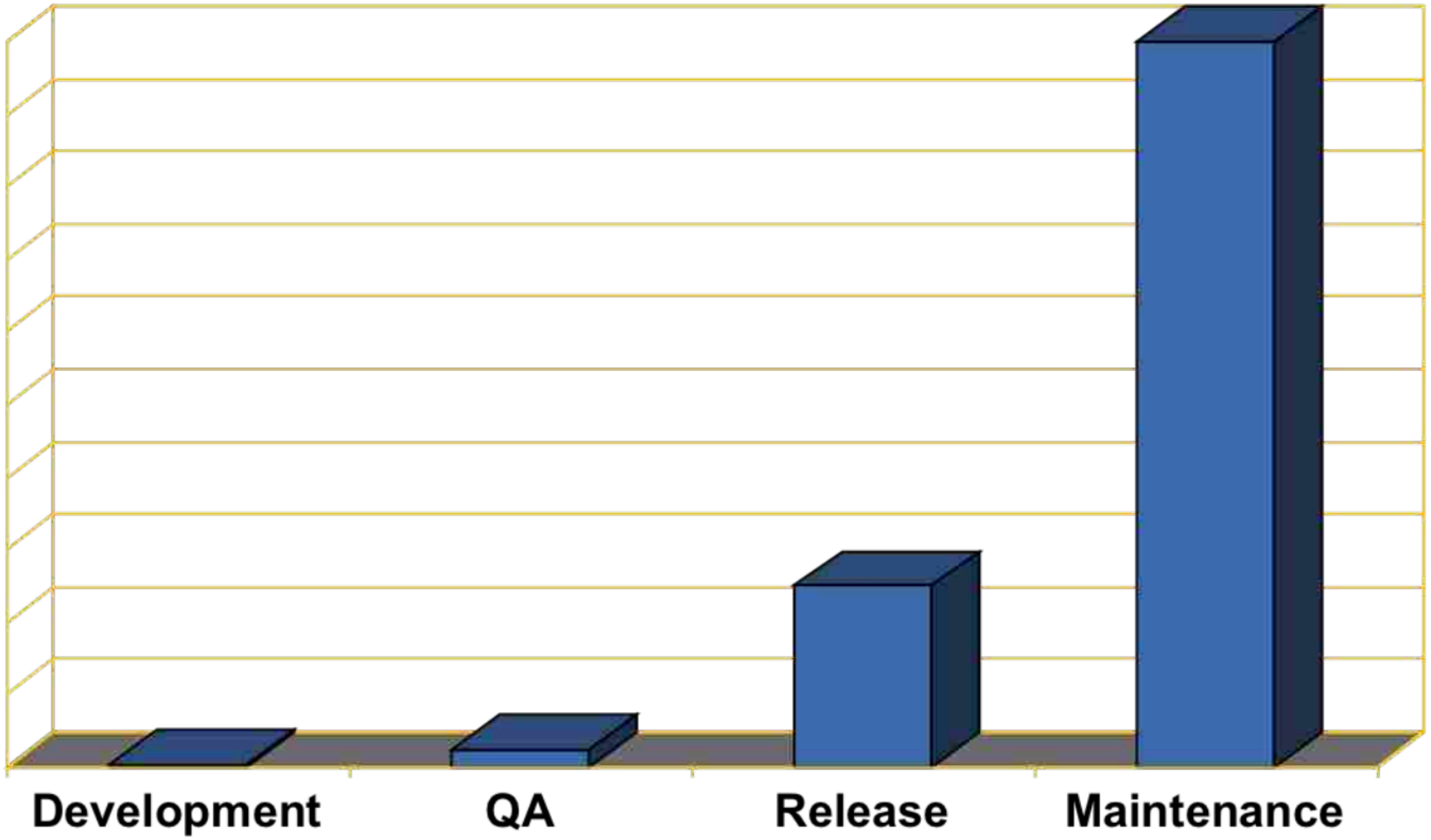
Property	Definition
Soundness	“Sound for reporting correctness” Analysis says no bugs → No bugs or equivalently There is a bug → Analysis finds a bug
Completeness	“Complete for reporting correctness” No bugs → Analysis says no bugs

