CSE467: Computer Security

24. Dynamic Analysis

Seongil Wi

Department of Computer Science and Engineering

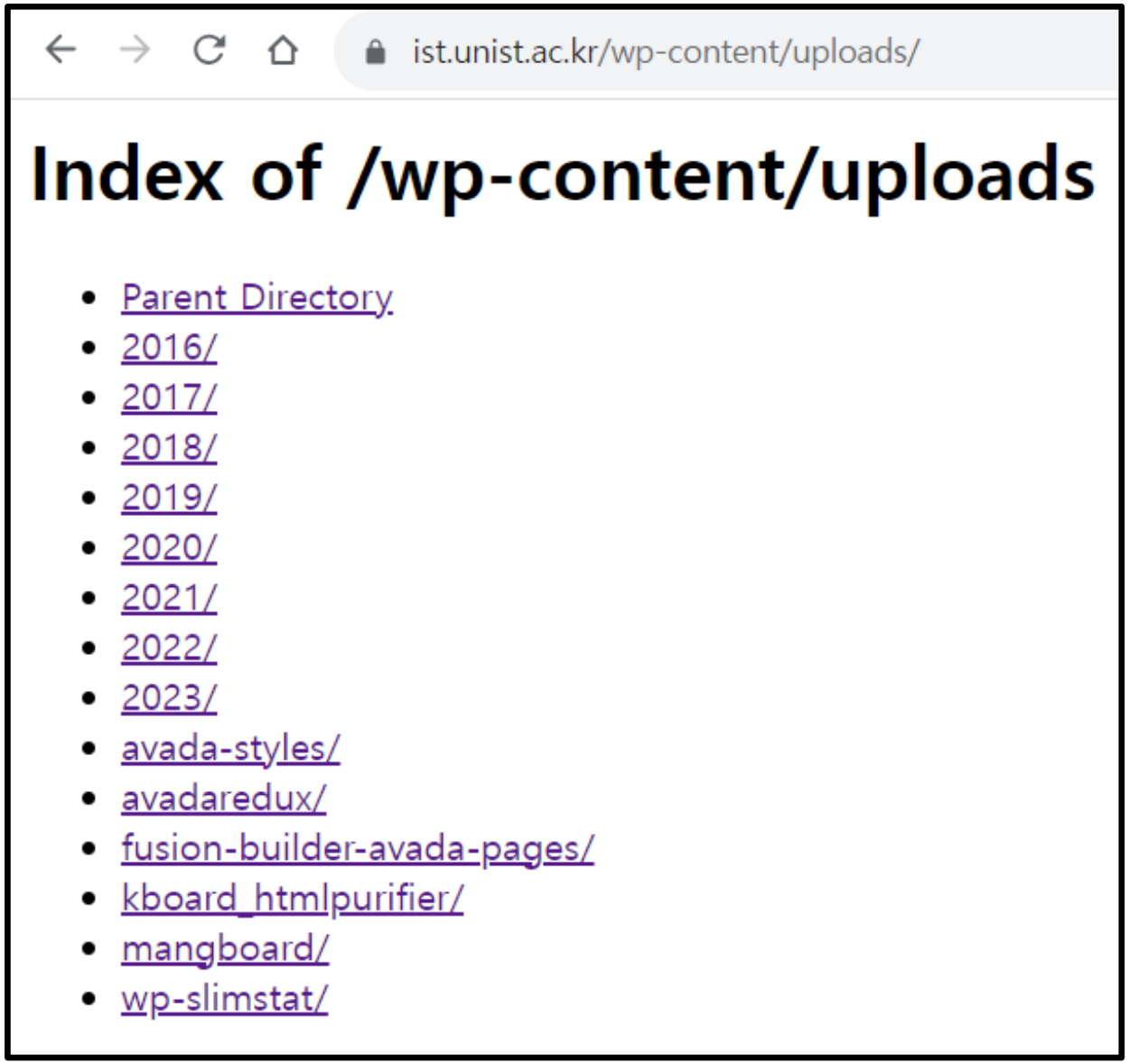
Final-term Overview

- Date: Thursday, 12/14
- Time: Class time (17:30 ~ 18:45)

- **Important notice #1:**
  - I will clarify the scope of the final exam in the next class

- **Important notice #2:**
  - A Q&A session will be held during the class time on12/7
  - I will be in the classroom during class time
  - If you have any questions about what you have learned so far, please come and ask
    - You are not required to attend this session!

SaveUNIST



Exposure of Information Through Directory Listing!

Nice finding from Jaewoo Heo ☺

Recap: 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

Recap: Symbolic Execution

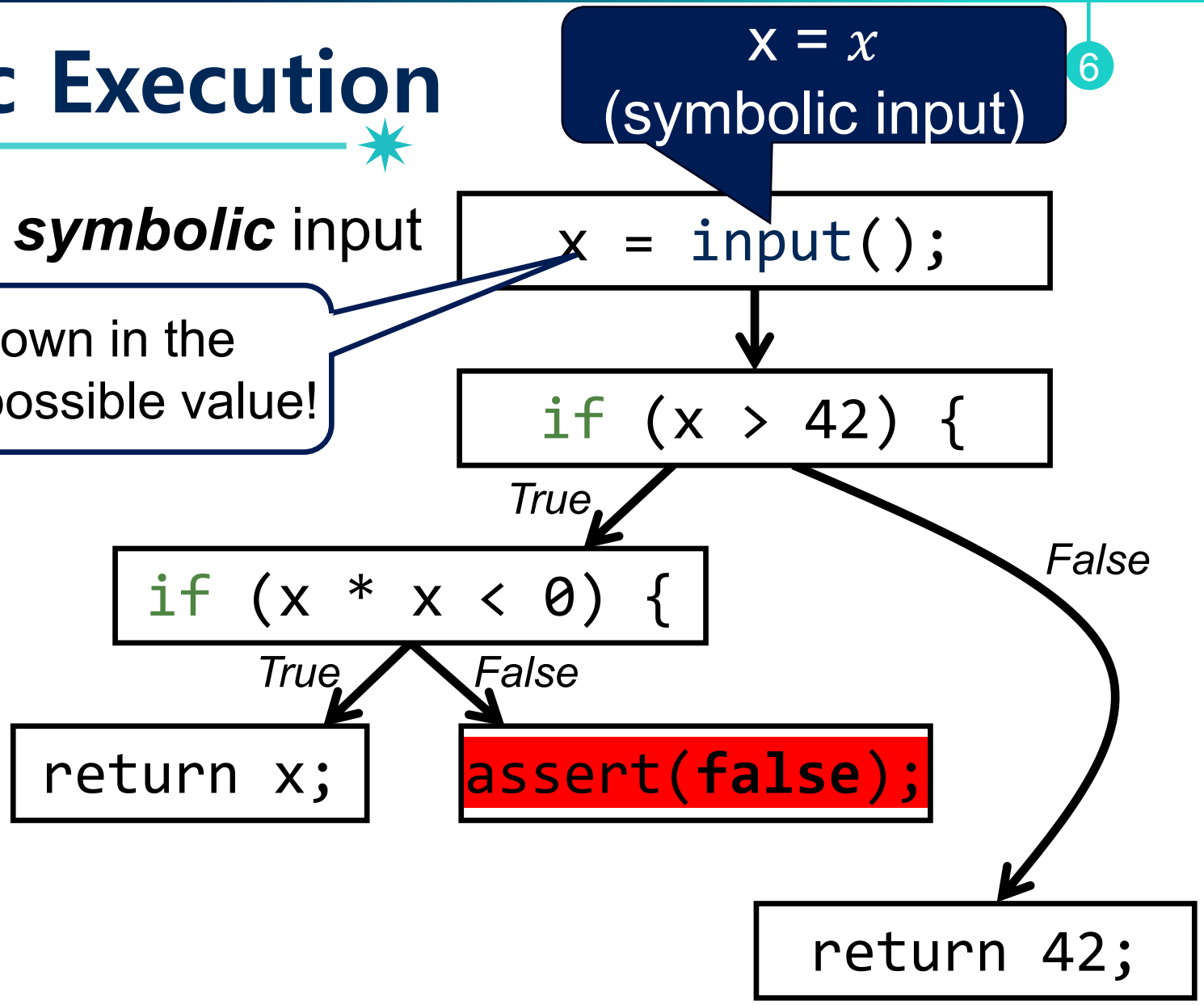
- A program analysis technique that executes a program with symbolic – rather than concrete – input values.

- For each execution path, construct a ***path formula*** that describes the input constraint to follow the path

Recap: Symbolic Execution

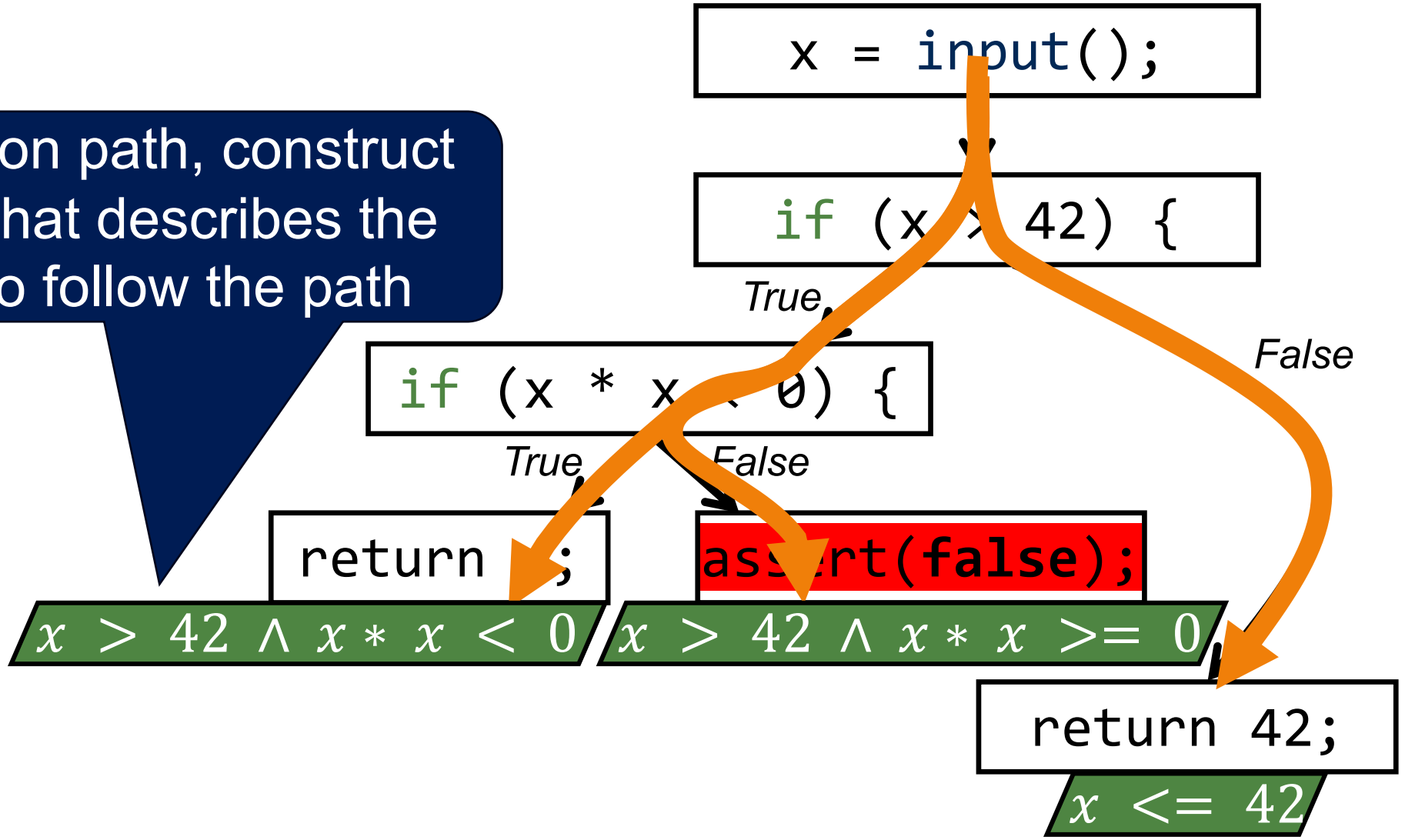
- Runs a program with a ***symbolic*** input

Let's treat it as an unknown in the equation. It can have any possible value!

