Lecture 16: Automated Test Generation

CS 5150, Spring 2026

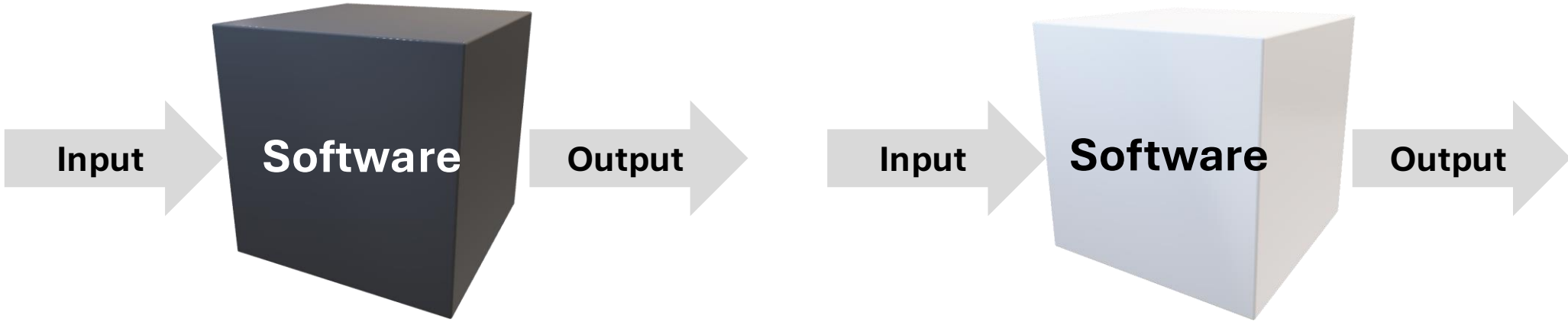
Lecture Goals

- Leverage black-box and white-box testing techniques to find bugs automatically
- Understand the pros and cons of each technique

How to automate bug detection?

