UNIVERSITY OF PADUA
UNIVERSITA' DEGLI STUDI DI PADOVA

FUZZING

Lazari Alberto - 2089120
Protopapa Francesco - 2079466
Scandaletti Elia - 2087934

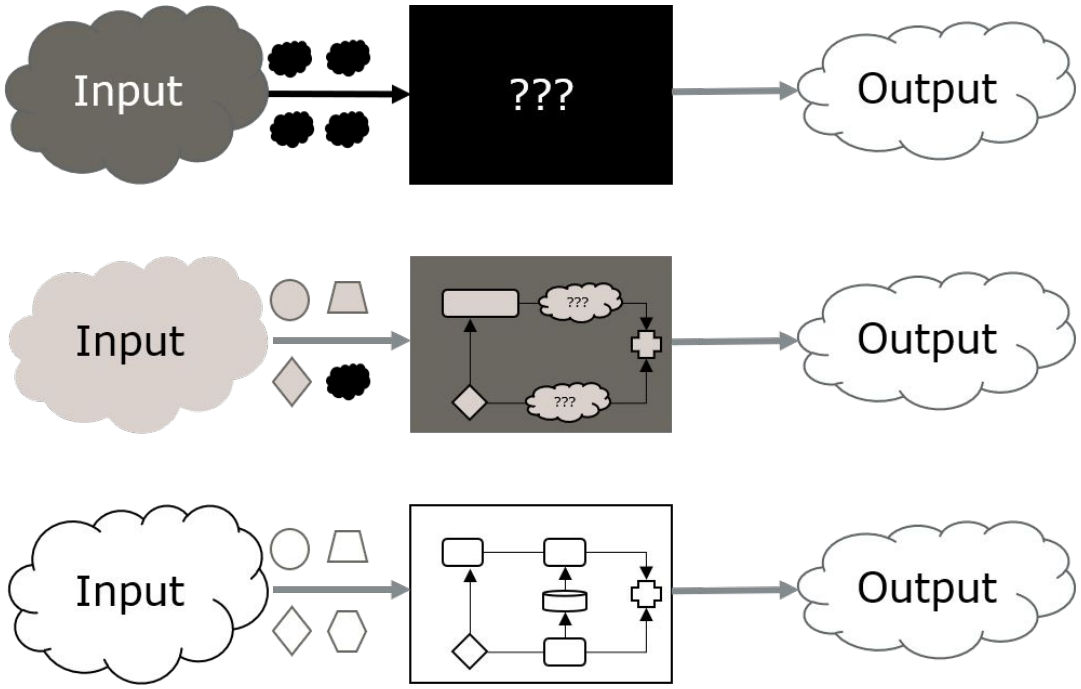
November 23, 2023

Fuzzing: Key Concepts

- Fuzzer
- Bug Oracle
- Fuzz Configuration
- Fuzz Campaign

Taxonomy of Fuzzers

- Black-box Fuzzer
- Grey-box Fuzzer
- White-box Fuzzer



Fuzz Testing Algorithm

1. Preprocess
2. Scheduling
3. Input Generation
4. Input Evaluation
5. Configuration Updating
6. Continue

ALGORITHM 1: Fuzz Testing

```
Input : ℂ, t_limit
Output : 𝔹 // a finite set of bugs
1 𝔹 ← ∅
2 ℂ ← PREPROCESS(ℂ)
3 while t_elapsed < t_limit ∧ CONTINUE(ℂ) do
4     conf ← SCHEDULE(ℂ, t_elapsed, t_limit)
5     tcs ← INPUTGEN(conf)
      // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← INPUTEVAL(conf, tcs, O_bug)
7     ℂ ← CONFUPDATE(ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9 return 𝔹
```

AFL/AFL++

- State of the art fuzzing tool
- AFL is in the Google cemetery
- AFL++ is an actively maintained fork to AFL

- Fuzzing steps:
  - Instrument target
  - Prepare campaign
  - Fuzz target
  - Monitor campaign

AFL/AFL++ - Instrumenting target

- Instrumenting = injecting code in the executable

- Source code → compile code with custom compiler *afl-cc*
- Binary code → QEMU use User Emulation Mode

- Instrumenting options:
  - Compiler
  - Custom sanitizers
  - Selectively execute code
  - Persistent mode

AFL/AFL++ - Preparing campaign

- Collecting inputs from any source
- Minimizing the number of inputs:
  a. Avoid input duplication
- Minimizing the size of each input

AFL/AFL++ - Fuzzing target

- Run *afl-fuzz*
- Setting maximum memory limit
- Parallelism
  a. One main process
  b. Many secondary processes
- Distribution
  a. One main process per server
  b. Servers may communicate or not

AFL/AFL++ - Monitoring campaign

- Manually introducing new inputs:
  a. Stopping fuzzing
  b. Expanding corpus
  c. Restarting fuzzing
- Monitoring the coverage → focus the campaign on certain inputs
- Triaging crashes

Benchmarking

How to benchmark fuzzing?

- Code coverage: who's better?
- Unique crashes: are they really unique?

How to protect from overfitting?

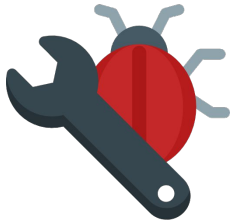
An ideal benchmark

Goals:

1. Real programs
2. Relevant bugs (crash, memory corruption)
3. Identifiable bugs
4. No overfitting

RevBugBench

- 10 real programs
- ~8000 known bugs injected, based on real fixes
- Overfitting? Tool to change programs



FixReverter

Automatically introduce bugs (undo fixes):

- Find fixes to remove
- Bug injection
- Triage phase

Fix patterns

- ABORT

```
- if (x == NULL) return;
```

- EXEC

```
- if (a == 2 && b != NULL) {
+ if (b != NULL) {
    do_stuff();
  }
```

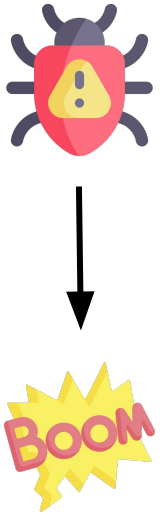
- ASSIGN

```
- if (x < 0) x = 1;
```

Triage

Test combination of causes to find:

- Individual-causes
- Combined-causes



Results

Benchmark run on 5 fuzzers

Best is AFL++ → 146/219 individual bugs

Thank you for your attention!