Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Andrea Fioraldi[1], Alessandro Mantovani[1], Dominik Maier[2], Davide Balzarotti[1]

[1]EURECOM, [2]TU Berlin

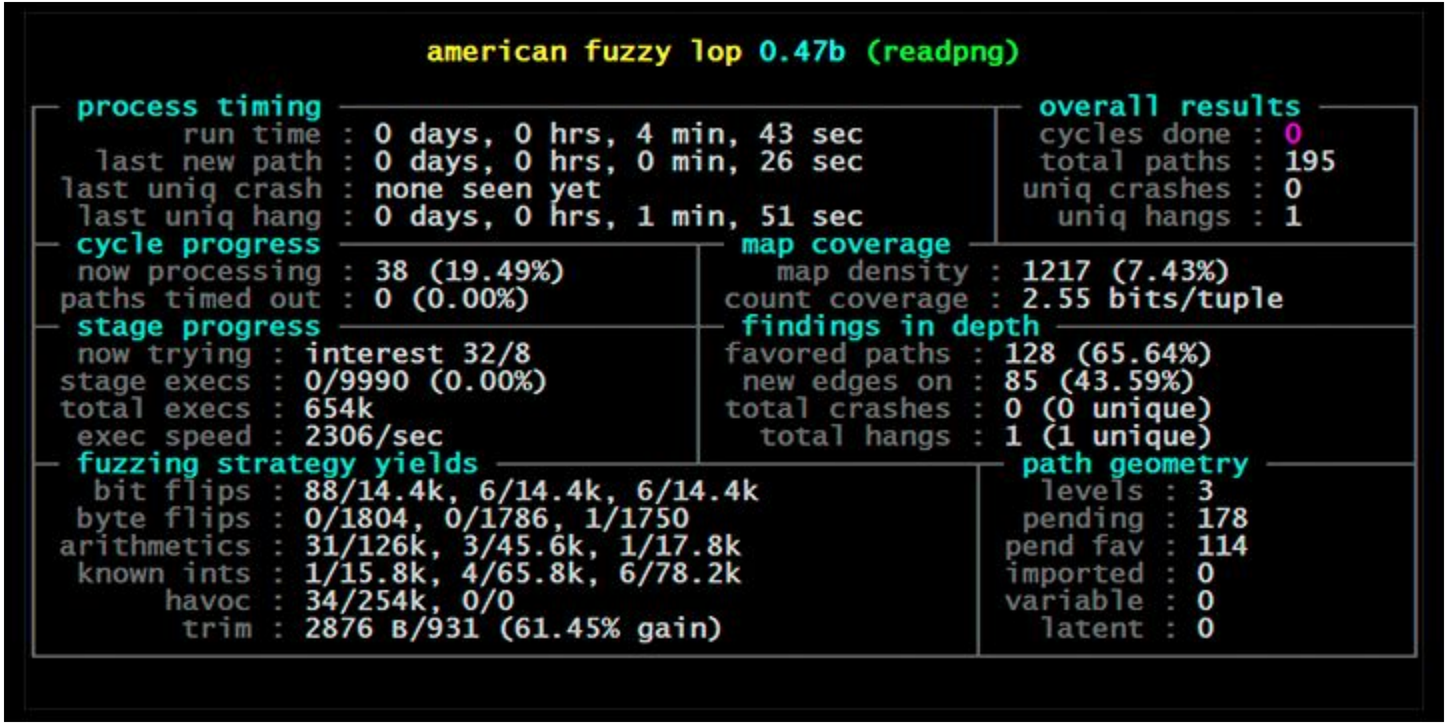
@andreafioraldi

fioraldi@eurecom.fr

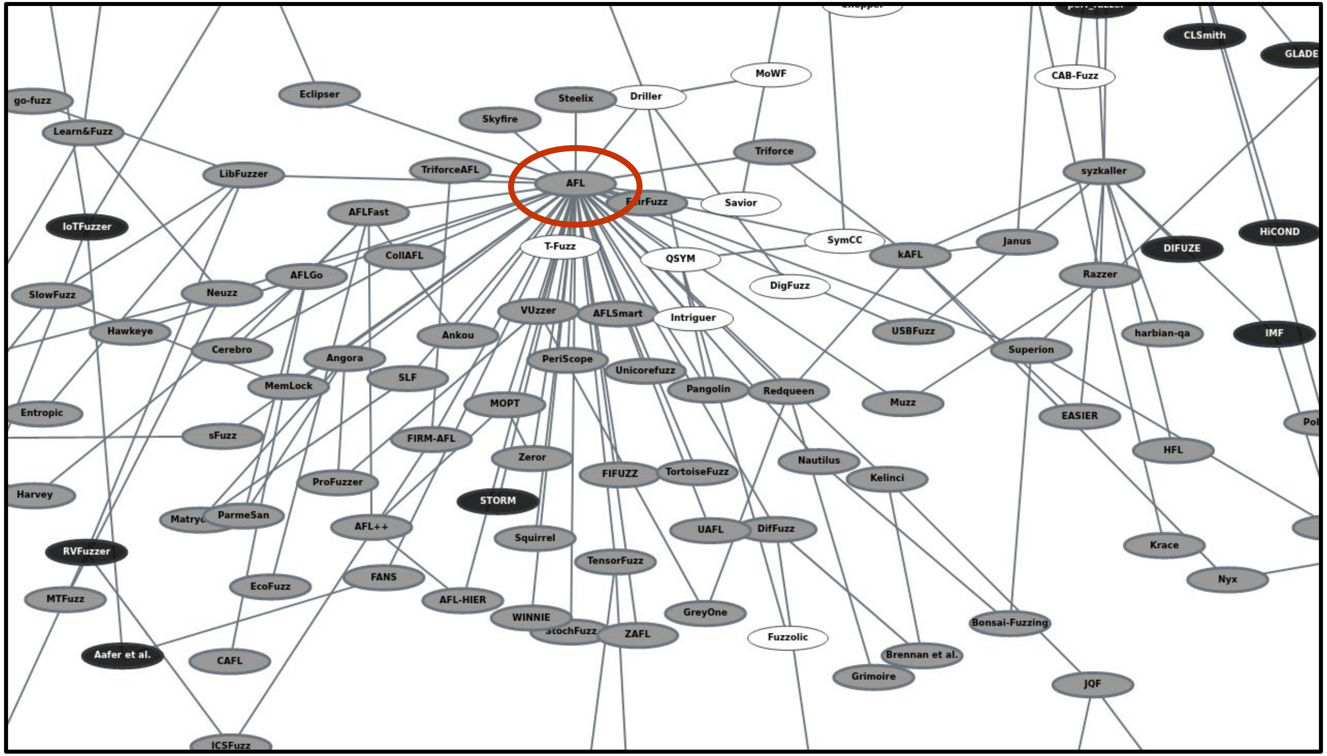
EURECOM
Sophia Antipolis

Technische Universität Berlin

American Fuzzy Lop



Why



From https://fuzzing-survey.org/

Core Principles

- **speed**
  - forkserver, bitwise operations for coverage evaluation, L2-sized shared map, lightweight inline instrumentation

