Introduction to Information Security

12. Dynamic Analysis and Fuzzing

Kihong Heo

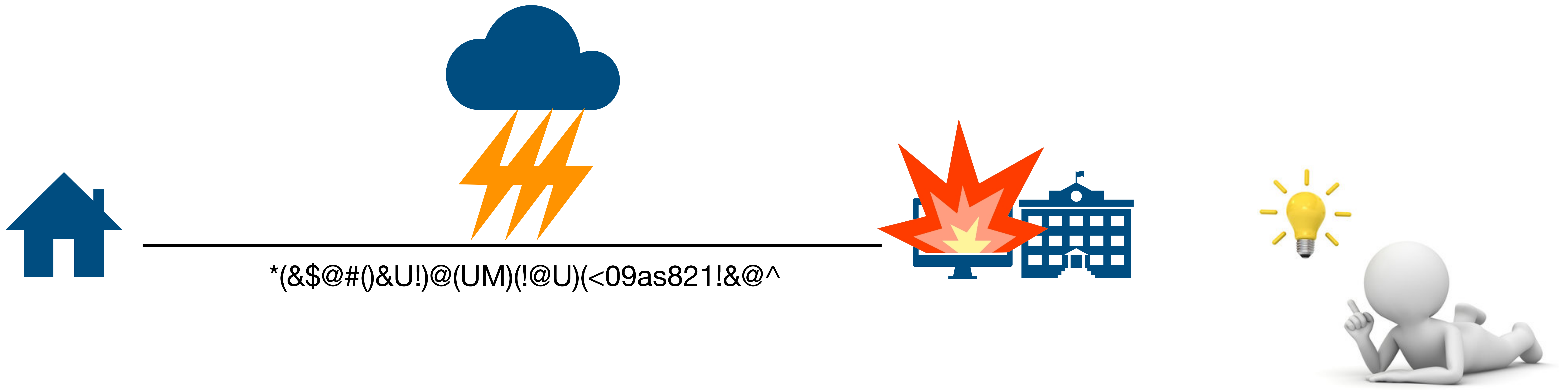
KAIST

Testing

- Check a set of **finite executions**
  - e.g., random testing (a.k.a fuzzing)
- In general, **unsound yet complete**
  - Unsound: cannot prove the absence of errors
  - Complete: produce counterexamples (i.e., erroneous inputs)
- Example: Google's oss-fuzz (https://github.com/google/oss-fuzz)

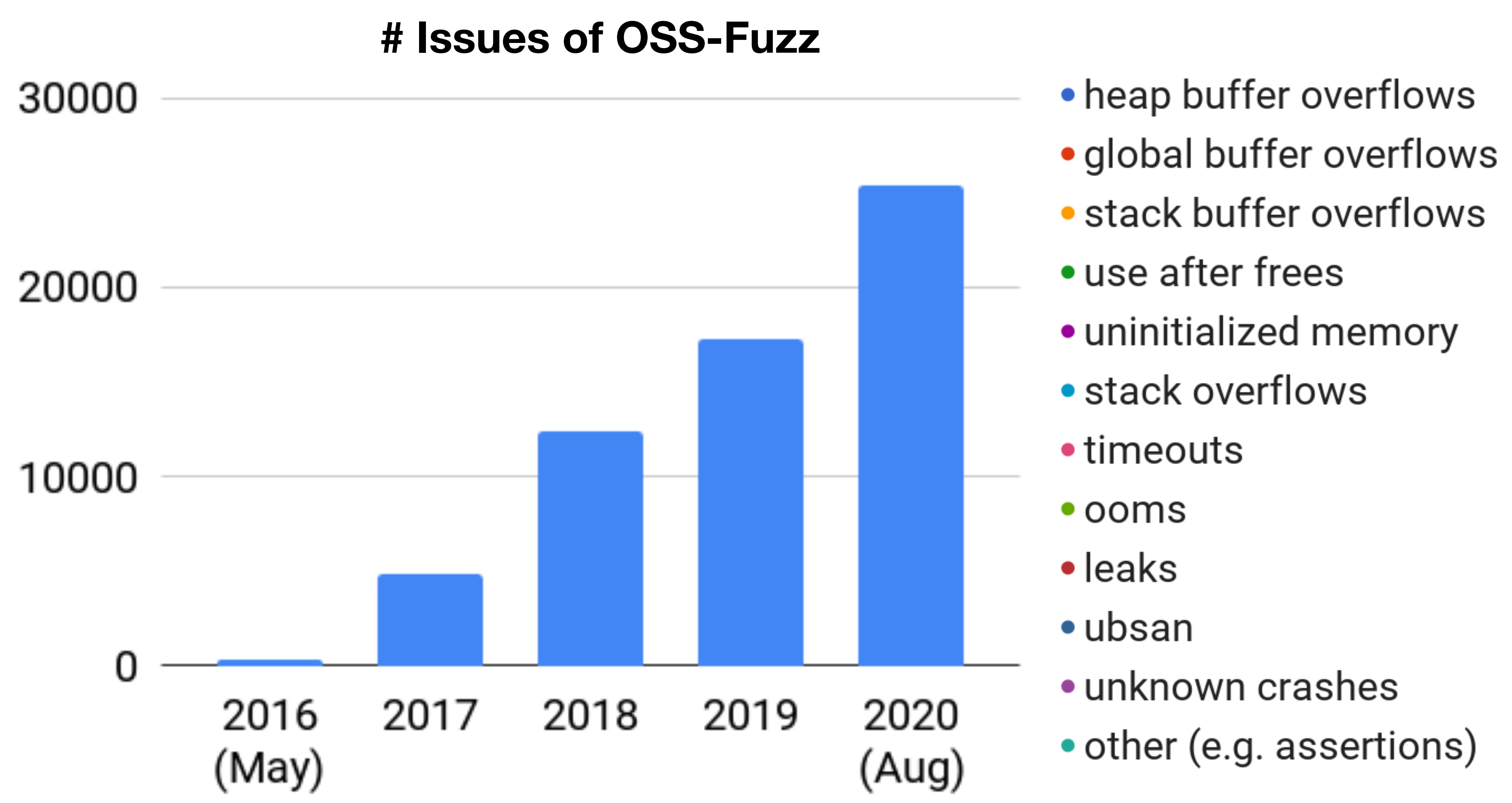
Fuzzing마구실행

- Initially, developed by Barton Miller in 1988
- Thunderstorm → noise on a network line → random characters → crash
- “Can we mimic the thunder-generated noise to check robustness?”