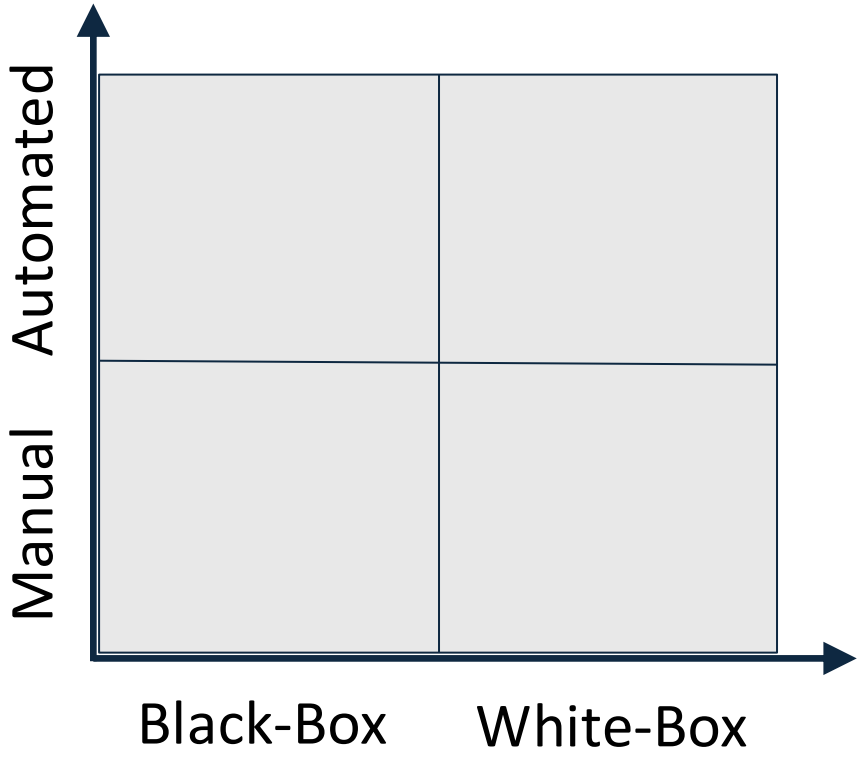
Types of test generation

- Black-box (functional) vs. white-box (structural) testing

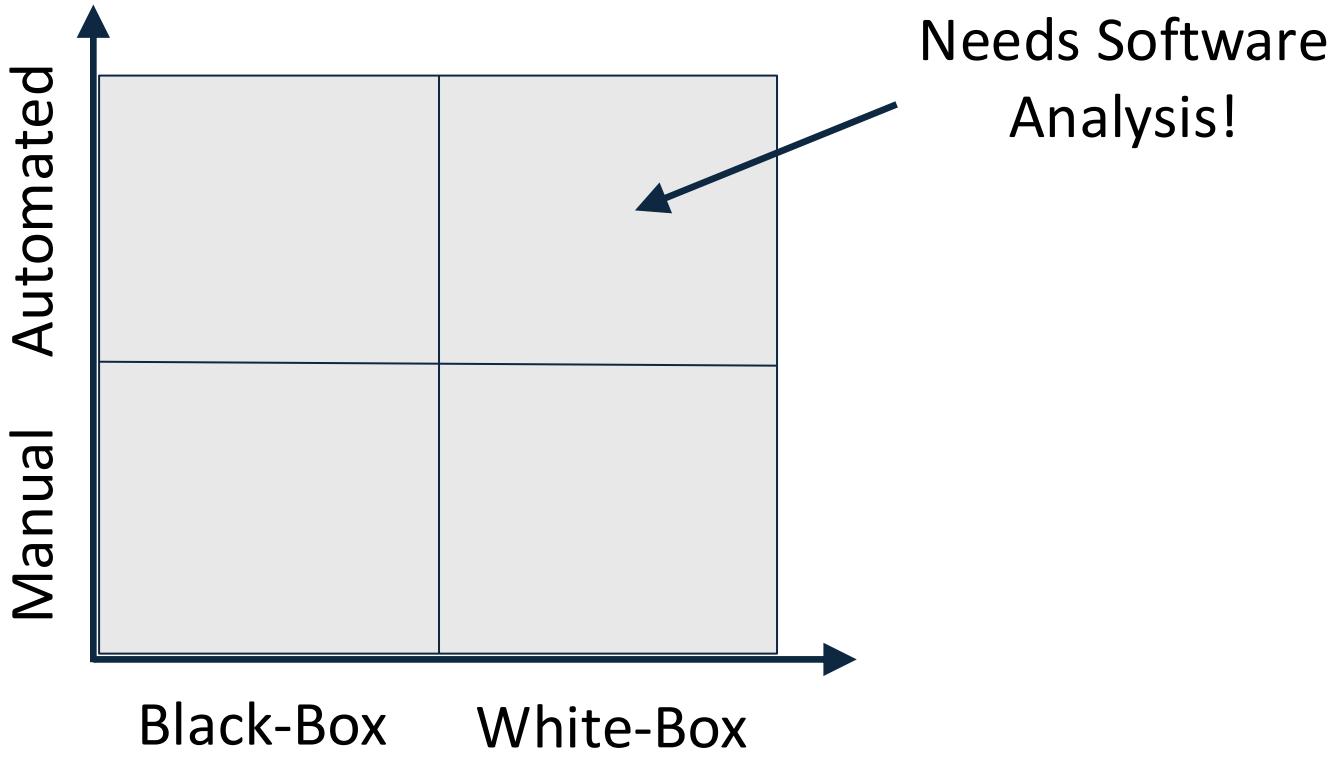


- **Black-box test generation**: generates tests based on the functionality of the program
- **White-box test generation**: generates tests based on the source-code structure of the program

Classification of Testing Approaches



Classification of Testing Approaches



Automated vs. Manual Testing

- Automated Testing:
  - Find bugs more quickly
  - No need to write tests
  - If software changes, no need to maintain tests
- Manual Testing:
  - Efficient test suite
  - Potentially better coverage

Black-Box vs. White-Box Testing

- Black-Box Testing:
  - Can work with code that cannot be modified
  - Does not need to analyze or study code
  - Code can be in any format (managed, binary, obfuscated)
- White-Box Testing:
  - Efficient test suite
  - Potentially better coverage

Today's focus

- **Unit testing**: involves testing individual units (e.g., methods or classes) of a software to ensure that each part is correct, typically
  - Unit level
  - White-box
  - Deterministic
  - …

