IA159 Formal Methods for Software Analysis

American Fuzzy Lop

Jan Strejček

Faculty of Informatics
Masaryk University

Focus and sources

focus

- main concepts under AFL
- demo of AFL++

sources

- `https://github.com/google/AFL/blob/master/docs/technical_details.txt`
- `https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer)`
- A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti: Dissecting American Fuzzy Lop: A FuzzBench Evaluation, ACM TOSEM 2023.

Thanks to Marek Trtík for help with demonstration preparation.

Basic facts about AFL

- developed by Michał Zalewski, initial release in 2013
- last version by the original author: 2.52b (2017)
- current stable version: 2.57b (2020)
- open source, available under Apache License 2.0
- considered as a state of the art fuzzer for many years
- discovered bugs in OpenSSH, Firefox, Safari, MySQL, ...
- many clones with various features, in particular AFL++

Basic facts about AFL

- developed by Michał Zalewski, initial release in 2013
- last version by the original author: 2.52b (2017)
- current stable version: 2.57b (2020)
- open source, available under Apache License 2.0
- considered as a state of the art fuzzer for many years
- discovered bugs in OpenSSH, Firefox, Safari, MySQL, ...
- many clones with various features, in particular AFL++
- American Fuzzy Lop is a rabbit breed

Kguzman99, CC BY-SA 3.0

Basic ideas

"The only governing principles are
speed, reliability, and ease of use"

(M. Zalewski)

- `afl-fuzz` is a greybox fuzzer: program (target) is instrumented to measure the coverage of each run
- given input seeds are mutated
- inputs that discover something new are collected and mutated again
- don't do anything too expensive or specific for some program class

Coverage measurement

- the instrumented program captures edge coverage (for each edge between basic blocks, we track the number of visits in the current execution)

Coverage measurement

- the instrumented program captures edge coverage (for each edge between basic blocks, we track the number of visits in the current execution)
- each basic block is instrumented with

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

- random location identifiers simplify linking complex projects and keep XOR (^) uniformly distributed
- `shared_mem` array is a 64 kB region (fits into L2 cache)
  - indices (2 bytes) represent pairs (`prev_location`, `cur_location`)
  - values (1 byte) represent numbers of edge visits (hitcounts)
  - indices can collide, hitcounts can overflow
- shift (>>) used because of loops: `A^A = B^B = 0x00`
- shared memory survives a crash of the program (another thread can read it)

Hitcounts bucketing

- positive edge hitcounts are assigned to the buckets

1	2	3	4-7	8-15	16-31	31-127	128-255

- behavior of the run is given by these bucketed hitcounts
- fuzzer maintains another 64 kB table that remembers the bucketed hitcounts for individual edges seen so far
- an input is interesting if it produces a new bucketed hitcount for some edge; it is discarded otherwise
- differences within one bucket are considered not important

Algorithm

- takes the instrumented program and at least one input
- inputs are consumed via standard input or as an input file

Algorithm

- takes the instrumented program and at least one input
- inputs are consumed via standard input or as an input file

```
foreach input ∈ seeds do                                   // initial phase
    execute the program on the input
    trim the input such that the behavior is not changed
    insert the trimmed input to queue
set up limits for a single execution (e.g. timeout)
while true do                                              // the main fuzzing loop
    while queue is not empty do
        take input from queue
        if input should be skipped then continue
        trim the input
        mutate the input, execute, add interesting mutants to queue
    put all inputs that were in queue back to queue
    determine favored inputs
```

Trimming inputs

- small inputs = less mutations and faster executions
- some mutations increase input size

Trimming inputs

- small inputs = less mutations and faster executions
- some mutations increase input size

```
do
    oldinput ← input
    remove some block from the input
while the behavior remains unchanged
return oldinput
```

Trimming inputs

- small inputs = less mutations and faster executions
- some mutations increase input size

```
do
    oldinput ← input
    remove some block from the input
while the behavior remains unchanged
return oldinput
```