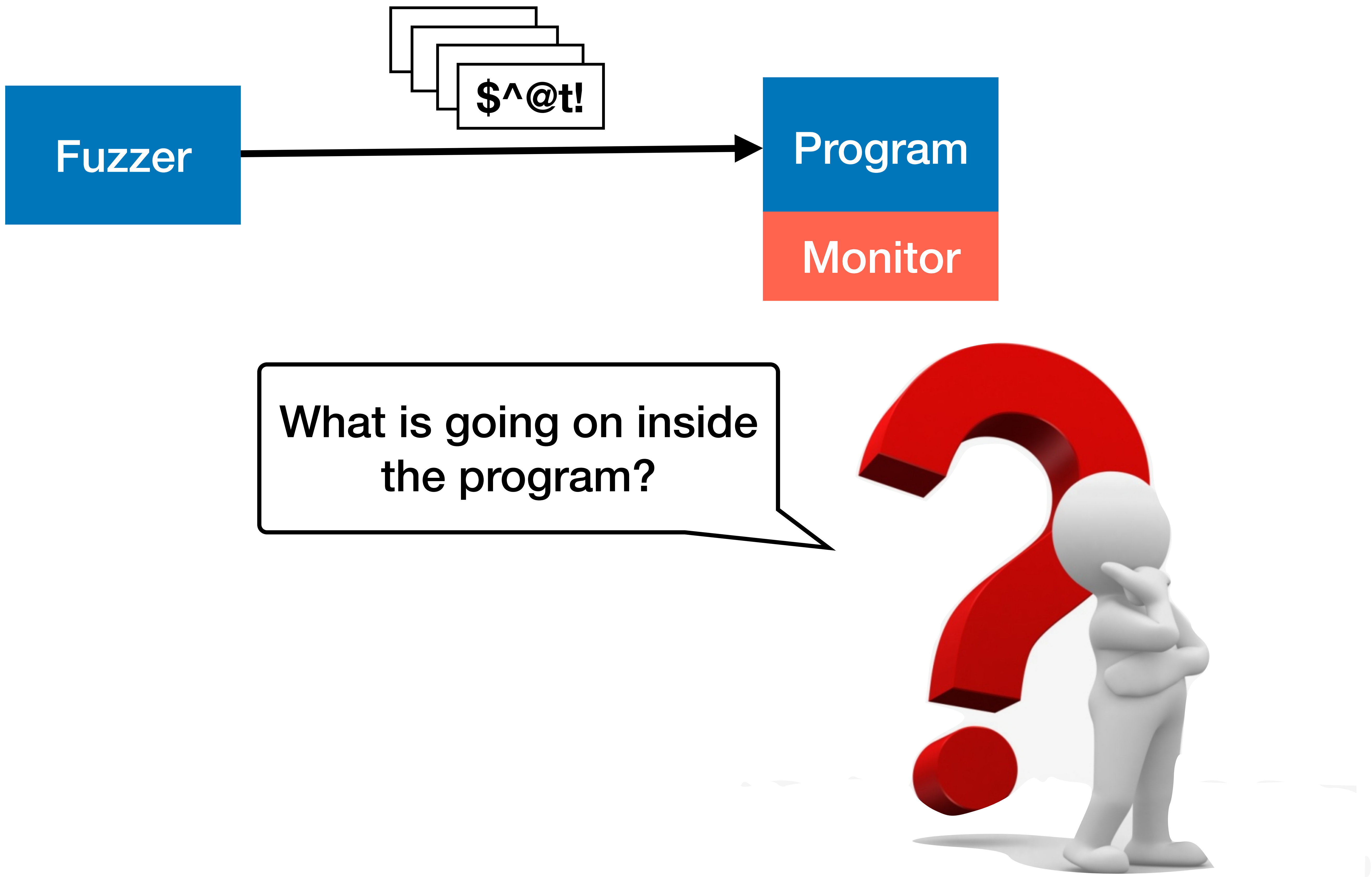


Success Stories

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



Fuzzing Overview



Runtime Monitoring

- Observe program executions
  - Explicitly trap the execution if an error happens
  - Give more detailed information (e.g., bug type, coverage, location)
- Important: achieving low overhead at runtime
- How?
  - If source code available: instrumentation via compilation (e.g., LLVM's sanitizers)
  - If no source code available: binary rewriting (e.g., Pintool) or emulation (e.g., QEMU)

A Simple Fuzzing

- (Generate random inputs; run a given program w/ the inputs; see if it crashes)+
- Without considering the target program's detailed behavior
  - So called, **blackbox** fuzzing
- Initial success: a command-line fuzzer for UNIX utilities*
- The infinite monkey theorem actually works!

"A monkey randomly hitting keys for an infinite amount of time will produce any given text, such as Shakespeare's Hamlet"

*Barton Miller, et al., An Empirical Study of the Reliability of UNIX Utilities, *CACM*, 1990

Blackbox불투명 vs Whitebox투명

- Blackbox: generate inputs **regardless** of program's logic and structure
  - Easy to implement and low cost
  - Hard to explore deeper parts

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

- Whitebox: generate inputs by **observing** program's logic and structure
  - Can explore deeper parts
  - Require constraint solving (high overhead)

```
x = input();
y = input();
if (x < y)
  if (x^2 > 3)
    bug();
```

