EC4219: Software Engineering

Lecture 2 — Fuzzing

Sunbeom So
2024 Spring

Software Testing (Dynamic Analysis)

- Automatically generating test inputs to discover errors.
- Broadly classified into two groups according to how they generate tests.
  - Fuzzing: generate concrete inputs directly.
  - Symbolic execution: create inputs by generating symbolic constraints and solving them.
- This lecture aims to cover fuzzing.

Fuzzing

- Also known as fuzz testing or random testing.
- The basic idea is **surprisingly simple.**
  - Generate a set of random program inputs.
  - Observe whether the program behaves "correctly" on each such input.
- Correctness can be defined in various ways. For example,
  - The execution does not crash.
  - The execution satisfies the specification (e.g., pre- and postcondition, assertion).

Fuzzing History

- The first popular fuzzing experiment was conducted by Barton Miller at the Univ of Wisconsin.

- In 1990, developed a command-line fuzzer to test the reliability of UNIX utility programs by bombarding them with random data.
  - Covered a substantial part of those that were commonly used at the time (e.g., the mail program, screen editors, compilers, document formatting packages).
- Discovered two kinds of errors in 25–33% of the tested programs: crash and hang (loops indefinitely).
- Fuzzing techniques have been significantly improved.

Classification of Fuzzing

Depending on the amount of access to the tested program's source code:

- Blackbox: generate inputs regardless of the source code.
- Coverage-guided (greybox): generate inputs by observing some information related to the source code.

Depending on how test inputs are generated:

- Mutation-based: generate new inputs by mutating existing valid inputs (seeds).
- Generation-based: generate inputs based on a given model (can be manually defined or automatically inferred).

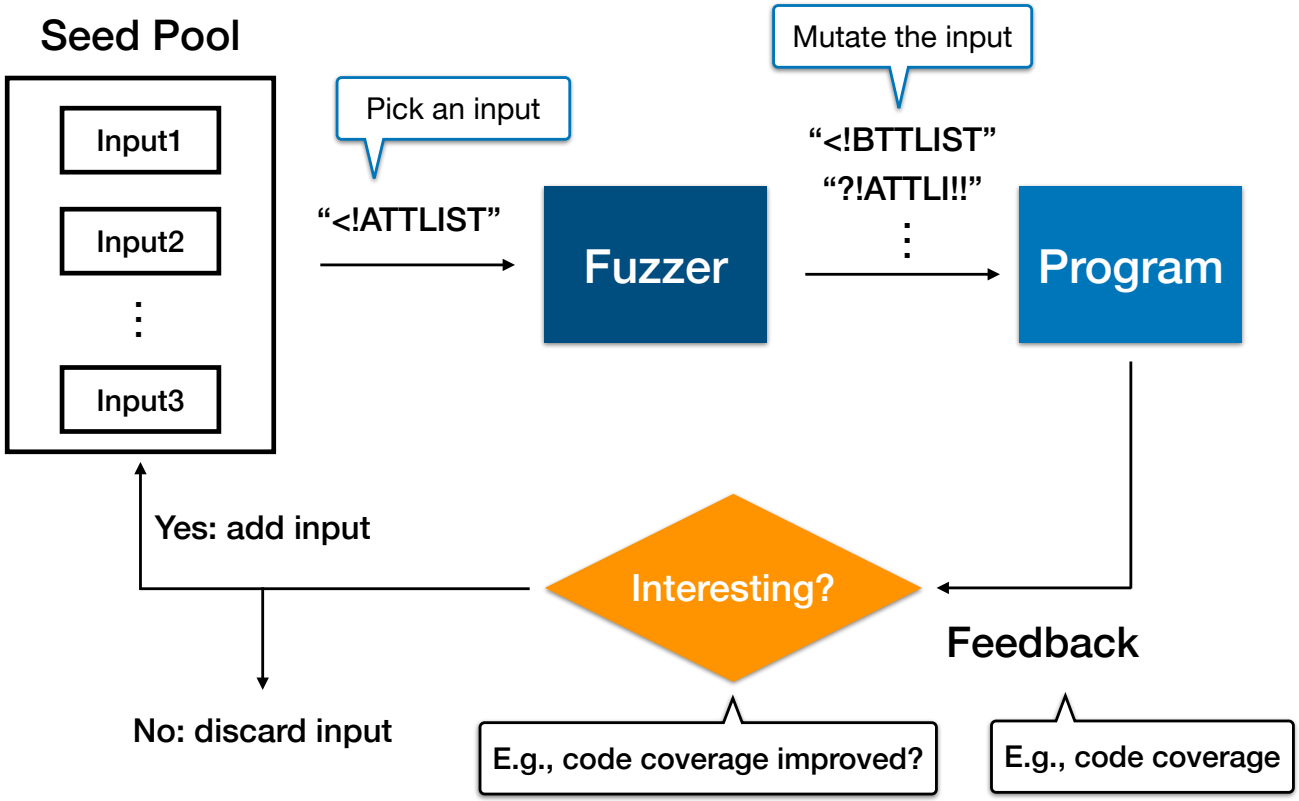
There are other lots of classifying standards for fuzzing techniques.