Recall: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$

Soundness & Completeness

	Complete	**Incomplete**
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

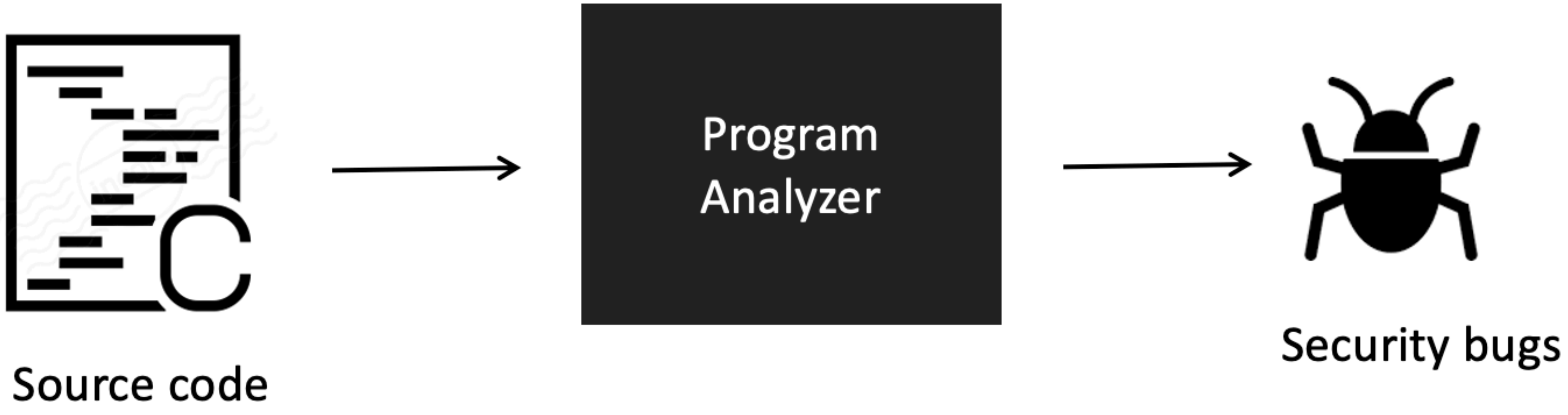
When to find bugs?



Cost of bug finding

Credit: Andy Chou, Coverity

Static Analysis for Security



Source code

Program Analyzer

Security bugs

Program analyzer must be able to understand program properties (e.g., can a variable be NULL at a particular program point?)

→ Must perform control and data flow analysis

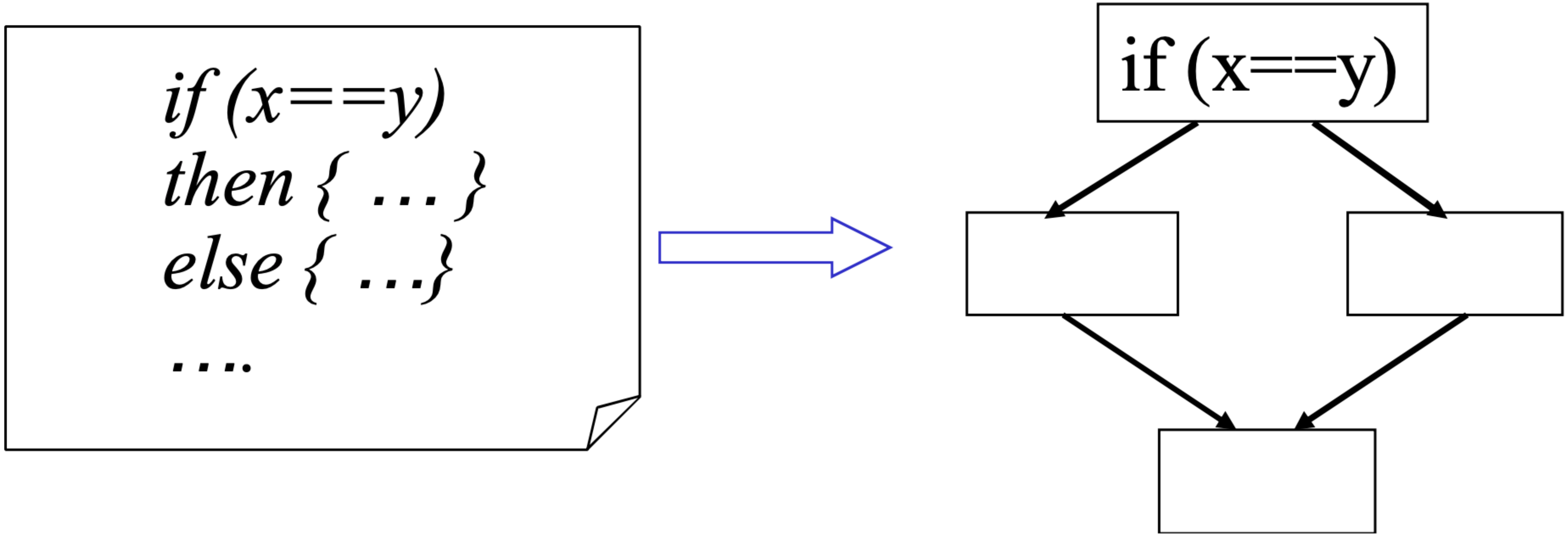
Control Flow Analysis

- Control flow
  - Sequence of operations
  - Representations
    - Control flow graph
    - Control dependence
    - Call graph
- Control flow analysis
  - Analyzing program to discover its control structure

