AFL Fast Optimized

Abhishek Badjatya, Akanksha Gupta, Jiamin Huang, Min Fan

Columbia University

***Abstract*—AFL or American Fuzzy Lop uses the concept of Coverage-based Greybox Fuzzing (CGF). CGF uses lightweight (binary) instrumentation to determine a unique identifier for the current path being explored by exercising on an input. New test inputs are generated by mutating the seed input to find unexplored and interesting paths. If the latest mutation provides a new interesting path, further mutations are carried on with that seed input, else the input is discarded. Our motivation from AFL and AFL Fast was to analyze the two fuzzing tools in market and understand the efficiency achieved by AFL Fast over AFL. We did in depth analysis of the five power schedules that AFL Fast introduces. Power schedules are the various strategies to control the number of inputs generated from a seed with the objective to exercise a larger number of low-frequency paths at the same time. Further, we bring our own version of AFL Fast Optimized with six additional power schedules, namely Triala, Trialb, MCOE, MINS, Hybrid and Average, out of which the "Average" power schedule has found to be better than the current AFL Fast in terms of performance.**

Keywords—AFL, AFL-Fast, Fuzzing, Power Schedule, CFG (Control Flow Graphs), Markov Model

I. Introduction

Currently, AFL Fast makes use of 6 types of power schedules. These power schedules make use of circular queue to store the mutated inputs.

A. EXPLOIT

The exploitation based constant schedule assigns constant energy to the mutated input based on execution time, block transition coverage, and creation time. It is the power schedule used in AFL.

B. EXPLORE

The exploration based constant schedule also assigns constant but fairly low energy.

C. COE

The Cut-Off Exponential schedule prevents high-frequency paths to be fuzzed totally until they become low-frequency paths.

D. FAST

The exponential schedule is an extension of COE. Instead of cutting off the high-frequency paths totally, it calculates the energy of the input which is inversely proportional to the fuzzing that has taken place till now for it. This is the default power schedule that is very efficient and finds bugs and crashes faster than AFL.

E. LINEAR

The linear power schedule increases the energy in a linear fashion based upon the number of times the input has been chosen from queue but is also inversely proportional to the number of times it has been fuzzed.

F. QUAD

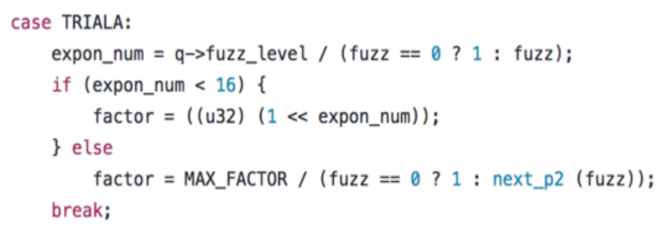
The quadratic schedule increases the energy in quadratic fashion instead of linear manner as done in linear mode.

Currently AFL Fast can be run in any one of the above mentioned power schedule modes, AFL Fast mode being the default. The Fast schedule results in highest performance among the six power schedules and is able to find maximum crashes and hangs during the same time period.

II. Our Approach

We approached to expand our understanding of AFL and AFL Fast with analyzing the research papers about AFL and AFL Fast. We crafted test cases that had hidden bugs for AFL, AFL Fast and our trails to detect them. We also made use of open source codes like that of Gravity compiler and the CPython compiler to validate our results. We first ran AFL and the five power schedules of AFL Fast on our test cases and compared the results. We analyzed the Markov Model of Coverage-based Greybox Fuzzing (CGF) to improve the efficiency and speed of the fuzzing process. Next, we designed and implemented our own power schedules. Our various attempts and schedules are explained below.

A. TRIAL A



```
case TRIALA:
    expon_num = q->fuzz_level / (fuzz == 0 ? 1 : fuzz);
    if (expon_num < 16) {
        factor = ((u32) (1 << expon_num));
    } else
        factor = MAX_FACTOR / (fuzz == 0 ? 1 : next_p2 (fuzz));
    break;
```