cf) Code Coverage

```
1  int foo (int x, int y)
2    int z = 0;
3    if (x <= y) { z = x; }
4    else { z = y; }
5    return z;
6  }
```

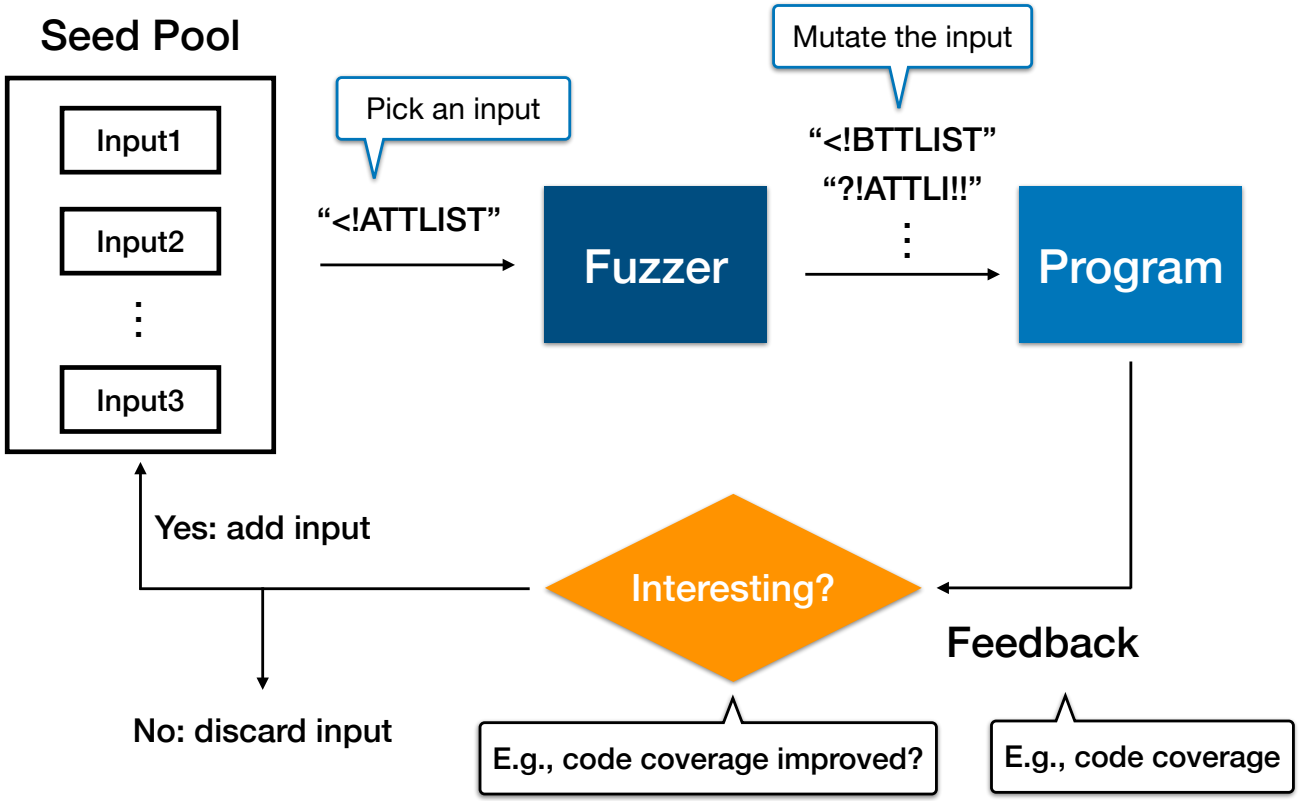
- A metric to quantify the extent to which a program's code is tested by a given test suite.
  - Function coverage: # of functions called out of the total number of functions in a program.
  - Statement coverage: # of statements that are executed out of all statements in a program.
  - Branch coverage: # of branches of each control structure taken so far. For example, given `foo(1, 0)`, the branch coverage is 50%.
  - Many others: line coverage, path coverage, etc.
- Higher code coverage is generally indicative of a better test suite.