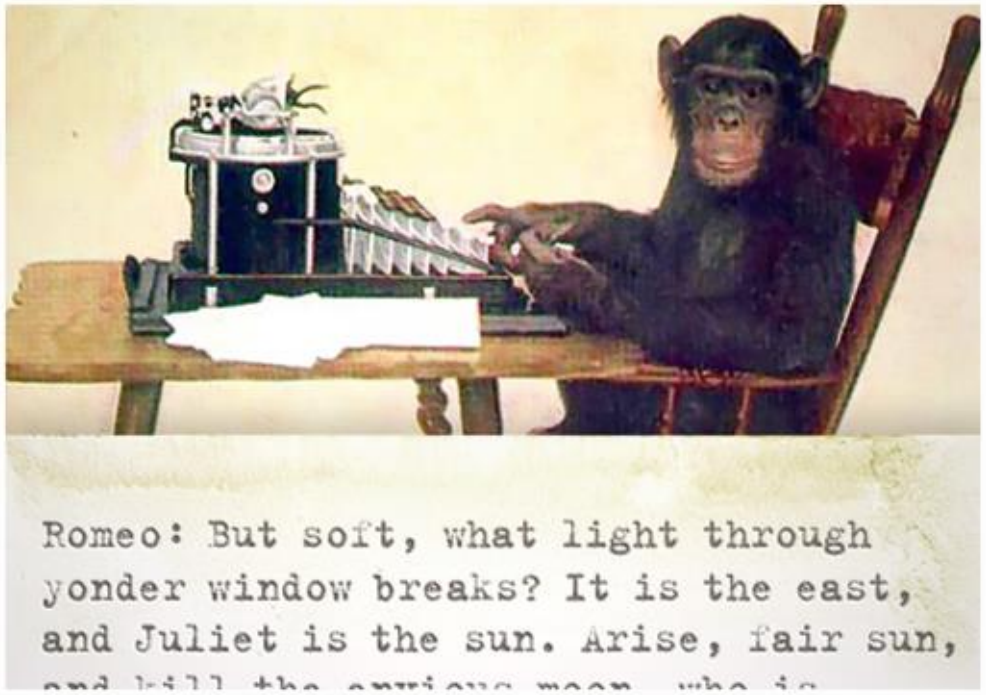
- **Fuzz testing** (fuzzing): involves providing invalid, unexpected, or random data as inputs to a software, typically
  - System level
  - Black-box
  - Non-deterministic
  - …

Random Testing (Fuzzing)

- Idea: Feed random inputs to a program
- Observe whether it behaves “correctly”
  - Execution satisfies given specification
  - Or simply doesn’t crash
    - A simple specification

The Infinite Monkey Theorem

“A monkey hitting keys at random on a typewriter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases.”



https://en.wikipedia.org/wiki/Infinite_monkey_theorem

The First Fuzzing Study

- Conducted by Barton Miller @ Univ of Wisconsin
- 1990: Command-line fuzzer, testing reliability of UNIX programs
  - Bombards utilities with random data
- 1995: Expanded to GUI-based programs (X Windows), network protocols, and system library APIs
- Later: Command-line and GUI-based Windows and OS X apps

pages.cs.wisc.edu/~bart/fuzz/fuzz.html
[The 1990 study://ftp.cs.wisc.edu/par-distr-sys/technical_papers/fuzz.pdf]
[The 1995 study ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf]

Fuzzing UNIX Utilities: Aftermath

- 1990: Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)
  - 88 tested programs across 7 versions of UNIX
- 1995: Systems got better... but not by much!

  “Even worse is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995.”