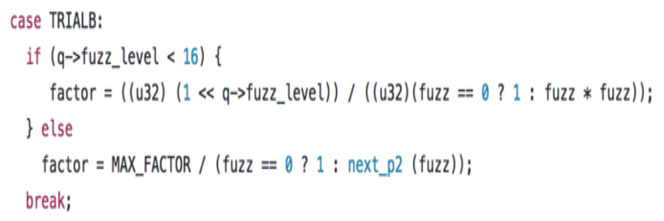
Fig. 1. Code for Triala

This is the first power schedule we designed to test and explore the code base of AFL-Fast. The key idea is derived from the proposed Markov Chain model and the optimizations performed by AFL-Fast [1] The increased efficiency of fast schedule comes from the assigned energy to each state, which equals to the number of inputs generated by fuzzing the seed t_i. The assigned energy should increase as the number of times (s(i)), t_i has previously been chosen, increases and decrease as the number of generated inputs (f(i), a measure of distribution density) which exercise path i, increases. The fast schedule set the energy p(i) proportional to $2^{s(i)}$ and inversely proportional to f(i). Our first Triala schedule set the energy proportional to $2^{[s(i)/f(i)]}$. The idea is that if the seed is in high density, the assigned energy will be low and around 4 since s(i) will be around 2*f(i). when the seed is in low density, the schedule will assign high energy to it since s(i)/f(i) will be high and $2^{[s(i)/f(i)]}$ will be even larger. As the analysis shows, this schedule resembles COE schedule except that it still assigns some energy to high-density seeds.

Below is the formula to calculate the energy of seed t_i:

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{\frac{s(i)}{f(i)}}, M\right)$$

B. TRIAL B



```
case TRIALB:
  if (q->fuzz_level < 16) {
    factor = ((u32) (1 << q->fuzz_level)) / ((u32)(fuzz == 0 ? 1 : fuzz * fuzz));
  } else
    factor = MAX_FACTOR / (fuzz == 0 ? 1 : next_p2 (fuzz));
  break;
```

Fig. 2. Code for Trialb

The design principle of Trialb schedule is that the distribution density (f(i)) of the seed is penalized by dividing the assigned energy by the square of f(i) instead of f(i). After we implemented Triala and tested, the schedule is not as fast as Fast schedule and the reason is speculated to be that the schedule decreased the energy for medium-density seeds too much. Instead of dividing f(i) inside the exponential term, the Trialb schedule divides f(i) twice, compared to Fast schedule.

Below is the formula to calculate the energy of seed t_i:

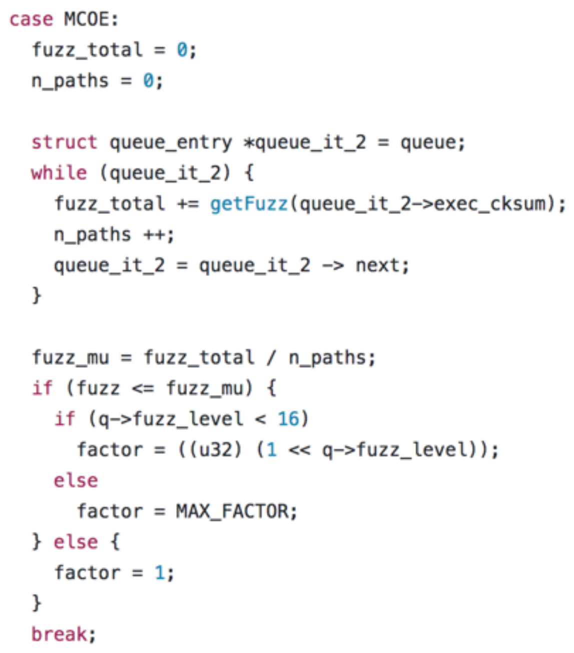
$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{[f(i)]^2}, M\right)$$

C. MCOE

The MCOE schedule is a modified cut-off exponential (COE) schedule. After carefully studying the implemented schedules in AFL Fast, we found that the Fast schedule and COE schedule are both exponential schedules and the performances are almost equally good at 24 hour run mark. However, COE schedule starts slower than Fast and COE schedule either assigns high energy for low-density seeds or 0 for high-density seeds. The idea behind MCOE schedule is that even if the high-density seed should be fuzzed less, they should not be assigned zero energy. There are still possibilities high-density seeds can produce new inputs that explore new paths, that is, not all inputs exploring new paths are generated from low-density seeds. Therefore, in the MCOE schedule, we still give the high-density seeds a chance (energy set to 1), although it is very low.