Coverage-guided and Mutation-based Fuzzing



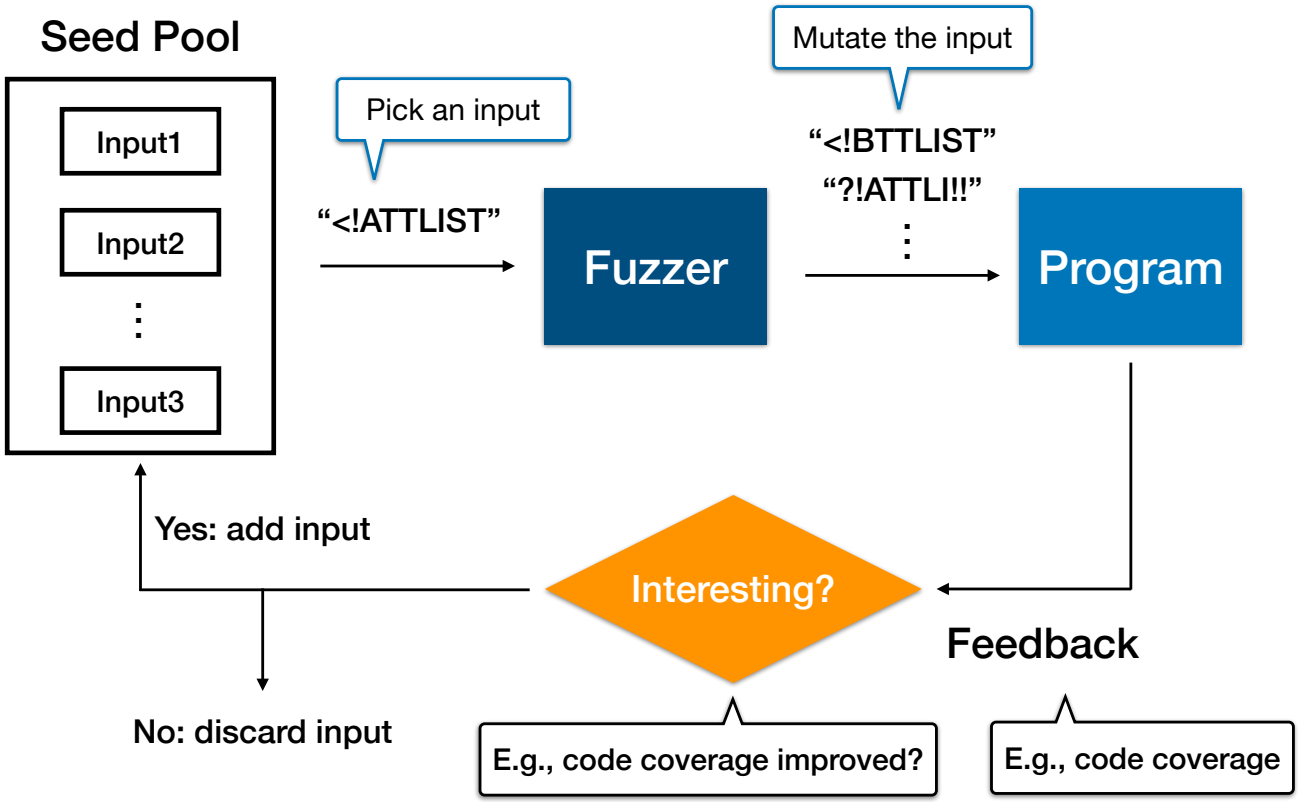
- Arguably the most popular form of fuzzing (e.g., AFL).
- A kind of genetic algorithm.
- Overall steps: (1) Selection, (2) Mutation, (3) Augmentation

Step 1. Selection



- An input will be loaded from the initial set of seeds (seed pool).
- The input corpus can be augmented with a new input during fuzzing.

