Software Security 06

Sven Schäge

Fuzzing

Sven Schäge

mostly based on

The Art, Science, and Engineering of Fuzzing: A Survey by Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo (IEEE Transactions on Software Engineering'21)

Fuzzing and Dynamic Application Security Testing (DAST)

- + Automatic
  - Scales well
- + Cannot raise false alarms
- - Computationally more expensive
  - May scan year-round
- - Needs the software to be run (dynamic analysis)
- - Not guaranteed to find all bugs
- + Can find security vulnerabilities in deployed environments
- + Does not require the code
- - Testing may launch an attack!
  - Testing should be performed in production-like but non production environment.
- - Cannot cover all of the source code and thus application
- + Can find errors in runtime environment
- + Can be used to verify results of Static Code Analysis
- - After finding a vulnerability, we have to find the code position that is responsible for it

Fuzzing

- At a high level refers to a process of repeatedly running a program with generated inputs that may be syntactically or semantically malformed
- Intuitively, fuzzing tries to decrease the entire search space of all inputs to ‘interesting’ candidates
- To decide which inputs are ‘interesting’ the tester/attacker might have **access to different information**
  - Depending on that we may distinguish black-box, grey box, white-box fuzzing
- Generation of inputs can be adaptive, i.e. the results achieved from previous inputs influences the choice of current inputs

Some Heuristics Underlying Fuzzing

- Ideally testing must be smarter than random testing/brute force
- Speed of test case evaluation is very important metric
- For optimization: avoid generation of hopeless test case candidates whenever possible
- For optimization: generate and use valuable feedback from tested program to design test cases
- Fuzzing tries to achieve high coverage of the tested program
- Fuzzing aims at generating most important test cases, avoid redundant test
- Conceptually valuable tests often…
  - trigger a violation of security policy in few components of the PUT
  - are not too far away from well-formed inputs
- Heuristic 1: Test configurations that have previously lead to found bugs are favorable
- Heuristic 2: Test configurations that test new execution paths (increase coverage) are favorable

Interlude: Code Coverage

- Metric to quantify extent to which a program’s code is tested
- Given as percentage of some aspect of the program
- 100% coverage rare in practice: e.g., inaccessible 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, …

Code Coverage Metrics

- Assume that the grayed lines have been executed in an analysis because of the function call smaller(1,0).
- What is the Statement and Branch Coverage?
- Statement Coverage ?
- Branch Coverage ?
- Give arguments for another call to smaller(x,y) to increase both coverages to 100%.
  - x=?, y=?

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

Code Coverage Metrics

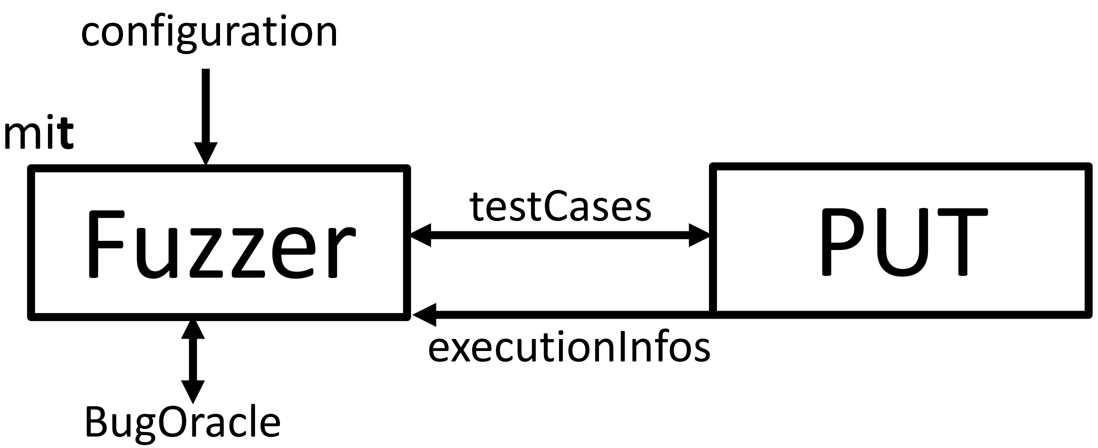
- Assume that the grayed lines have been executed in an analysis because of the function call smaller(1,0).
- What is the Statement and Branch Coverage?
- Statement Coverage 80%
- Branch Coverage 50%
- Give arguments for another call to smaller(x,y) to increase both coverages to 100%.
  - x=1, y=1