Below is the formula to calculate the energy of seed t_i:

$$p(i) = \begin{cases} 1, & if\ f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right), & otherwise \end{cases}$$



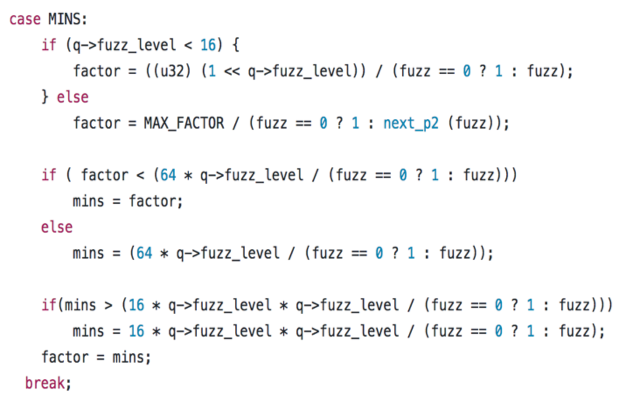
```
case MCOE:
  fuzz_total = 0;
  n_paths = 0;

  struct queue_entry *queue_it_2 = queue;
  while (queue_it_2) {
    fuzz_total += getFuzz(queue_it_2->exec_cksum);
    n_paths ++;
    queue_it_2 = queue_it_2 -> next;
  }

  fuzz_mu = fuzz_total / n_paths;
  if (fuzz <= fuzz_mu) {
    if (q->fuzz_level < 16)
      factor = ((u32) (1 << q->fuzz_level));
    else
      factor = MAX_FACTOR;
  } else {
    factor = 1;
  }
  break;
```

Fig. 3. Code for MCOE

D. MINS



```
case MINS:
    if (q->fuzz_level < 16) {
        factor = ((u32) (1 << q->fuzz_level)) / (fuzz == 0 ? 1 : fuzz);
    } else
        factor = MAX_FACTOR / (fuzz == 0 ? 1 : next_p2 (fuzz));

    if ( factor < (64 * q->fuzz_level / (fuzz == 0 ? 1 : fuzz)))
        mins = factor;
    else
        mins = (64 * q->fuzz_level / (fuzz == 0 ? 1 : fuzz));

    if(mins > (16 * q->fuzz_level * q->fuzz_level / (fuzz == 0 ? 1 : fuzz)))
        mins = 16 * q->fuzz_level * q->fuzz_level / (fuzz == 0 ? 1 : fuzz);
    factor = mins;
  break;
```

Fig. 4. Code for MINS

The MINS schedule is designed with the speculation that AFL Fast schedules are assigning too much energy to both high-density seeds and low-density seeds. Although the Fast schedules and other schedules perform much better than AFL fuzzing, the Fast schedule may not be optimal. In the design of Fast schedule, the key idea is to assign high energy (exponentially high energy) to low-density seeds and suppress the amount of fuzzing of high-density seeds. However, this introduces a potential problem that the energy assigned to low-density seeds may be too high and slows down the overall fuzzing. In this first hybrid schedule, it takes the minimum of calculated energy from Fast schedule, linear schedule, and quadratic schedule, for each seed. The naïve way of taking minimum will not work since linear schedule will always be smaller than the other two schedules for s(i)>1. To cope with this problem, the energy of linear schedule is multiplied with 64 and the energy of quadratic schedule is multiplied with 16 to compete with exponential schedule. In this way, the minimum of all three schedules are regularized.

Below is the formula to calculate the energy of seed t_i:

$$p(i) = \min\left\{\frac{\alpha(i)}{\beta}\cdot\frac{s(i)}{f(i)}\cdot 64, \frac{\alpha(i)}{\beta}\cdot\frac{[s(i)]^2}{f(i)}\cdot 16, \frac{\alpha(i)}{\beta}\cdot\frac{2^{s(i)}}{f(i)}, M\right\}$$

E. HYBRID

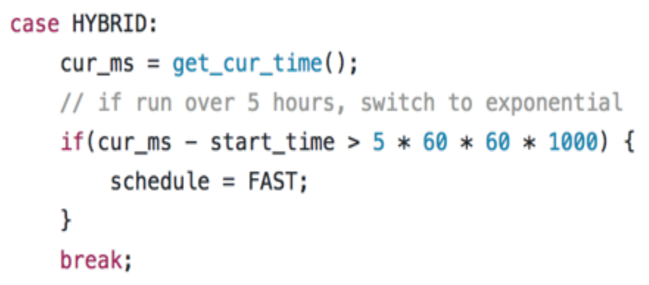

```
case HYBRID:
    cur_ms = get_cur_time();
    // if run over 5 hours, switch to exponential
    if(cur_ms - start_time > 5 * 60 * 60 * 1000) {
        schedule = FAST;
    }
    break;
```