Step 2. Mutation



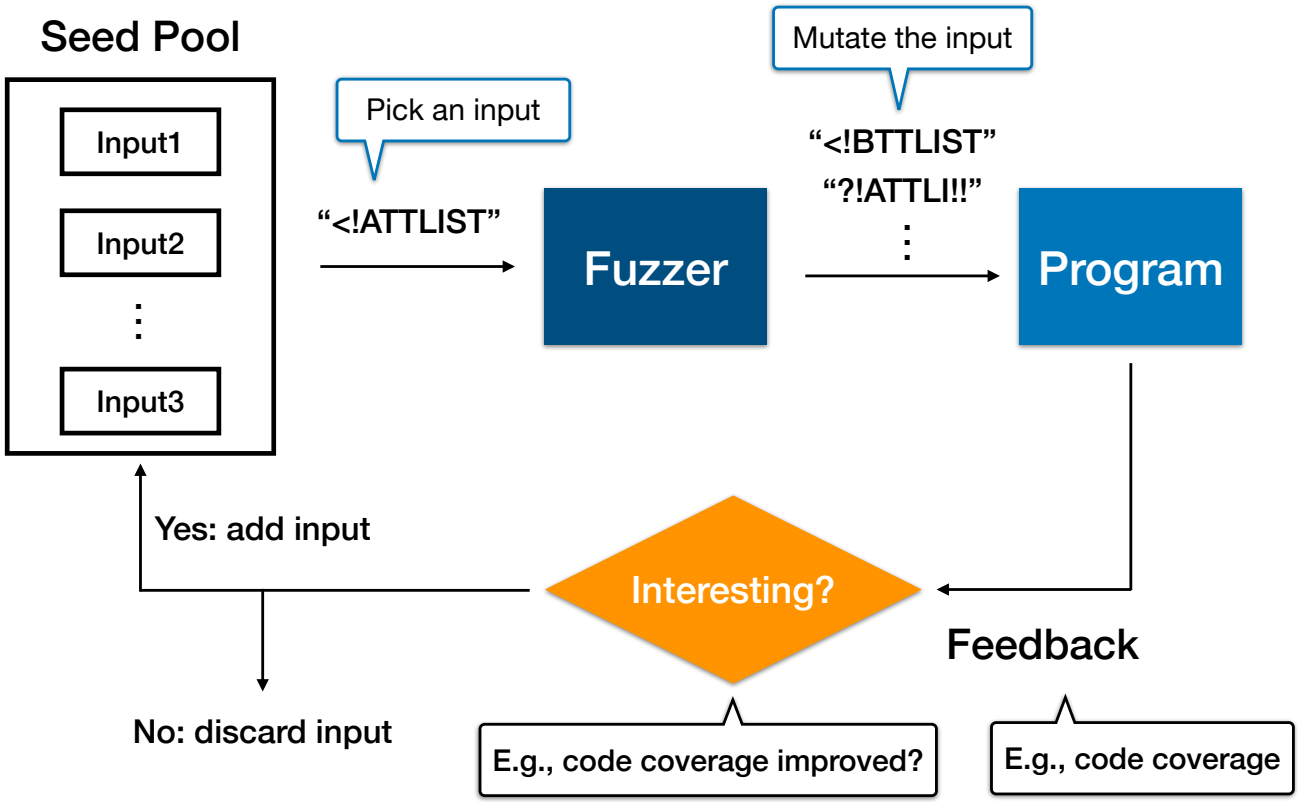
- Mutate the minimized input using a variety of mutation strategies.
- Very small conservative changes will suffer from poor code coverage.
- An aggressive mutation strategy can also produce useless mutants.
- Mutation strategies must be carefully engineered to find a sweet spot.

Example: Mutation in AFL

Mutation strategies used by AFL include the following.