A Silver Lining: Security Bugs

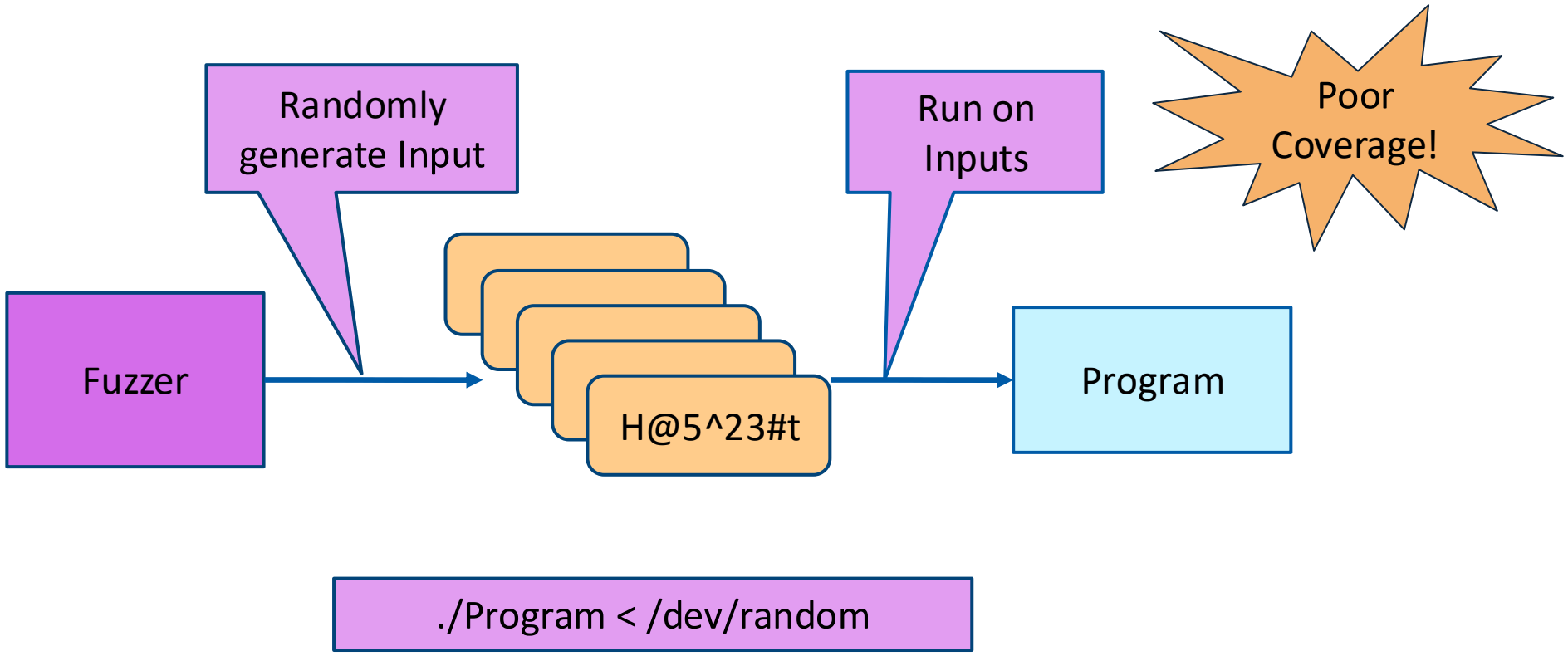
- gets() function in C has no parameter limiting input length
  ⇒ programmer must make assumptions about structure of input

- Causes reliability issues and security breaches (buffer overflow)
  - Second most common cause of errors in 1995 study

- Solution: Use fgets(), which includes an argument limiting the maximum length of input data