Fig. 5. Code for Hybrid

The Hybrid power schedule is a hybrid schedule of Explore schedule and Fast schedule, which are described in details in the AFL Fast paper.[1] As is shown in the comparison of power schedules from the original paper, the Explore schedule starts out very fast exploring many paths through both high-density and low-density seeds, by assigning a constant low energy to all seeds. In the long run, however, the efficiency for finding the number of unique crashes drops and loses the lead to the Fast schedule. The underlying idea of Hybrid schedule is to combine the high efficiency of Explore schedule at the beginning and that of Fast schedule after a certain amount time. From the diagram of the total number of unique crashes over time for various schedules, the Explore curve and the Fast curve intersects at five-hour mark. Therefore, the Hybrid schedule utilizes the Explore schedule for the first five hours and switches to the Fast schedule afterwards.

Below is the formula to calculate the energy of seed t_i.

$$p(i) = \begin{cases} \frac{\alpha(i)}{\beta}, & if\ time < 5\ hours \\ \min\left(\frac{\alpha(i)}{\beta}\cdot\frac{2^{s(i)}}{f(i)}, M\right), & otherwise \end{cases}$$

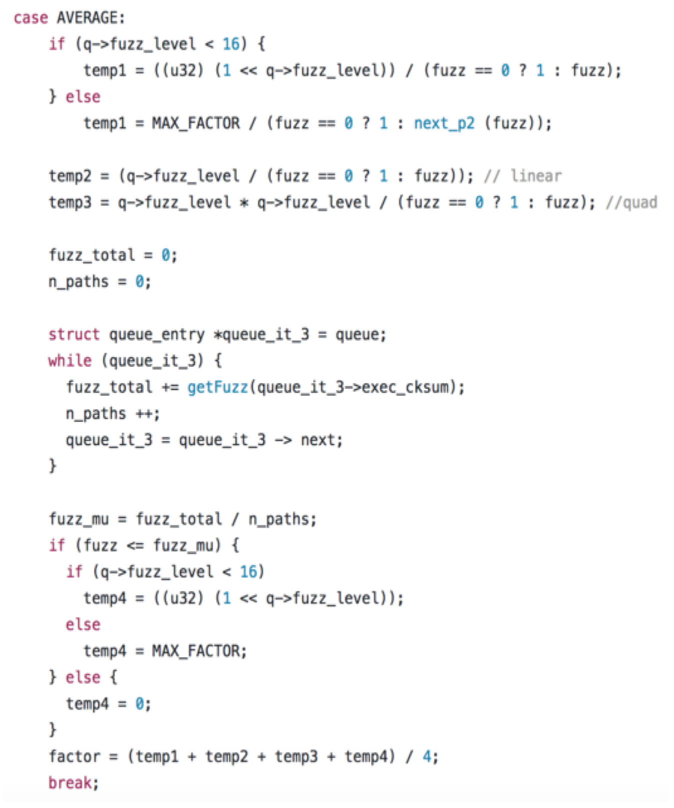
F. AVERAGE

Finally, we designed and implemented Average schedule, which incorporates all our knowledge about fuzzing, Markov Chain model, and power schedules. After careful analysis of the schedules and results of our new power schedules, we found the Fast schedule and Trialb schedule perform consistently than other schedules and sometimes Trialb runs faster than Fast schedule. With such a finding, we went back analyzing the proposed Markov Chain model and its probability distribution over the paths and states. We speculate that the Fast schedule assigns too much energy to the low-density seeds, which caused the Markov Model to overfit. This also explains the result that sometimes Trialb outperforms the Fast schedule. More detailed analysis is presented in the Results section. The idea used to design the Average schedule is that using the knowledge from all the relatively good schedules the Average schedule takes the average of their proposed energy. Each of the four schedules, COE, Fast, Linear, and Quadratic schedule, contributes equally and the Average schedule takes the mean of the four proposed energy, avoiding assigning too much (Fast) or too little (Linear) schedule. The formula is the following.