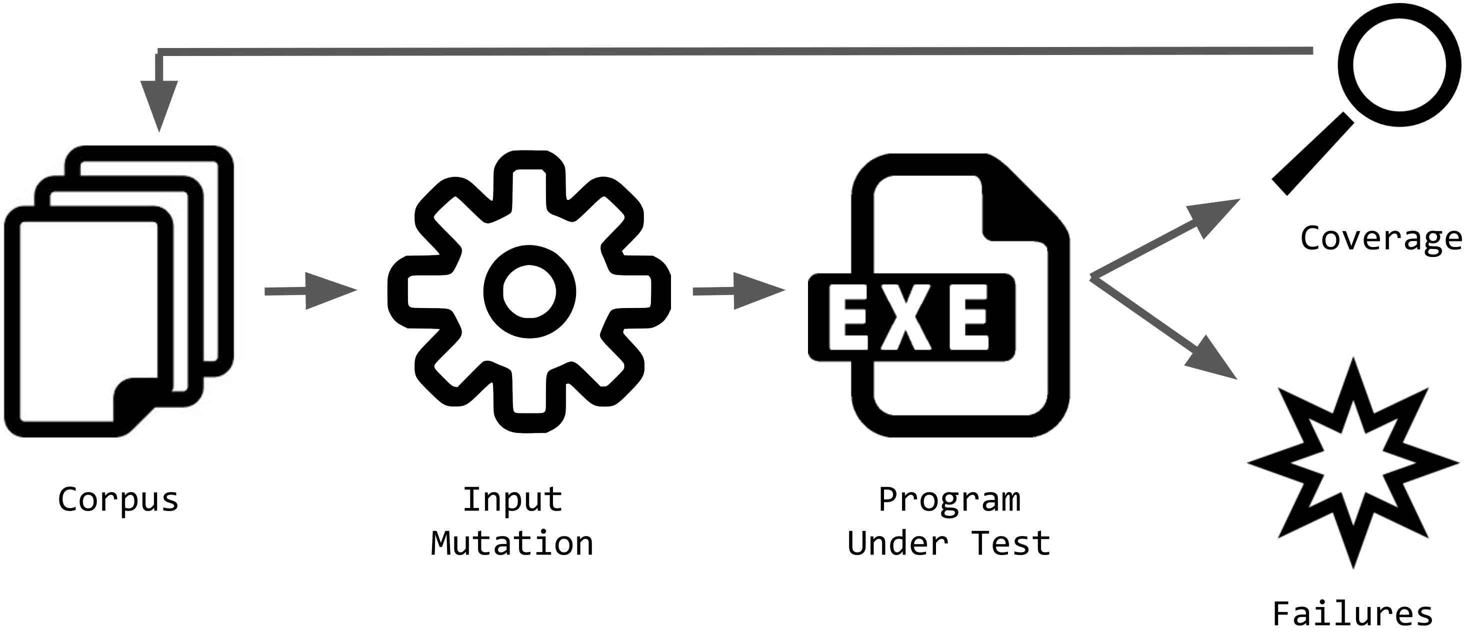
Core Principles

- **speed**
- **reliability**
    - forkserver, calibration and stability detection, low memory usage

