Structure-aware fuzzing

for real-world projects

Réka Kovács
Eötvös Loránd University, Hungary
rekanikolett@gmail.com

Overview

- tutorial, no groundbreaking discoveries

Motivation

- growing code size -> growing number of bugs
- big tech companies started to systematically fuzz their code recently
- we all should

Quality assurance

- coding guidelines
- compiler warnings
- code review
- test suite
- static analysis
- dynamic analysis
- random testing

Let’s look at who’s using this technology today.

Who is fuzzing their code today?

- Microsoft
  - every untrusted interface of every product is fuzzed (Security Development Lifecycle)
  - 670 machine-years devoted to fuzz Microsoft Edge & Internet Explorer, more than 400 billion DOM manipulations generated from 1 billion HTML files
  - Project Springfield (2016)

https://docs.microsoft.com/en-gb/microsoft-edge/deploy/group-policies/security-privacy-management-gp
https://www.microsoft.com/en-us/security-risk-detection/

Who is fuzzing their code today?

- Google
  - Chromium is fuzzed continuously with 15.000 cores
  - external reporters invited to write fuzzers
  - OSS-fuzz (2016): 158 open-source projects including Boost, Coreutils, CPython, FFmpeg, Firefox, LLVM, OpenSSH, OpenSSL, …

https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf
https://security.googleblog.com/2014/01/ffmpeg-and-thousand-fixes.html
https://opensource.google.com/projects/oss-fuzz

When did this all start?

History of fuzzing

- recently became a synonym for penetration testing
- term “fuzzing ”coined by prof. Bart Miller, University of Wisconsin-Madison

- 1990: original “fuzzing” paper

  Miller, B.P., Fredriksen, L. and So, B., 1990. **An empirical study of the reliability of UNIX utilities.** *Communications of the ACM*, *33*(12), pp.32-44.
  - completely random input to UNIX utilities
  - 25-33% crashed

History of fuzzing

- 1995: “Fuzz Revisited”: network apps, GUI apps

  Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A. and Steidl, J., 1995. ***Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.*** Technical report.

- 2000: Windows NT applications

  Forrester, J.E. and Miller, B.P., 2000, August. **An empirical study of the robustness of Windows NT applications using random testing.** In *Proceedings of the 4th USENIX Windows System Symposium* (Vol. 4, pp. 59-68).

- 2006: MacOS applications: 22/30 GUI apps crashed

  Miller, B.P., Cooksey, G. and Moore, F., 2006, July. **An empirical study of the robustness of macos applications using random testing.** In *Proceedings of the 1st international workshop on Random testing* (pp. 46-54). ACM.

History of fuzzing

“smart” fuzzers:

- 2011: CSmith https://embed.cs.utah.edu/csmith/

  Yang, X., Chen, Y., Eide, E. and Regehr, J., 2011, June. **Finding and understanding bugs in C compilers**. In *ACM SIGPLAN Notices* (Vol. 46, No. 6, pp. 283-294). ACM.

  - generates well-formed C programs from scratch
  - created to test compilers
  - ~80 gcc bugs, ~200 clang bugs reported

History of fuzzing

"smart" fuzzers:

- 2012: SAGE

  Godefroid, P., Levin, M.Y. and Molnar, D., 2012. **SAGE: whitebox fuzzing for security testing**. *Queue*, *10*(1), p.20.

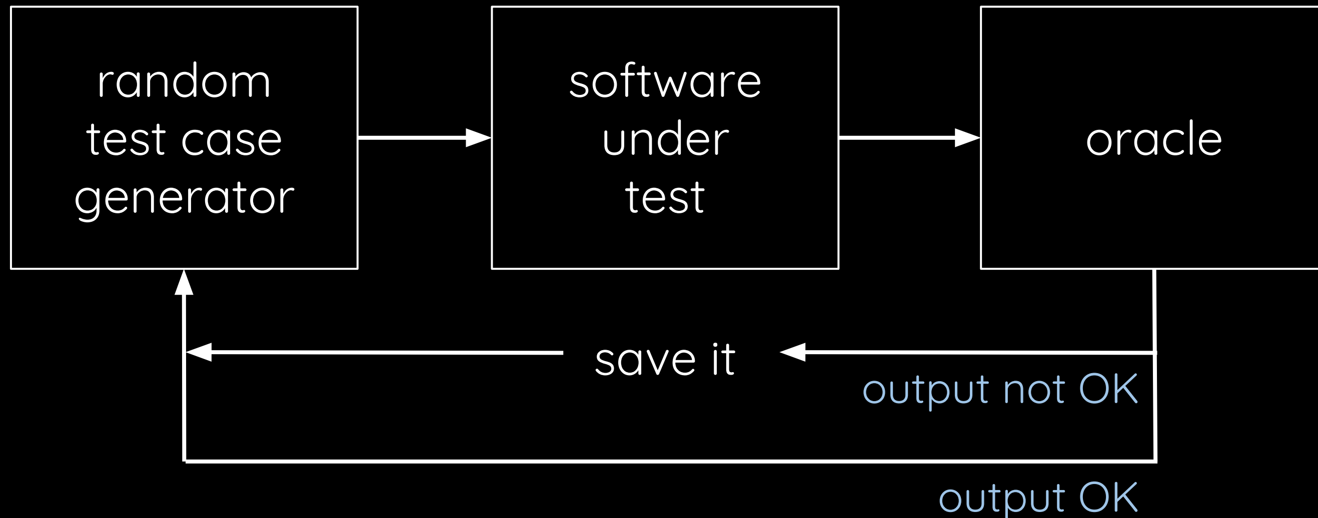
  - discovers new corner cases efficiently by combining symbolic execution and dynamic analysis

```
if (x == 179000)
    abort(); // error
```