Control Flow Graph

- CFG models flow of control in the program
  - G = (N, E) as a directed graph
  - Node $n \in N$: basic blocks
    - A basic block is a maximal sequence of statements with a single entry point, single exit point, and no internal branches
  - Edge $e=(n_i, n_j) \in E$: possible transfer of control from block n_i to block n_j

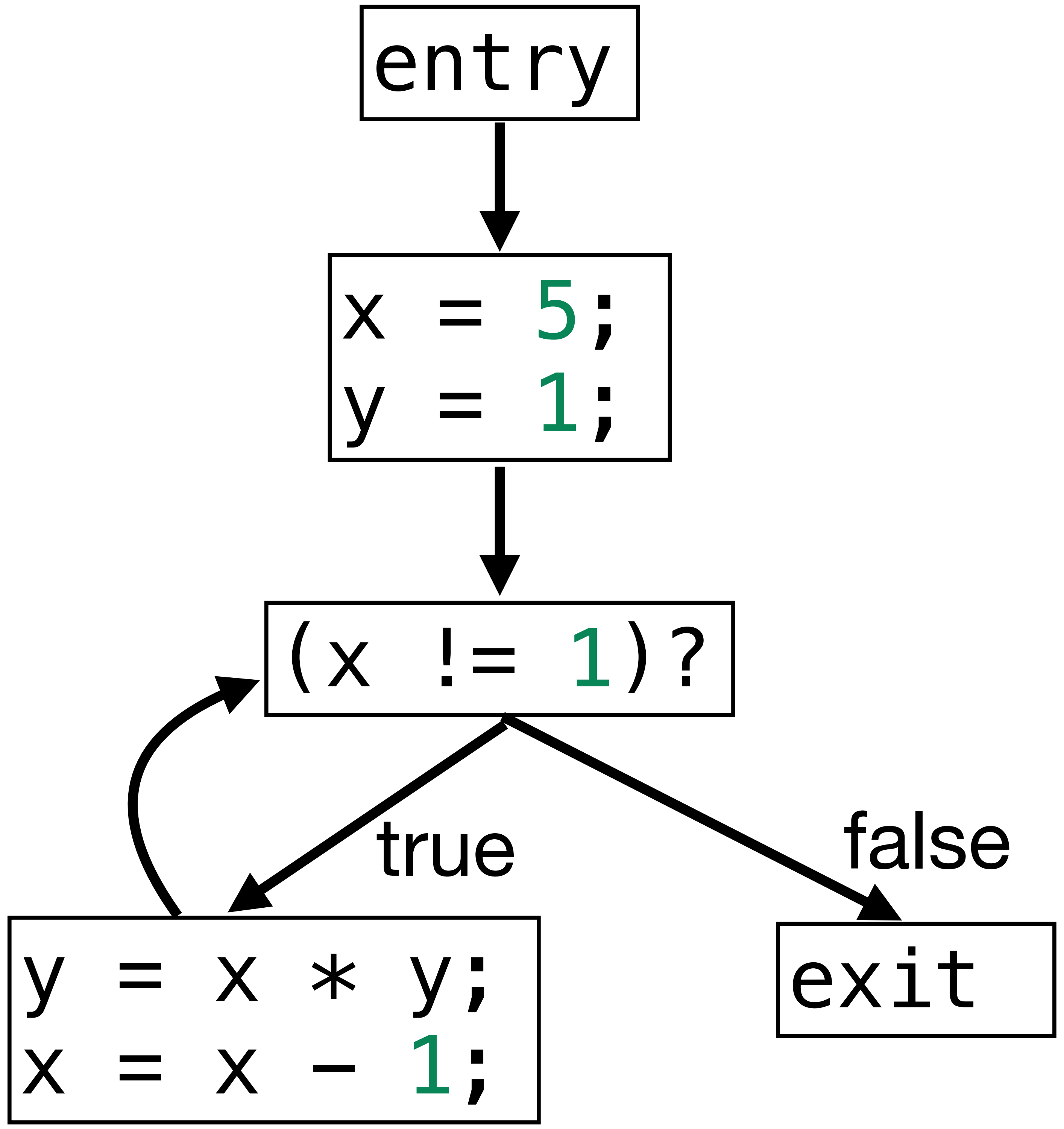
Control Flow Graph Example



if (x==y)
then { … }
else { …}
….

