AFL++ & KLEE demos (Lecture 5 part 1)

Prof. Mathy Vanhoef

DistriNet – KU Leuven – Belgium

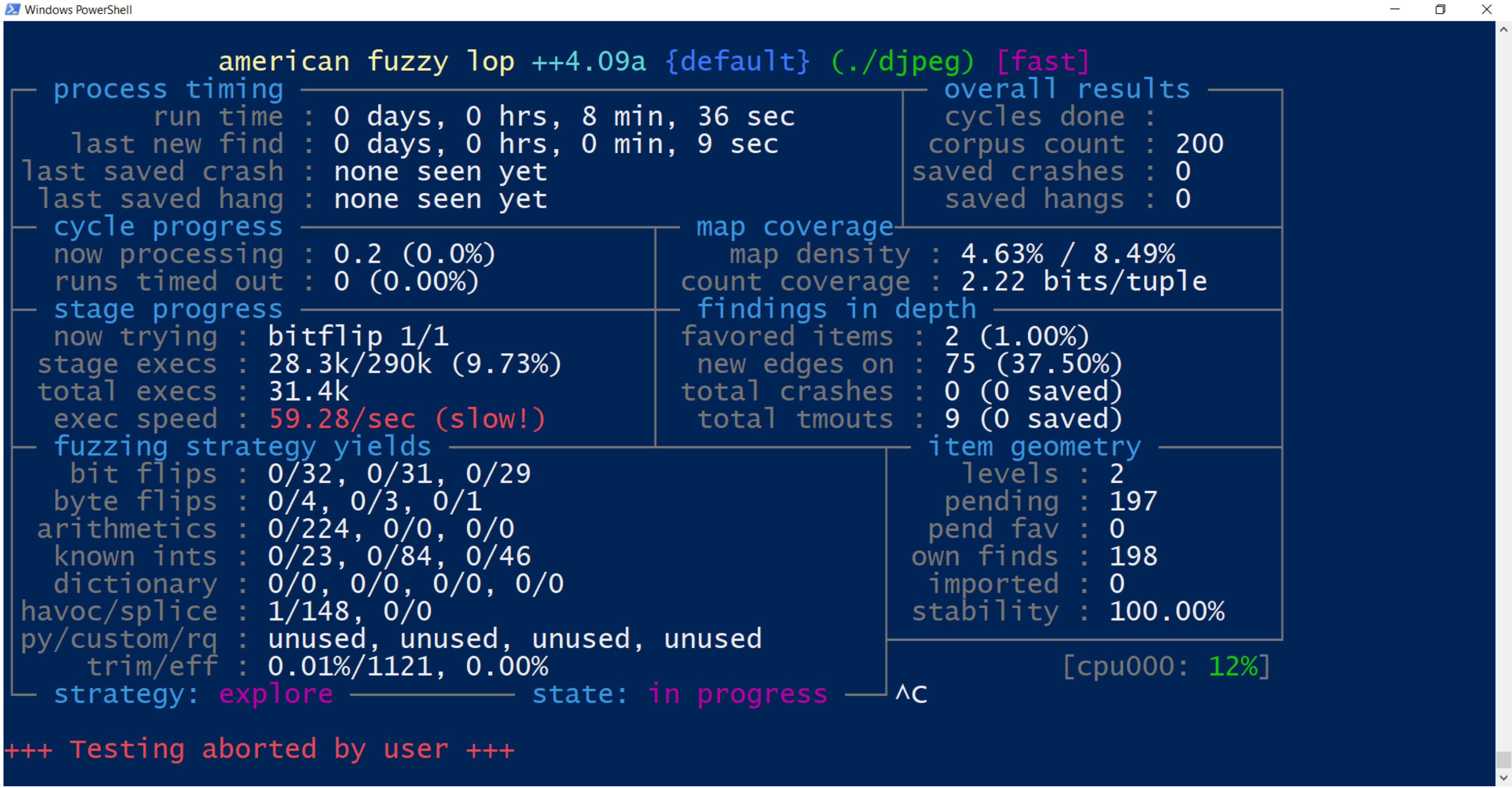
AFL++ Demo: “JPEGs out of thin air”

AFL guide (!!): https://aflplus.plus/docs/fuzzing_in_depth/

- Target: https://github.com/libjpeg-turbo/libjpeg-turbo
- Read build instructions of project → uses cmake
  - If needed, search how to change the compiler
  - Example solution for cmake: https://stackoverflow.com/a/17275650
- Essential commands:

```
mkdir build && cd build
CC=afl-cc CXX=afl-c++ cmake ..
# create in_dir and out_dir and fill with seeds
afl-fuzz -i in_dir/ -o out_dir/ ./djpeg @@
```