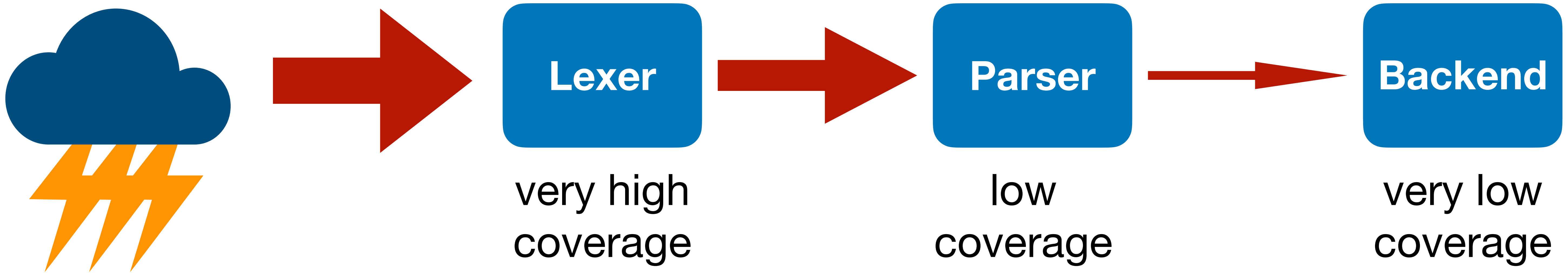
```
a = input();
b = input();
c = input();
n = input();

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

Examples

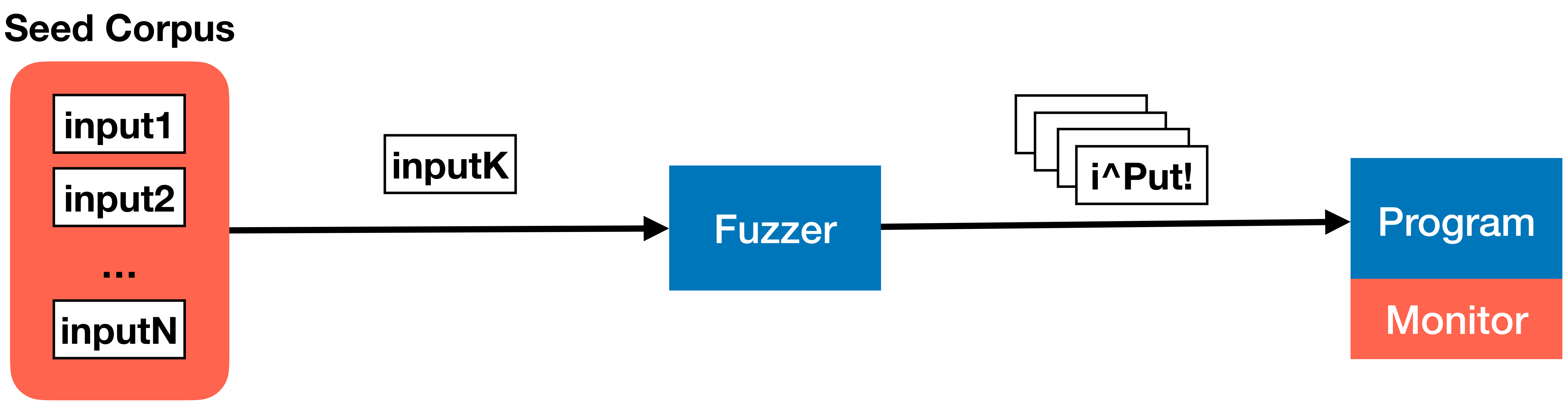
- Monkey: a random testing tool for Android apps
  - Randomly generate streams of events such as clicks, touches, etc
  - A part of Android Studio
- Cuzz: a random testing tool for finding concurrency bugs
  - Randomly generate thread schedules
  - A part of Microsoft Application Verifier

Problems

- 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?
  - URL format: `scheme://netloc/path?query#fragment`



Fuzzing Overview



Mutation-based Fuzzing

- Idea: generate new inputs by **mutating** existing valid inputs (seeds)
  - valid inputs usually reach deeper parts