Three Generations of Fuzzers

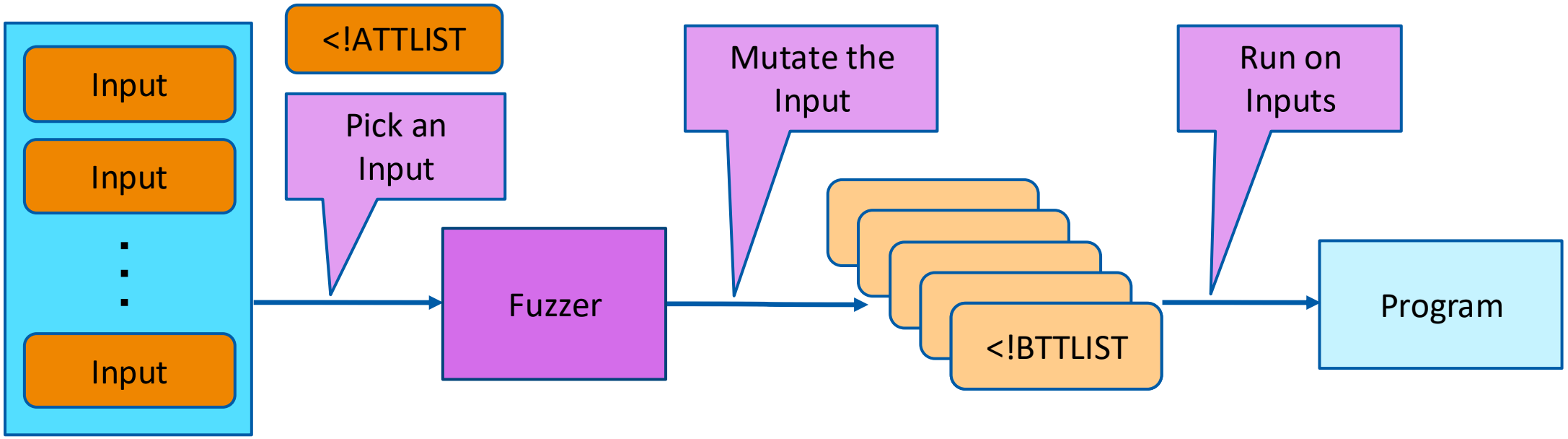
The First Generation



Many inputs easily rejected by input sanitization

Poor coverage of deeper logic in program

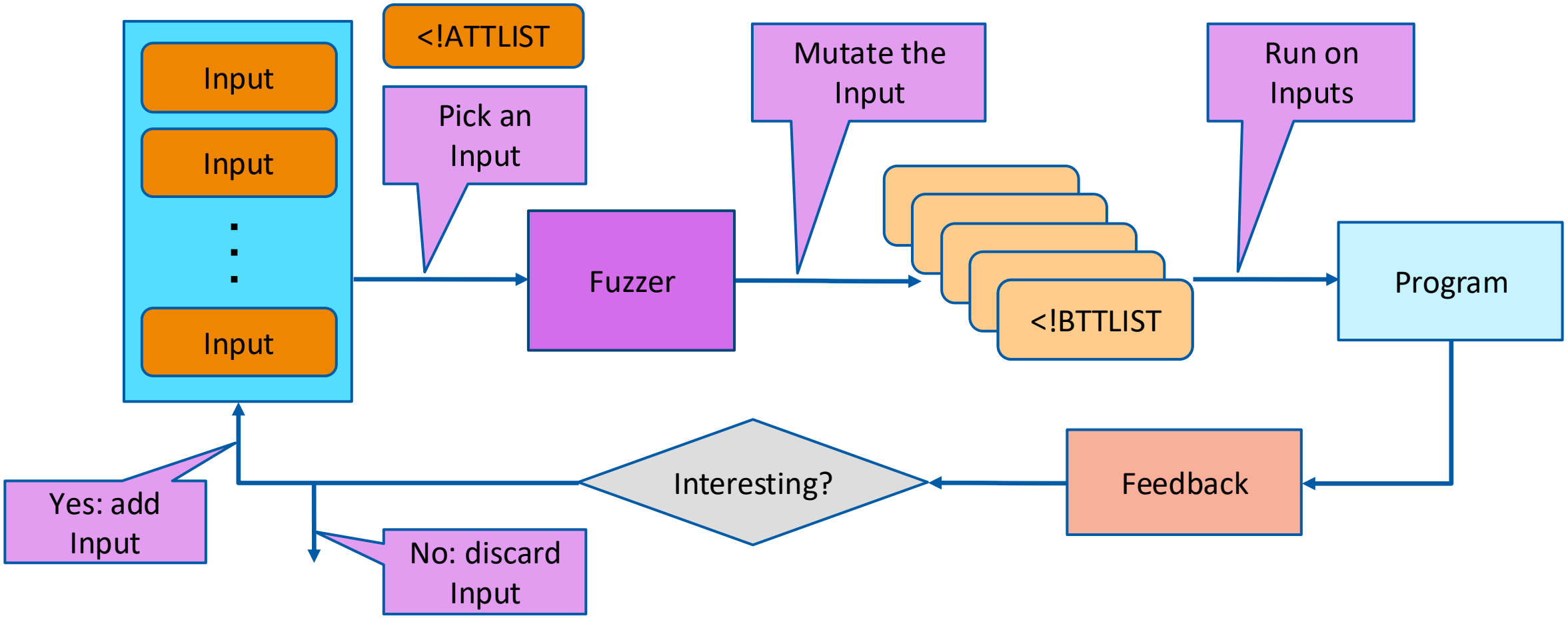
The Second Generation



Inputs not easily rejected by input sanitization logic

But coverage is still poor!