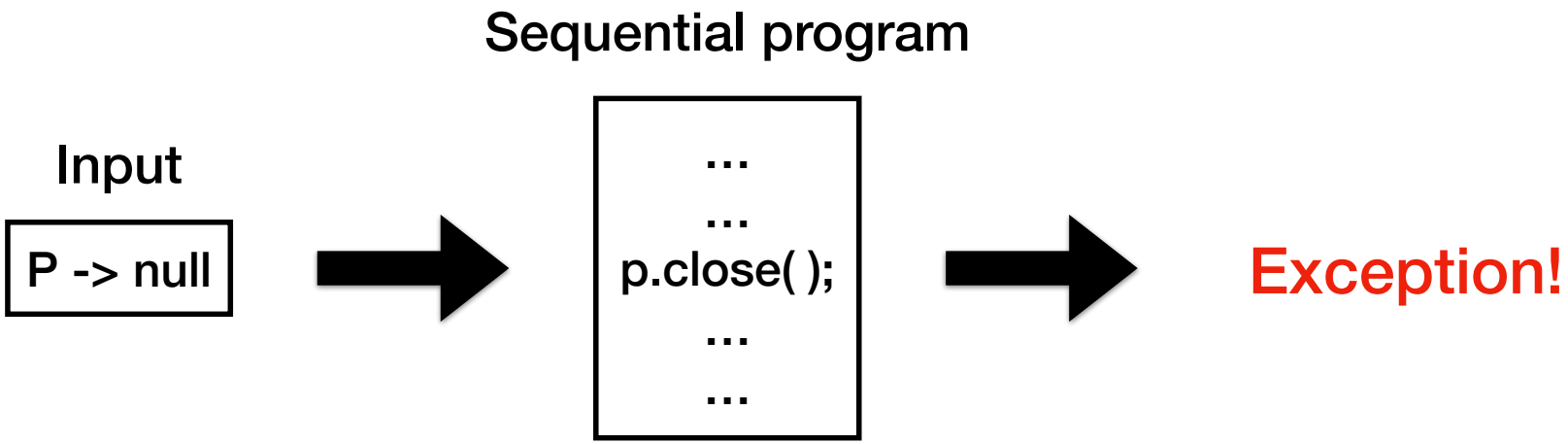
- Flipping bits and bytes
- Incrementing or decrementing constants: involves modifying integer value constants in the input.
- Replacing integers: replaces integer values from a carefully selected set of hardcoded values (e.g., -1, 256, 1024).

Step 3. Augmentation



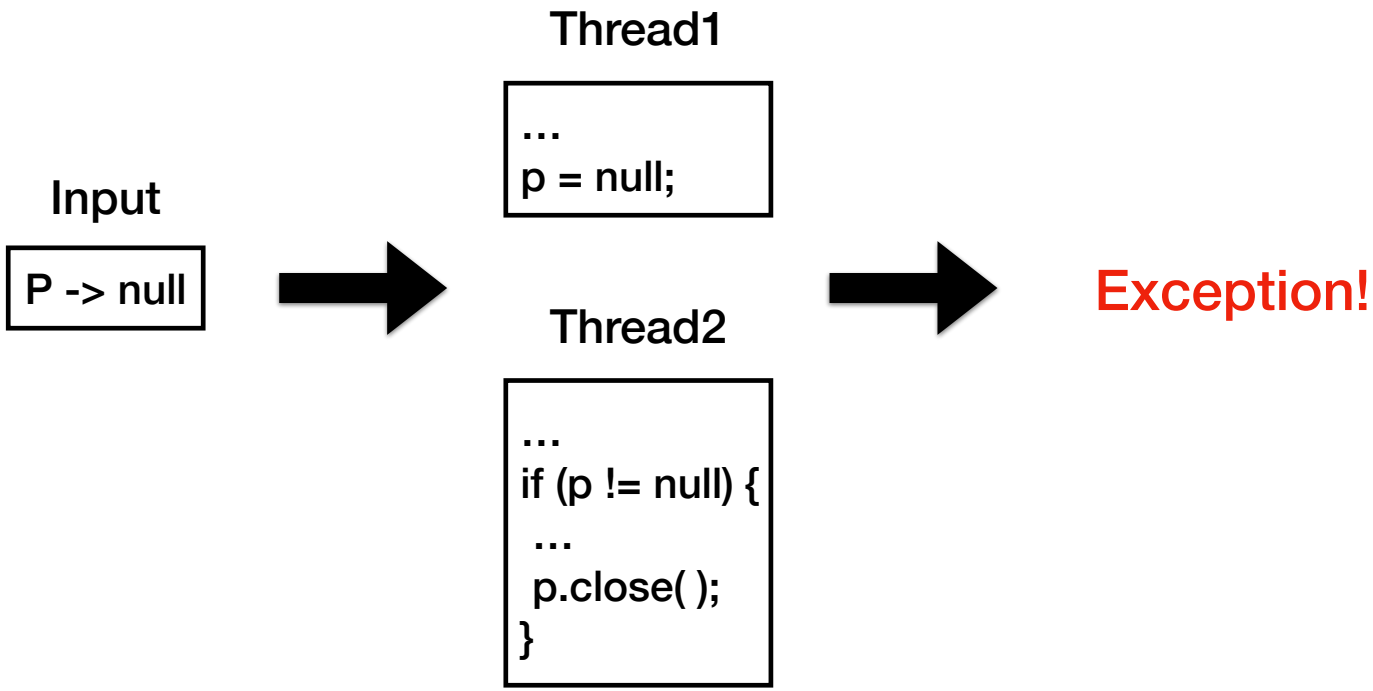
- Employ feedback to augment the seed pool if any of the generated mutants show some "interesting" behaviors.
- For instance, "does the input achieve new code coverage compared to all previously attempted inputs?"