- Mutation operators: random flips or heuristics
- Success story: AFL (2013)
  - See this blog post by the author of AFL*

*https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

Example: Mutating URLs

Seed input: http://www.google.com/search?q=fuzzing

randomly insert/delete/flip characters

0 mutations: http://www.google.com/search?q=fuzzing
5 mutations: http:/L/www.googlej.com/seaRchq=fuz:ing
10 mutations: http:/L/www.ggoWglej.com/seaRchqfu:in
15 mutations: http:/L/wwggoWglej.com/seaR3hqf,u:in
20 mutations: htt://wwggoVgle"j.som/seaR3hqf,u:in
25 mutations: htt://fwggoVgle"j.som/eaRd3hqf,u^:in
30 mutations: htv://>fwggoVgle"j.qom/ea0Rd3hqf,u^:i
35 mutations: htv://>fwggozVle"Bj.qom/eapRd[3hqf,u^:i
40 mutations: htv://>fwgeo6zTle"Bj.\'qom/eapRd[3hqf,tu^:i
45 mutations: htv://>fwgeo]6zTle"BjM.\'qom/eaR[3hqf,tu^:i

*https://www.fuzzingbook.org/html/MutationFuzzer.html

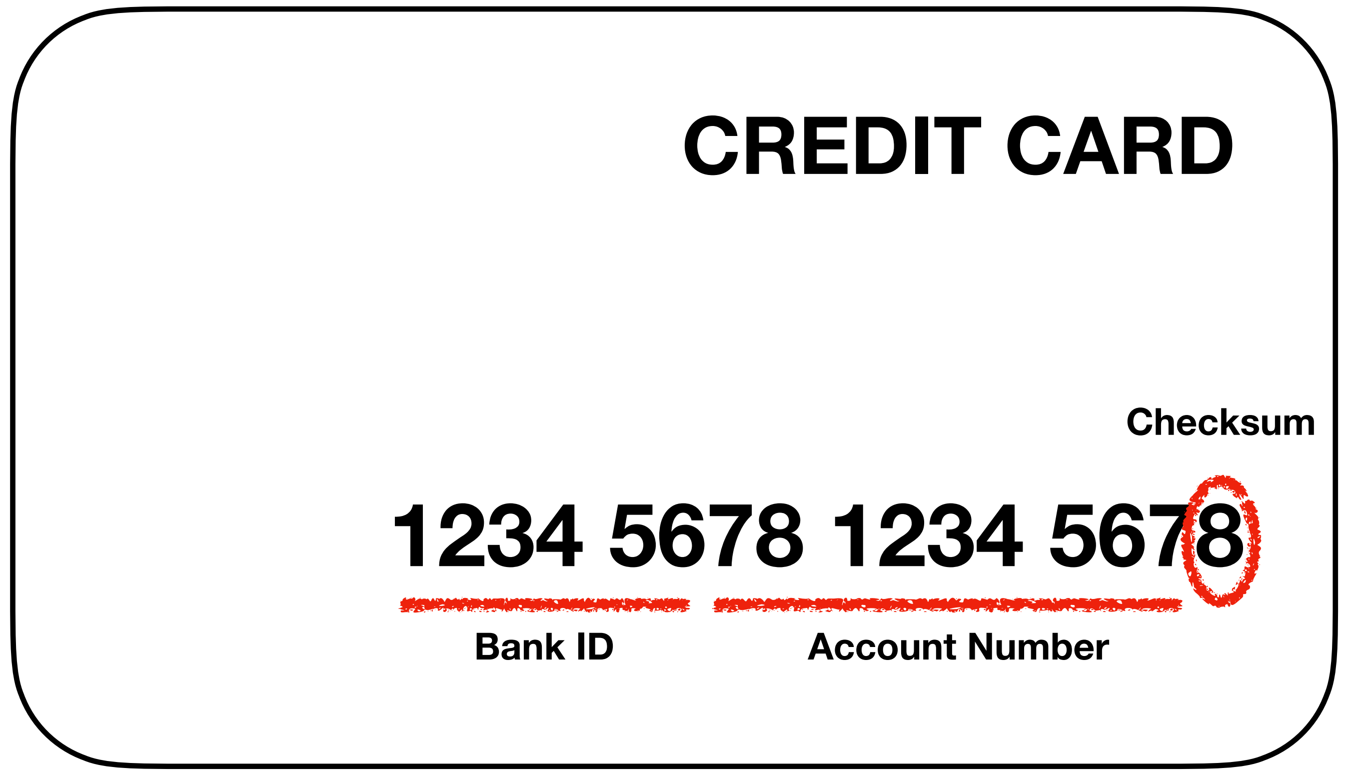
Example Mutation Operators

- **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”}

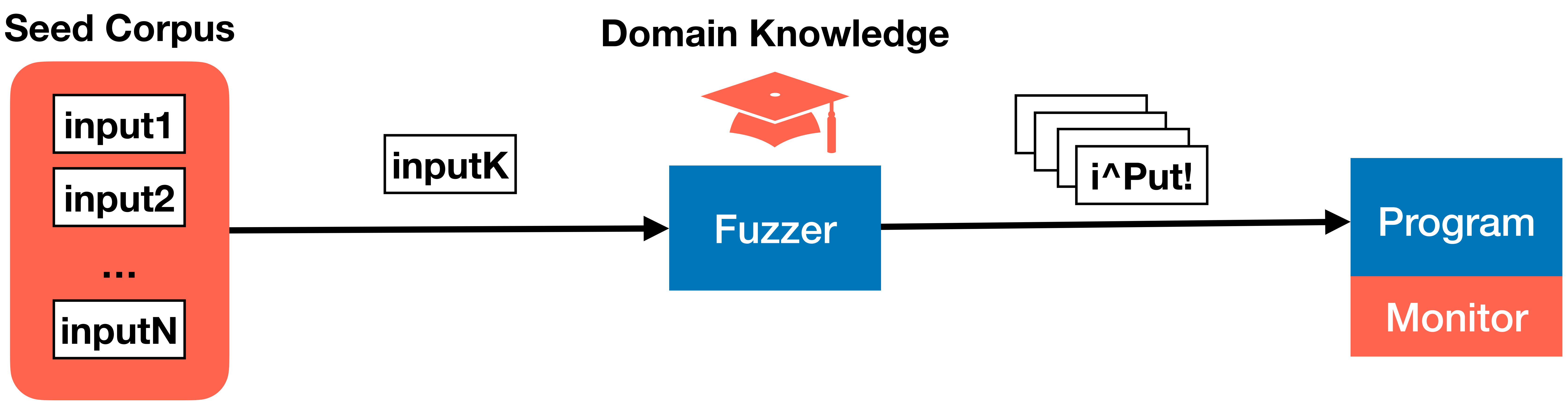
*Manes et al., The Art, Science, and Engineering of Fuzzing: A Survey, TSE, 2019

Problems

- Limited by seed inputs
- Random mutations often violate complicated syntactic or semantics rules
  - E.g., XML, Javascript, magic number, checksum



Fuzzing Overview



Generation-based Fuzzing

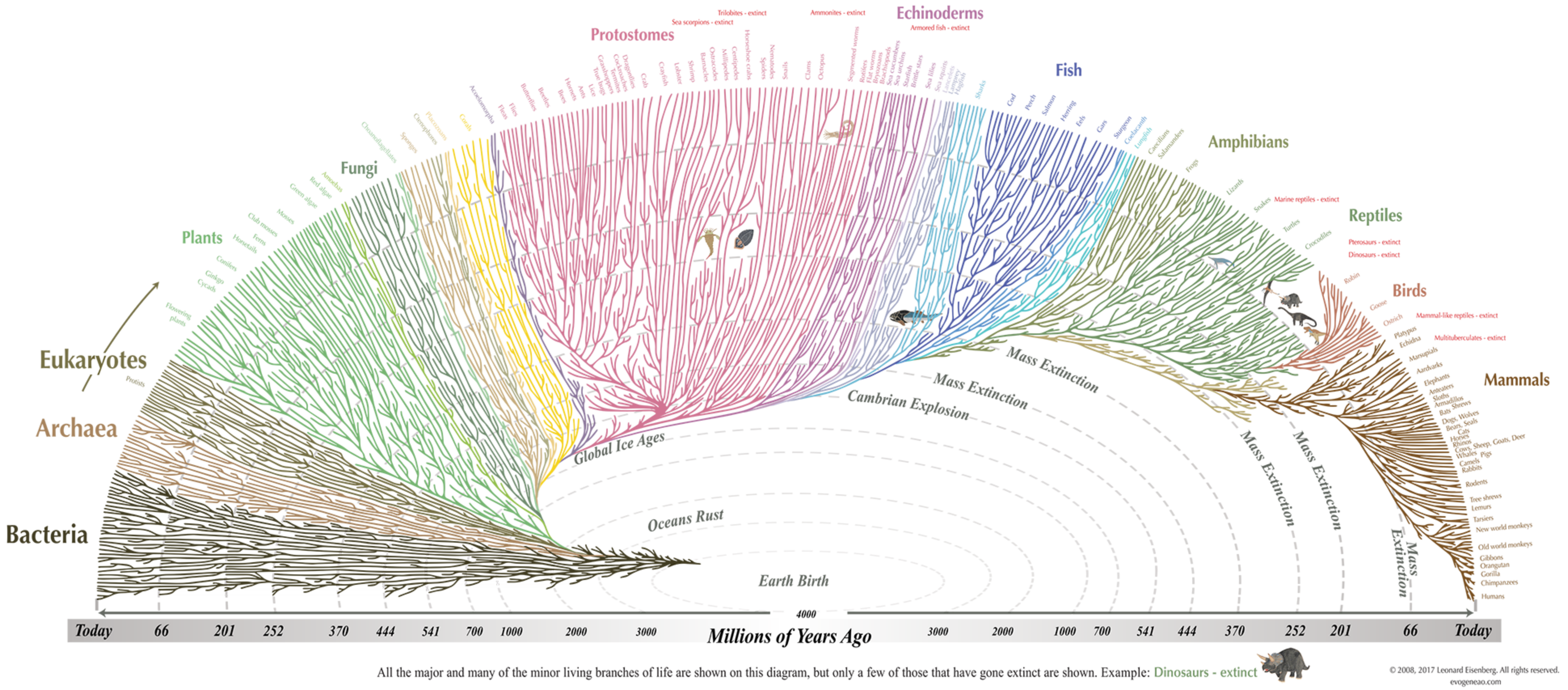
- Generate inputs based on a **given model**
  - manually defined or automatically inferred
- Examples:
  - Kernel APIs: system call templates (i.e., function signatures)
