Fuzzing

Arnav Gupta

April 5, 2024

Contents

1 Fuzzing

Set of automated testing techniques that identifies abnormal program behaviours by evaluating how the tested program responds to various random inputs.

Basic fuzzer is purely random:

- easy to implement and fast, but relies on luck and has shallow program exploration.

Fuzzer categories:

- **generation-based**: inputs generated from scratch
- **mutation-based**: inputs from modifying existing inputs

- **dumb** or **smart** depending on awareness of input structure
- **white/grey/black box** depending on awareness of program structure

1.1 Goal

1. Find real bugs
2. Reduce the number of false positives
   (a) generate reasonable input

1.2 Mutation-Based Fuzzing

1. Take existing input
2. Randomly modify it
3. Pass it to the program

1.2.1 Grammar-Based Fuzzing

Grammars can easily formally specify input languages.

A simple grammar fuzzer:

1. starts with the start symbol then keeps expanding it
2. to avoid expansion to infinite inputs, place a limit on the # of nonterminals
3. to avoid being stuck in a situation where we cannot reduce the number of symbols further, limit the total # of expansion steps

Grammar generated inputs can be seeds in mutation-based fuzzing.

1.2.2 Guiding by Coverage

Retrieve coverage of a test run and evolve inputs that are successful (where a new path was found during test execution).

1.3 Gerybox Fuzzing

Power schedule: distributes fuzzing time among the seeds in the population

Objective: maximize the time spend fuzzing the seeds that lead to higher coverage increase in shorter time.

Seed Energy: likelihood with which a seed is chosen from a population

1.3.1 Directed Gerybox Fuzzing

Implement a power schedule that assigns more energy to seeds with a low distance to the target function:

- build a call graph among function in a program
- for each function executed in the test, calculate its shortest path to the target function, then do an average distance, using the distance to calculate the power schedule
- can do more complex calculation by considering finer grained info

1.4 Search-Based Fuzzing

For deriving specific test inputs that achieve some objective. Uses domain knowledge of which inputs is closest to what one is looking for.

1.4.1 Hillclimbing Algorithm

1. Take a random starting point
2. Determine fitness value of all neighbours
3. Move to the neighbour with the best fitness value
4. If solution not found, back to step 2.

1.4.2 Genetic Algorithm

Based on the idea that solutions can be genetically encoded, based on natural selection. A fitness function takes the info contained in the description and evaluates properties.

Emulates natural evolution with the following process:

- create an initial population of random chromosomes
- select fit individuals for reproduction
- generate new population through reproduction of selected individuals (selecting parents and creating mutations)

- continue until an optimal solution or limit reached

1. Challenge 1: Feeding Inputs The fuzzing engine executes the fuzz target many times, so the target must:
    - tolerate any kind of input
    - not exit on any input
    - join all threads at the end
    - be fast and as deterministic as possible
    - not modify any global state
    - be as narrow as possible
2. Challenge 2: Detecting Abnormal Behaviour Refers to crashes, triggering user-provided assertion failure, hanging, or allocating too much memory.

    For early crash detection use sanitizers for address, thread, memory, undefined behaviour, or leaks.
3. Challenge 3: Ensuring Progress Use coverage to evolve test cases that find new paths through program execution, with Gerybox or search-based approaches.

    Coverage with American Fuzzy Lop captures branch coverage by instrumenting compiled programs which is fast.
4. Challenge 4: Coming up with Interesting Inputs Must understand input type and structure, using model, grammar, or protocol based fuzz.
5. Challenge 5: Speed To help with speed:
    - initialize once and fork for other inputs
    - replace costly resources with cheaper ones
    - run many inputs on a single process
    - minimize the # of test cases, discarding redundant ones
    - run in parallel, distributed

1.5 Problems with Fuzzing

Many false positive that are expensive.

Focus on code coverage, which is less important than reasonable inputs.

Cleaning to make random input more reasonable:

- **minimization**: eliminate redundant test failures through diffing
- **triage**: finding similar outputs/stackdumps and grouping them in bug report