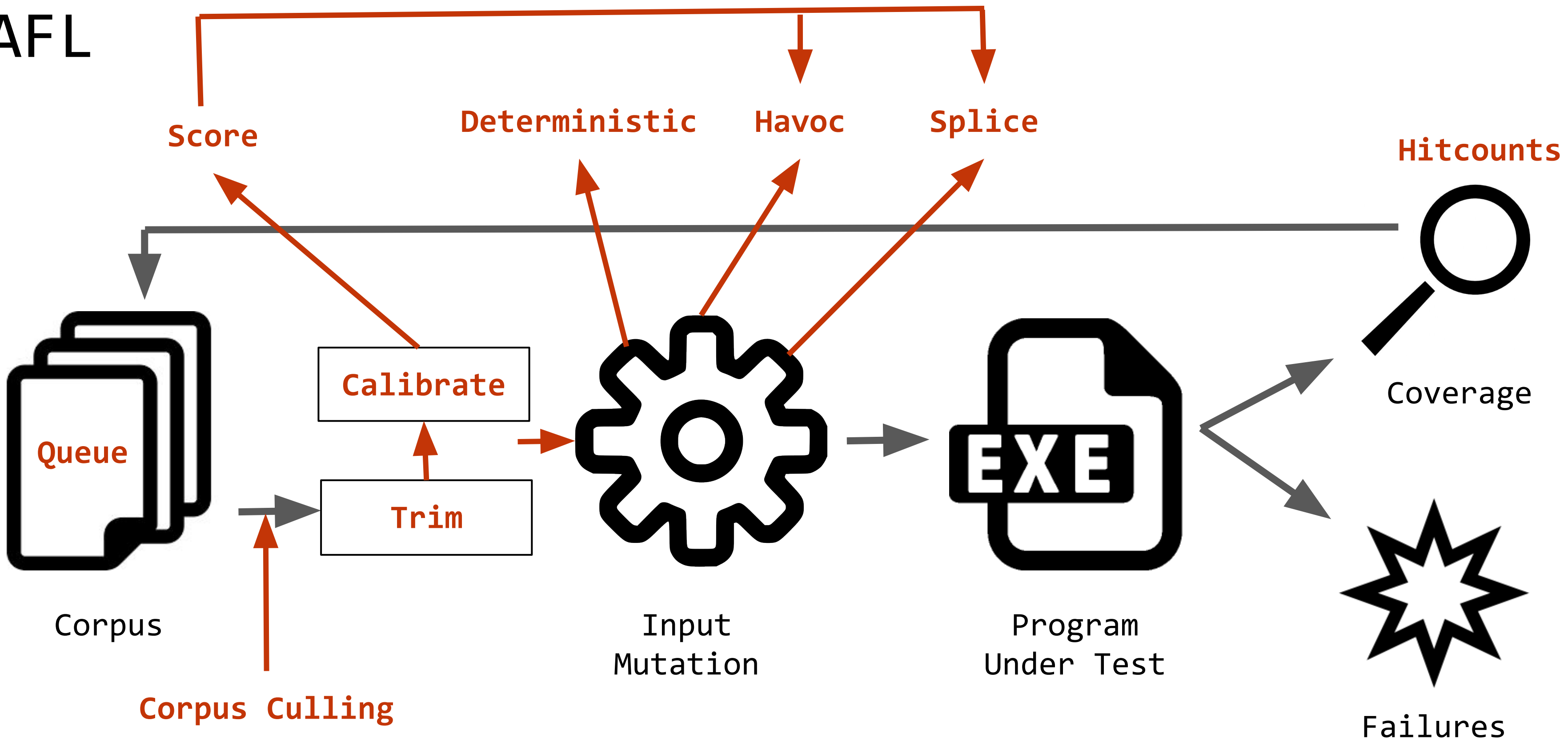
Core Principles

- **speed**
- **reliability**
- **ease of use**
    - corpus as a queue, deterministic mutations, testcases minimization, dictionaries

Coverage-guided Fuzzing



AFL


Score
Deterministic
Havoc
Splice
Hitcounts
Queue
Calibrate
Trim
EXE
Coverage
Corpus
Input
Mutation
Program
Under Test
Corpus Culling
Failures

Evaluating AFL aspects

By reviewing the implementation and the internals of AFL, we identified nine characteristics to assess in our tests.

We mainly use the bug benchmark of FuzzBench, which consists of 25 targets known to contain bugs.

Each program is executed for 23 hours. The reported results are median values over 20 trials to mitigate the effects of randomness