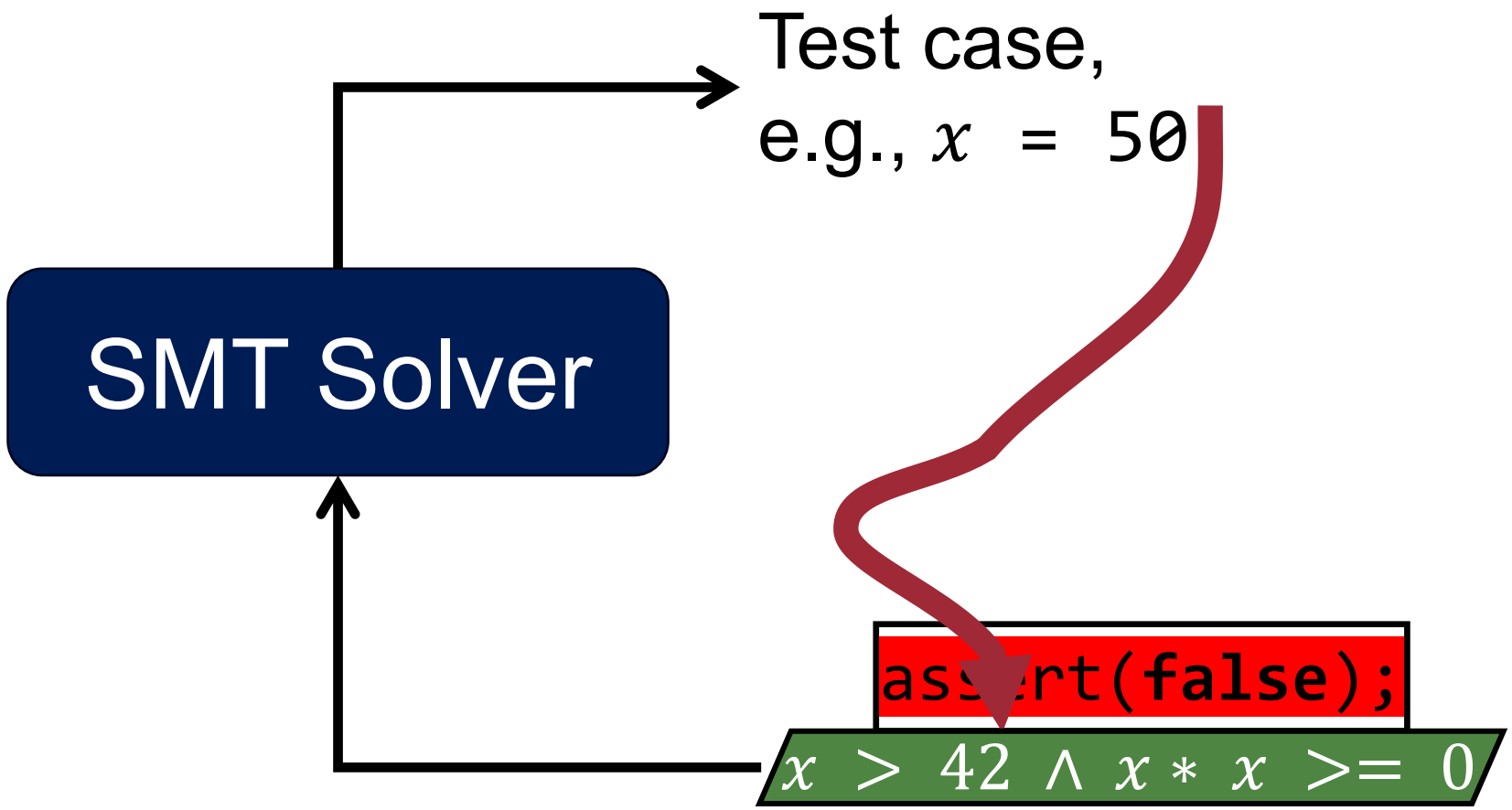


Recap: Path Formulas

For each execution path, construct a ***path formula*** that describes the input constraint to follow the path



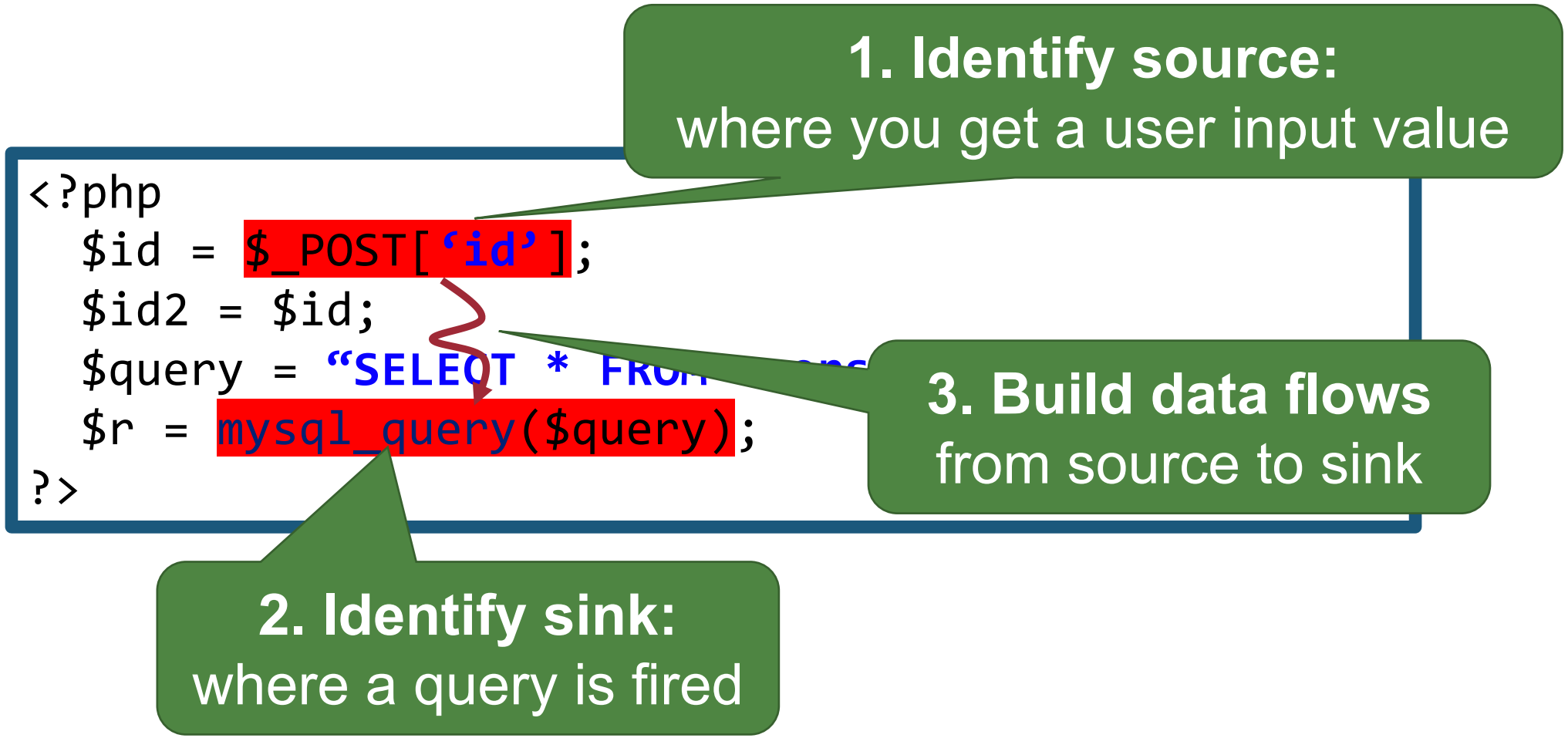
Recap: Symbolic Execution



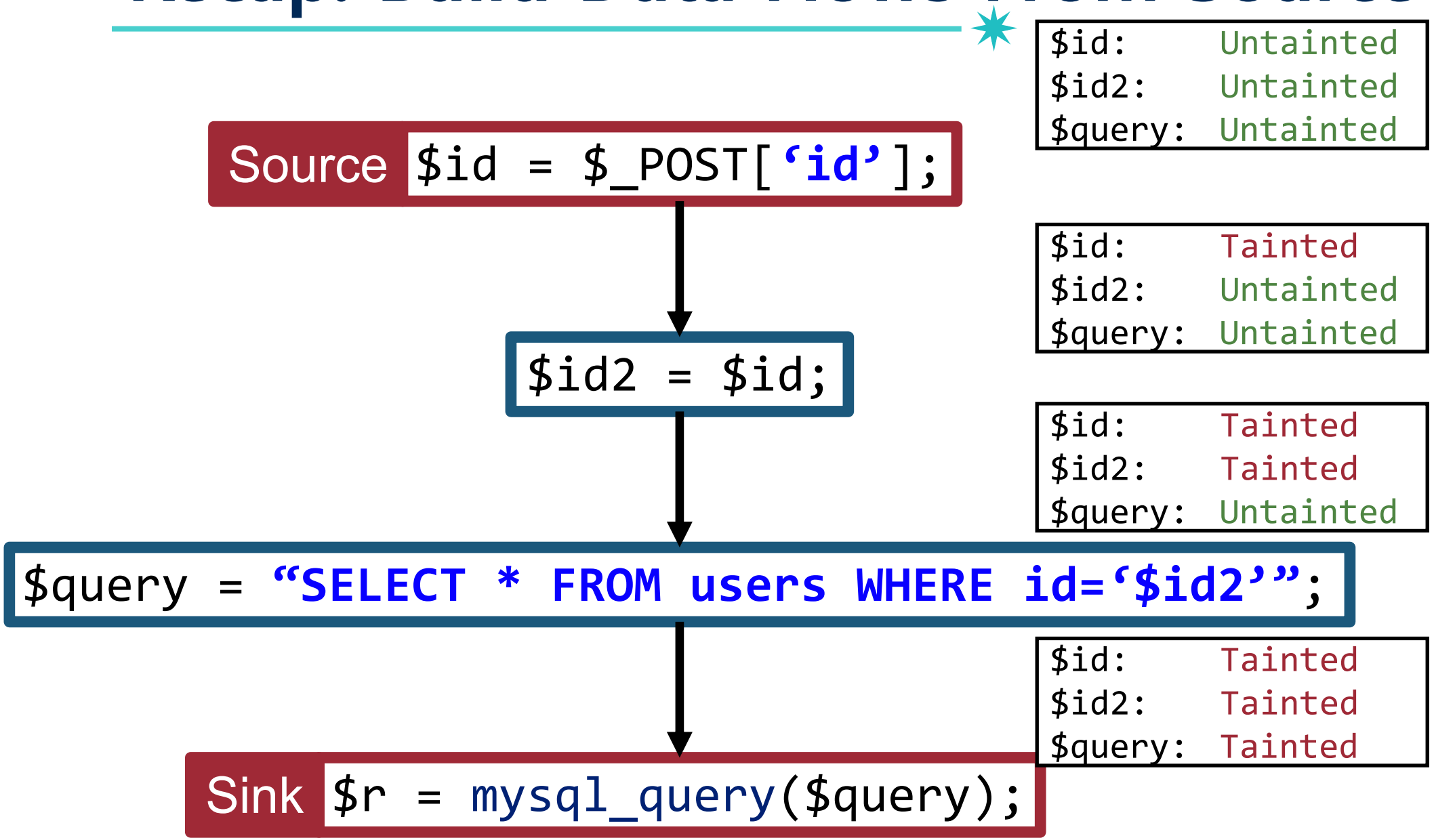
Recap: Taint Analysis

- More abstract than Symbolic Execution
- Basic idea: identify whether “tainted” values can ***reach*** “sensitive” points in the program
  - Tainted source: input values that come from the user
  - Sensitive sink: any point in the program where a value is

Recap: Taint Analysis Procedure



Recap: Build Data Flows From Source to Sink

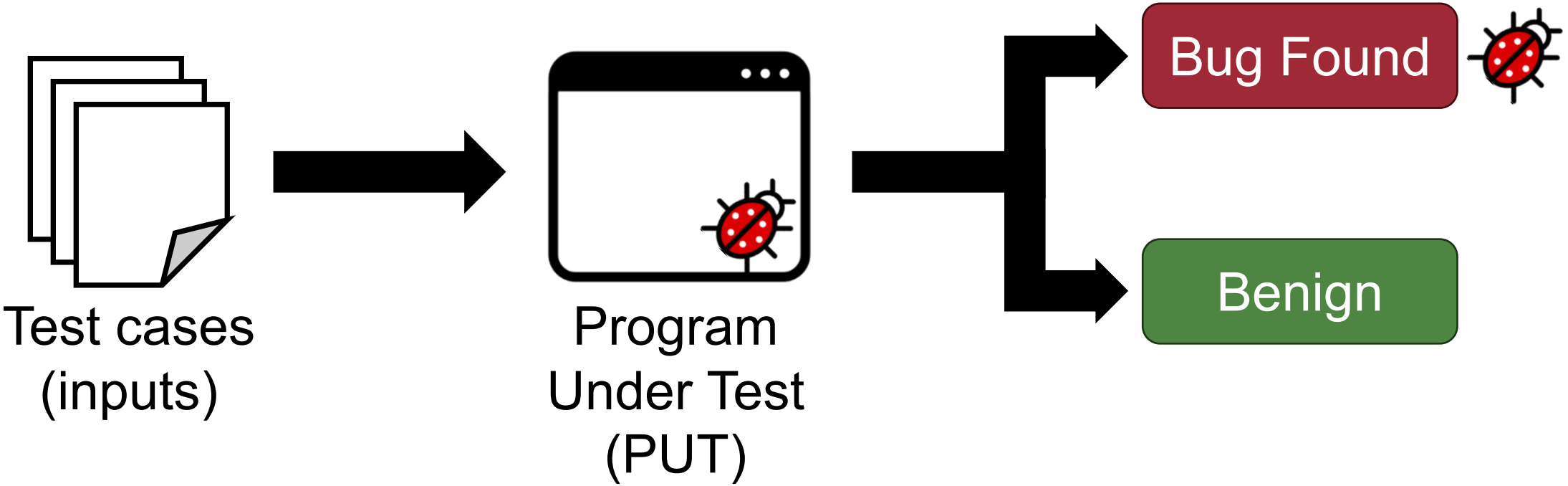


Today's Topic!