Below is the formula to calculate the energy of seed t_i.

$$avg = \begin{cases} 1, & if\ f(i) > \mu \\ \frac{1}{4}\cdot\left(\frac{\alpha(i)}{\beta}\cdot 2^{s(i)} + \frac{\alpha(i)}{\beta}\cdot\frac{2^{s(i)}}{f(i)} + \frac{\alpha(i)}{\beta}\cdot\frac{s(i)}{f(i)} + \frac{\alpha(i)}{\beta}\cdot\frac{[s(i)]^2}{f(i)}\right), & otherwise \end{cases}$$

$$p(i) = \min(avg, M)$$


```
case AVERAGE:
    if (q->fuzz_level < 16) {
        temp1 = ((u32) (1 << q->fuzz_level)) / (fuzz == 0 ? 1 : fuzz);
    } else
        temp1 = MAX_FACTOR / (fuzz == 0 ? 1 : next_p2 (fuzz));

    temp2 = (q->fuzz_level / (fuzz == 0 ? 1 : fuzz)); // linear
    temp3 = q->fuzz_level * q->fuzz_level / (fuzz == 0 ? 1 : fuzz); //quad

    fuzz_total = 0;
    n_paths = 0;

    struct queue_entry *queue_it_3 = queue;
    while (queue_it_3) {
      fuzz_total += getFuzz(queue_it_3->exec_cksum);
      n_paths ++;
      queue_it_3 = queue_it_3 -> next;
    }

    fuzz_mu = fuzz_total / n_paths;
    if (fuzz <= fuzz_mu) {
      if (q->fuzz_level < 16)
        temp4 = ((u32) (1 << q->fuzz_level));
      else
        temp4 = MAX_FACTOR;
    } else {
      temp4 = 0;
    }
    factor = (temp1 + temp2 + temp3 + temp4) / 4;
    break;
```

Fig. 6. Code for Average

III. Evaluation Of Test Cases

We evaluated our power schedules by running them on a set of open source programs as well our own written C programs. We categorized our test cases into three major buckets:

- *Fast Programs*: These included C programs which were supposed to report crashes when fuzzed for a span of 30 minutes.

- *Medium Programs:* These programs, based on our rationale, took more time than the fast programs and were run for 45 minutes, averages over three trials.
- *Slow Programs:* These programs comprised of the Gravity compiler, Cpython compiler and other C programs in which the crash would take place after extensive fuzzing was done. These were open source codes that had close to 3K lines of code and took most time for fuzzing. Such codes were fuzzed for around 10 hours and analyzed.

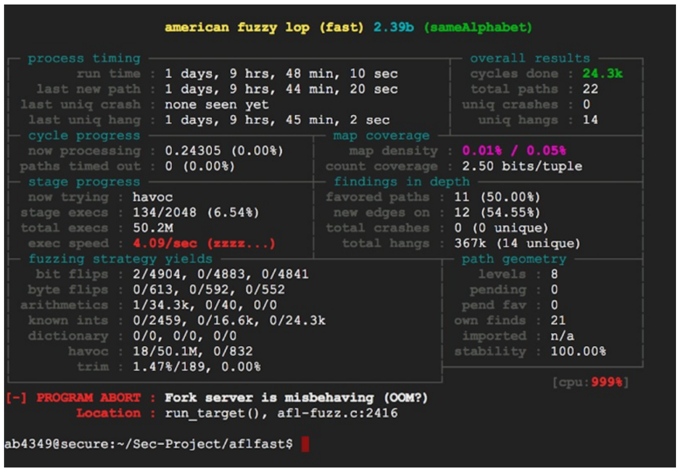
We used the below mentioned binaries/test cases:

- *Alphabet Occurences:* This test case fuzzes the seed to try reaching the target of simultaneous 1500 occurrences of alphabet 'a'. If the fuzz is successful, the program results in buffer overflow.
- *Recursive Function:* We made use of a recursive function to study the unique crashes and compare results for AFL, AFL Fast and the Average schedule.
- *Gravity Compiler:* We made use of Gravity which is a powerful dynamically typed, lightweight, embeddable programming language written in C. It is based on class based concurrent scripting language with modern Swift-like syntax. Gravity helps write easy portable code for the IOS and Android platforms. The code is around 3K lines.
- *Long Jump:* This test case fuzzes the initial seed introduces long jumps to cause the program to crash. Using long jumps, we were able to introduce more paths to our test case for our AFL to explore and crash accordingly.
- *Hit DEADBEEF:* This test case fuzzes the initial seed and results in a crash as soon as the program finds it way to DeadBeef. It is a medium running test case resulting in abort on reaching deadbeef.
- *CPython:* CPython is the original Python implementation. The python language has been developed in C. So we tried to fuzz the python compiler by providing a sample input program to check if we are able to find any vulnerability.