in fuzzing and the Mann-Whitney U test is used to verify the statistical significance of the results. The aggregation of the results is done using an average normalized score.

Hitcounts

```
cur_location = <COMPILE_TIME_RANDOM>;

shared_mem[cur_location ^ prev_location]++;

prev_location = cur_location >> 1;
```

To avoid path explosion each entry is then divided into buckets:

1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

Preliminary Evaluation

Fuzzer	Average normalized score
AFL edge coverage	88.09
AFL	74.36

TABLE I: Hitcounts vs. plain edge coverage bug-based experiment score

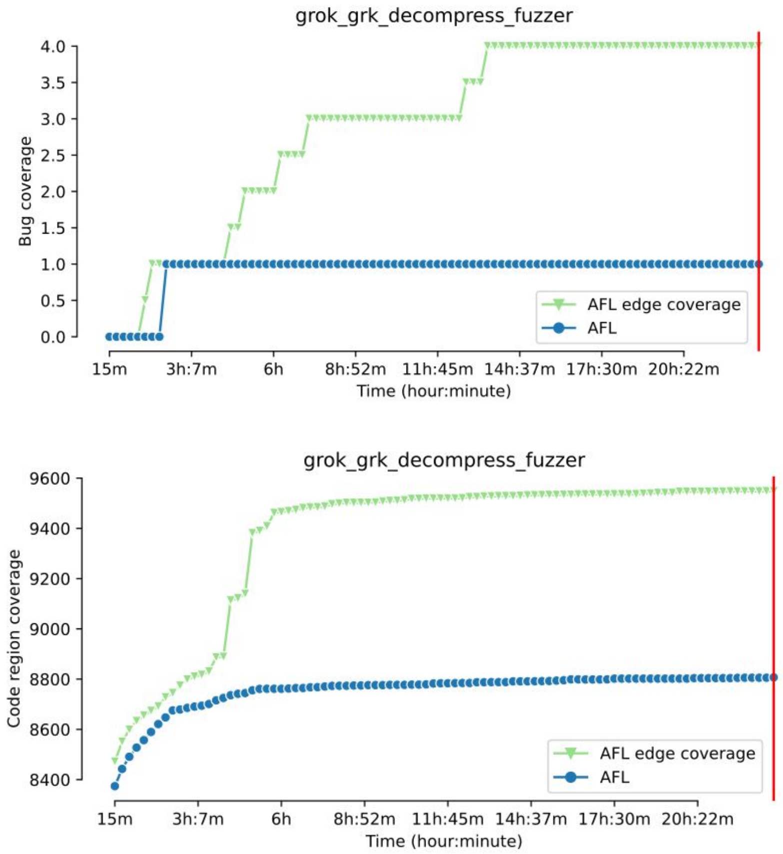


Fig. 2: Median code coverage growth on grok (Hitcounts vs. plain edge coverage experiment)