- **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



Fuzzing

Fuzzing

- Original definition: feed a string of random characters into a program for finding ***software bugs***
- Extended definition: a software testing technique for finding ***software bugs***

- Goal: find as many security-related bugs as possible

Why Fuzzing

- Simple (simpler than symbolic execution) and fast

History of Fuzzing

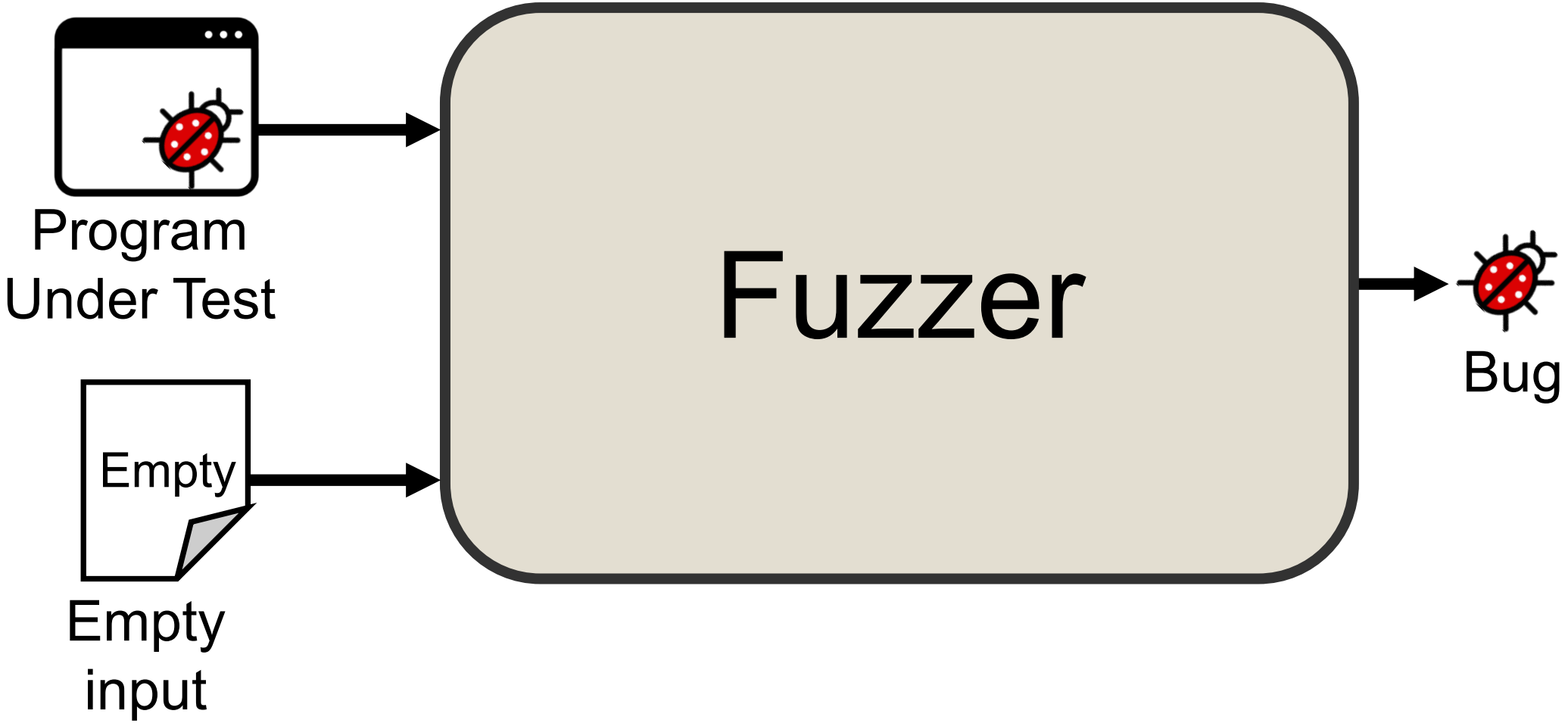
The original work was inspired by being logged on to a modem during a storm with lots of line noise.

Thunderstorm → noise on a network line → random characters → crash
(*fuzz*)

The term was coined by ***Barton Miller*** in 1990

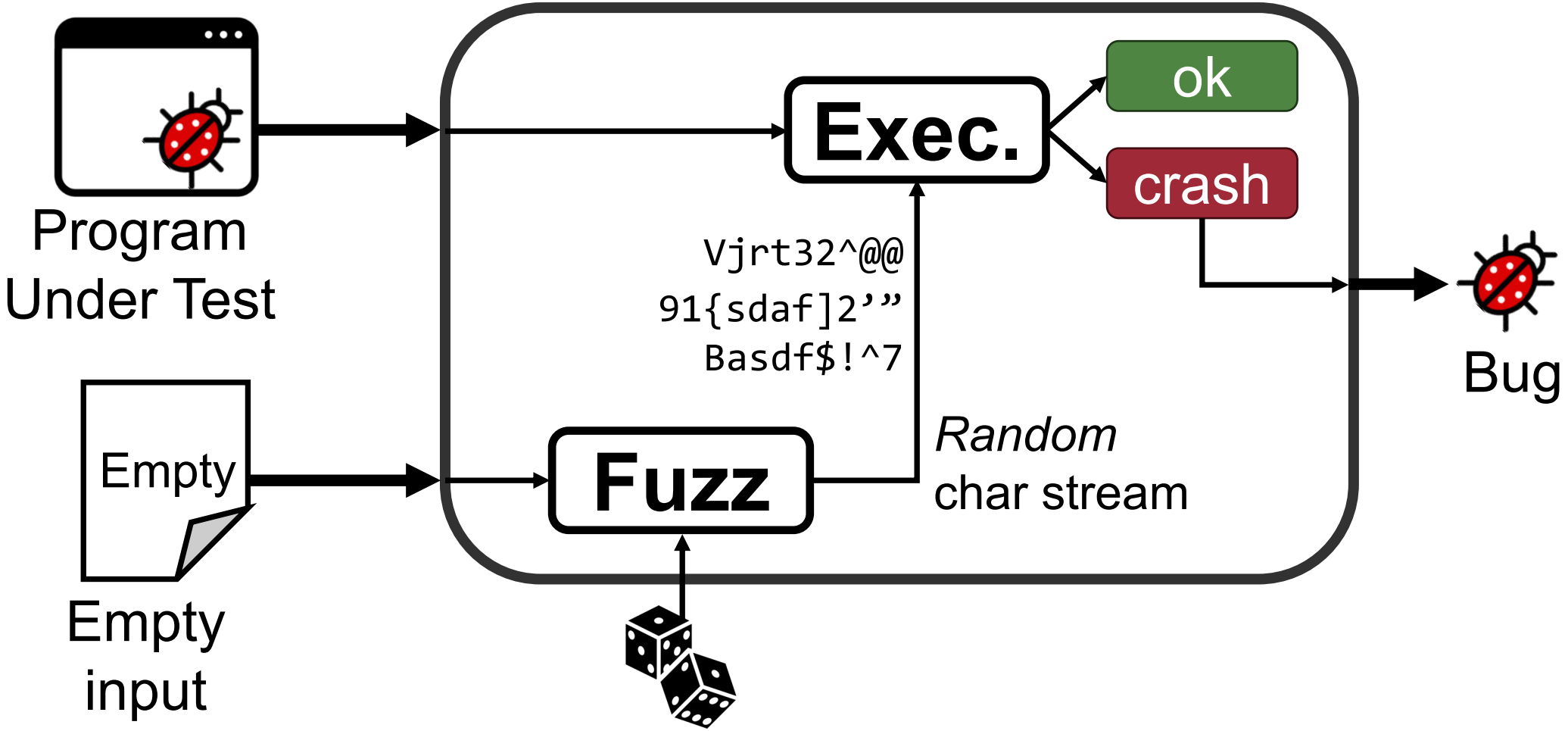
Fuzzing in 1990s

- An Empirical Study of the Reliability of UNIX Utilities, ***CACM 1990***



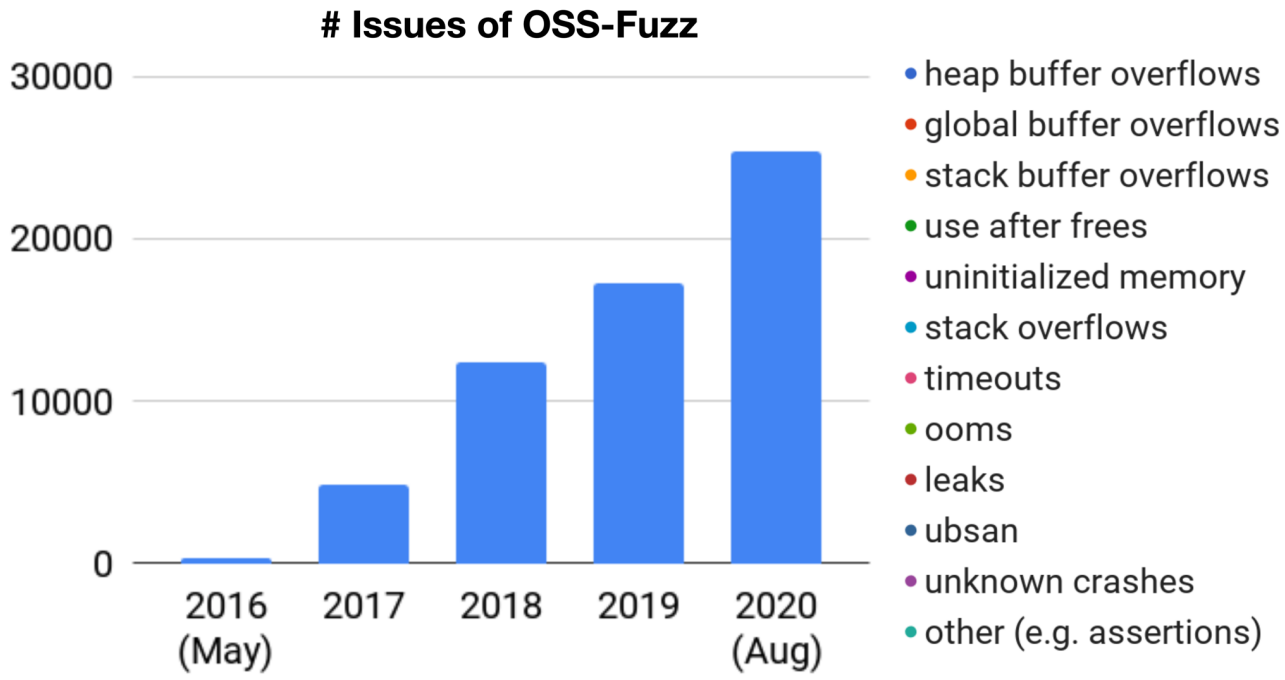
Fuzzing in 1990s

- An Empirical Study of the Reliability of UNIX Utilities, ***CACM 1990***

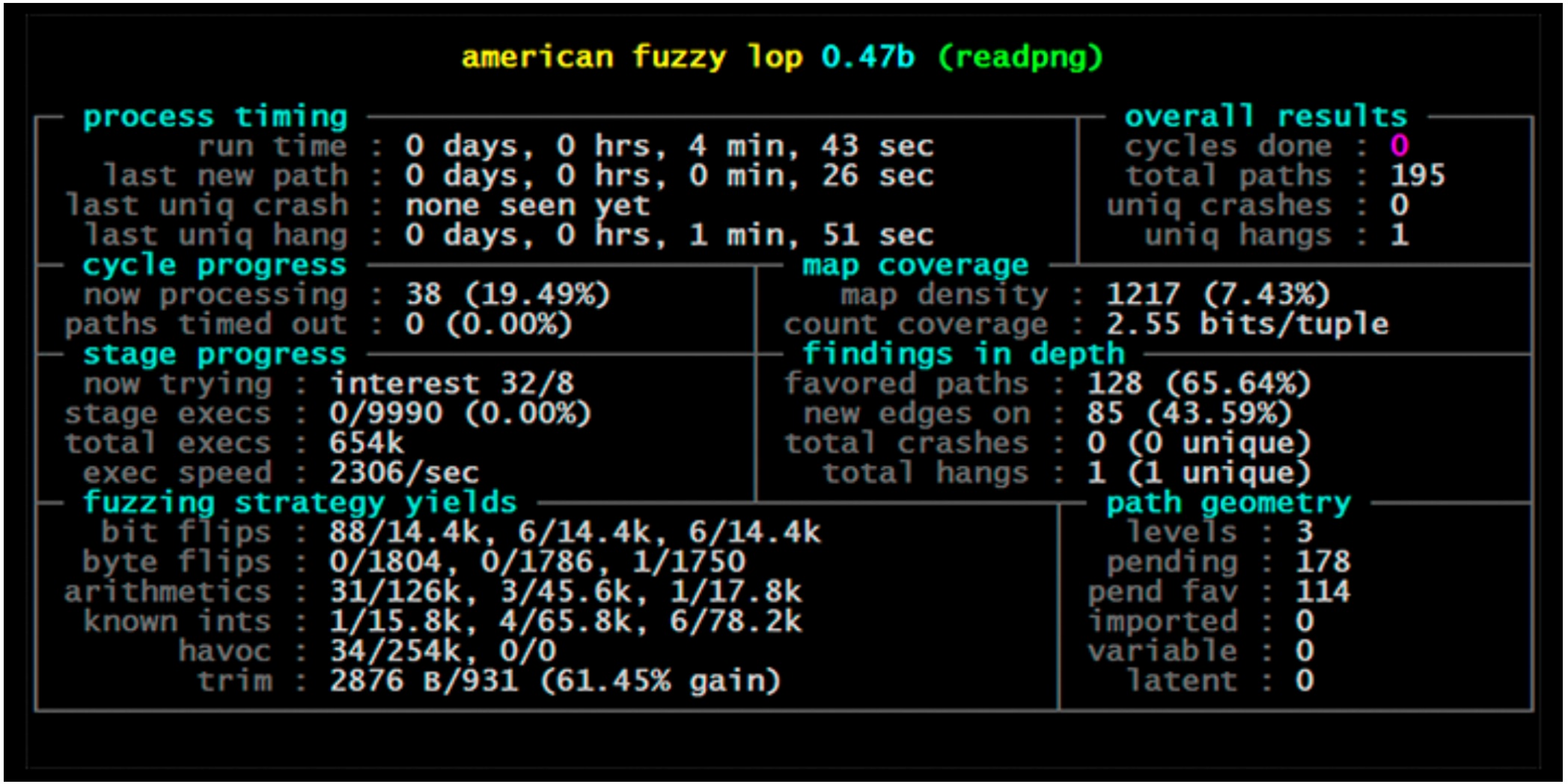


Success Stories

- Miller et al. found many crashes in UNIX utilities
- AFL (American Fuzzy Lop) has found a lot of security vulnerabilities
  - https://lcamtuf.coredump.cx/afl/
- Google's OSS-Fuzz: continues fuzzing platform for open-source software



AFL (American Fuzzy Lop)



Fuzzing is …

- Simple, and popular way to find security bugs
- Used by security practitioners
- But, ***not studied systematically until 2013***
  - Why fuzzing works so well in practice?
  - Are we maximizing the ability of fuzzing?