- removed blocks are of increasing size
- average per input gain is 5–20%
- tool `afl-tmin` implements a more expensive algorithm, used e.g. to minimize inputs that exhibit some program bug

Skipping inputs

- inputs are skipped to get to the favored ones faster

Skipping inputs

- inputs are skipped to get to the favored ones faster

```
if the current input is favored then
    do not skip
else
    if the queue contains favored inputs then
        skip the input with probability 99%
    else
        if the current input was mutated before (in previous cycles) then
            skip the input with probability 95%
        else
            skip the input with probability 75%
```

Favored inputs computation (aka culling the corpus)

- favored inputs have to jointly cover all edges covered so far
- small inputs with short execution times are preferred

Favored inputs computation (aka culling the corpus)

- favored inputs have to jointly cover all edges covered so far
- small inputs with short execution times are preferred

mark all inputs as non-favored
to each input assign a score propositional to execution time and input size
foreach edge covered by the inputs so far **do**
    if the edge is not covered by any favored input **then**
        select the input with the lowest score that covers the edge
        mark this input as favored

Favored inputs computation (aka culling the corpus)

- favored inputs have to jointly cover all edges covered so far
- small inputs with short execution times are preferred

```
mark all inputs as non-favored
to each input assign a score propositional to execution time and input size
foreach edge covered by the inputs so far do
    if the edge is not covered by any favored input then
        select the input with the lowest score that covers the edge
        mark this input as favored
```

- usually 10–20% of inputs are marked as favored
- tool `afl-cmin` provides a more sophisticated and slower algorithm (e.g. for prunning the resulting corpus of inputs)

Mutating inputs (aka fuzzing)

mutations are generated in this order

1. deterministic mutations
2. nondeterministic mutations (havoc)
3. splicing (combines two inputs into one)

Deterministic mutations

- flipping (i.e., inverting) 1-32 bits with various stepovers
- incrementing or decrementing 8-, 16-, and 32-bit integers, in both little- and big-endian encodings
- overwriting parts of the input with “approximately two dozen ’interesting’ values”, including 0 and maximum and minimum signed and unsigned integers of various widths, again in both little- and big-endian encodings
- replacing parts of the input with data drawn from a dictionary of user-specified or auto-detected tokens

Havoc mutations

- new input is produced by 2 to 128 mutations of the following types
  - the deterministic mutations described before
  - overwriting bytes with random values
  - deleting a multi-byte block
  - duplicating a multi-byte block
  - setting each byte in a block to a single value

Mutations by splicing

- activated only after the fuzzer goes through a full cycle of the entire queue without any new finding
- combines the current input with another input in the queue
- truncates both of them at arbitrary positions, concatenates them together, and applies the havoc stage to the result

De-duplication of crashes

- need to identify different reasons of crashes
- identification by the faulting instruction is insufficient
  (e.g. when the instruction is in a common library function)
- `afl-fuzz` considers a crash unique if
  - the crash trace includes an edge not seen in any of the previous crashes or
  - the crash trace is missing an edge that was always present in earlier crashes

Architecture

- repeated `execve()`, linking and `libc` initialization of the instrumented program takes time
- `afl-fuzz` uses fork-server that forks the execution of the instrumented code using copy-on-write
- performance gain on fast programs is usually between 1.5x and 2x

Notes

- parallel fuzzing: use of multiple cores or remote machines
- if sources are not available, instrumentation of binaries is used
- dictionaries are very important

Notes

- parallel fuzzing: use of multiple cores or remote machines
- if sources are not available, instrumentation of binaries is used
- dictionaries are very important

- AFL++ is available as a package
  `sudo apt install afl++`
- documentation available at
  `https://github.com/AFLplusplus/AFLplusplus`

Try it on your code!

Advertisement

Fizzer

- fuzzer developed at FI MU since 2023 by Marek Trtík and students
- slower program executions, more targeted input generation
- more information obtained from executions, aimed to flip the results of branching statements
- success in Test-Comp
- topics for bachelor and master theses