  - Formatted data: DOM objects, PNG, MP3, etc
  - Programs: Javascript, PHP, C, etc
  - Network protocols: TLS, NFC, etc

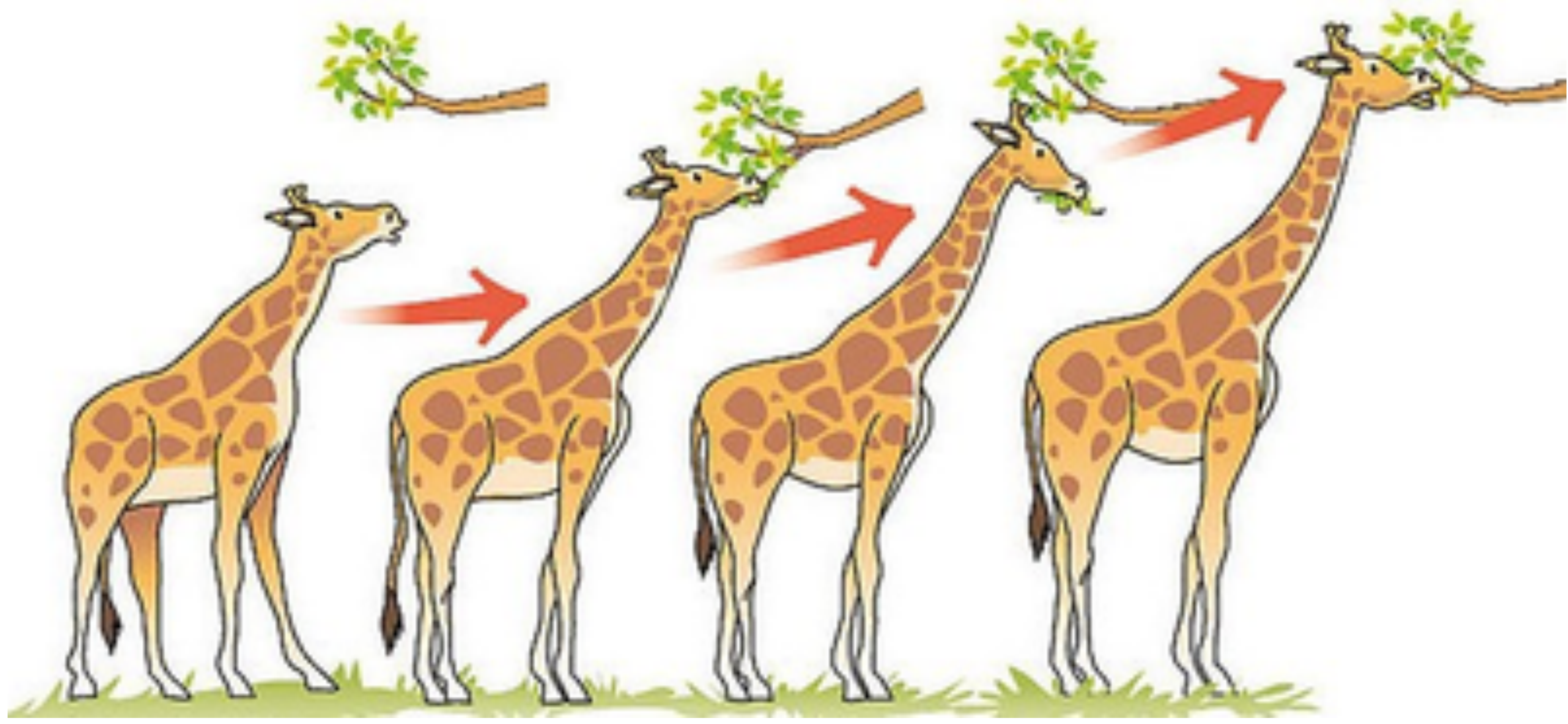
Problem 1: Still in the Dark

- How much fuzzing is enough?
- How to evaluate fuzzer’s performance?
    - Are we going in the right direction?
    - Or stuck at some point?
- Which mutant is better than others?
    - Does it explore new execution paths?
    - Or revisit redundant paths?
- Solution: gradual, continuous, fine-grained feedback

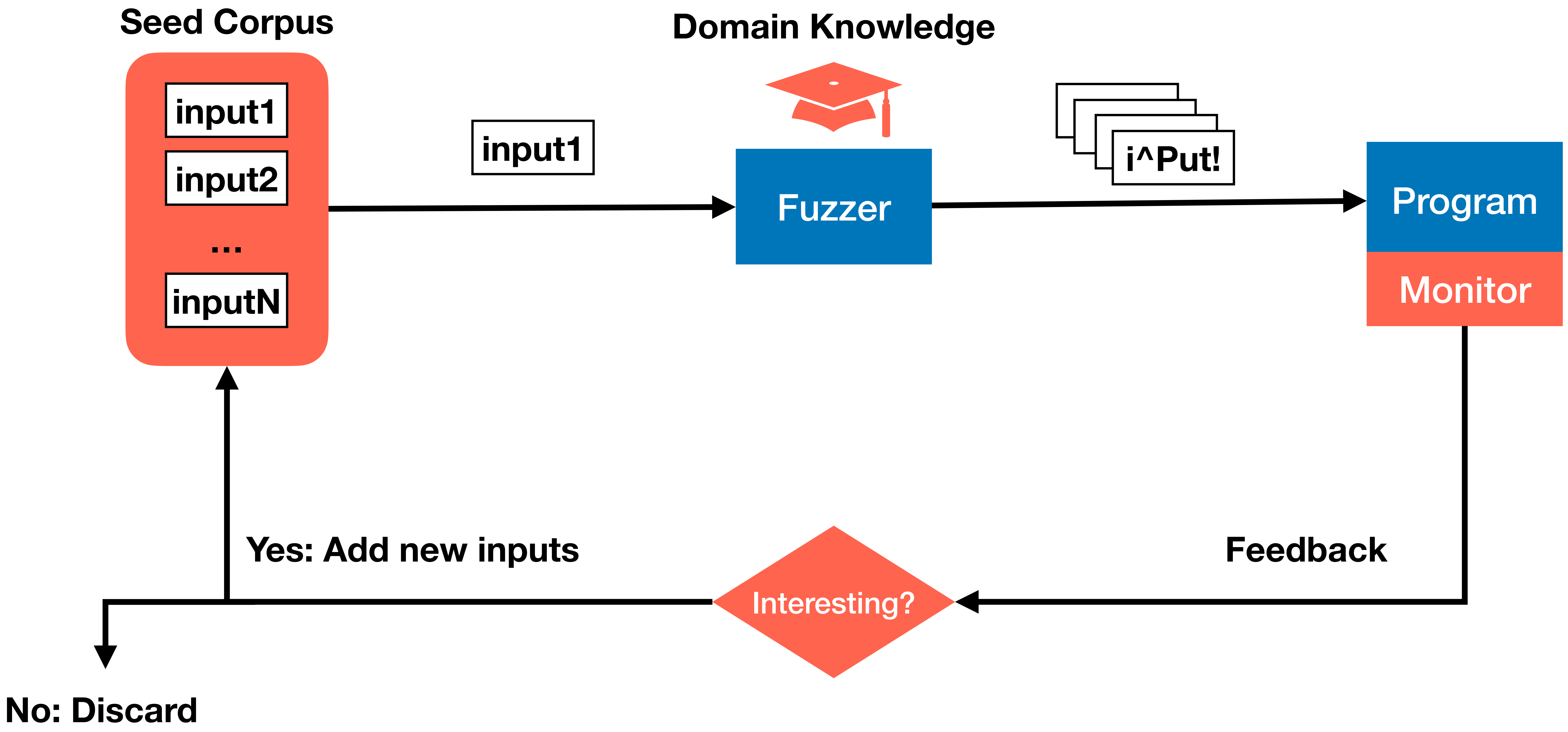
Problem 2: Monotonous Mutants

- Same set of corpus + same set of mutation operators = ??
- How to achieve divergence?
- Idea: divergence and selection



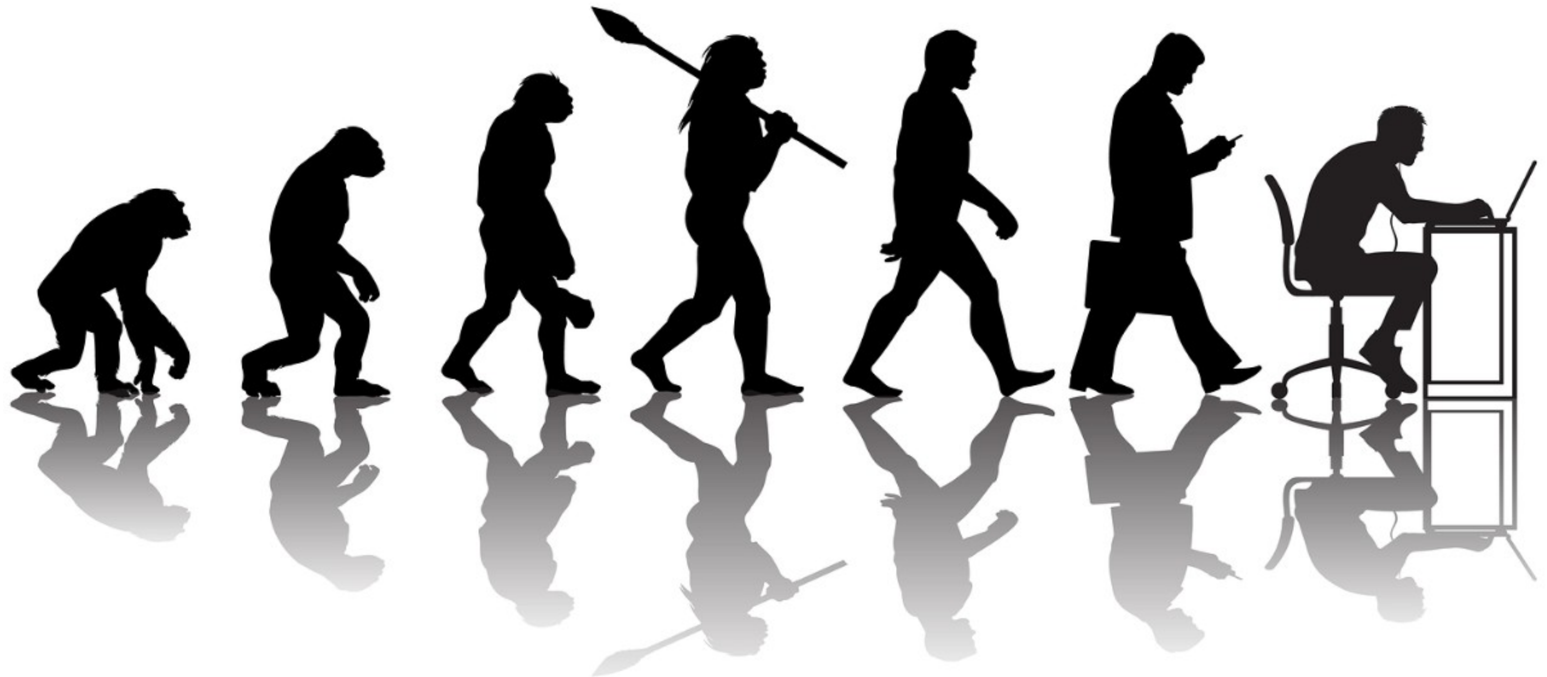


Fuzzing Overview



Coverage-guided Fuzzing