Fuzzing is an Overloaded Term

- White-box, black-box, and grey-box fuzzing
- Directed fuzzing and undirected fuzzing
- Feedback-driven fuzzing
- Generational fuzzing and mutational fuzzing
- Grammar-based fuzzing
- Seed-based fuzzing
- Model-based fuzzing and model-less fuzzing
- Etc

Let’s organize the terms

Definitions

- Based on ***the granularity of what we observe*** in each run:
  - Black-box fuzzing
  - White-box fuzzing
  - Grey-box fuzzing

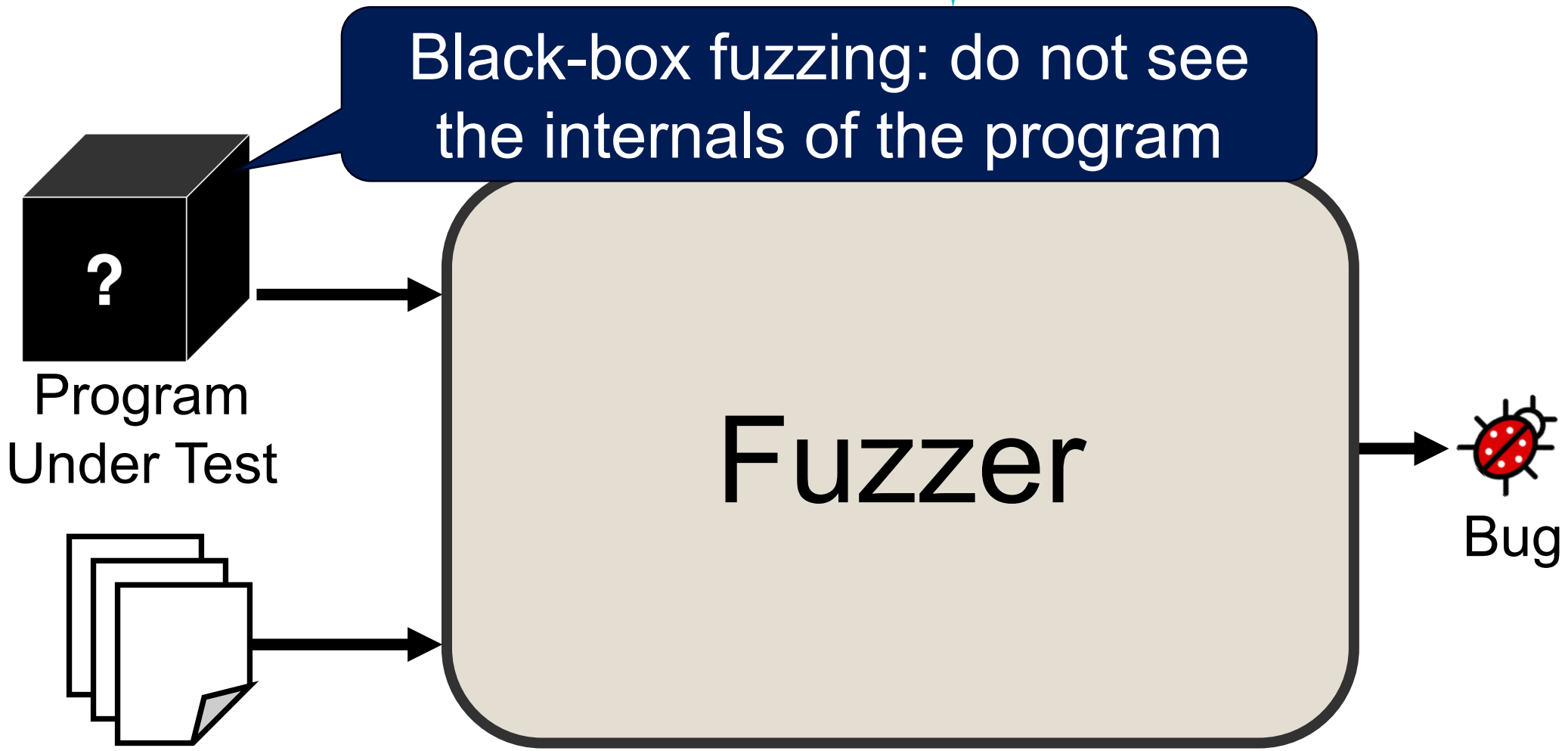
- Based on ***input production techniques***:
  - Mutation-based fuzzing
  - Grammar-based fuzzing

Definitions

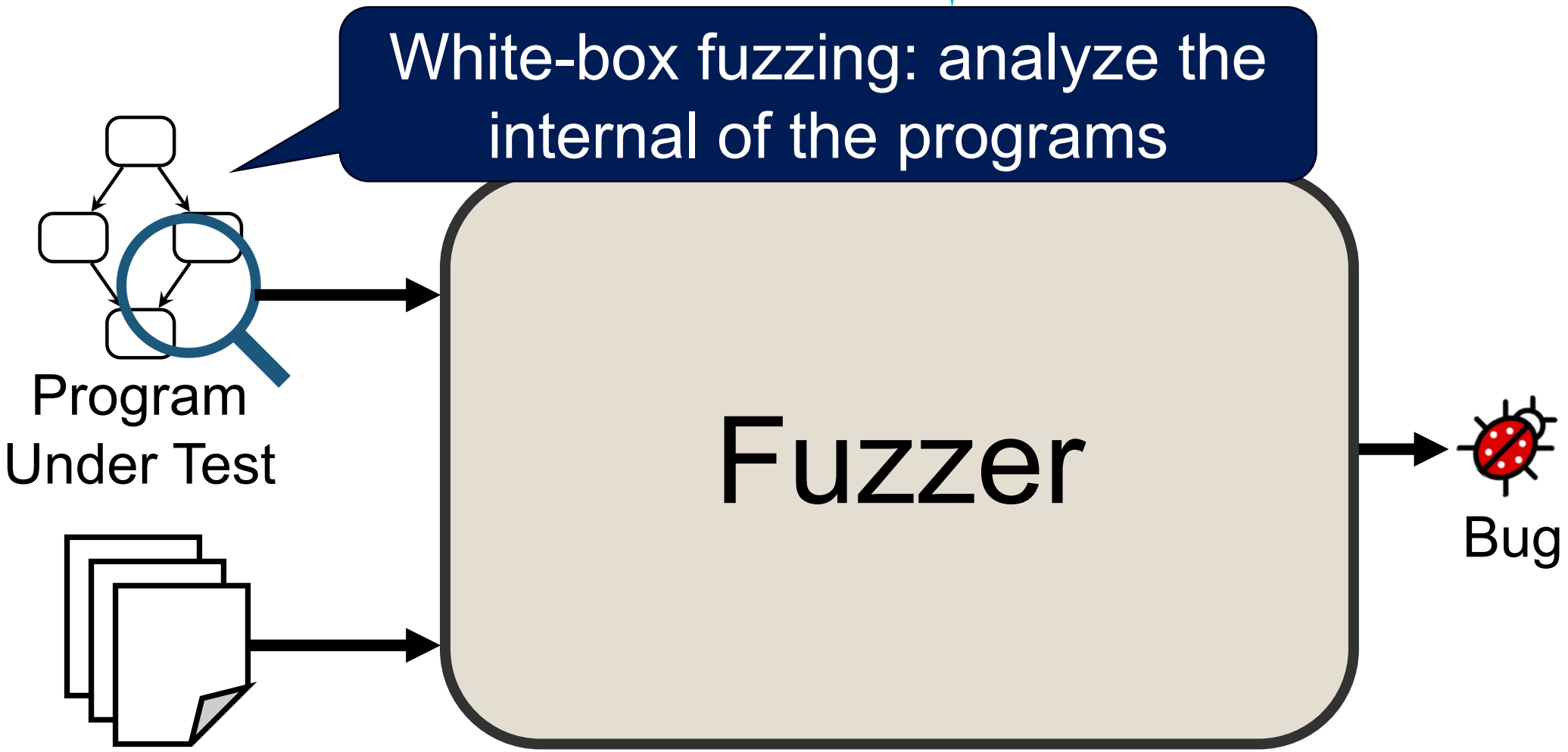
- Based on ***the granularity of what we observe*** in each run:
  - Black-box fuzzing
  - White-box fuzzing
  - Grey-box fuzzing

- Based on ***input production techniques***:
  - Mutation-based fuzzing
  - Grammar-based fuzzing

Black-box vs. White-box Fuzzing



Black-box vs. White-box Fuzzing



Black-box vs. White-box Fuzzing

- **Blackbox**: generate inputs *regardless* of program's logic and structure
  - Pros: easy to implement and low cost
  - Cons: hard to explore deeper parts

```
x = input();
if (x == 482716115)
    bug();
```

- **Whitebox**: generate inputs by *observing* program's logic and structure (a.k.a., dynamic symbolic execution)
  - Pros: can explore deeper parts
  - Cons: Require constraint solving (high overhead)

```
a = input();
b = input();
c = input();
n = input();
if (n > 2)
    if(a^n +b^n == c^n)
        bug();
```

Grey-box Fuzzing

- White-box fuzzing (strictly speaking)
- Obtain "*some*" partial information about the program execution
  - E.g., code coverage information

- Pros: easy to implement and low cost (easier than white-box fuzzing)
- Pros: can explore deeper parts (deeper than black-box fuzzing)

Definitions

- Based on ***the granularity of what we observe*** in each run:
  - Black-box fuzzing
  - White-box fuzzing
  - Grey-box fuzzing

- Based on ***input production techniques***:
  - Mutation-based fuzzing
  - Grammar-based fuzzing

Definitions

- Based on ***the granularity of what we observe*** in each run:
  - Black-box fuzzing
  - White-box fuzzing
  - Grey-box fuzzing

- Based on ***input production techniques***:
  - Mutation-based fuzzing
  - Grammar-based fuzzing

Mutation- vs. Generation-based Fuzzing

- ***Mutation-based***: mutate a given **seed* to generate test cases
- ***Generation-based***: generate test cases from a model

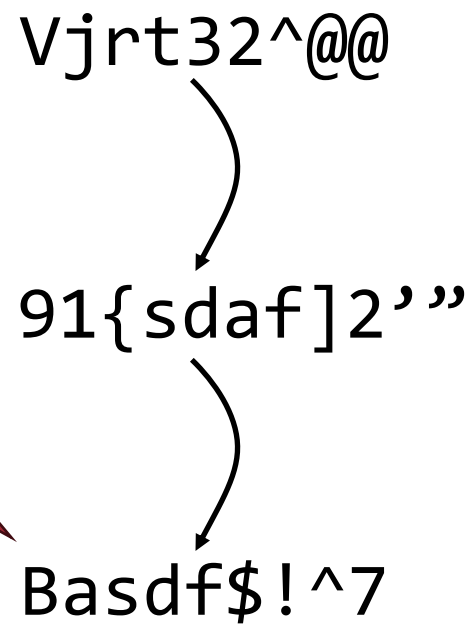
* ***Seed***: an input to a program

Problems of Miller's Design


```
int x = source();
if (x[0] == 'f') {
    if (x[1] == 'd')
        bug()
    elif (x[1] = 'c')
        y = 2 * x;
}
```

Random inputs are likely to be rejected

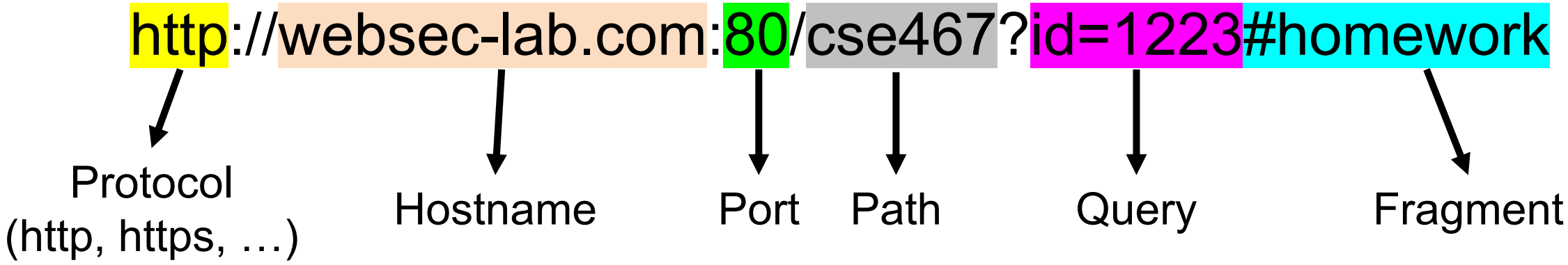
Random input generation



Problems of Random Input Generation

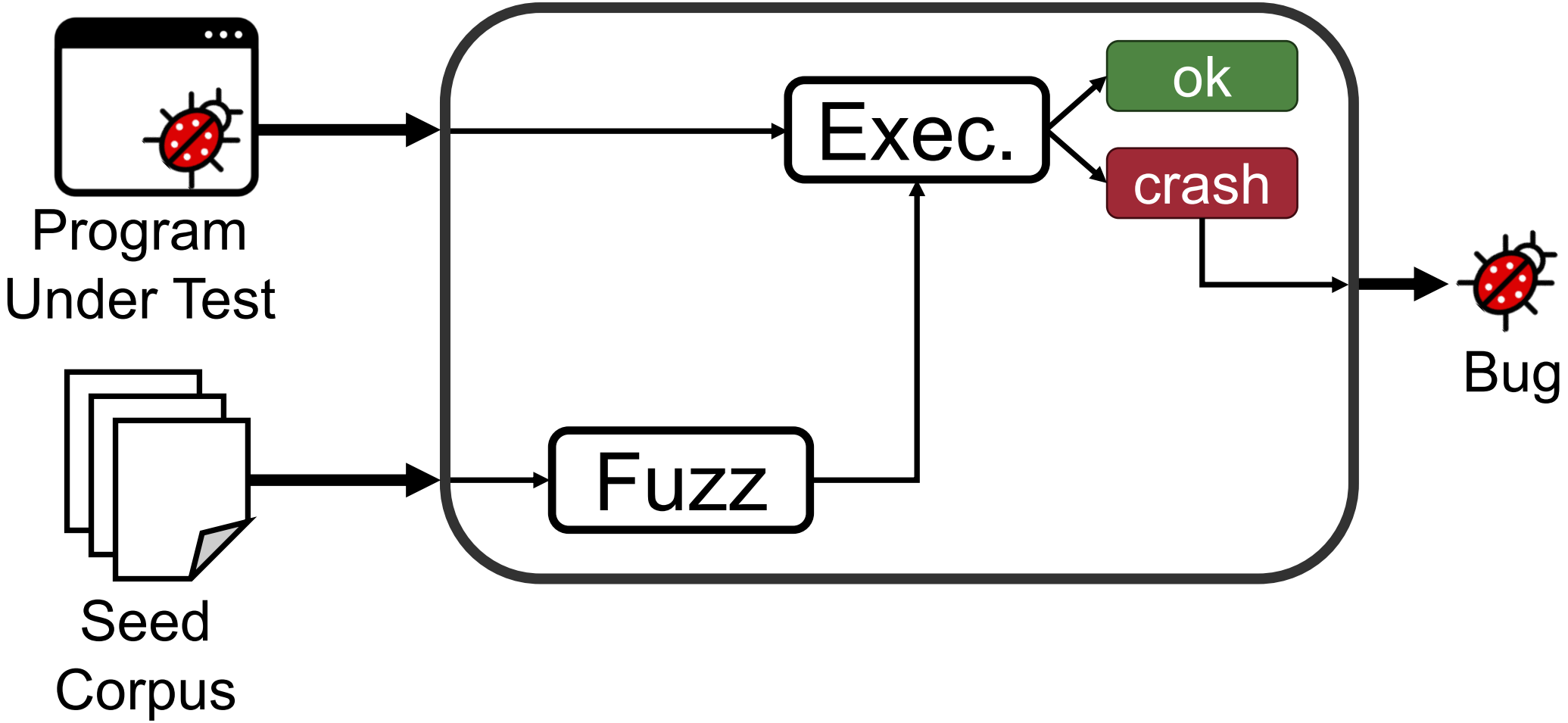
- Very low test coverage!