Preliminary Evaluation

Fuzzer	Average normalized score
AFL edge coverage	88.09
AFL	74.36

TABLE I: Hitcounts vs. plain edge coverage bug-based experiment score

Fuzzer	Average normalized score
AFL	99.63
AFL edge coverage	97.99

TABLE II: Hitcounts vs. plain edge coverage code coverage-based experiment score

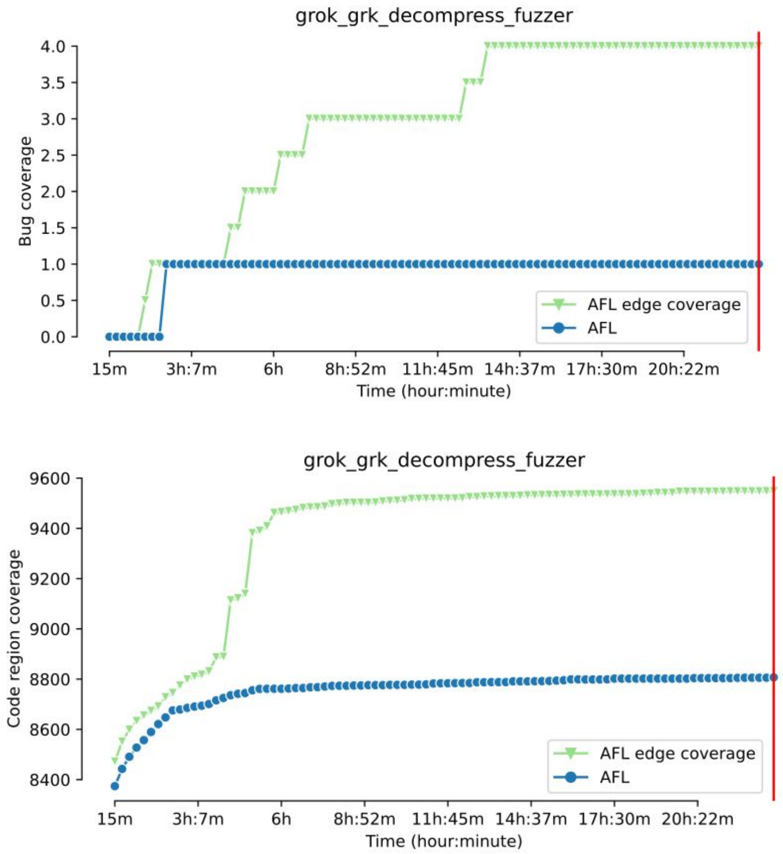
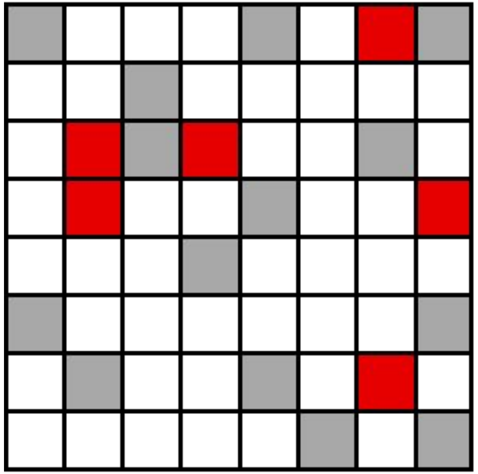


Fig. 2: Median code coverage growth on grok (Hitcounts vs. plain edge coverage experiment)

Novelty search vs. fitness maximization

$$f(i) = |\mathrm{BB}(i)| \begin{cases} \dfrac{\sum_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b))}{\log_2(\mathrm{len}(i))} & \text{if } \mathrm{len}(i) > 50000 \\ \displaystyle\sum_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b)) & \text{otherwise} \end{cases}$$



Preliminary Evaluation

Fuzzer	Average normalized score
`AFL`	83.32
`AFL fitness`	83.08
`AFL fitness only`	70.17