AFL++ Demo: “JPEGs out of thin air”



```
Windows PowerShell

                american fuzzy lop ++4.09a {default} (./djpeg) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 8 min, 36 sec         │  cycles done :       │
│   last new find : 0 days, 0 hrs, 0 min, 9 sec          │ corpus count : 200   │
│last saved crash : none seen yet                        │saved crashes : 0     │
│ last saved hang : none seen yet                        │  saved hangs : 0     │
├─ cycle progress ─────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0.2 (0.0%)          │    map density : 4.63% / 8.49%       │
│  runs timed out : 0 (0.00%)           │ count coverage : 2.22 bits/tuple     │
├─ stage progress ─────────────────────┼─ findings in depth ───────────────────┤
│  now trying : bitflip 1/1             │ favored items : 2 (1.00%)            │
│ stage execs : 28.3k/290k (9.73%)      │  new edges on : 75 (37.50%)          │
│ total execs : 31.4k                   │ total crashes : 0 (0 saved)          │
│  exec speed : 59.28/sec (slow!)       │  total tmouts : 9 (0 saved)          │
├─ fuzzing strategy yields ────────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 0/32, 0/31, 0/29                        │    levels : 2         │
│  byte flips : 0/4, 0/3, 0/1                           │   pending : 197       │
│ arithmetics : 0/224, 0/0, 0/0                         │  pend fav : 0         │
│  known ints : 0/23, 0/84, 0/46                        │ own finds : 198       │
│  dictionary : 0/0, 0/0, 0/0, 0/0                      │  imported : 0         │
│havoc/splice : 1/148, 0/0                              │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused          ├───────────────────────┘
│    trim/eff : 0.01%/1121, 0.00%                       │          [cpu000: 12%]
└─ strategy: explore ────────── state: in progress ─────┘^C

+++ Testing aborted by user +++
```

AFL++ Demo: “JPEGs out of thin air”

Why is fuzzing slow?

- Not running in persistent mode: process is constantly restarted by AFL.
- Running in docker/VM, this slows things down.
- Only using single CPU, not fuzzing on each CPU in parallel.

How to discover (more) bugs?

- Also enable sanitizers during compilation!
- This slows down fuzzing… but can now find more bugs.

How to fuzz libraries?

Not every program/library accepts input from stdin

- Need to write custom harness to provide fuzzed inputs to the code/function being tested
- Typical harness structure:

```
int main() {
    /* 1. Initialize library */
    /* 2. Read input from fuzzer */
    /* 3. Call function being fuzzed with given input */
}
```

Example harness: fuzzing capstone disassembler

```
int main(int argc, char** argv) {
  uint8_t buf[128]; // The buffer we will pass to the library
  csh h; cs_insn *insn; size_t count;

  ssize_t numread = read(stdin, buf, 128);
  if (cs_open(CS_ARCH_X86, CS_MODE_64, &h) == CS_ERR_OK) {
    count = cs_disasm(h, buf, numread, 0x1000, 0, &insn);
    cs_free(insn, count); // clean up after ourselves
  } else return -1;
  cs_close(&h); // close the capstone library
}
```

Writing fuzz harness can be hard

- Need to understand library, know which function to call,…
- Harness may need updates to support new library version

Alternative: developer writes the harness! Use libFuzzer:

- Developer adds `LLVMFuzzerTestOneInput` function
- This function is part of the project. Can initialize needed functionality and call the code to be tested.
- Integrated into the latest clang compiler.

libFuzzer: demo

```
bool FuzzMe(const uint8_t *Data, size_t DataSize) {
  return DataSize >= 3 && Data[0] == 'F' &&
      Data[1] == 'U’ && Data[2] == 'Z’ && Data[3] == 'Z';
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t len) {
  FuzzMe(data, len);
  return 0;
}

// clang++ -g -fsanitize=address,fuzzer fuzzme.cpp
// ./a.out
```

KLEE demos

1. check-sign
   - `-posix-runtime` and `–sym-arg X` to provide a command-line argument consisting of X symbolic bytes.
2. strtol-sign
   - `-libc=uclibc` to symbolically analyze libc functions.
3. symmalloc.c
   - KLEE limitation: cannot handle symbolic malloc sizes.
   - Length will be concretized.

General remarks

- Docker images take up space:

Name	Tag	Status	Created	Size
aflplusplus/aflplusplus 13fdf7e8b621	latest	Unused	8 days ago	3.06 GB
klee/klee 673d72c9b2d7	latest	In use	5 months ago	10.44 GB

- First do AFL, then do KLEE. Periodically remove temp files.
- Windows: first start Docker, then run commands. Otherwise you get errors when starting the instances.

References

Optional reading:

- Fuzzing capstone using AFL persistent mode
- LLVMFuzzerTestOneInput integrated into capstone