- Random inputs are often filtered out in earlier stages of programs
  - E.g., “Invalid syntax”
  - What are the chances of getting valid URL from random strings?



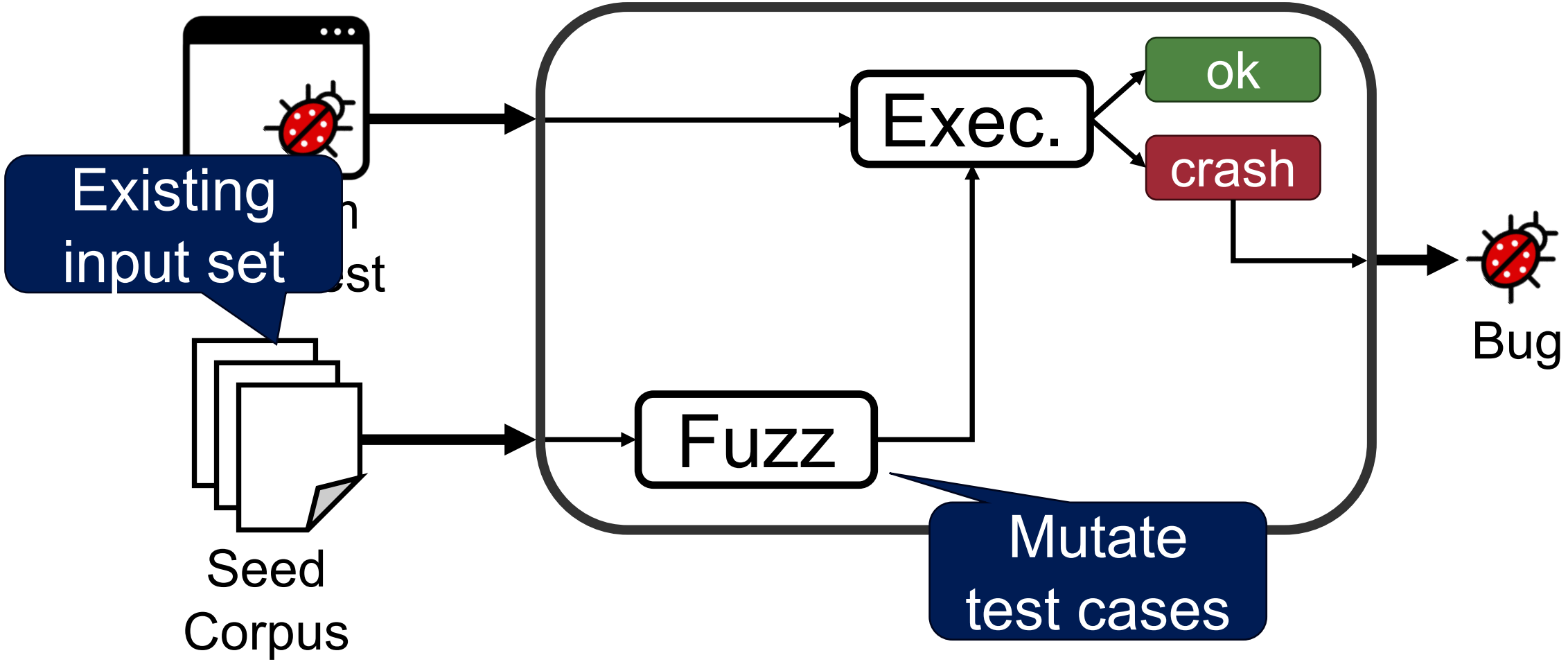
Mutation-based Fuzzing

- Produce new inputs by mutating existing valid inputs (seeds)



Mutation-based Fuzzing

- Produce new inputs by mutating existing valid inputs (seeds)



Mutation-based Fuzzing

- Produce new inputs by mutating existing valid inputs (seeds)

```
int x = source();
if (x[0] == ‘f’) {
    if (x[1] == ‘d’)
        bug()
    elif (x[1] = ‘c’)
        y = 2 * x;
}
```

Mutation-based fuzzing

Seed: fc

mutate!

fa

mutate!

fd

Found a bug!

Random input generation

Vjrt32^@@

91{sdaf]2’”

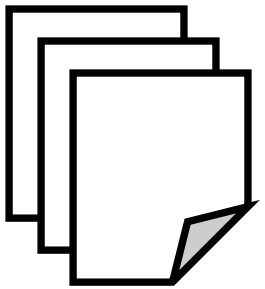
Basdf$!^7

Example Mutation Operations

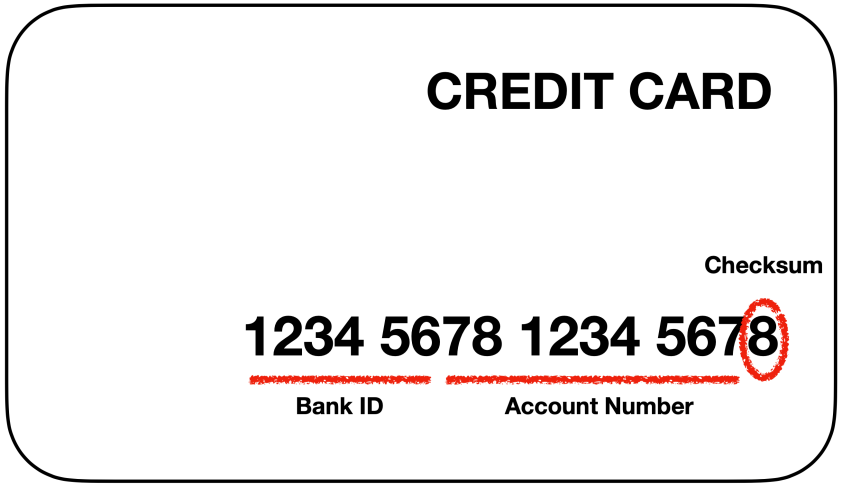
- **Random bit-flipping**: randomly flip bits with a certain probability
- **Arithmetic mutation**: perform simple arithmetic on a value (e.g., x + r)
- **Block-based mutation**: insert/delete/replace/permute/resize a subpart of an input
- **Dictionary-based mutation**: use a set of pre-defined values for mutation such as {0, -1, 1} or {“%s”, “%x”}
- **Crossover**: recombine two parents to generate new input (offspring)

Problems of Mutation-based Fuzzing

- Testing capability is limited by seed inputs

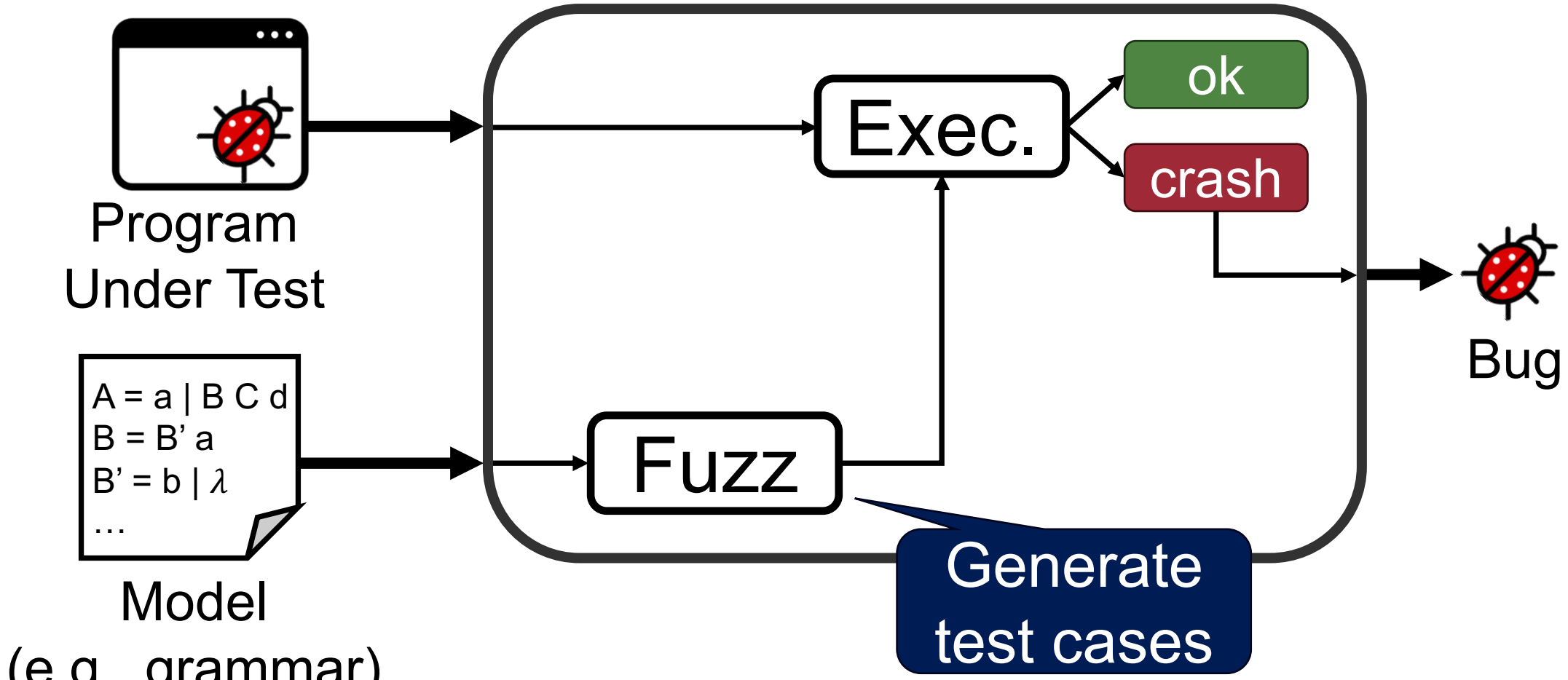


- Random mutations often violate complicated syntactic or semantics rules
  - E.g., XML, JavaScript, length field, checksum



Generation-based Fuzzing

- Generate inputs based on a **given model**
  - Manually defined or automatically inferred



Generation-based Fuzzing

- Generate inputs based on a **given model**
  - Manually defined or automatically inferred

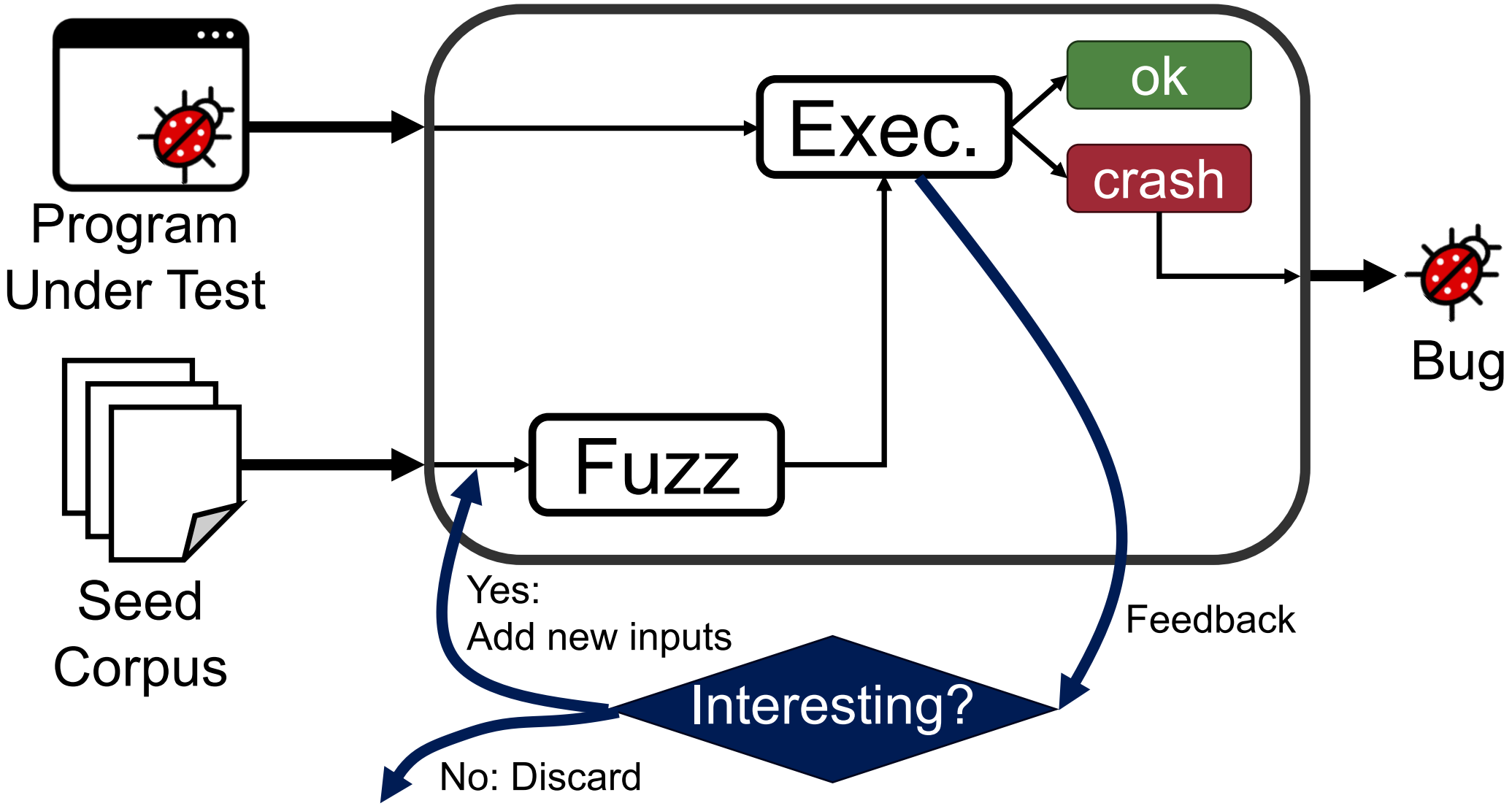
- If the input ***model*** is precise, we can generate valid inputs
  - JavaScript interpreter fuzzing (with JavaScript grammar as a model)
  - PNG parser fuzzing (with PNG file format as a model)

- Examples
  - Formatted data: PNG, MP3, etc.
  - Programs: JavaScript, PHP, C, etc.
  - Networks protocols: TLS, NFC, etc.