Fuzzing Concurrent Programs



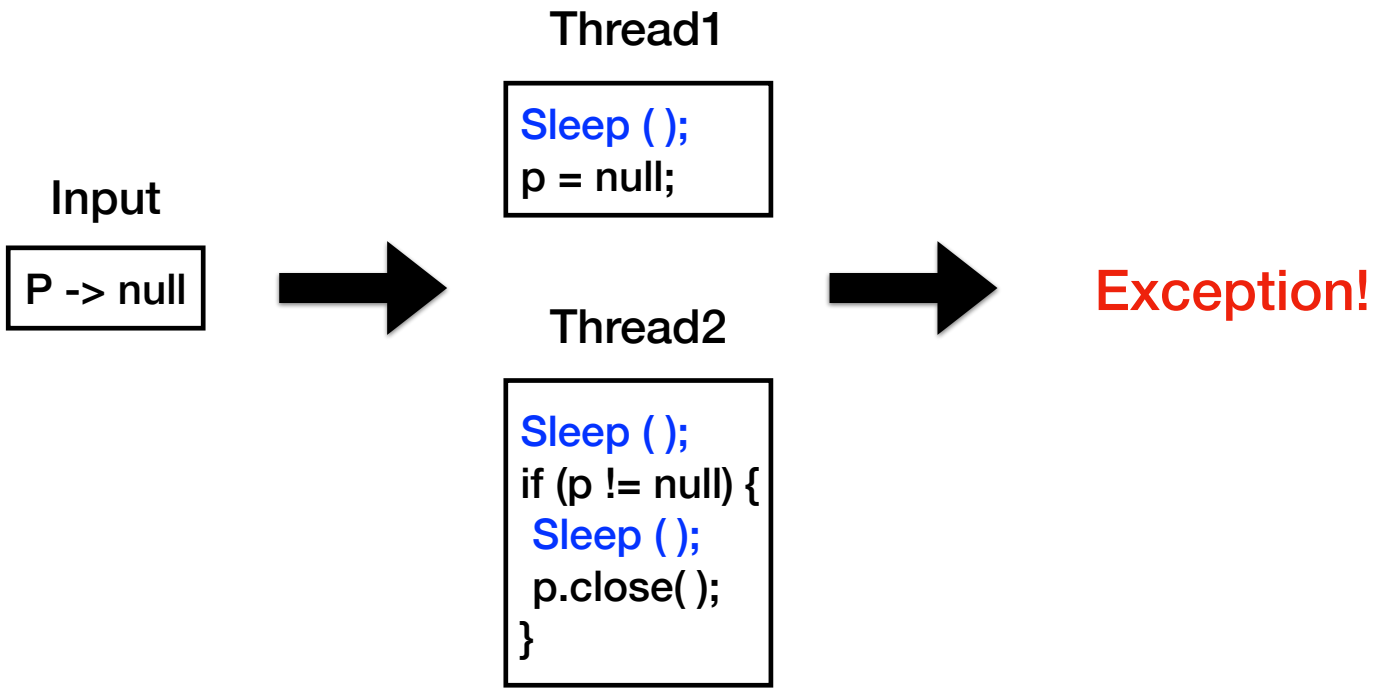
- An important domain in which fuzzing is exceedingly useful is the testing of concurrent programs.
- In a sequential program, a bug is triggered under a specific program input, and testing sequential programs is primarily concerned with techniques to discover such an input.
- This Java program would crash due to the null File handle.