Great! I want to fuzz my code.
How do I go about it?

How does fuzzing work?

John Regehr & Sean Bennett: Software Testing
https://eu.udacity.com/course/software-testing--cs258



Oracles

John Regehr & Sean Bennett: Software Testing
https://eu.udacity.com/course/software-testing--cs258

Weak

- crash (hardware, OS)
- rule violation of enhanced execution environment
  - Valgrind
  - sanitizers

Medium

- assertions

Strong

- alternative implementation
  - differential testing
  - old version of software
  - reference implementation
- inverse function pair
  - e.g. encrypt/decrypt
- null space transformation

Input structure

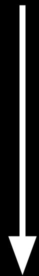
e.g. web browsers

random bits

protocol-correct code

valid HTML

scripts, forms



“dumb” fuzzer

“smart” fuzzer

Program structure

Black-box fuzzer

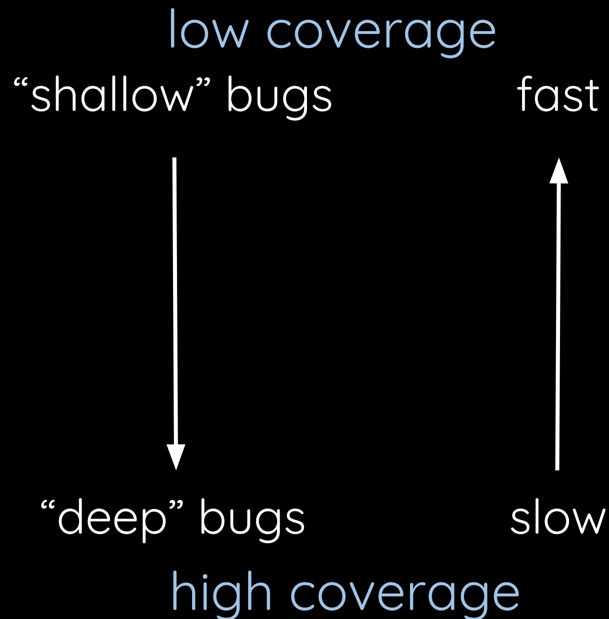
- no coverage feedback

Grey-box fuzzer

- lightweight instrumentation
- e.g. AFL, libFuzzer

White-box fuzzer

- heavyweight program analysis
- e.g. SAGE



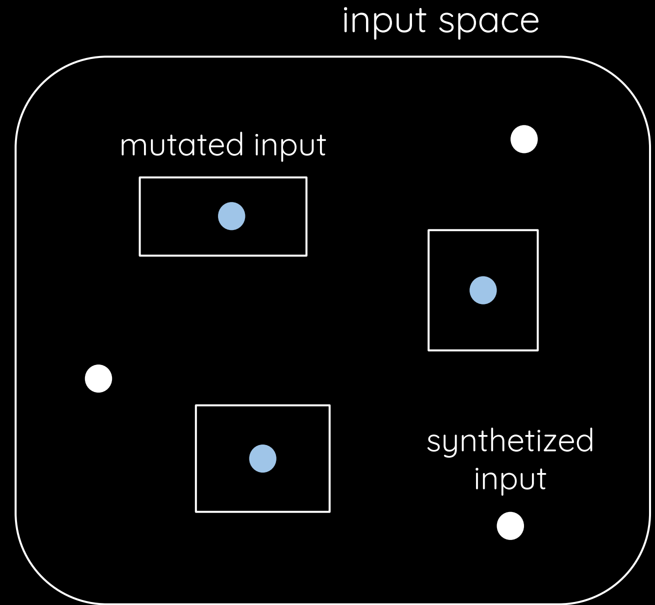
Reuse of input seeds

Generative

- synthetize test cases from scratch
- complex, a lot of work
- e.g. CSmith

Mutation-based

- modify (non-)random test cases
- treats input as a bag of bits
- e.g. AFL, libFuzzer



This is too complicated. I want to set it up easily.
What are my options?