Control Flow Graph Example

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1;
}
```



Control Flow Graph

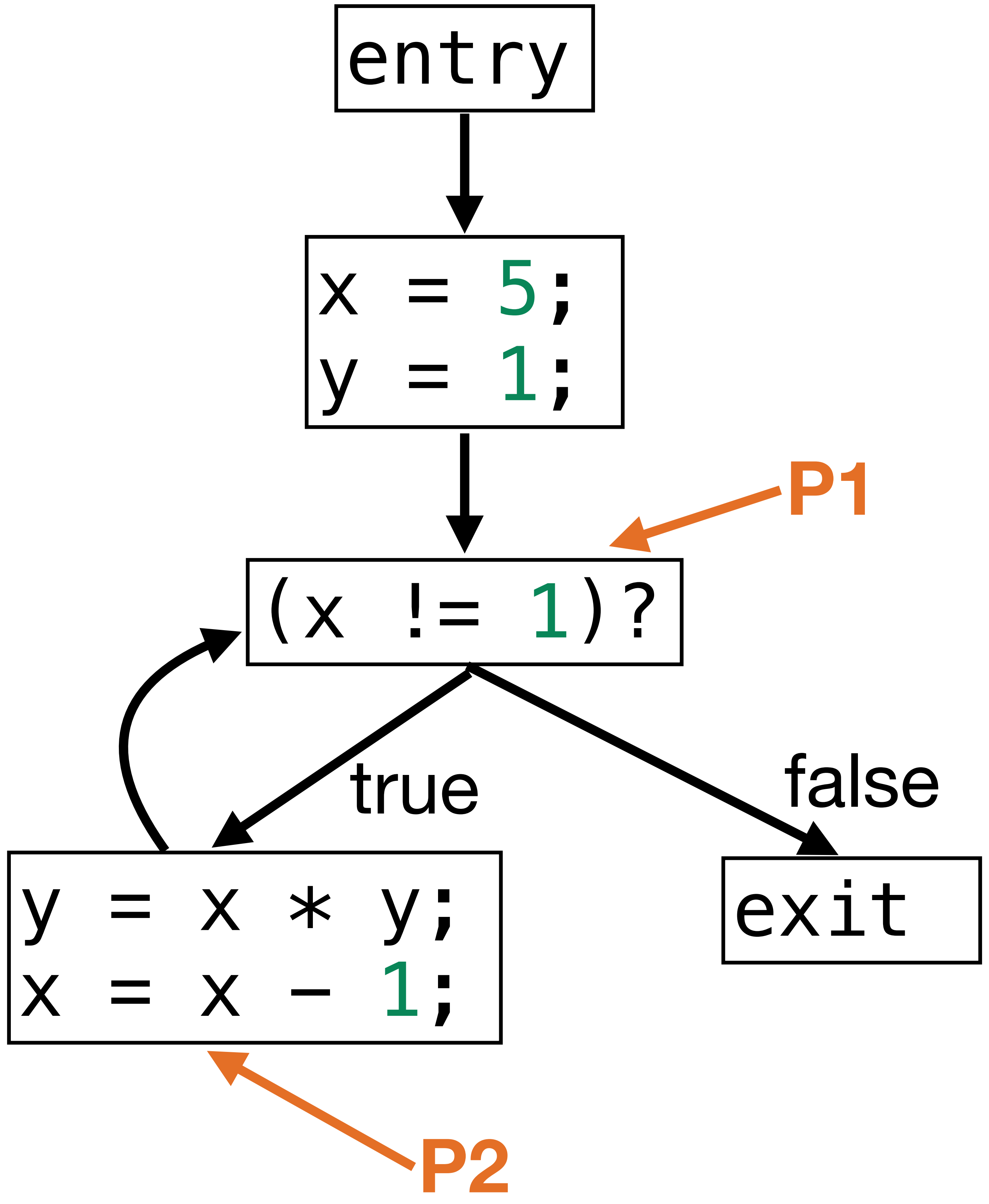
- CFGs are commonly used to propagate information between nodes (basic blocks)
  - e.g., For data flow analysis
- Useful for dynamic analysis
  - e.g., fuzzing

Data Flow Analysis

- **Data-flow analysis** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Derives information about the dynamic behavior of a program by only examining the static code
- Examples:
  - Reaching definition analysis
  - Live variable analysis
  - Dead code detection
  - …

Data Flow Analysis Example

- Reaching definition analysis:
  - At each program point, which assignments (definitions) have been made, and not overwritten, when the execution reaches that point along some path.
- Example: assignment x = 5 reaches P1, but does not reach P2, since x = x-1 overwrites x.