TABLE III: Novelty search vs. maximization of a fitness bug-based experiment score

Preliminary Evaluation

Fuzzer	Average normalized score
AFL	83.32
AFL fitness	83.08
AFL fitness only	70.17

TABLE III: Novelty search vs. maximization of a fitness bug-based experiment score

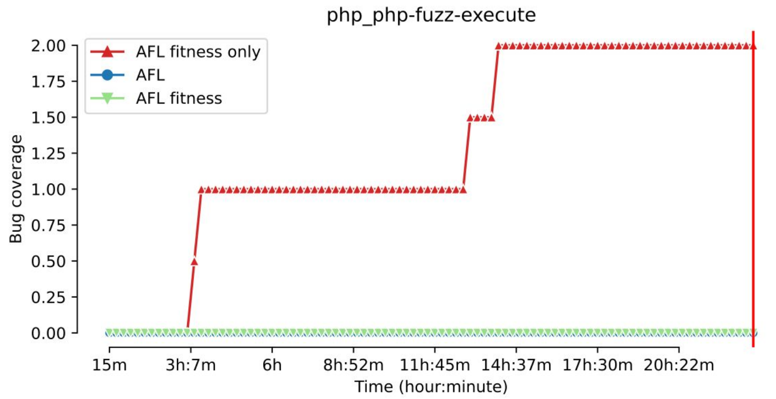


Fig. 3: Median bug coverage growth on PHP (Novelty search vs. maximization of a fitness)

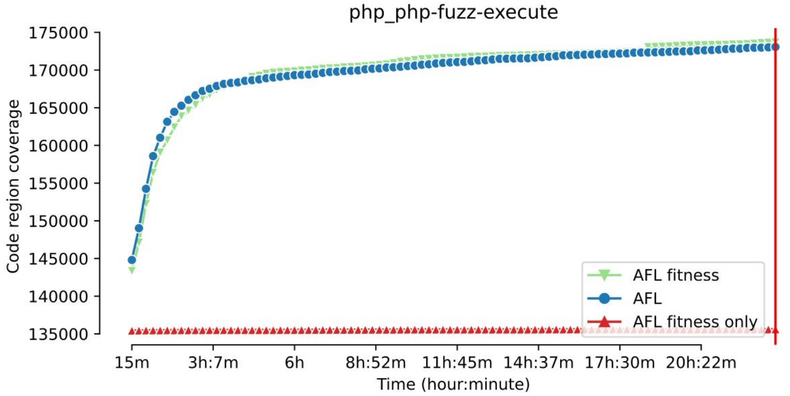


Fig. 4: Median code coverage growth on PHP (Novelty search vs. maximization of a fitness)

Corpus culling

AFL, periodically, evaluates the testcases in queue. It assigns a score proportional on execution latency and file size. Then, for each index of the bitmap, it selects the testcase with lowest score.

A minimized set of the corpus is devised in this way:

1. Find next index not yet in the temporary working set,
2. Locate the winning queue entry for this index,
3. Register *all* indexes present in that entry's trace in the working set,
4. Go to #1 if there are any missing indexes in the set.

All the located winning queue entries are marked as favored.

Corpus culling

AFL, periodically, evaluates the testcases in queue. It assigns a score proportional on execution latency and file size. Then, for each index of the bitmap, it selects the testcase with l

A minimized set of

1. Find next inde
2. Locate the win
3. Register *all* ing set,
4. Go to #1 if there are any missing indexes in the set.

All the located winning queue entries are marked as favored.

Can a fuzzer that reasons on the entire queue and not on a minimized set of testcase trigger different bugs due to the increased diversity?

Score calculation

The performance score used to calculate how many times to mutate and execute the input in the havoc and splice stages are derived from many variables, mainly testcase size and execution time.

In this experiment, we want to measure the delta between the AFL solution and the baseline, represented by a constant (two variants, minimum and maximum score) and a random score.

In addition, we include in the experiment a variant that does not prioritize novel corpus entries as this was a significant optimization in the AFL history.

Score calculation

The performance score used to calculate how many times to mutate and execute the input in the havoc and splice stages are derived from many variables, mainly testcase size and execution t

In this experiment, tion and the baseline, represent m score) and a random score.

In addition, we inc oritize novel corpus entries as t ory.

Is the different score changing drastically the outcome of the fuzzer? We foresee that the major contribution is the prioritization of the novelties, with a small delta between the other variants.

