VASET, 18 August 2016

American Fuzzy Lop

Bernie Pope, bjpope@unimelb.edu.au

Outline

- Fuzz testing.
- Key features of AFL.
- Example use case.
- Program instrumentation.
- Test case mutation.
- Impressive results.

Fuzz Testing

- A program is applied to a wide variety of inputs, including unexpected, invalid and random inputs, in an attempt to provoke an error.
- Especially popular in security; searching for inputs which trigger security flaws.
- B.P. Miller, L. Fredriksen, and B. So, "*An Empirical Study of the Reliability of UNIX Utilities*", Communications of the ACM 33, 12 (December 1990).

Originally a graduate class assignment in *Advanced Operating Systems* subject at UW-Madison, 1988.

Fuzz Testing

- How to get good program coverage in reasonable time?
- Purely randomised inputs are unlikely to efficiently explore the input search space.
- Naive techniques probably only find shallow bugs.

American Fuzzy Lop (AFL)

- Author: Michał Zalewski
- License: Apache License, Version 2.0
- Platforms: most Unix-like systems, and there is a fork which runs on Windows.
- Most of this talk was inspired by the AFL docs, the AFL source code, and the Michał Zalewski's blog:

  http://lcamtuf.blogspot.com

Main Features

- Compile-time program instrumentation.
- Employs a carefully tuned test-case generation algorithm.
- Test case minimisation.
- Produces a corpus of test cases which can be used for other testing purposes.
- Has relatively low runtime overheads.

Overall Process

```
queue := initial_test_cases
seen := ∅

forever:
    new_queue := copy(queue)
    for next in queue:
        for test_input in mutate(next):
            signature := execute(program, test_input)
            if signature ∉ seen:
                new_queue.append(test_input)
                seen.add(signature)
    queue := cull(new_queue)
```

Example Use Case

```
#define MIN_DIGITS 6

int main(int argc, char **argv)
{
    char buf[MAXBUF];

    fgets(buf, MAXBUF-1, stdin);

    if (str_is_digits(buf) && (strlen(buf) >= MIN_DIGITS))
    {
       if (is_prime(atoi(buf)))
       {
          abort();
       }
    }
    return 0;
}
```

Toy program, for the sake of demonstration.

Example Use Case

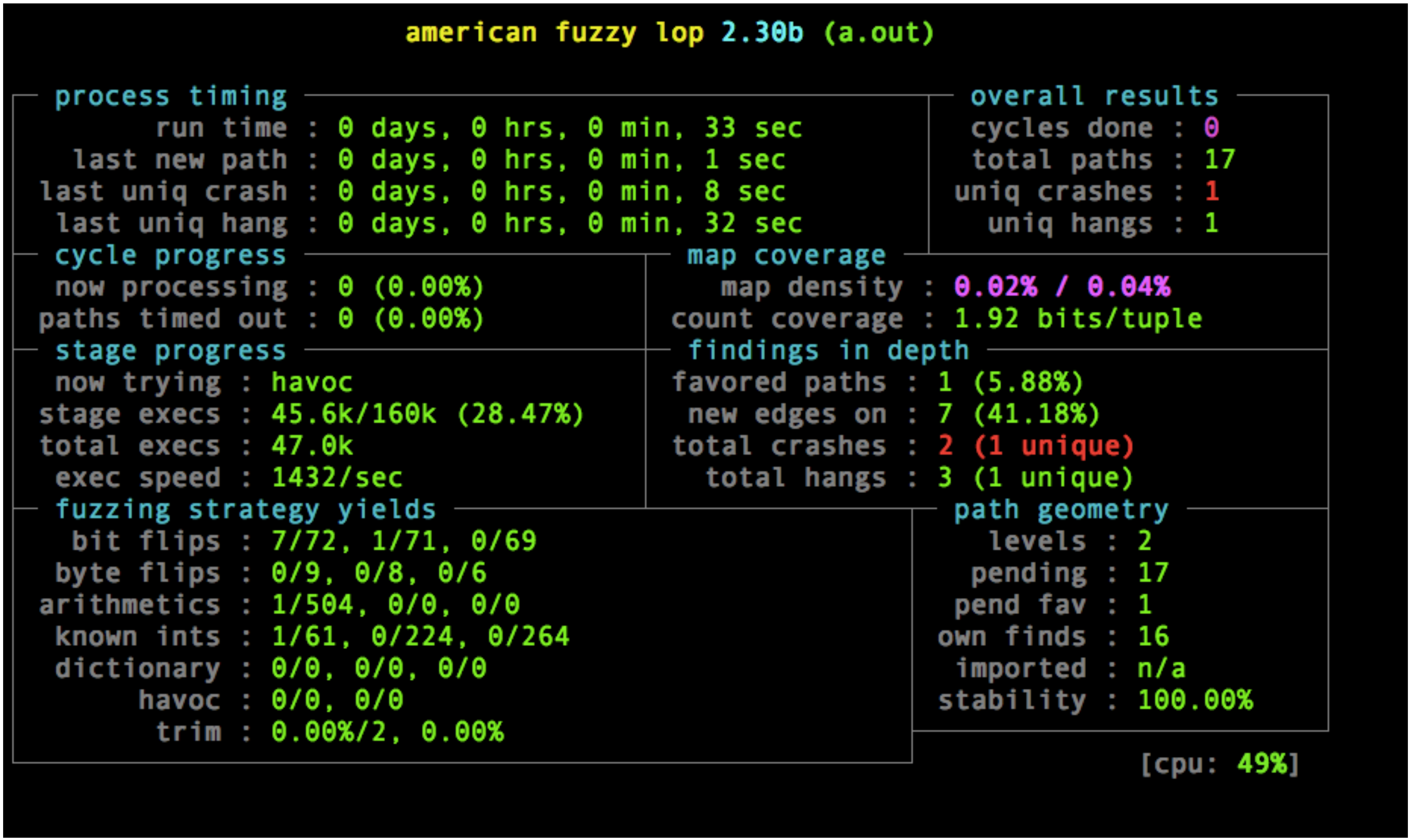
```
#define MIN_DIGITS 6

int main(int argc, char **argv)
{
    char buf[MAXBUF];

    fgets(buf, MAXBUF-1, stdin);

    if (str_is_digits(buf) && (strlen(buf) >= MIN_DIGITS))
    {
        if (is_prime(atoi(buf)))
        {
            abort();
        }
    }
    return 0;
}
```

