Fuzzing project

- Steer clear of FFmpeg
- Look for evidence of fuzzing in the code repo

Fuzzing - last week

1. Basic fuzzing with random/long inputs
2. 'Dumb' mutational fuzzing

   example: OCPP
3. Generational fuzzing aka grammar-based fuzzing

   example: GSM
4. Whitebox fuzzing with SAGE

   using symbolic execution

Still left for today

1. Code-coverage guided evolutionary fuzzing with `afl`

   aka grey box fuzzing or 'smart' mutational fuzzing

Coverage-guided evolutionary fuzzing with `afl`

(American Fuzzy Lop)

Evolutionary Fuzzing

Use evolution:

try random input mutations, and

observe the effect on some form of coverage, and

let only the interesting mutations evolve further

where “interesting” = resulting in ‘new’ execution paths

Aka coverage-guided evolutionary greybox fuzzing,

but terminology is a bit messy/non-standard.

alf: observing jumps to find interesting inputs/input changes

input

code

line	instruction
1	**JMP 6**
2	..
3	..
4	..
5	**JZ (Jump If Zero) 7**
6	..
7	*arraycopy (dst, input[i..j]);*
8	
9	..
10	**JCXZ 2**
11	..
12	..
13	*println (part of input);*
14	..
15	**JNE 103131**
16	..

afl bitmap `shared_mem`

	1	2	3	4	5	6	7	8	9	10	11	12
1						☑[1]						
2												
3												
4												
5							☑[3]					
6												
7												
8												
9												
10		☑[2]										
11												
12												
13												
14												
15												

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

- Code instrumented to observe execution paths:
  - if source code is available, by using modified compiler
  - if source code is not available, by running code in an emulator
- Code coverage represented as a 64KB bitmap:
  each control flow jumps is mapped to a change in this bitmap
  - different executions could result in same bitmap, but chance is small
- Mutation strategies include: bit flips, incrementing/decrementing integers, using pre-defined interesting values (eg. 0, -1, MAX_INT,....) or user-supplied dictionary, deleting/combining/zeroing input blocks, ...
- The fuzzer forks the SUT to speed up the fuzzing
- Big win: no need to specify the input format, but still good coverage

afl's instrumentation of compiled code

Code is injected at every branch point in the code

```
cur_location = <SOME_RANDOM_NUMBER_FOR_THIS_CODE_BLOCK>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

where `shared_mem` is a 64 KB memory region

Intuition: for every jump from L_1 to L_2 a different byte in `shared_mem` is changed (increased).

Which byte is determined by random values chosen at compile time inserted at source and destination of every jump