Maximizing the Ability of Fuzzing

- Problem: monotonous mutants
  - Same set of corpus + same set of mutation operators = ??
  - Limited search space!
- How to achieve divergence?
- Idea: divergence and selection (using find-grained feedback)

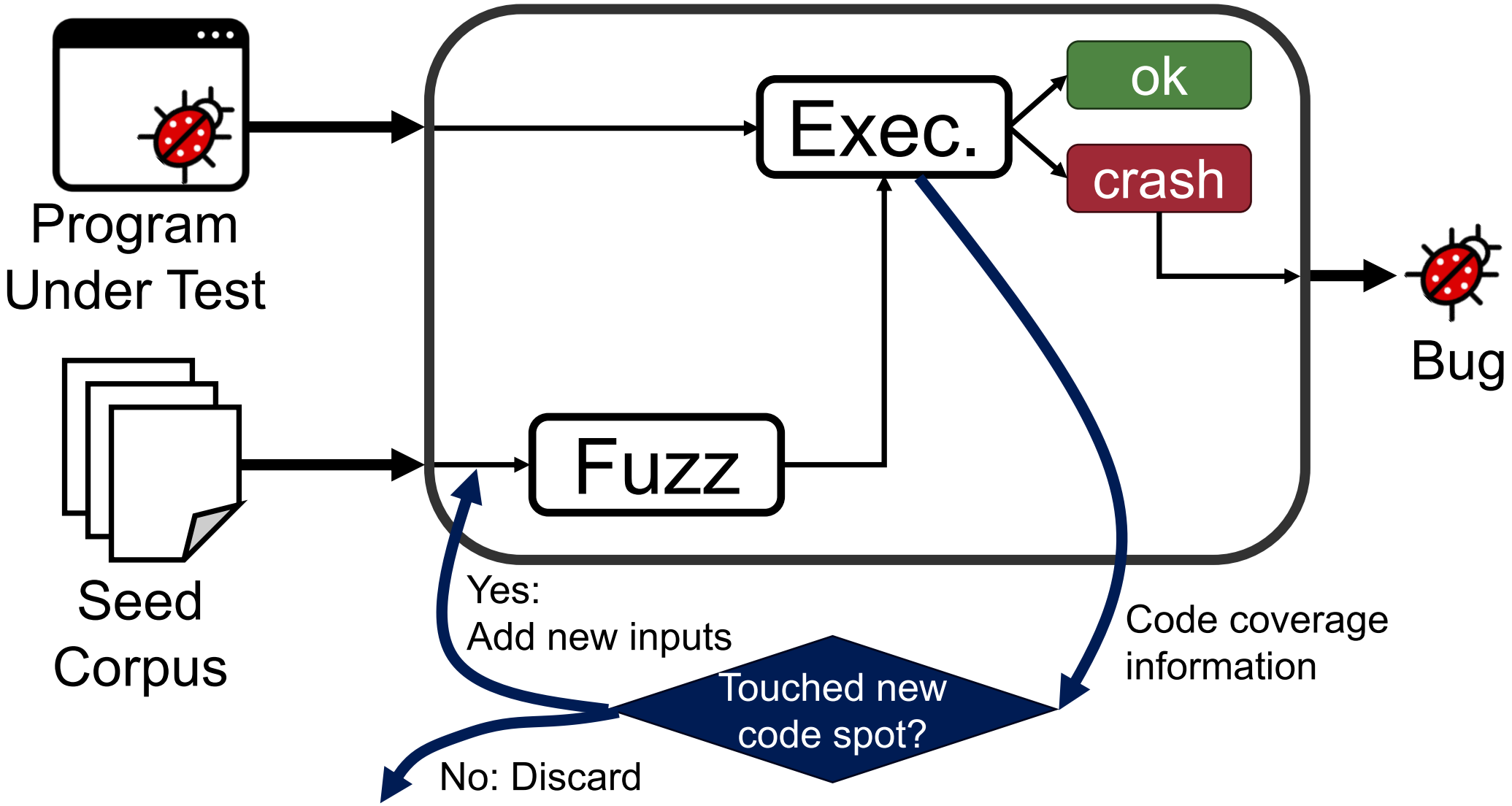
Feedback-driven Fuzzing



Recap: Grey-box Fuzzing

- White-box fuzzing (strictly speaking)
- Obtain "*some*" partial information about the program execution
  - E.g., code coverage information

Coverage-guided fuzzing:
an example of the Feedback-driven Fuzzing

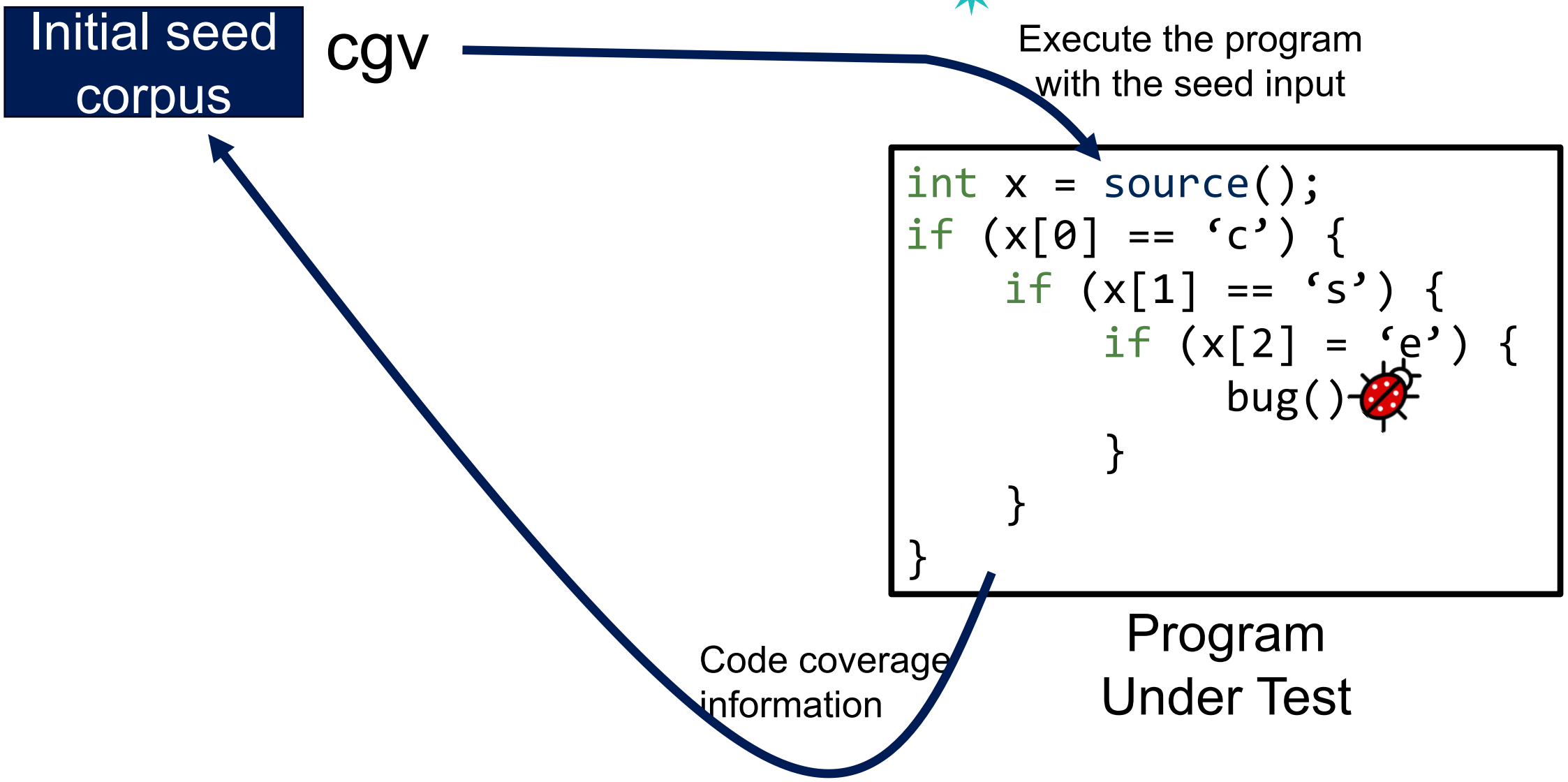


Coverage-guided Fuzzing

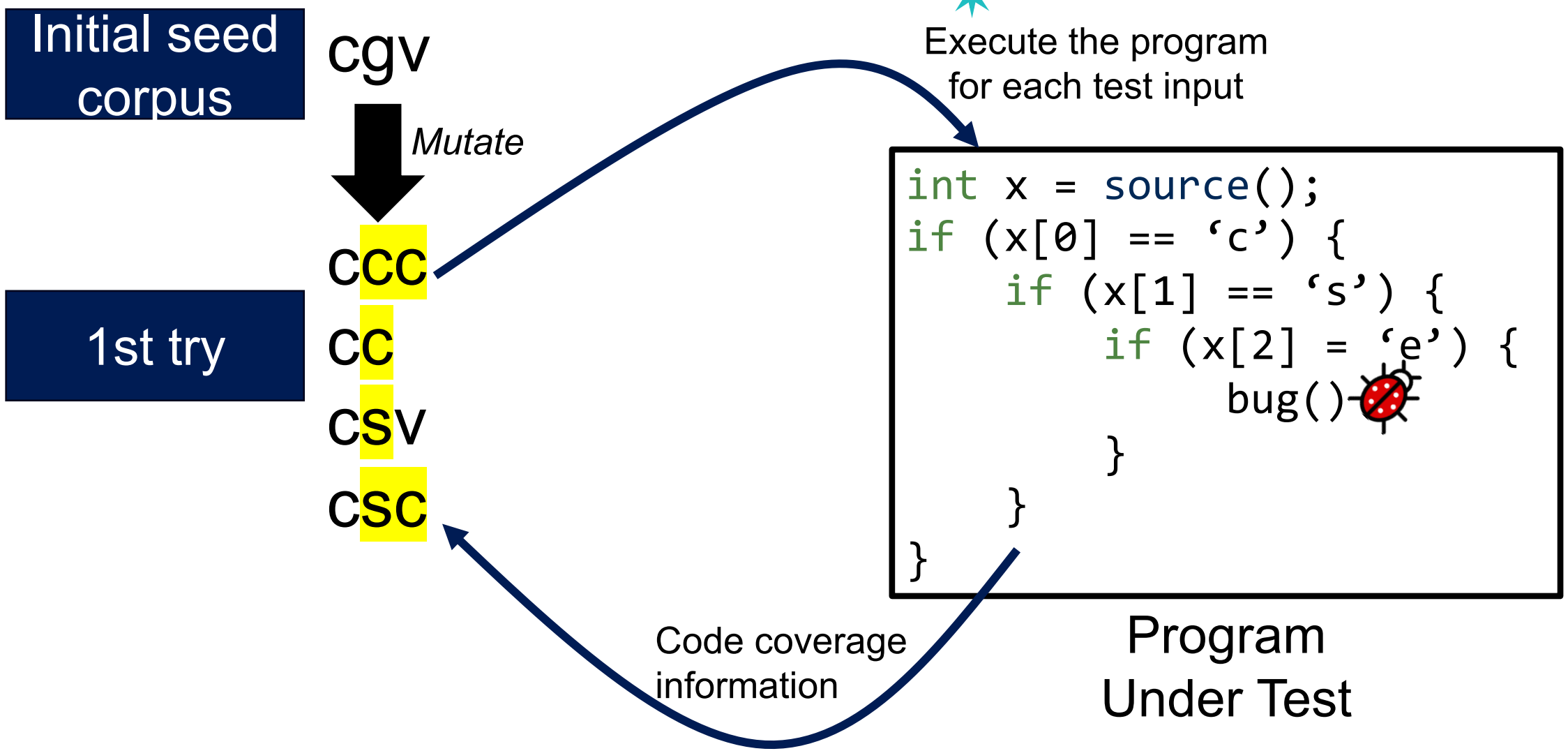
- Fuzzing as a **genetic algorithm**
  - Chromosome population: seed corpus
  - Genetic mutation: input mutation
  - Fitness function: coverage

- **Key idea**: keep mutants that increase ***code coverage*** for future mutations!
  - Grey-box fuzzing!
  - E.g., AFL, LLVM's libFuzzer