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

Some Terminology

- Program Under Test: PUT
- To fuzz, intutitively: feed in random data for the program to consume
- Definition 1 (Fuzzing). Fuzzing is the execution of the PUT using input(s) sampled from an input space (the "fuzz input space") that protrudes the expected input space of the PUT.
- Definition 2 (Fuzz Testing). Fuzz testing is the use of fuzzing to test if a PUT violates a security policy.
- Definition 3 (Fuzzer). A fuzzer is a program that performs fuzz testing on a PUT.
- Definition 4 (Fuzz Campaign). A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific security policy.
- Definition 5 (Bug Oracle). A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific security policy, e.g.: ''program should not crash on inputs.''
- Definition 6 (Fuzz Configuration). A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.
    - e.g. {(PUT, seed1, mutation ratio1), (PUT, seed2, mutation ratio2), …) or {(PUT)}
- Seed: A seed is a (commonly well-structured) input to the PUT, used to generate test cases by modifying it.
- Seed Pool: Fuzzers typically maintain a collection of seeds, and some fuzzers evolve the collection as the fuzz campaign progresses. This collection is called a seed pool.

Algorithm 1: Fuzz Testing



Overview **InputEval**

Input: C, timeLimit //set of test configurations and time limit

Output: B //finite set of bugs

B← Ø

C← **PreProcess**(C)

WHILE (timeElapsed < timeLimit AND Continue(C)) DO

  configuration ← **Schedule**(C, timeElapsed, timeLimit)

  testCases ← **InputGen**(configuration)

  \\BugOracle is inherent in a fuzzer, e.g. a signal that the program has crushed

  (B’, executionInfos) ← **InputEval**(configuration, testCases, BugOracle)

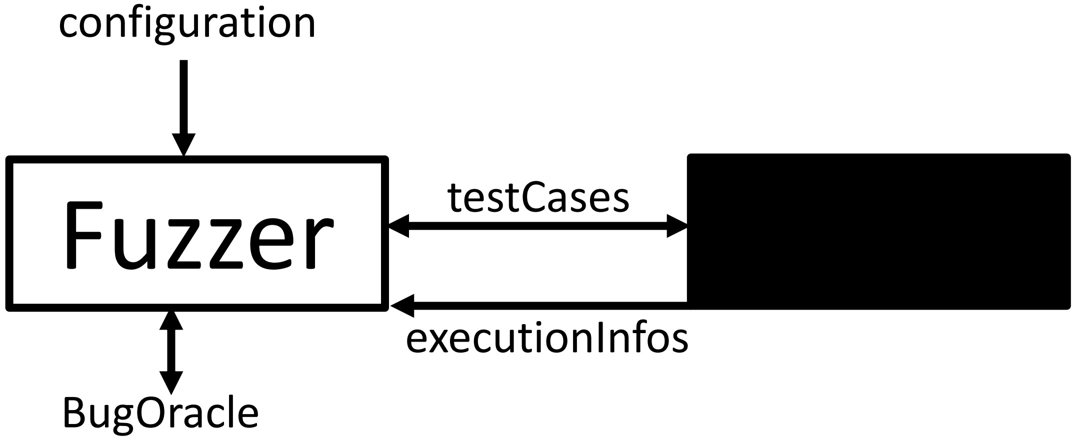
  C ← **ConfUpdate**

  B ← B U B’