- Fuzzing as a **genetic algorithm**
  - Chromosome population: seed corpus
  - Genetic mutation: mutation
  - Fitness function: coverage
- Key idea: keep mutants that increase **code coverage** for future mutations
  - So-called **grey-box** fuzzing (반투명 마구실행)
  - E.g., AFL, LLVM's libFuzzer



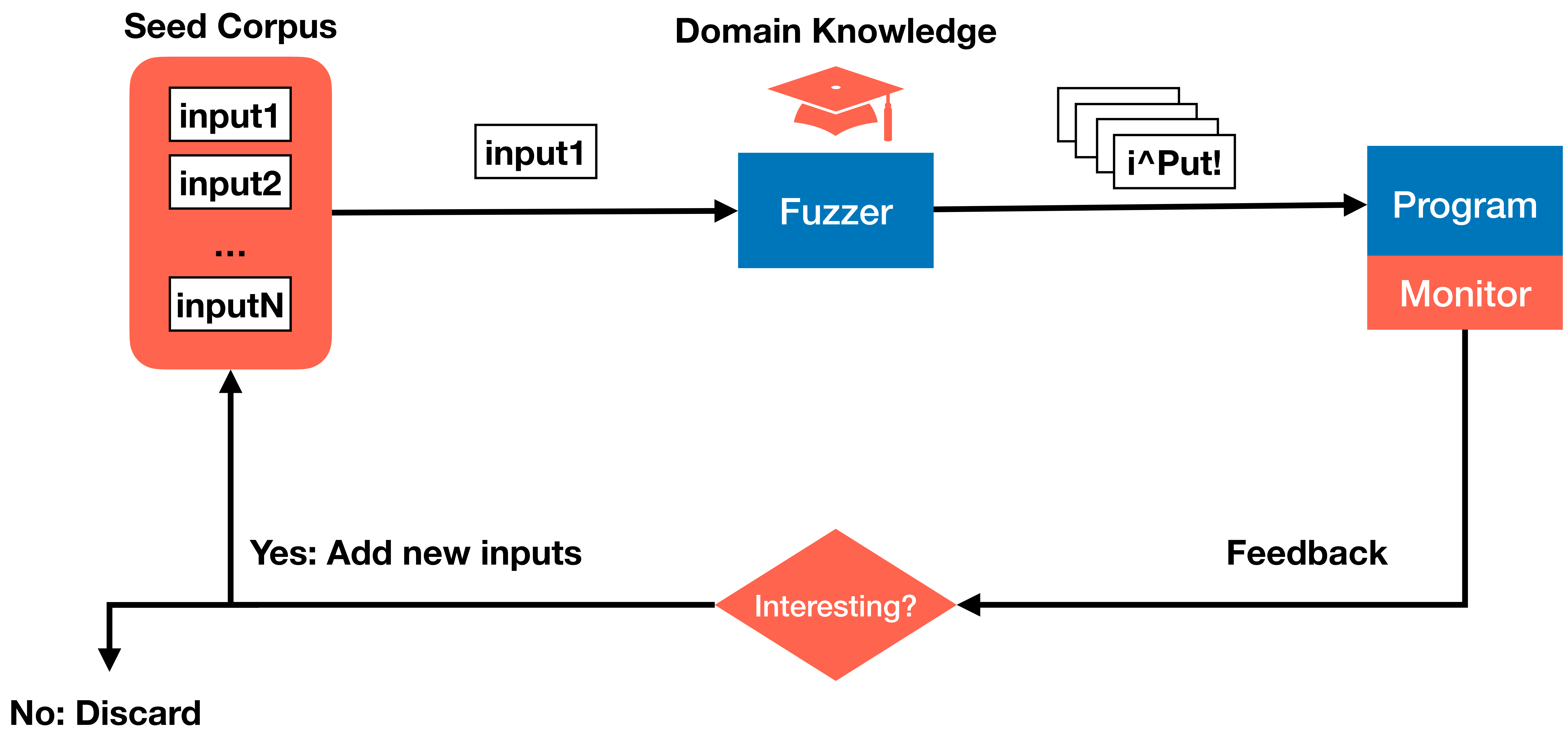
```
arr = input();
if (arr[0] > 0) {
  if (arr[1] > 10) {
    if (arr[2] > 100) {
      bug();
    }
  }
}
```

Code Coverage덮이

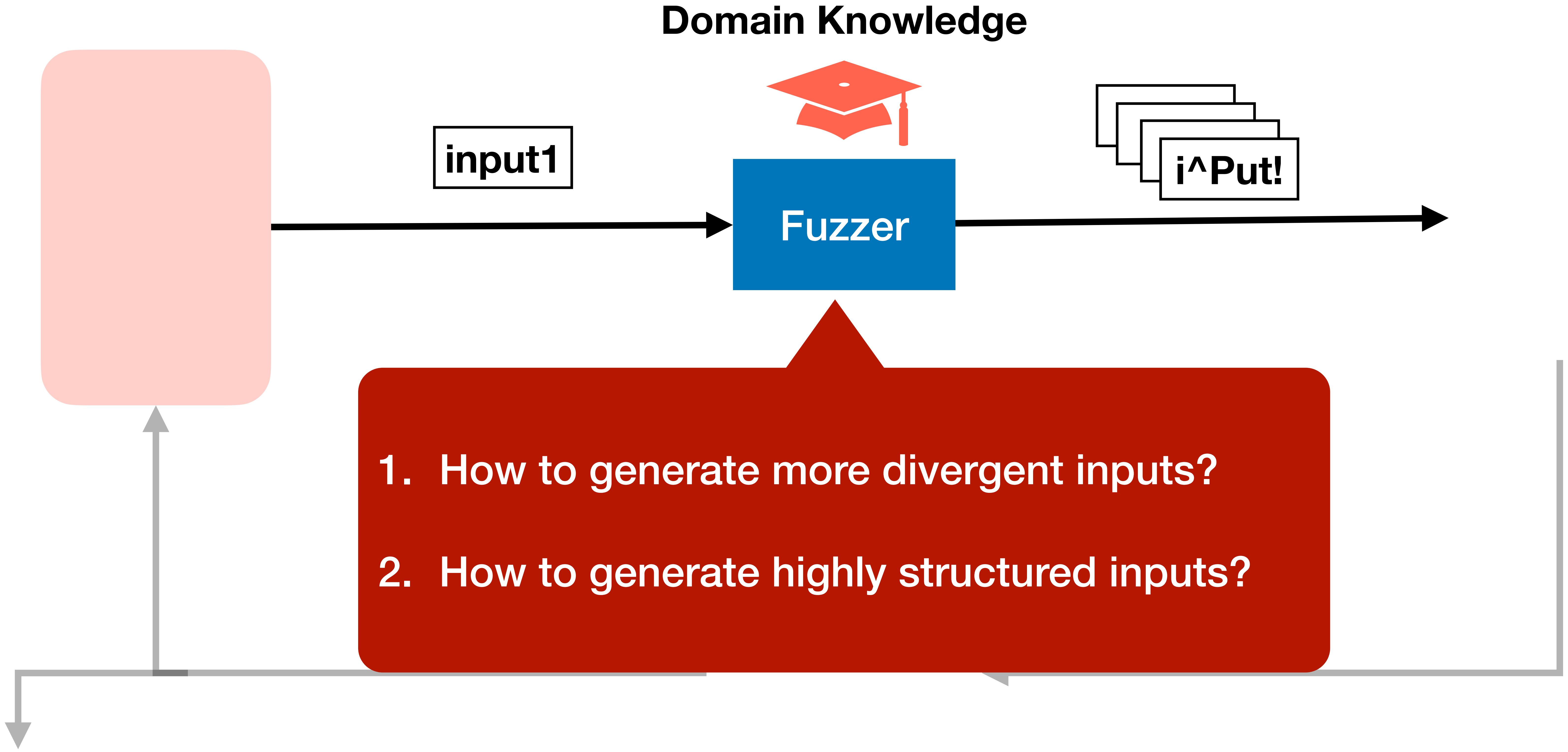
- A metric that determines how much code has been executed
- Obtained from the runtime monitor
  - E.g., LLVM’s SanitizerCoverage, gcov, etc
- Many criteria: line/stmt coverage, branch coverage, path coverage, etc
- Caveat: 100% coverage does not mean exhausted exploration