The Third Generation

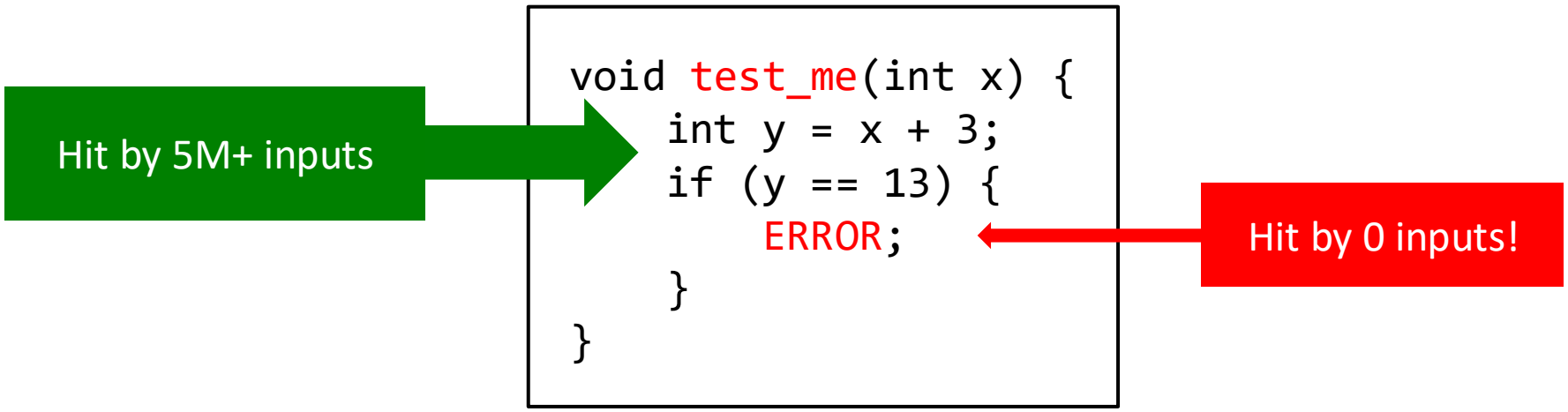


Inspiration: Evolutionary Algorithms

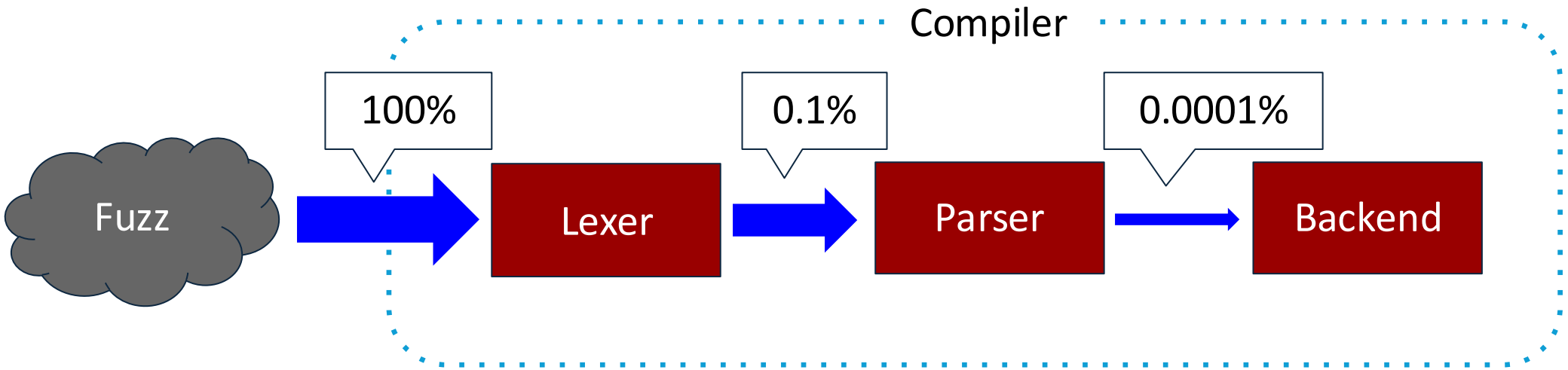
What Kinds of Bugs can Fuzzing Find?