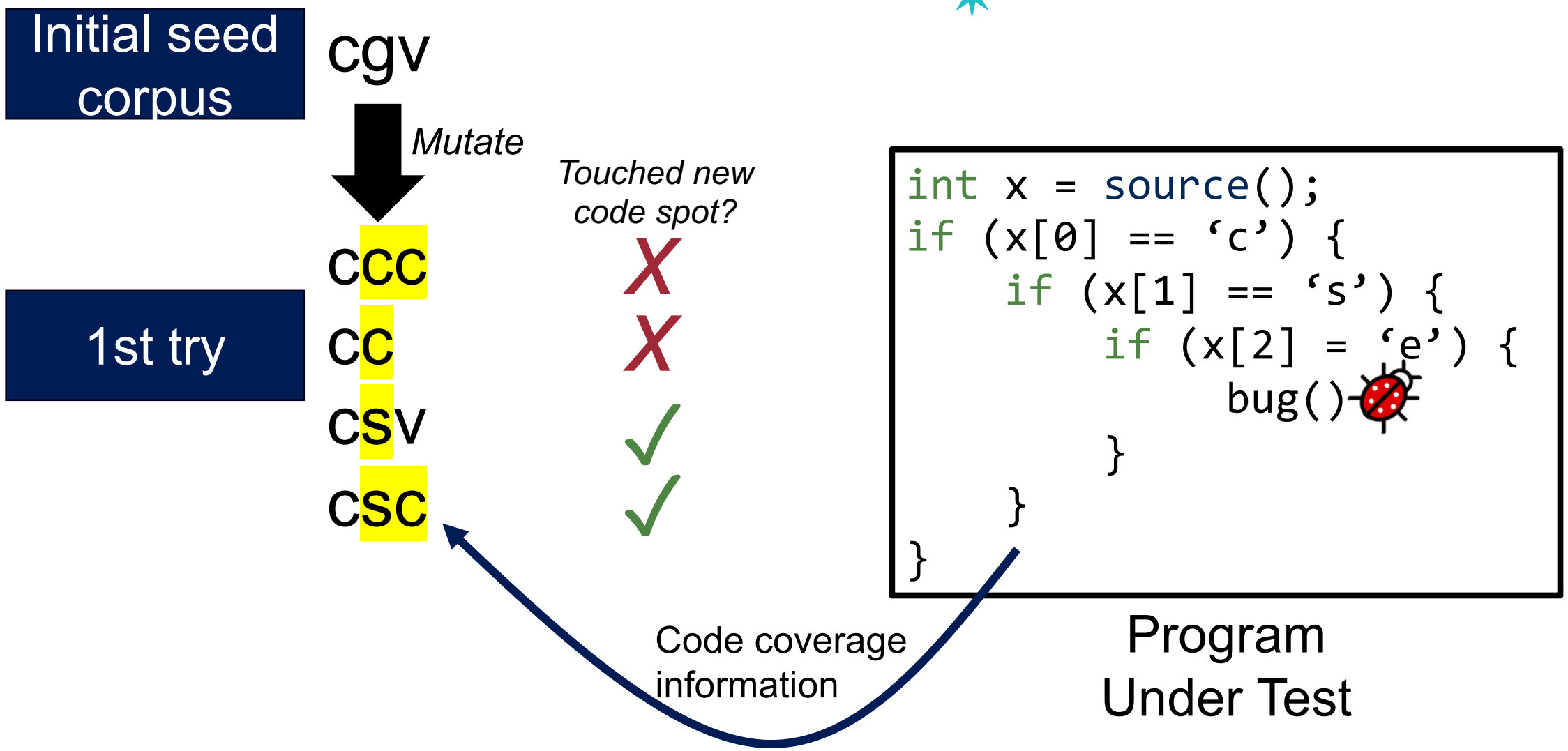
Coverage-guided Fuzzing



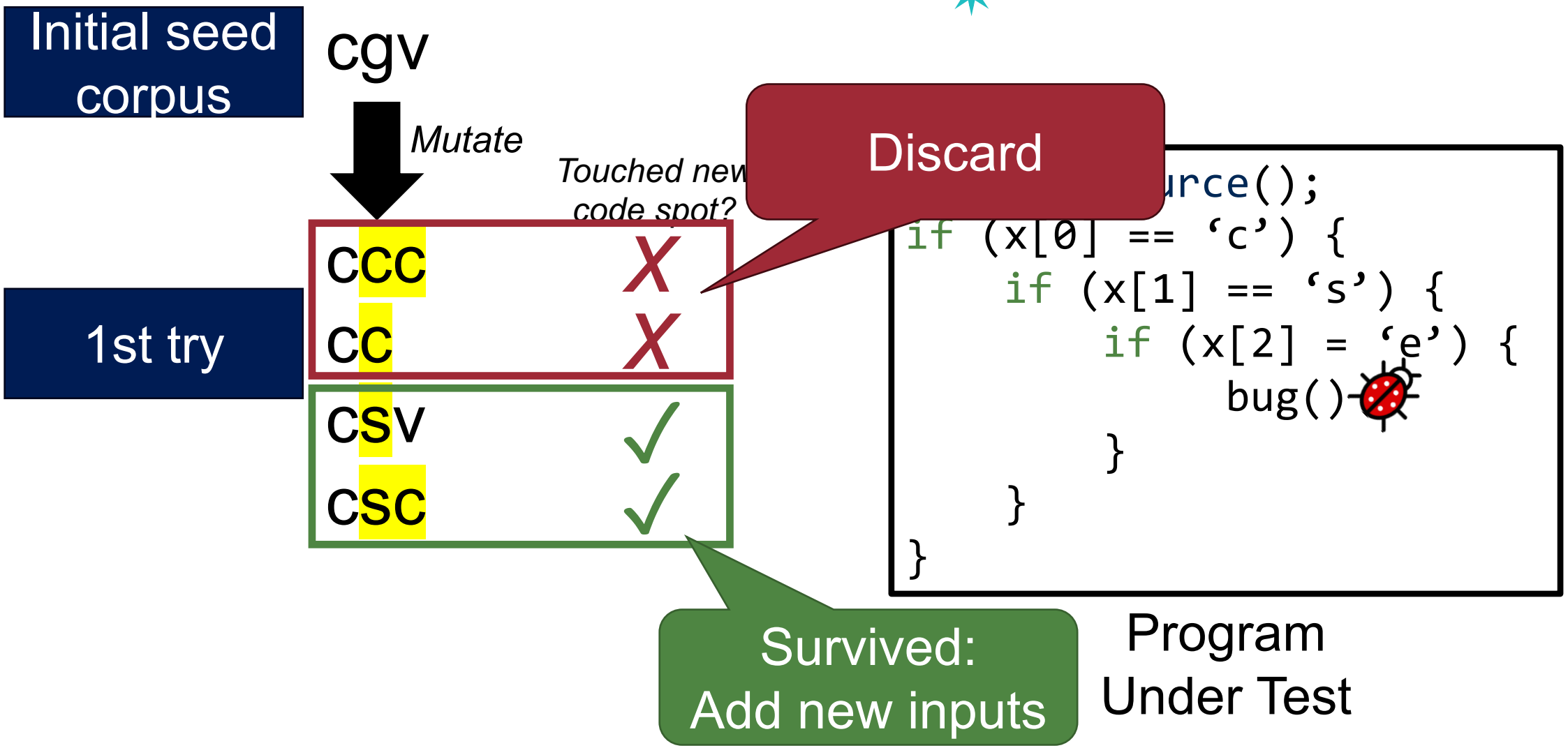
Coverage-guided Fuzzing



Coverage-guided Fuzzing



Coverage-guided Fuzzing



Coverage-guided Fuzzing

Initial seed corpus

cgv

Mutate

1st try

csv

csc

Mutate

csvv

2nd try

cev

csf

cse

Found a bug!

Execute the program for each test input

```
int x = source();
if (x[0] == 'c') {
    if (x[1] == 's') {
        if (x[2] = 'e') {
            bug()
        }
    }
}
```

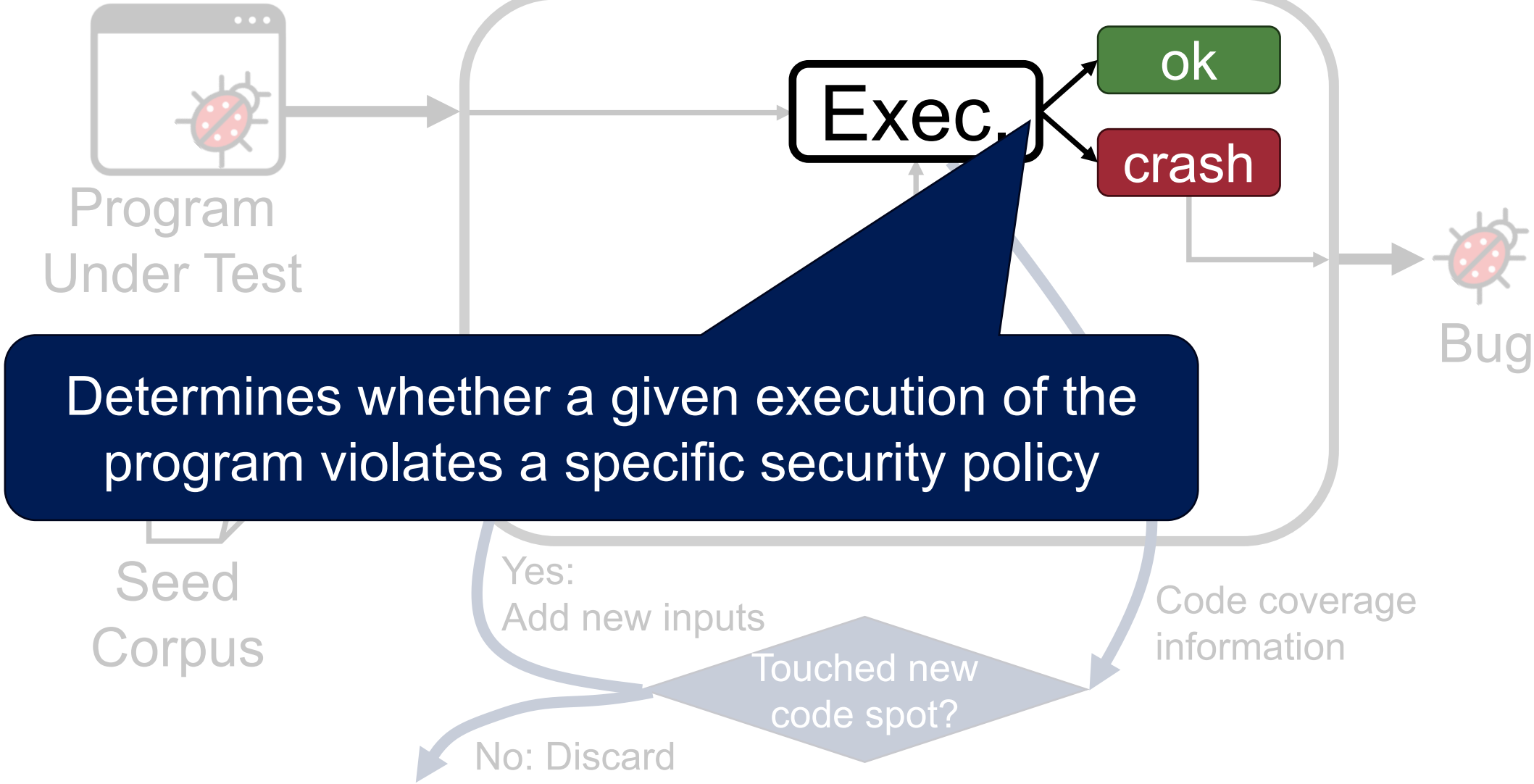
Program Under Test

Code Coverage

- A metric that determines how much code has been executed
- Obtained from the runtime information
  - E.g., LLVM's SanitizerCoverage, gcov, etc.
- Many criteria: line/statement coverage, branch coverage, path coverage, etc.
- How to measure? Instrumentation!
  - If source code available: instrumentation via compilation (e.g., LLVM's sanitizers)
  - If no source code available: binary rewriting (e.g., Pin tool) or emulation (e.g., QEMU)

Add specific code to the source code or binary under analysis

Bug Oracle



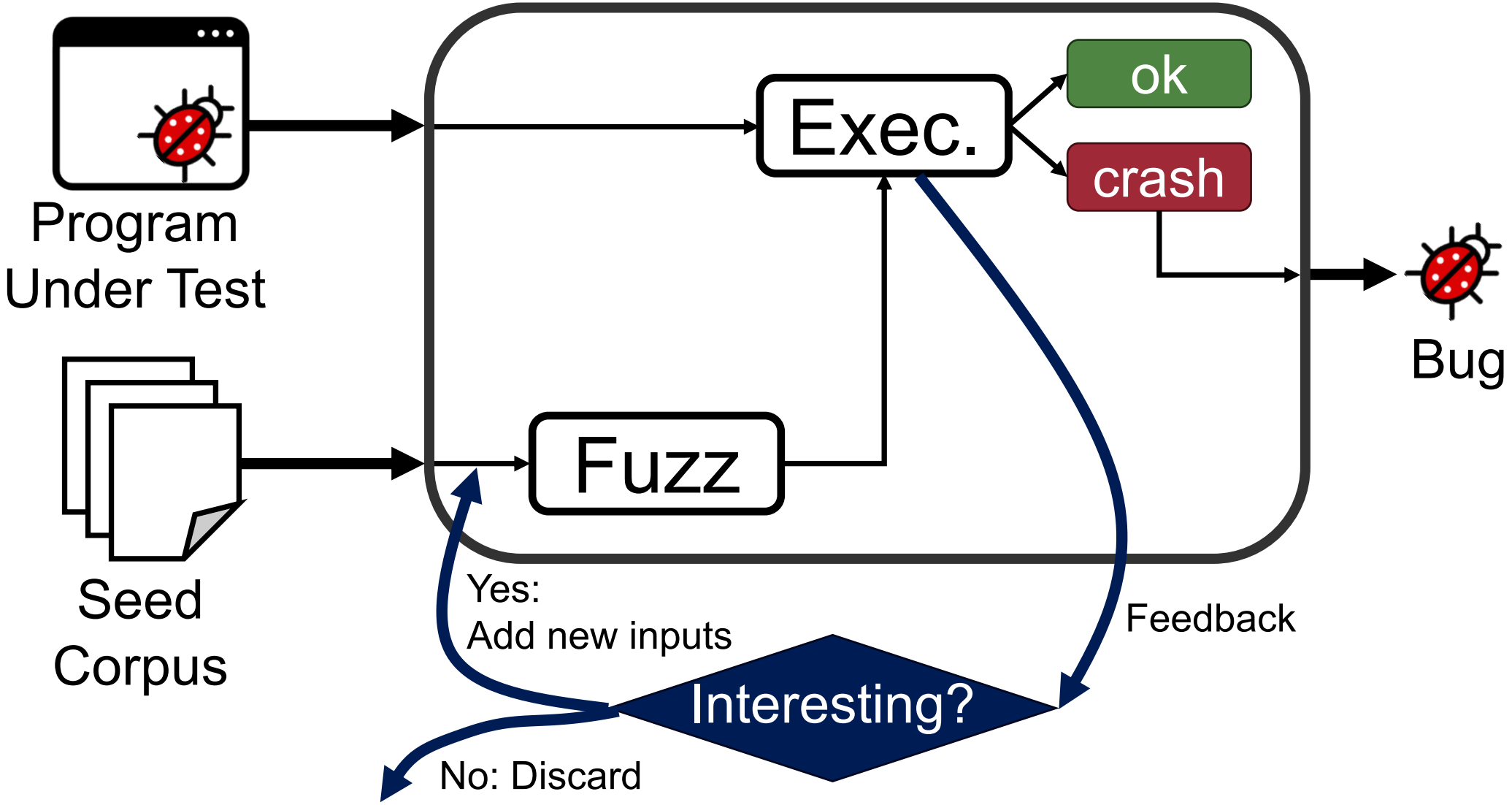
Determines whether a given execution of the program violates a specific security policy