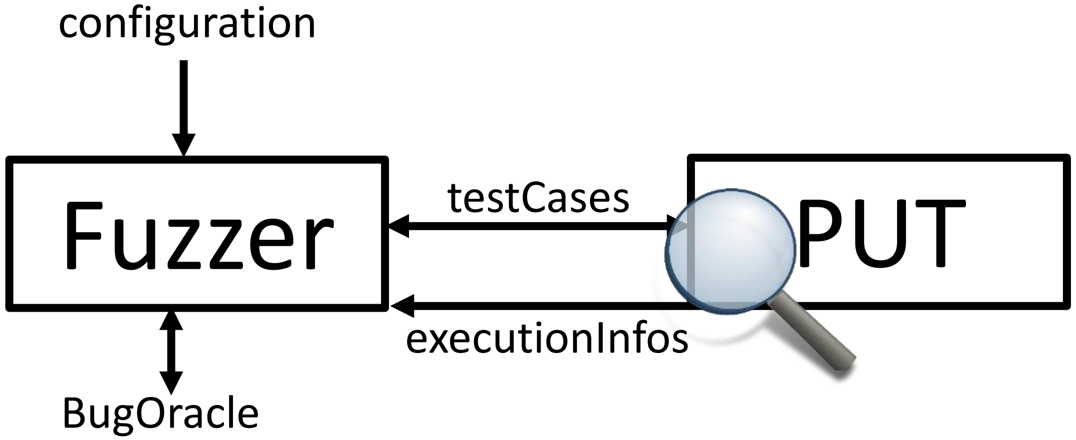
RETURN B

Fuzzer

- Black-box Fuzzer

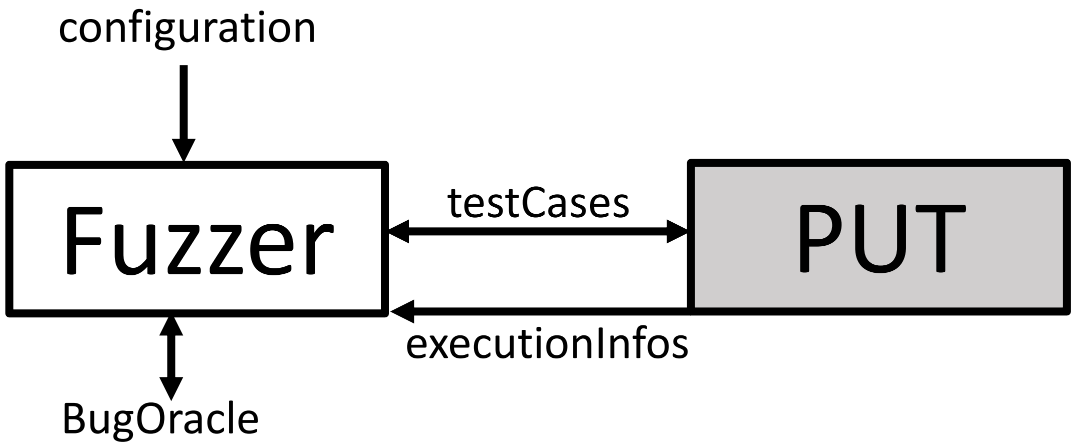


- White-box Fuzzer
  - Can explore the state-space of PUT systematically
  - Can instrument PUT to additionally deliver interesting information



Fuzzer

- Grey-box Fuzzer



- Middle ground
- Rely on approximated, imperfect information in order to gain more speed and thus test more inputs
- Know some information about the PUT
  - Dynamic: code coverage
  - Lightweight static analysis

5 Algorithms: **PreProcess(C)**

- Instrumentation (grey-box and white-box only)
    - Static (before **PreProcess** runs) on source code or intermediate code (at compile time)
    - Dynamic while running **InputGen** (at runtime)
        - Can instrument dynamically linked libraries
    - Similar to **ConfUpdate**
- Seed Selection (Idea: weed out potentially redundant information)
    - Problem of decreasing the size of initial seed pool: **seed selection problem**
    - minset, i.e. minimal set of seeds that maximizes a coverage metric
- Seed Trimming
    - Prioritizing smaller inputs is likely to yield higher throughput
- Driver programming (only once)
    - E.g. to be able to call functions in a library L in case our PUT is L
- Prepare a model for future input generation (**InputGen**)