- Memory errors
  - Spatial (e.g., out-of-bounds access) and temporal (e.g., use-after-free)
- Other undefined behaviors
  - Integer overflow, divide-by-zero, null dereference, uninitialized read, …
  - Use sanitizers
- Assertion violations
- Infinite loops (using timeout)
- Concurrency bugs
  - Data races, deadlocks, …

Uneven Code Coverage: Example 1



Uneven Code Coverage: Example 2

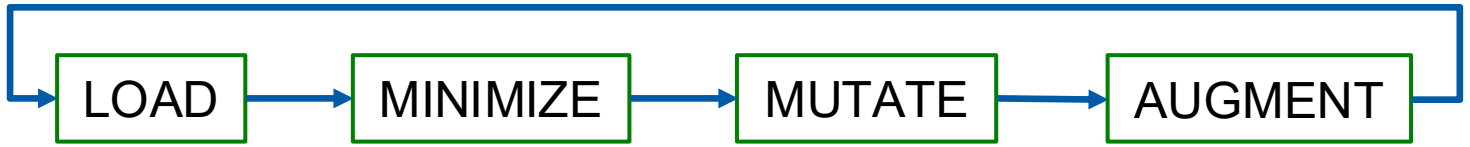


- The lexer is very heavily tested by random inputs
- But testing of later stages is much less efficient …

Fuzzers in the Wild

AFL: American Fuzzy Lop

- Arguably the best-known coverage-guided fuzzing tool
- Core ideas:
  - Genetic algorithm
  - Efficient source-code instrumentation
  - Effective heuristics for input mutation



Flipping bits and bytes

Incrementing/decrementing constants

Replacing with potentially troublesome integers

Test-case splicing

LibFuzzer

- Motivation: enable to fuzz libraries (i.e., program components) instead of whole programs
- User provides fuzzing entry points called **fuzz targets**
- Intuition: if program has X lines of code and Y fuzz targets, then fuzzer only has to cover X / Y lines of code on average per target

LibFuzzer – Fuzz Target

- Fuzz Target: a function that takes as input an array of bytes and performs something interesting with the bytes using the API under test
- Fuzz target is executed by the fuzzer multiple times with different data
- Fuzz Target Skeleton:

```
extern “C” int LLVMFuzzerTestOneInput(const unit8_t *Data, size_t Size) {
    PerformInterestingStuff(Data, Size)
    return 0;
}
```

Detecting the Heartbleed Bug in OpenSSL

- The Heartbleed bug in OpenSSL (2014) enabled eavesdropping on information protected by the SSL/TLS encryption protocol through the heartbeat protocol.
- The heartbeat protocol is used to verify that the server is alive. The server sends back an exact copy of the data sent as a positive acknowledgment.
- The Heartbleed vulnerability is triggered when the user specifies the payload size to be larger than the actual length of the message it is sending. The server then responds with the exact message copy PLUS any additional data in the buffer. In this way, users can eavesdrop on additional data.

Detecting the Heartbleed Bug in OpenSSL

```
extern “C” int LLVMFuzzerTestOneInput(const unit8_t *Data, size_t Size) {
    static SSL_CTX *sctx = Init();
    SSL *server = SSL_new(sctx);
    BIO *sinbio = BIO_new(BIO_s_mem());
    BIO *soutbio = BIO_new(BIO_s_mem());
    SSL_set_bio(server, sinbio, soutbio);
    SSL_set_accept_state(server);
    BIO_write(sinbio, Data, Size);
    SSL_do_handshake(server);
    SSL_free(server);
    return 0;
}
```

LibFuzzer finds the Heartbleed bug in < 10 seconds!

The Heartbleed bug in OpenSSL (2014) enabled eavesdropping on information protected by the SSL/TLS encryption protocol through the heartbeat protocol.