Fuzzing Concurrent Programs



- In a **concurrent program**, a bug is triggered not only under a specific **program input**, but also under a specific **thread schedule**.
- If the non-null check in Thread 2 is executed first, followed by the assignment of null to `p` in Thread 1, followed by the `p.close()` statement in Thread 2, then the program will throw a null pointer exception at this statement.

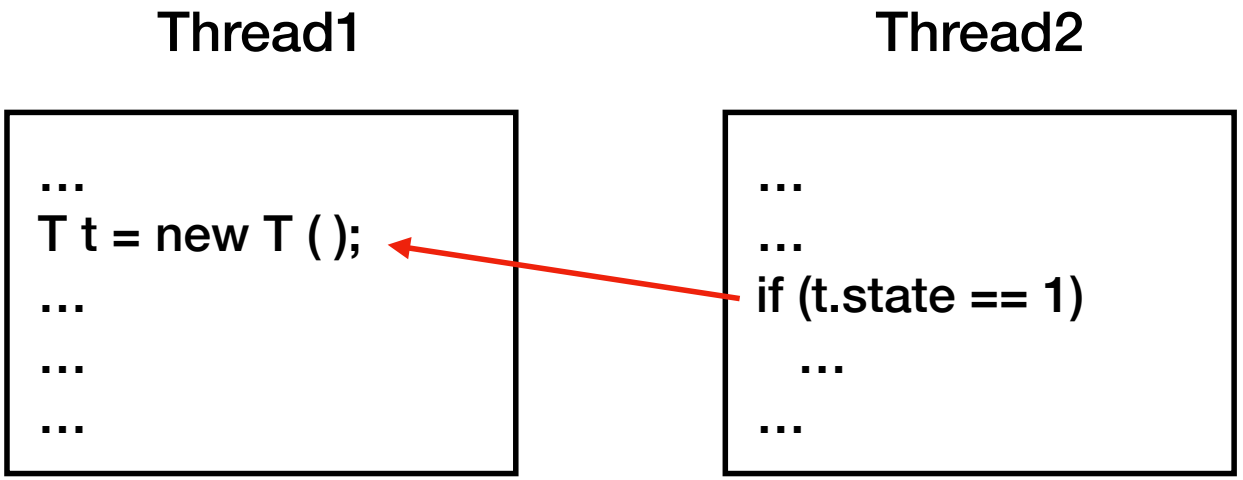
Fuzzing Concurrent Programs



- The predominant approach to introduce random delay, indicated by the calls to a system function `Sleep( )` that has the the effect of lowering the priority of the current thread.
- Making these random delays has the effect of attempting different thread schedules in the hope of finding one that triggers any lurking concurrency bug.

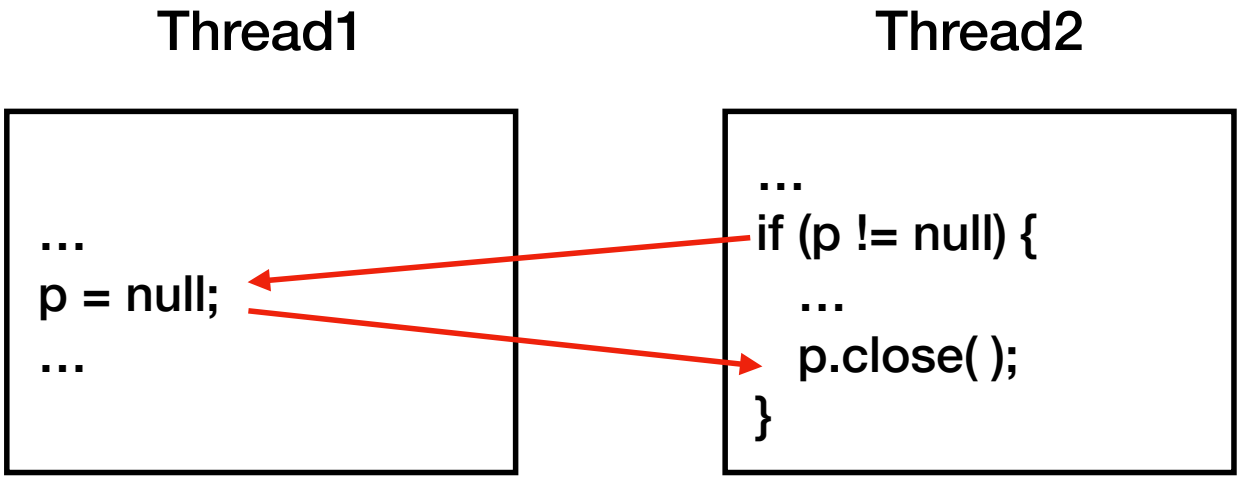
Key Obersvation in Fuzzing Thread Schedules