Program aborts if input is a string of at least 6 digits denoting a prime number in base 10.

Example Use Case

```
# compile the program with the AFL compiler wrapper

afl-clang is_prime.c

# create an initial test case (a large non-prime)

mkdir test_cases
echo -n ‘492876842’ > test_cases/test.txt

# run the fuzzer on the compiled program
#    - specify directory containing initial test cases
#    - specify directory to store results (findings)

afl-fuzz -i test_cases -o findings -- ./a.out

# wait, monitor output, and hit control-c when done
```

AFL dashboard



Examine Findings

```
# inspect test case(s) which cause crashes

cat findings/crashes/id:000000,sig:06,src:000000,op:havoc,rep:4
449287?

# hmm, what is going on here?

od -a findings/crashes/id:000000,sig:06,src:000000,op:havoc,rep:4
0000000    4   4   9   2   8   7 nul soh   ?
0000011
```

Null byte was inserted at the 7th position. The program checks if 449287 is prime (which it is), and aborts.

Execution Signatures

- AFL computes a signature for each program execution.
- The signature approximates the set of branches taken by a program, and their counts.
- A signature is considered interesting if a new branch is taken, or a significant change occurs in the number of times a branch is taken.
- The signature does not retain any information about the order in which branches were taken.

Execution Signatures

- Branches (edges) are represented by tuples:

  (p_1, p_2)

  where p_1 and p_2 are program points

  p_1 is branch source

  p_2 is branch destination

- Branch counts are binned to: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

Execution Signatures

- Suppose the first execution of the program consists of this trace (ignoring counts):

  $A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow E$

- AFL records this set of tuples:

  (A, B), (B, C), (C, D), (D, E)

- And the next execution gives rise to this trace:

  $A \Rightarrow B \Rightarrow C \Rightarrow A \Rightarrow E$

- This is interesting because it includes a new tuples (C, A) and (A, E).
- However, this trace does not produce any new tuples, and is therefore not considered interesting:

  $A \Rightarrow B \Rightarrow C \Rightarrow A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow E$

Program Instrumentation

- Code inserted at branch points is (roughly):

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

Program Instrumentation

- Code inserted at

Only a fixed set of tuples is considered. Tuple keys are made by XORing program point identities.

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

`shared_mem` is a 64 kB array of 8 bit counters.

Program Instrumentation

Compile time random simplifies the generation of identifiers for program points, and keeps XOR distribution uniform.

- (roughly):

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

Program Instrumentation

- Edge directionality is recorded by giving each program point 2 identities.
  destination: `COMPILE_TIME_RANDOM`
  source: `COMPILE_TIME_RANDOM >> 1`

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

Program Instrumentation

- Tuple key collisions increase with branch count.
- Colliding tuples grows to 30% at 50,000 branches. However, many real test cases contain fewer discoverable branches.
- The 64 kB table can easily fit into L2 cache, and can be analysed in microseconds.
- The 8 bit counters can overflow (and wrap).

Program Instrumentation

- afl_clang (afl_gcc, etc) is a compiler wrapper, applying a transformation on the output assembly stream.
- The transformation looks for branch labels emitted by the compiler, and conditional branch instructions.

Test Case Mutation

- Initial mutations are deterministic changes:
    - bit flips
    - addition and subtraction of small integers
    - insertion of interesting values, 0, 1, INT_MAX …
- Randomised mutations are tried next, including splicing of different test cases.
- AFL can monitor the success rate of each mutation strategy for a given program and modulate the choice of strategy to try to increase yield.
- Experiments have been run on many different input formats to get a feeling for effectiveness of strategies. E.g. walking bit flips of a single bit tends to yield 70 new execution signatures per million test cases tried:

  https://lcamtuf.blogspot.com.au/2014/08/binary-fuzzing-strategies-what-works.html

Ornate Input Grammars

- Bit flipping style changes are quite effective for simple "binary" formats, but will have difficulty navigating input formats from complex grammars (e.g. HTML files, computer programs).
- To combat this you can feed AFL a list of tokens from the input language (e.g. keywords of a programming language).
- It can find interesting rearrangements of input tokens and thus "discover" some of the underlying grammar.

Impressive Results

- Synthesised valid JPEG images from a starting input string of “hello” (after a couple of days fuzzing).
- Lots of bugs found in many popular libraries and tools, including some significant security issues (e.g. Shellshock)

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