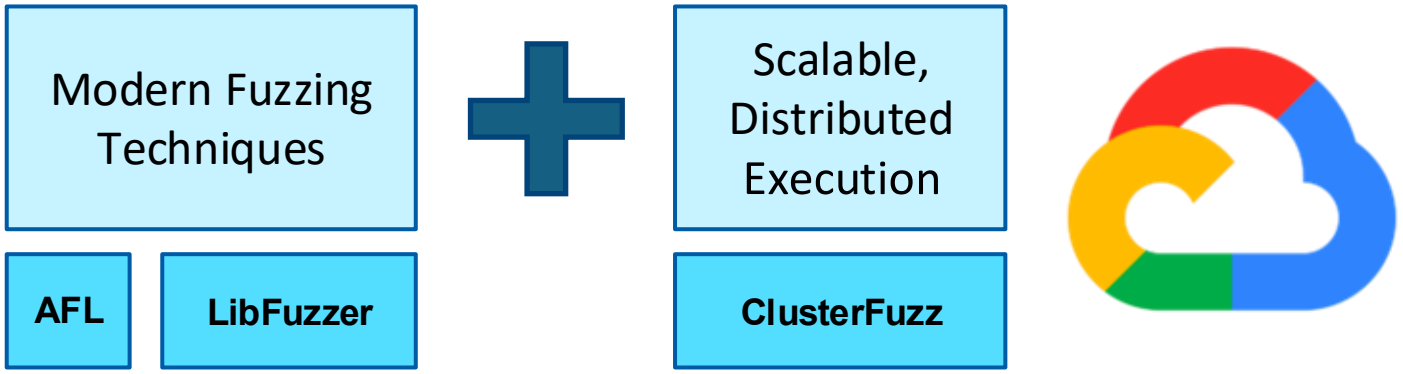
Complete Fuzz Target: https://github.com/google/fuzzer-test-suite/blob/master/openssl-1.0.1f/target.cc.

Fuzz Target Guidelines

- Must tolerate different input kinds (e.g. malformed, null, huge, etc.)
- Should be fast (e.g. avoid long-running computation, logging I/O, etc.)
- Should not consume too much memory
- Narrower the better (e.g., write different fuzz targets if library handles different data formats)

OSS-Fuzz

- Continuous fuzzing infrastructure hosted on the Google Cloud Platform



- Runs on over 25,000 machines!
- OSS-Fuzz has discovered over 17,400 bugs from 2016 to 2019 in many large projects (e.g. openssl, llvm, postgresql, git, firefox)

ClusterFuzz

- Google's Scalable fuzzing infrastructure
  - Used to fuzz Chrome browser
  - Till 2019, 16k bugs in Chrome, 11k in 160 OSS

- Features:
  - Highly scalable (runs on 1000's of machines)
  - Accurate deduplication of crashes
  - Fully automatic bug filing for issue trackers
  - Testcase minimization
  - Statistics for analyzing fuzzer performance
  - Easy-to-use web interface for report

Poll: PollEv.com/cs5150sp26

- Which tool is more appropriate for fuzzing binary targets?
  - A. AFL
  - B. LibFuzzer

Poll: PollEv.com/cs5150sp26

- Which tool puts a greater overhead on the tester?
  - A. AFL
  - B. LibFuzzer

How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

Code Coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., (provably) unreachable code
  - Often required for safety-critical applications

Types of Code Coverage

- Function coverage: which functions were called?
- Statement coverage: which statements were executed?
- Branch coverage: which branches were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, …

Mutation Analysis

- Founded on “competent programmer assumption”:

  The program is close to correct to begin with

- Key idea: Test variations (mutants) of the program
  - Replace x > 0 by x < 0
  - Replace w by w + 1, w - 1
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

A Problem

- What if a mutant is equivalent to the original?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
  - Try to prove program equivalence automatically
  - Often requires manual intervention

QUIZ: Code Coverage Metrics

Test Suite: `{ foo(1, 0) }`

Statement Coverage: ☐ %

Branch Coverage: ☐ %

Give arguments for another call to `foo(x, y)` to add to the test suite to increase both coverages to 100%.

x = ☐ y = ☐

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

QUIZ: Code Coverage Metrics

Test Suite: `{ foo(1, 0) }`

Statement Coverage: 80 %

Branch Coverage: 50 %

Give arguments for another call to `foo(x, y)` to add to the test suite to increase both coverages to 100%.

x = 1 y = 1

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☐
Mutant 2 x <= y → x != y	☐	☐

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite adequate with respect to both mutants? ○ Yes ○ No

QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite adequate with respect to both mutants?

○ Yes ☑ No

QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:

foo(☐ , ☐) == ☐

QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:

foo(1 , 0) == 0