Corpus scheduling

The FIFO policy used by AFL is only one of the possible policies that a fuzzer can adopt to select the next testcase. This is a usability feature. However, derived works tend to take the corpus structure as a queue for granted.

We want to evaluate AFL versus a modified version that implements the baseline, random selection, and the opposite approach, a LIFO scheduler.

Corpus scheduling

The FIFO policy used by AFL is only one of the possible policies that a fuzzer can adopt to select the next testcase. This is a usability feature. However, derived works tend to take the corpus structure as a queue for granted.

We want to evaluate baseline, random selection, a

We expect that the random performs equal or even better than the original AFL, while the LIFO approach may help in gaining coverage faster on some targets.

Splicing as stage vs. splicing as mutation

Splicing refers to the operation that merges two different testcases. In AFL, it is a stage in which the merge happens before the mutations and the the havoc mutator is applied on the merged testacase.

However, other fuzzers (e.g. Libfuzzer) often implement splicing as a mutation rather than a stage, thus applying it many more times for each testcase during their havoc stage.

Splicing as a stage has the roots in usability, as it leads to less convoluted testcases.

Splicing as stage vs. splicing as mutation

Splicing refers to the operation that merges two different testcases. In AFL, it is a stage in which the merge happens before the mutations and the the havoc mutator is applied on the merge

However, other fuzz mutation rather than a stage, thus ng their havoc stage.

Splicing as a stage onvoluted testcases.

We expect that a splicing as mutation in AFL can increase the exploration of the fuzzer while reducing the simplicity of the testcases and, therefore, complicating the a-posteriori triaging phase.

Trimming

Trimming the testcases allows the fuzzer to reduce the size of the input files and consequently give priority to small inputs, under the assumptions that large inputs introduce a slowdown in the execution and the mutations would be less likely to modify an important portion of the binary structure.

Despite the fact that this algorithm can bring the two important benefits described above, we argue that reducing the size of the testcases could lead to lose state coverage and this operation can be a bottleneck for slow targets.

Trimming

Trimming the testcases allows the fuzzer to reduce the size of the input files and consequently give priority to small inputs, under the assumptions that large inputs introduce a slowdown in the execution and the mutations would be less likely to modify an important

Despite the fact th fits described above, we argue tha lose state coverage and this c

Our hypothesis is that trimming can be either beneficial or detrimental depending on the type of target program and the structure of its input.

Timeout Calculation

AFL can automatically compute a timeout value for the program under test. More specifically, as a first step, AFL calibrates the execution speed during an initial phase by running the target several times and computing an average of the execution times. After that, the default heuristic applies a constant factor (x5) to this average value and rounds it up to 20 ms.

In our experiments, we try to modify the multiplicative factor (2x, 10x) to measure its effect on the fuzzing session.

Timeout Calculation

AFL can automatically compute a timeout value for the program under test. More specifically, as a first step, AFL calibrates the execution speed during an initial phase by running the target several times and computing an average of the execution times. After that, ... (5) to this average value and r...

In our experiments, ... 0x) to measure its effect on the f...

We expect that a higher timeout can lead to a better coverage, but also degrade the performance of the fuzzer, while a smaller one can detrimental in the long run.

Collisions

In our evaluation, we want to compare the AFL instrumentations approach against a collision-free one.SanitizerCoverage splits critical edges into basic blocks and trace them at runtime with guard variables. AFL assigns random indentifiers to the guards and so having collisions, but a simple incremental counter instead would remove the collisions.

We want to benchmark this feature as the collision-free variant is simpler than the original implementation with pcguard, raising the question why random identifiers are used in AFL. In addition, it is unclear if the lack of feedback from the indirect jumps affects the performance more than the collisions, so we include the classic approach too in order to benchmark this impact.

Collisions

In our evaluation, we want to compare the AFL instrumentations approach against a collision-free one.SanitizerCoverage splits critical edges into basic blocks and trace them at runtime ... entifiers to the guards and so havi ... tead would remove the collisio ...

We want to benchmar ... mpler than the original implementa ... identifiers are used in AFL. In add ... the indirect jumps affects the p ... the classic approach too in orde ...

We expect an improvement in coverage for the collision-free variant but it is unclear if it can outperforms the classic instrumentation with the hash of the previous and current block.

Thank you!

Questions?