- This could be useful for detecting many security vulnerabilities.



Do we need to implement control and data flow analysis from scratch?

- Most modern compilers already perform several types of such analysis for code optimization
  - We can hook into different layers of analysis and customize them
  - We still need to understand the details
- LLVM (http://llvm.org/) is a highly customizable and modular compiler framework
  - Users can write LLVM passes to perform different types of analysis
  - Clang static analyzer can find several types of bugs
  - Can instrument code for dynamic analysis

Soundness & Completeness

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

False positive rate is very high

Static analysis: consider all possible paths in a program, over report vulnerabilities

Soundness & Completeness

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Dynamic analysis: execute programs on concrete input, but may miss vulnerabilities

Soundness & Completeness

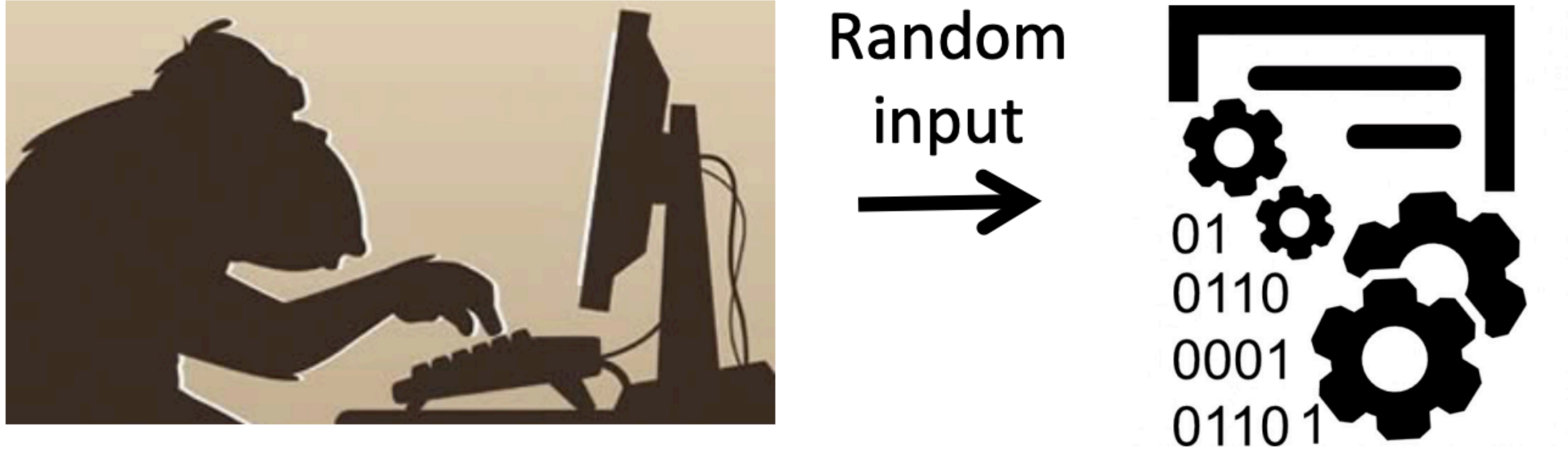
	Complete	**Incomplete**
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Implementations of some tools may belong here but it’s not very nice

Fuzzing

- **Fuzzing**, or **fuzz testing**, is an automated software testing technique that involves providing invalid, semi-valid, unexpected, or random data as inputs to a computer program.

Blackbox Fuzzing



Random input →

Test program

Miller et al. ‘89

Blackbox Fuzzing

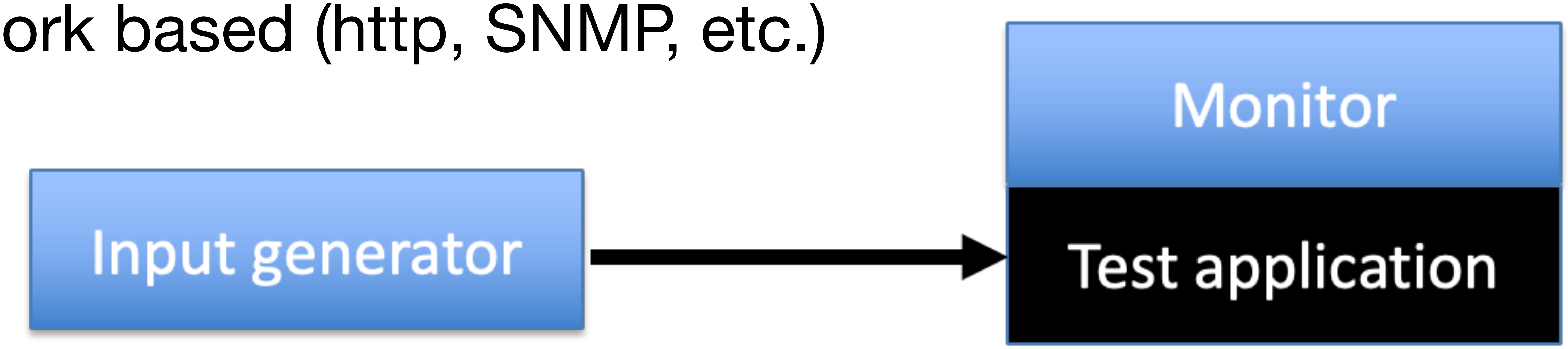
- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., crashes)

- Advantage: easy, low programmer cost

- Disadvantage: inefficient
  - Inputs often require structures, random inputs are likely to be malformed
  - Inputs that trigger an incorrect behavior is a a very small fraction, probably of getting lucky is very low