```
                american fuzzy lop 2.52b (dnsmasq)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 20 hrs, 31 min, 27 sec       │  cycles done : 3       │
│   last new path : 0 days, 0 hrs, 48 min, 28 sec        │  total paths : 3409    │
│ last uniq crash : 0 days, 2 hrs, 22 min, 39 sec        │ uniq crashes : 12      │
│  last uniq hang : none seen yet                        │   uniq hangs : 0       │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3138* (92.05%)    │    map density : 0.34% / 4.51%          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.92 bits/tuple        │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : user extras (insert)  │ favored paths : 686 (20.12%)            │
│ stage execs : 509k/1.38M (36.79%)   │  new edges on : 1022 (29.98%)           │
│ total execs : 29.4M                 │ total crashes : 363 (12 unique)         │
│  exec speed : 464.9/sec             │  total tmouts : 54 (18 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 151/1.22M, 104/1.22M, 47/1.22M         │    levels : 17         │
│  byte flips : 0/152k, 2/61.4k, 4/59.8k               │   pending : 2326       │
│ arithmetics : 133/3.47M, 0/1.04M, 0/286k             │  pend fav : 7          │
│  known ints : 32/264k, 29/1.62M, 10/2.55M            │ own finds : 1887       │
│  dictionary : 103/2.43M, 48/5.49M, 176/1.58M         │  imported : n/a        │
│       havoc : 1060/6.14M, 0/0                        │ stability : 100.00%    │
│        trim : 40.91%/56.3k, 58.16%                   ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000:150%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

```
                  american fuzzy lop 2.52b (dnsmasq)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 20 hrs, 31 min, 27 sec      │  cycles done : 3      │
│   last new path : 0 days, 0 hrs, 48 min, 28 sec       │  total paths : 3409   │
│ last uniq crash : 0 days, 2 hrs, 22 min, 39 sec       │ uniq crashes : 12     │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3138* (92.05%)    │    map density : 0.34% / 4.51%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.92 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : user extras (insert)  │ favored paths : 686 (20.12%)           │
│ stage execs : 509k/1.38M (36.79%)   │  new edges on : 1022 (29.98%)          │
│ total execs : 29.4M                 │ total crashes : 363 (12 unique)        │
│  exec speed : 464.9/sec             │  total tmouts : 54 (18 unique)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 151/1.22M, 104/1.22M, 47/1.22M        │    levels : 17         │
│  byte flips : 0/152k, 2/61.4k, 4/59.8k              │   pending : 2326       │
│ arithmetics : 133/3.47M, 0/1.04M, 0/286k            │  pend fav : 7          │
│  known ints : 32/264k, 29/1.62M, 10/2.55M           │ own finds : 1887       │
│  dictionary : 103/2.43M, 48/5.49M, 176/1.58M        │  imported : n/a        │
│       havoc : 1060/6.14M, 0/0                       │ stability : 100.00%    │
│        trim : 40.91%/56.3k, 58.16%                  ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000:150%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

afl statistics

- total execs
- total paths
- (unique) crashes
- (unique) hangs
- cycles

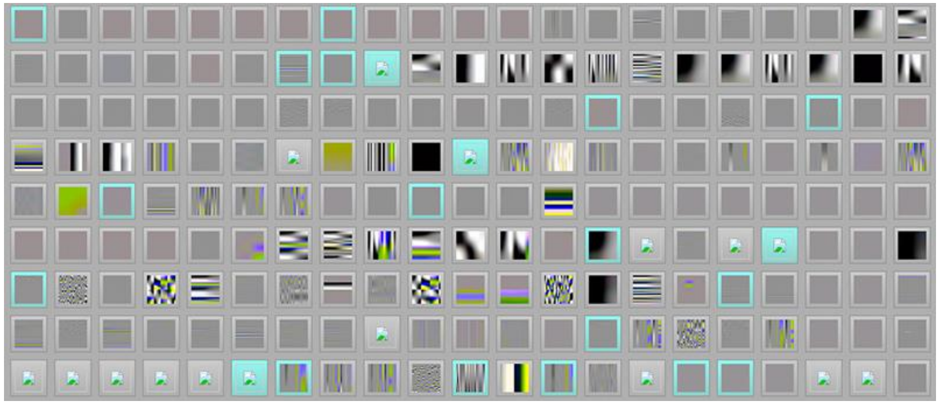
Cool example: learning the JPG file format

Fuzzing a program that expects a JPG as input, starting with 'hello world' as initial test input, afl can learn to produce legal JPG files

along the way producing/discovering error messages such as

- `Not a JPEG file: starts with 0x68 0x65`
- `Not a JPEG file: starts with 0xff 0x65`
- `Premature end of JPEG file`
- `Invalid JPEG file structure: two SOI markers`
- `Quantization table 0x0e was not defined`

and then JPGs like

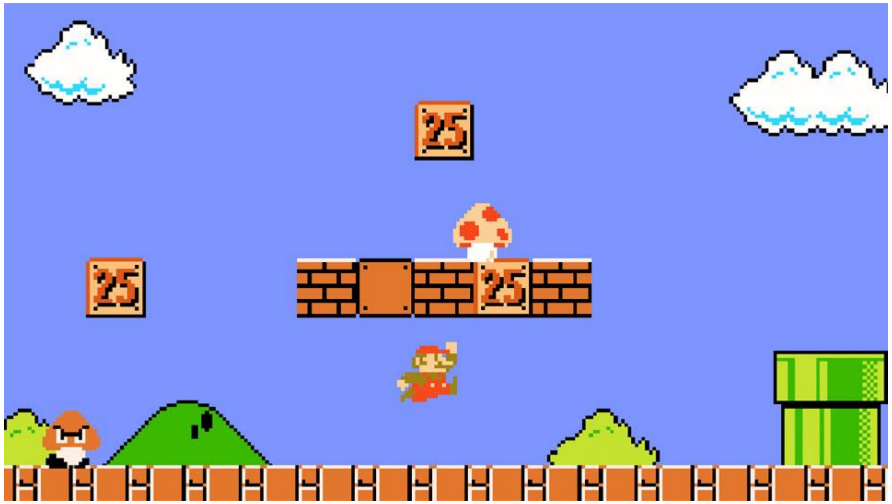


[Source http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html]

Other strategies in evolutionary fuzzing

Instead of maximizing path/code coverage, we can also let inputs evolve to maximize some other variable or property

- **Code may need to instrumented to let fuzzer observe that property**



Eg the x-coordinate of Super Mario

[Aschermann et al., *IJON: Exploring Deep State Spaces via Fuzzing*, IEEE S&P 2020]

https://www.youtube.com/watch?v=3PyhXlHDkNI

Conclusions

- Fuzzing is great technique to find (a certain class of) security flaws!
- If you ever write or use C(++) code, you should fuzz it.
- Challenge: getting good coverage fuzzing without too much effort

  Successful approaches include
  - White-box fuzzing based on symbolic execution with SAGE
  - Evolutionary fuzzing aka coverage guided greybox fuzzing with `afl`
- *Does fuzzing makes sense for code in other programming languages?*

  Yes, even if the kind of bugs found may have lower security impact.
- A more ambitious generation of tools not only tries to find security flaws, but also to then build exploits, eg. `angr`

To read (see links on the course page)

- Section 1 of technical white paper for afl
- Patrice Godefroid, *Fuzzing: Hack, Art, and Scienc*e CACM 2020