- Concurrency bugs typically have small *depths* (usually <3).
- Depth: # of ordering constraints a schedule has to satisfy to find the bug.
- The depth of this concurrency bug is 1.



Key Obersvation in Fuzzing Thread Schedules

- Concurrency bugs typically have small *depths* (usually <3).
- Depth: # of ordering constraints a schedule has to satisfy to find the bug.
- The depth of this concurrency bug is 2.



Quiz: Concurrency Bug Depth

- Q. Specify the depth of the concurrency bug and the reason.

Thread1

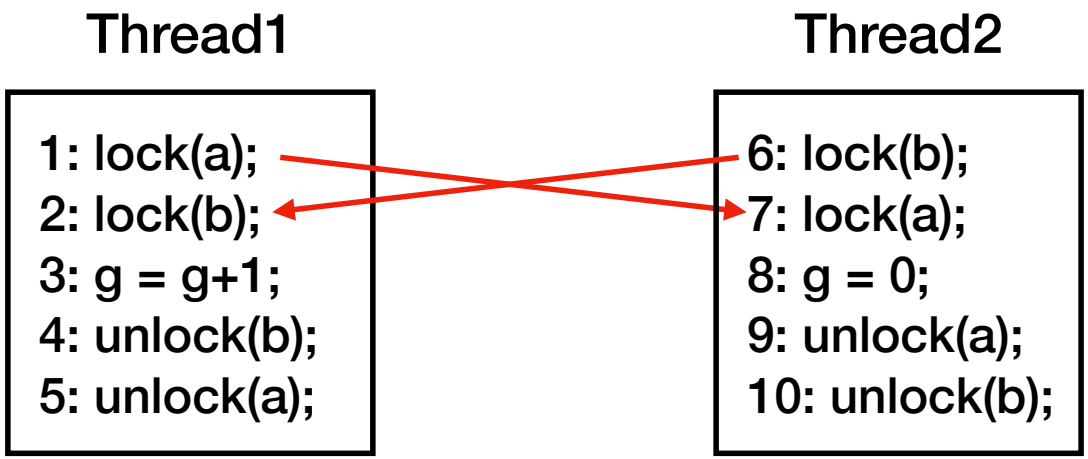
```
1: lock(a);
2: lock(b);
3: g = g+1;
4: unlock(b);
5: unlock(a);
```

Thread2

```
6: lock(b);
7: lock(a);
8: g = 0;
9: unlock(a);
10: unlock(b);
```

Quiz: Concurrency Bug Depth

- Q. Specify the depth of the concurrency bug and the reason.



- The concurrency bug in this program is a deadlock of depth two.
  - Line 1 in Thread 1 is executed before Line 7 in Thread 2, and
  - Line 6 in Thread 2 is executed before Line 2 in Thread 1.

Applications of Fuzzing

Fuzzing is arguably the most popular SW testing technique due to its simplicity (easy to implement and use).

- Violations of safety properties:
  - Assertion violation, integer overflow, division-by-zero, null dereference, uninitialized read, etc.
- Violations of liveness properties: infinite loops (using timeout).
- Concurrency bugs: data races, deadlocks.

Summary

- Fuzzing is arguably the most popular SW testing technique.
  - Depending on the access to the source: blackbox, coverage-guided
  - Depending on how test inputs are generated: mutation-based, generation-based
- Can be used to test not only sequential programs but also concurrent programs.
  - Key observation: Concurrency bugs typically have small depths.