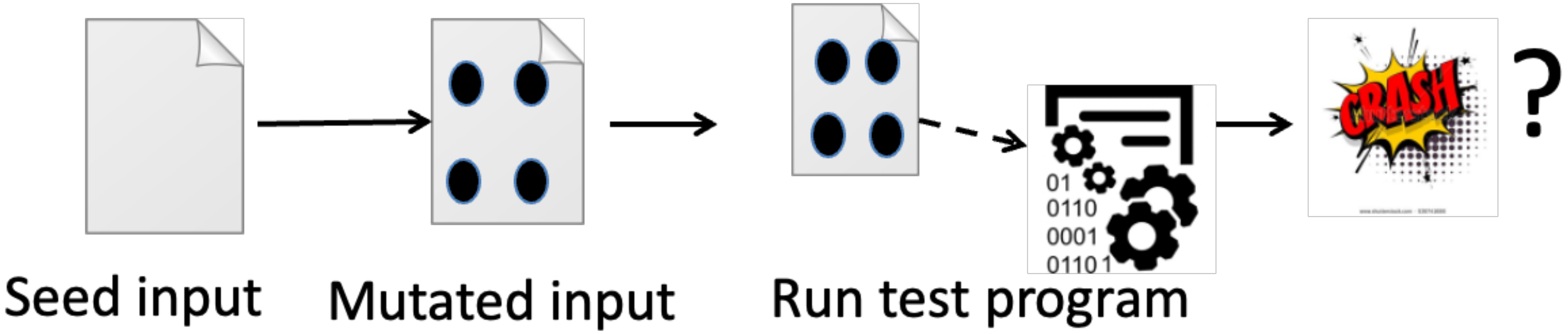
Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
  - See if program crashed, e.g., SEGV vs. assert fail
  - See if program locks up
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)



Enhancement 1:
Mutation-Based fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a PDF viewer

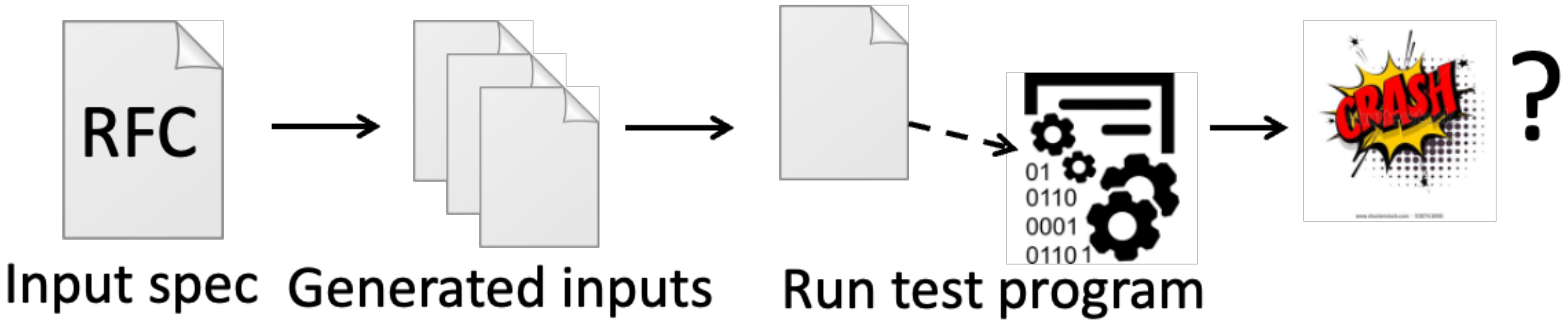
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
    - Collect seed PDF files
    - Mutate that file
    - Feed it to the program
    - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

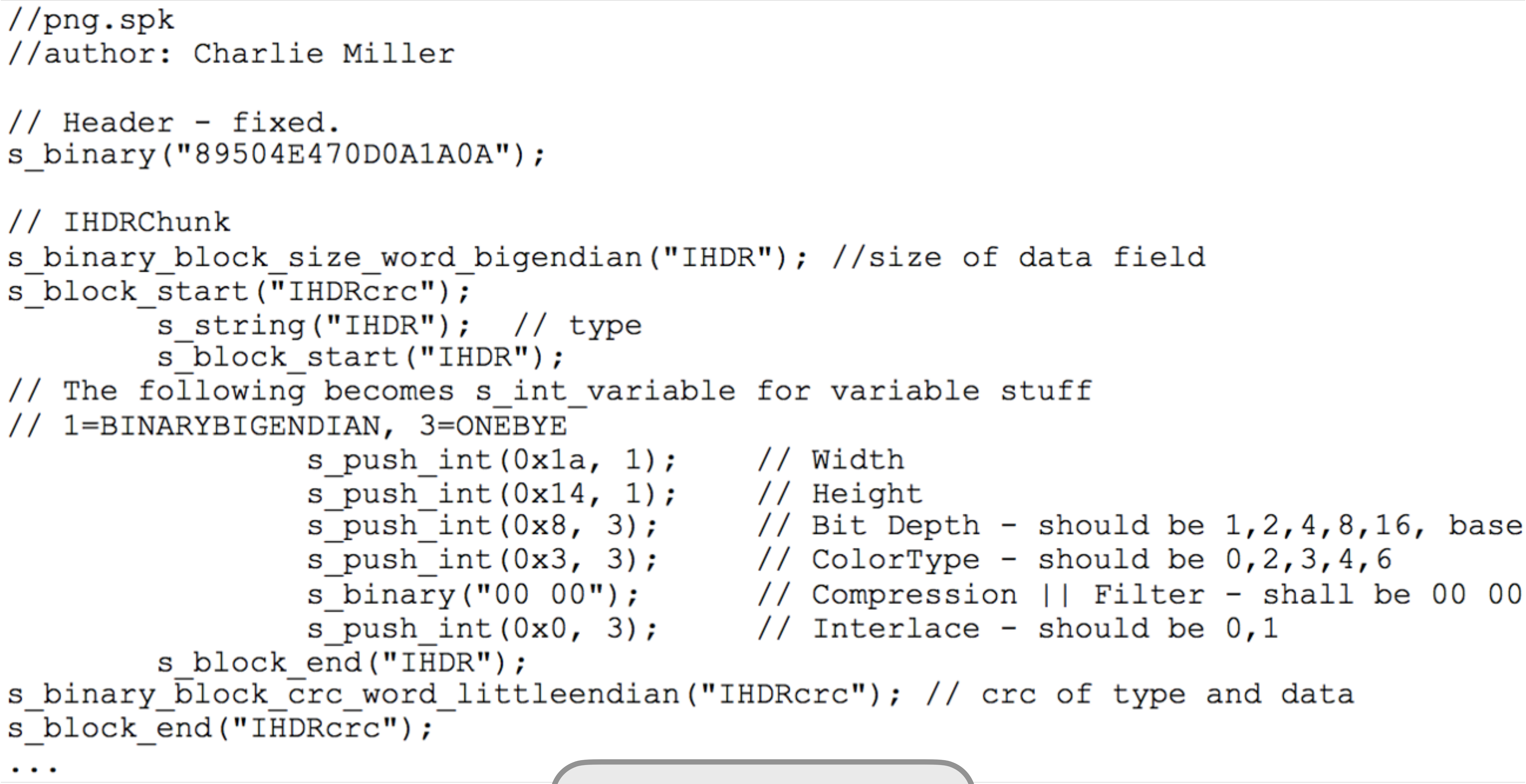
- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the input format: RFC, documentation, etc.
    - Using specified protocols/file format info
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Example: fuzzing a PNG file parser