5 Algorithms:
Schedule(C, timeElapsed, timeLimit)

- Goal: analyze currently available information about the configurations and pick a configuration that is likely to lead to the most favorable outcome
    - e.g. find highest number of unique bugs, maximize coverage of PUT
- Algorithms to address **Fuzz Configuration Scheduling (FCS) Problem**
    - time can either be spent on gathering more accurate information on each configuration to inform future decisions (explore) or
    - on fuzzing the configurations that are currently believed to lead to more favorable outcomes (exploit)
- Black-box FCS Algorithms
    - Available information: fuzz outcomes of a configuration, # bugs/crashes found, amount time spent so far
    - Postulate: configuration with higher ‘normalized’ success rate (#bugs/(computing time spent)) preferred
- Grey-box FCS Algorithms
    - Evolutionary development of configurations in configuration pool
    - Configurations that lead to control-flow jump/edge and have fastest and smallest input favorable

5 Algorithms: **InputGen(configuration)**

- Model-based Fuzzers
  - Predefined Model
  - Inferred Model
    - In **PreProcess**
    - In **ConfUpdate**
  - Encoder Model
    - Encoder program encodes data into a specific file format that will be decoded by the PUT
    - **InputGen** mutates encoder programm
- Mutation-based (model-less) Fuzzers, Seed-based Fuzzers
  - Random testing is often insufficient
  - Seed-based fuzzers modify a typically well-formed seed
  - Test cases that are close to the seed are usually **mostly valid** but also contain abnormal values that e.g. may trigger crashes of the PUT
  - Closeness
    - Bit-flip
    - Arithmetic mutation
    - Block-based mutation
    - Dictionary-based mutation

5 Algorithms: **InputGen(configuration)**

- White-box fuzzers may use access to PUT to find inputs that are accepted e.g. via program analysis
  - Symbolic Execution
    - Run program with symbolic values as inputs, which represent all possible values
    - If executed program so builds symbolic expressions instead of concrete values
    - Branches lead to forks
    - Paths can so be represented by formulas which in turn can be checked for satisfiability
  - Dynamic Symbolic Execution additionally executes concretely to help reduce the complexity of symbolic constraints
- Guided Fuzzing
  - Exploit program analysis techniques
- PUT Mutation

5 Algorithms: **InputEval(configuration, testCases, BugOracle)**

- BugOracles
  - Crash Yes/No
  - Memory Violation/Error
  - Undefined Behavior
  - Input Validation
  - Semantic Difference (differences in similar but different programs)
- Execution Optimization
  - Avoid costly loading processes of PUT by forking processes or loading memory images of program states => amortization of initial loading phase
- Triage: analyzing and reporting test cases that cause security violations
  - Deduplication ideal result: set of test cases that each trigger unique bug
  - Prioritization (**the fuzzer taming problem**): process of ranking violating test cases according to uniqueness and severity (determining exploitability of bug)
  - Test case minimation: identifying portion of test case that triggers the security violation and minimizing accordingly

5 Algorithms: **ConfUpdate(C, configuration, executionInfos)**

- Black-box fuzzer do not perform any program introspection beyond evaluating bug oracle
  => **ConfUpdate** typically leaves set C of fuzz configurations unmodified
- Grey-box and White-box have more sophisticated **ConfUpdate**
  - For example, an Evolutionary Algorithm may maintain a seed pool of promising seeds that evolves via biological evolution mechanisms like mutation, recombination, election
    - Node or branch coverage as fitness function: test cases that find new node or branch are added to test pool
    - Seed pool is intended as diverse subselection of all reachable paths representing current exploration of PUT
  - Maintaining a minset – minimal set of test cases that maximizes coverage metric