Bug Oracle

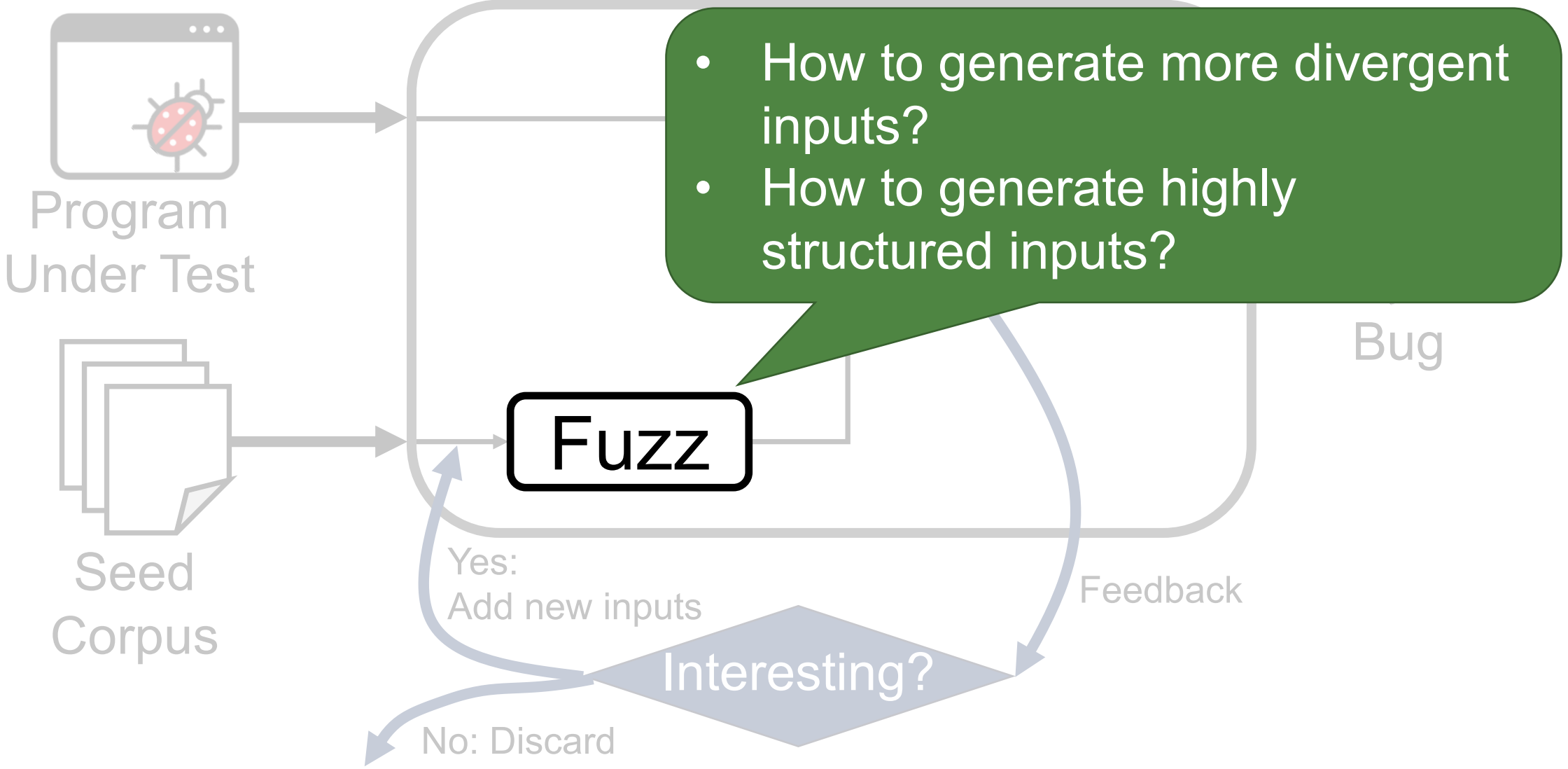
- Catch a fatal signal (SIGSEGV, SIGILL, SIGABRT, …)
- Also use sanitizers (ASAN, MSAN)

- In web security,
  - XSS found if alert occurs
  - Etc.

More Research Questions

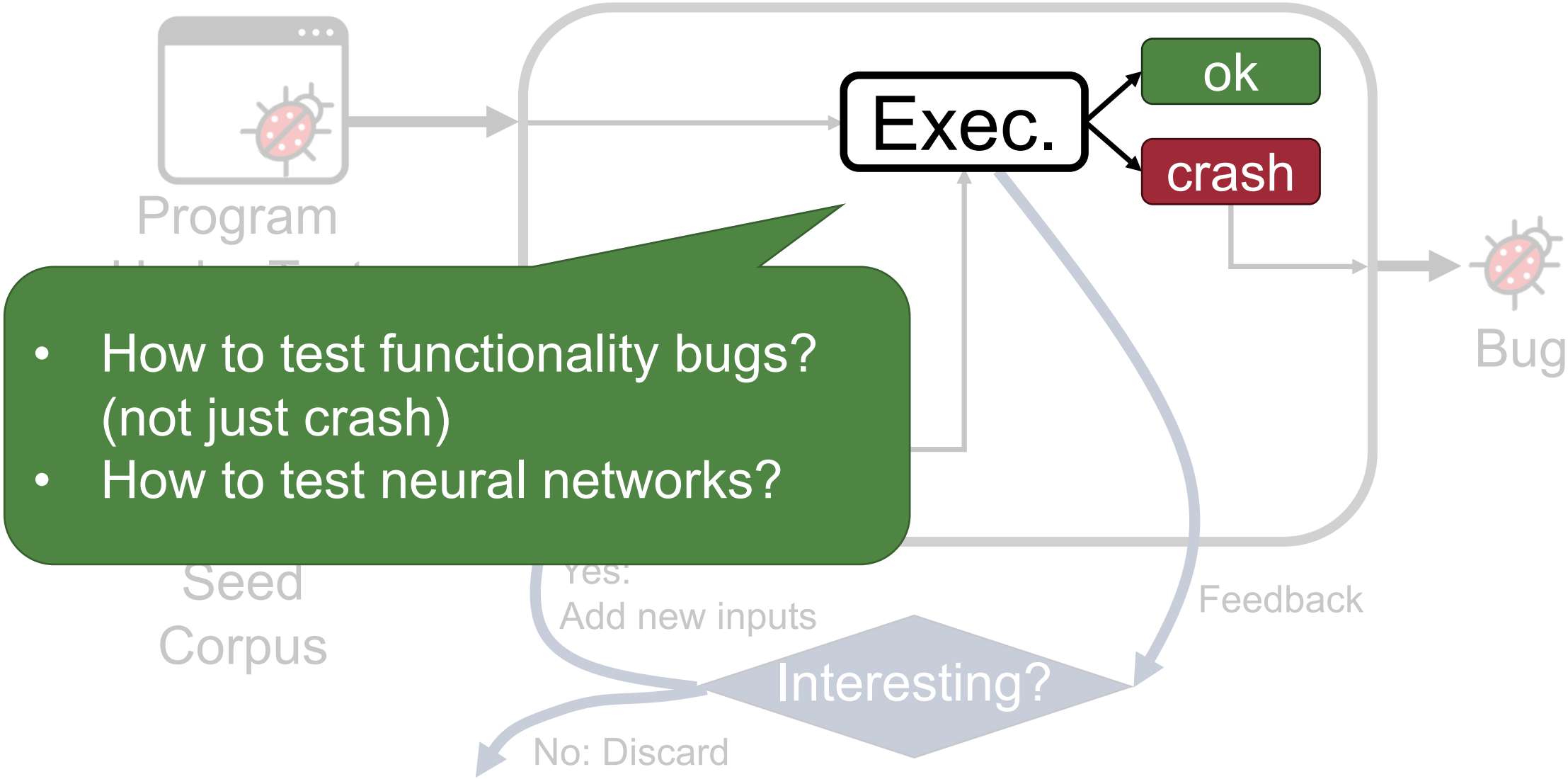


More Research Questions



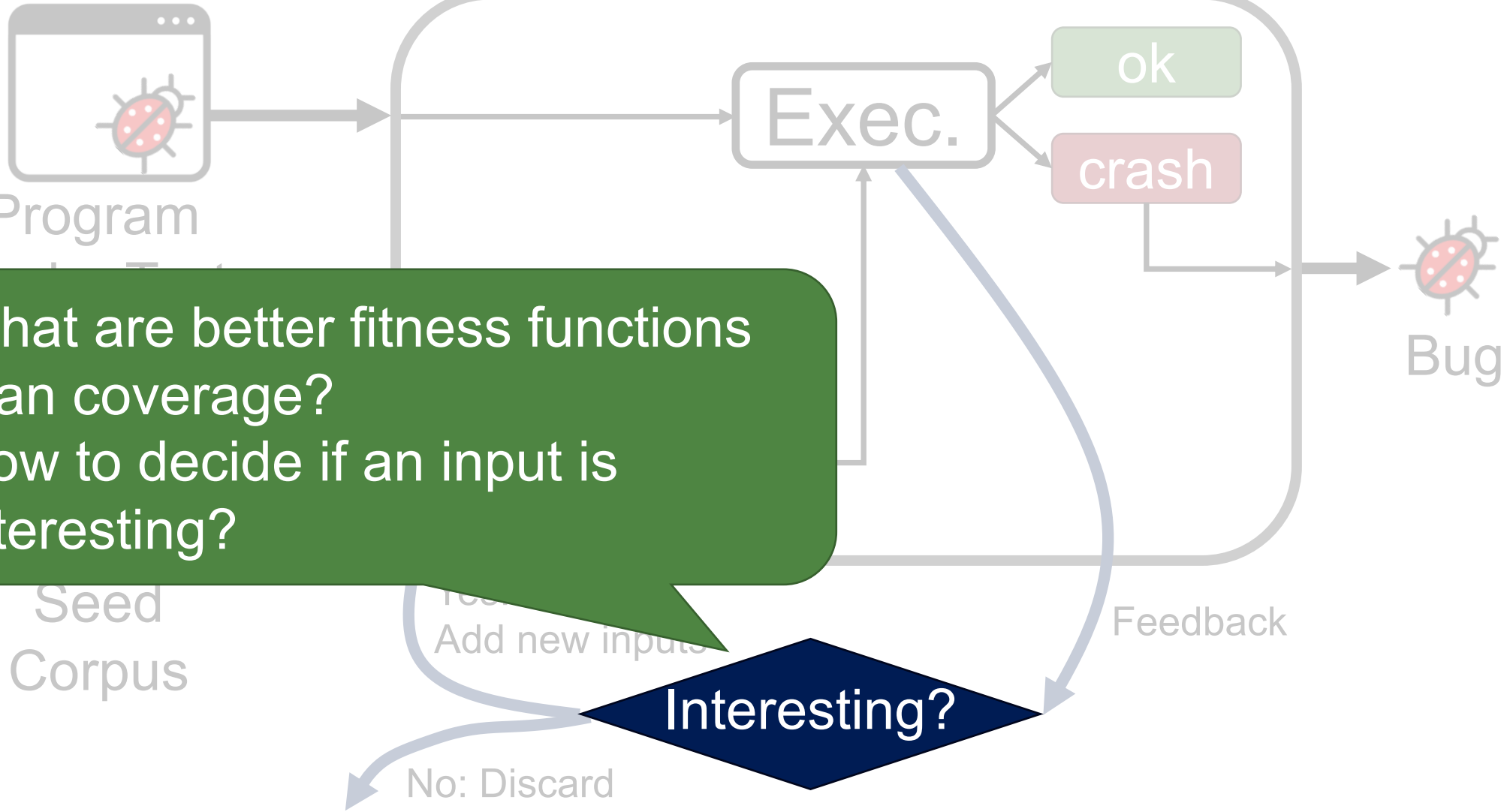
- How to generate more divergent inputs?
- How to generate highly structured inputs?

More Research Questions



- How to test functionality bugs? (not just crash)
- How to test neural networks?

More Research Questions



- What are better fitness functions than coverage?
- How to decide if an input is interesting?

Summary

- Fuzzing: efficient and effective testing technique
  - Input generation: mutation-based, generation-based
  - Feedback: black-box, grey-box, white-box

- Challenges
  - Efficiency (e.g., higher coverage)
  - Expressiveness (e.g., functionality errors)
  - Etc.