```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, base
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Sample PNG Spec

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fail for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependencies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they’re all run and no bugs are found?

Code coverage

- Some of the answers to these questions lie in code coverage
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Different Coverage Metrics

- **Line/block coverage:** Measures how many lines of source code have been executed
- **Branch coverage:** Measures how many branches in code have been taken (conditional jmps)
- **Path coverage:** Measures how many paths have been taken

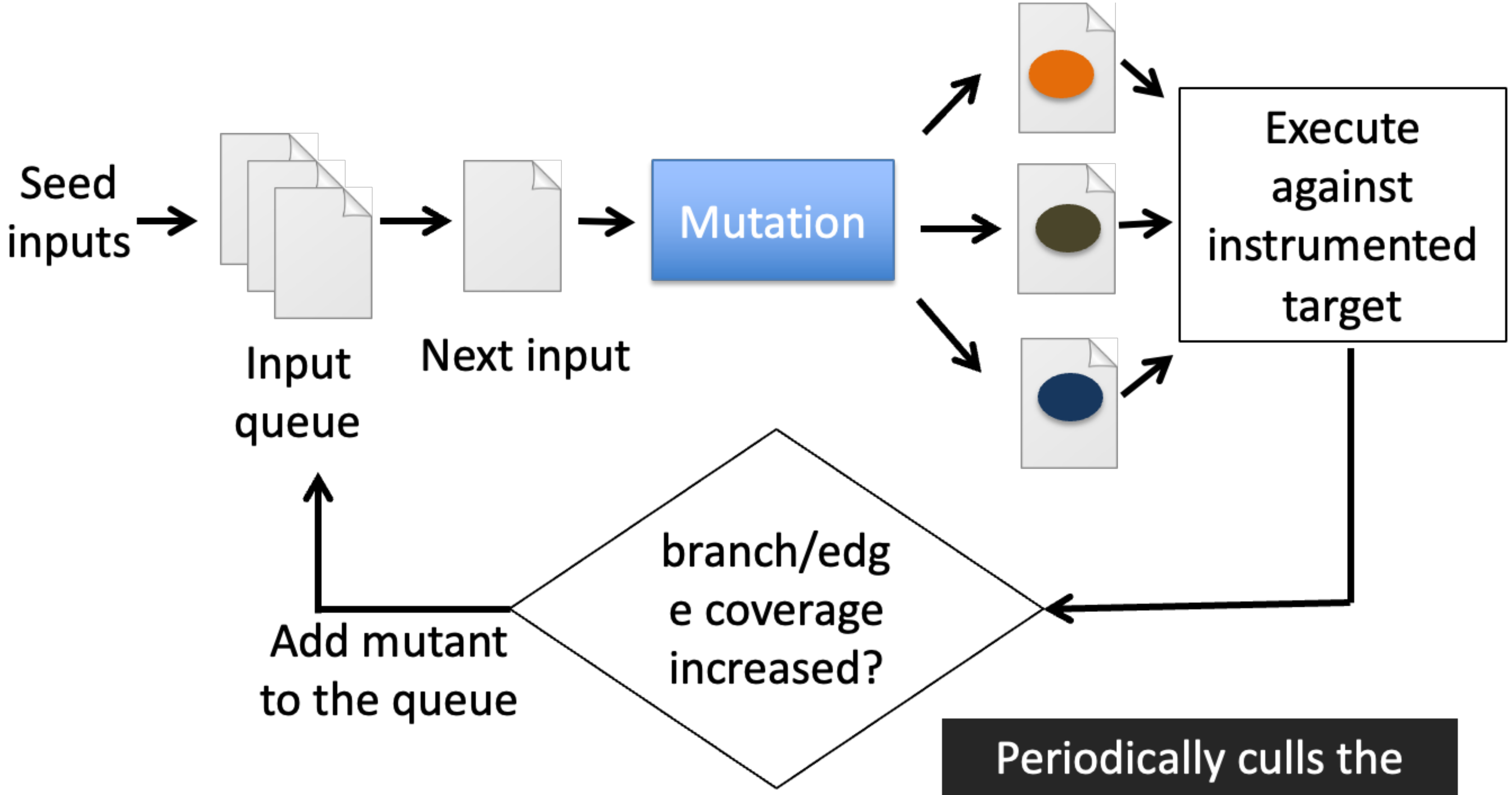
Code coverage

- Pros:
  - Can evaluate an input
  - Can compare fuzzers
  - Am I getting stuck somewhere?
- Cons:
  - Full coverage (any metric) does not guarantee finding the bug

Enhancement III:
Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic evolution algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)



Periodically culls the queue without affecting total coverage