```
// x : -1, 1
x = inputs();
if (x < 0) {
  skip;
} else {
  skip;
}
assert(x != 0);
```

More Research Questions

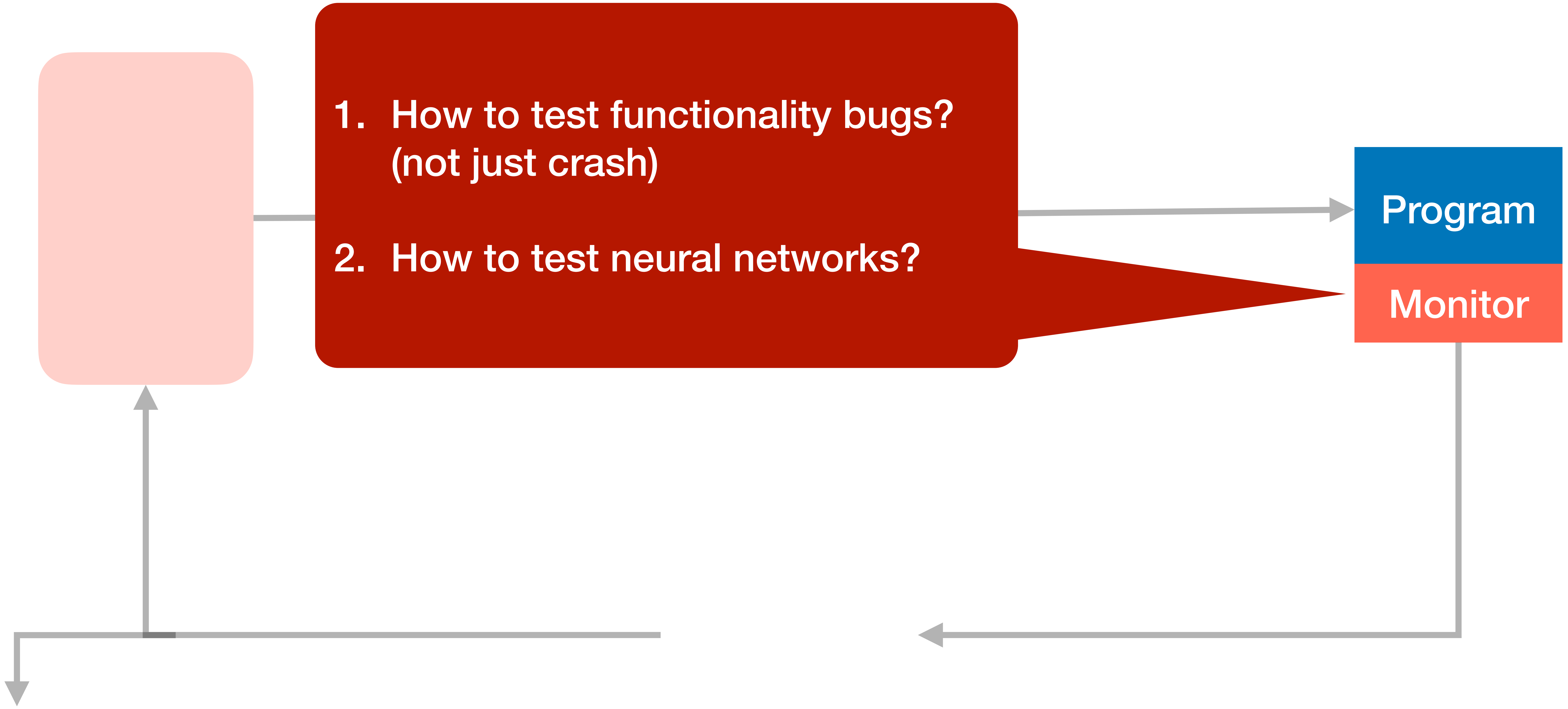


More Research Questions



1. How to generate more divergent inputs?
2. How to generate highly structured inputs?

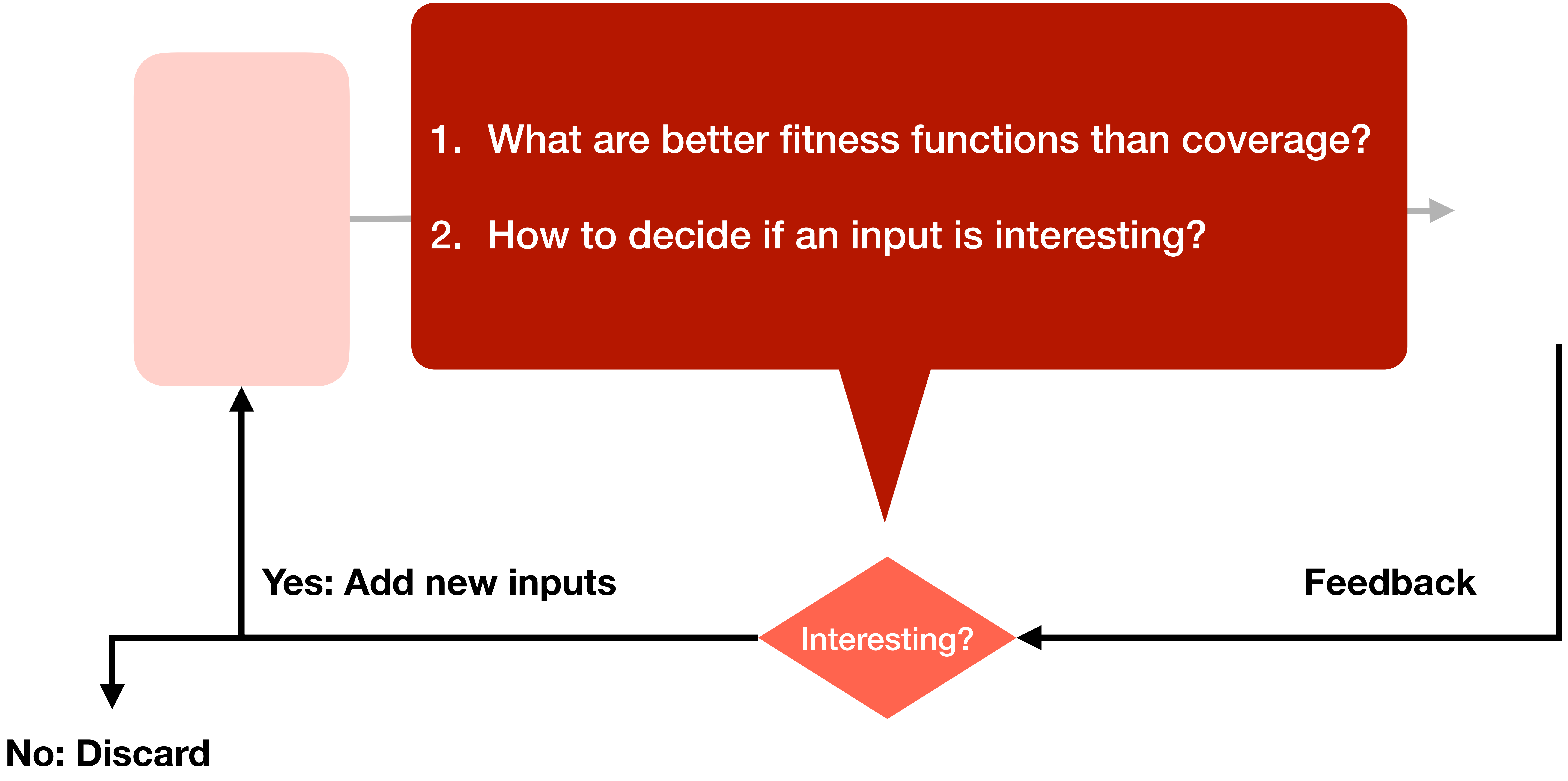
More Research Questions



1. How to test functionality bugs? (not just crash)
2. How to test neural networks?

More Research Questions

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



Summary

- Fuzzing: efficient and effective testing technique
  - Input generation: mutation-based, generation-based
  - Feedback: blackbox, greybox, whitebox
- Complete but unsound with respect to program correctness
- Challenges
  - Efficiency (e.g., higher coverage)
  - Expressiveness (e.g., functionality errors)
  - Etc