Tools

- if your code has never been fuzzed: black-box fuzzers
  - probably will find some bugs
- white-box fuzzers are a lot of work
- excellent grey-box fuzzers!
  - AFL, libFuzzer
  - coverage-guided
  - can generate fairly structured inputs
    - e.g. JPEGs, IR code, primitive C programs

AFL: American Fuzzy Lop

http://lcamtuf.coredump.cx/afl/

- brute-force fuzzer with an instrumentation-guided genetic algorithm
- uses a modified form of edge coverage to pick up changes to program control flow
- needs user-supplied test cases that it can mutate
- result: a corpus of interesting test cases

AFL: American Fuzzy Lop

- algorithm roughly:
  - load initial test cases into a queue
  - take next input from the queue
  - try to trim the test case
  - repeatedly mutate the file
  - if any of the mutations resulted in a new state, add the mutated output to the queue

```
#include <iostream>

int hi(const std::string &data, std::size_t size) {
  if (size > 0 && data[0] == 'H')
    if (size > 1 && data[1] == 'I')
      if (size > 2 && data[2] == '!')
        __builtin_trap();
  return 0;
}

int main() {
  std::string s;
  std::cin >> s;
  return hi(s, s.length());
}
```

```
                    american fuzzy lop 2.52b (hi)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 12 sec       │  cycles done : 14     │
│   last new path : 0 days, 0 hrs, 0 min, 5 sec        │  total paths : 7      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 3 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ───────────────────┬─ map coverage ──┴───────────────────────┤
│  now processing : 3 (42.86%)       │    map density : 0.01% / 0.03%         │
│ paths timed out : 0 (0.00%)        │ count coverage : 1.00 bits/tuple       │
├─ stage progress ───────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                │ favored paths : 6 (85.71%)             │
│ stage execs : 216/256 (84.38%)     │  new edges on : 7 (100.00%)            │
│ total execs : 68.8k                │ total crashes : 1 (1 unique)           │
│  exec speed : 5440/sec             │  total tmouts : 4 (1 unique)           │
├─ fuzzing strategy yields ──────────┴─────────────┬─ path geometry ──────────┤
│   bit flips : 2/280, 0/273, 0/259                │    levels : 4            │
│  byte flips : 0/35, 0/28, 0/16                   │   pending : 0            │
│ arithmetics : 0/1955, 0/469, 0/0                 │  pend fav : 0            │
│  known ints : 1/159, 0/772, 0/704                │ own finds : 6            │
│  dictionary : 0/0, 0/0, 0/0                      │  imported : n/a          │
│       havoc : 4/46.3k, 0/17.2k                   │ stability : 100.00%      │
│        trim : 42.86%/9, 0.00%                    ├──────────────────────────┘
└──────────────────────────────────────────────────┘          [cpu000: 25%]
```

libFuzzer

https://llvm.org/docs/LibFuzzer.html

- in-process, coverage-guided, evolutionary fuzzing engine
- code coverage information provided by LLVM’s SanitizerCoverage
- generates mutations on the corpus of input data in order to maximize the code coverage
- works without initial seeds

libFuzzer: input generation

- generic random fuzzing
  - e.g. clang-fuzzer, clang-format-fuzzer, …
    https://llvm.org/docs/FuzzingLLVM.html
- custom mutators
  - Justin Bogner: Adventures in Fuzzing Instruction Selection
    https://www.youtube.com/watch?v=UBbQ_s6hNgg
- structured fuzzing using libprotobuf-mutator
  - Kostya Serebryany: Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator
    https://www.youtube.com/watch?v=U60hC16HEDY

Protocol buffers

```
message Const {
  required int32 val = 1;
}

message BinaryOp {
  enum BinOp {
    PLUS = 0;
    MINUS = 1;
    MUL = 2;
    DIV = 3;
    MOD = 4;
  };
  required BinOp kind = 1;
  required Expr left = 2;
  required Expr right = 3;
}
```

```
message UnaryOp {
  enum UnOp {
    ABS = 1;
    SQRT = 2;
  };
  required UnOp kind = 1;
  required Expr arg = 2;
}

message Expr {
  oneof expr_oneof {
    Const constant = 1;
    BinaryOp binop = 2;
    UnaryOp unop = 3;
  }
}
```

Thank you!