IV. Experimentation Findings

A. *ALPHABET OCCURENCES*

The alphabet occurrences program leads to a buffer overflow when it finds continuous string of a's of length 1500. As we know every character can take any one of the 256 ASCII characters. So in this case for the 1500 character long string AFL needs to mutate the input such that it has alphabet 'a' out of the 256 ASCII characters at all the 1500 characters. This leads to a branching factor of $256^{1500} = 2^{12000}$. This means that mutation of the seed input has to go deep inside a single path out of the so many branches. We ran AFL Fast and the Average schedule on this program. The Average mode could find a crash while the AFL Fast was not able to able to find a crash during the same fuzzing duration. The results of the fuzzing experiment for AFL Fast and AFL Fast Average have been shown in Fig. 7 and Fig. 8 respectively.



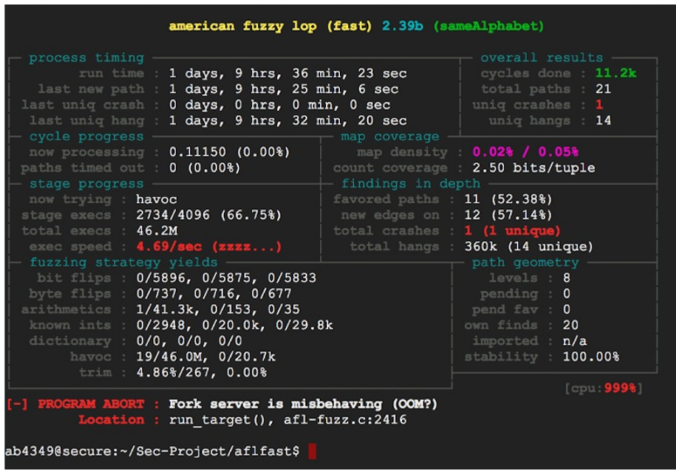
```
                 american fuzzy lop (fast) 2.39b (sameAlphabet)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 9 hrs, 48 min, 10 sec      │  cycles done : 24.3k  │
│   last new path : 1 days, 9 hrs, 44 min, 20 sec      │  total paths : 22     │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 1 days, 9 hrs, 45 min, 2 sec       │   uniq hangs : 14     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0.24305 (0.00%)   │    map density : 0.01% / 0.05%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.50 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 11 (50.00%)            │
│ stage execs : 134/2048 (6.54%)      │  new edges on : 12 (54.55%)            │
│ total execs : 50.2M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 4.09/sec (zzzz...)    │   total hangs : 367k (14 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 2/4904, 0/4883, 0/4841                │    levels : 8          │
│  byte flips : 0/613, 0/592, 0/552                   │   pending : 0          │
│ arithmetics : 1/34.3k, 0/40, 0/0                    │  pend fav : 0          │
│  known ints : 0/2459, 0/16.6k, 0/24.3k              │ own finds : 21         │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 18/50.1M, 0/832                       │ stability : 100.00%    │
│        trim : 1.47%/189, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu:999%]

[-] PROGRAM ABORT : Fork server is misbehaving (OOM?)
         Location : run_target(), afl-fuzz.c:2416

ab4349@secure:~/Sec-Project/aflfast$
```

Fig. 7. Output for AFL Fast



```
                 american fuzzy lop (fast) 2.39b (sameAlphabet)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 9 hrs, 36 min, 23 sec      │  cycles done : 11.2k  │
│   last new path : 1 days, 9 hrs, 25 min, 6 sec       │  total paths : 21     │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec        │ uniq crashes : 1      │
│  last uniq hang : 1 days, 9 hrs, 32 min, 20 sec      │   uniq hangs : 14     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0.11150 (0.00%)   │    map density : 0.02% / 0.05%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.50 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 11 (52.38%)            │
│ stage execs : 2734/4096 (66.75%)    │  new edges on : 12 (57.14%)            │
│ total execs : 46.2M                 │ total crashes : 1 (1 unique)           │
│  exec speed : 4.69/sec (zzzz...)    │   total hangs : 360k (14 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/5896, 0/5875, 0/5833                │    levels : 8          │
│  byte flips : 0/737, 0/716, 0/677                   │   pending : 0          │
│ arithmetics : 1/41.3k, 0/153, 0/35                  │  pend fav : 0          │
│  known ints : 0/2948, 0/20.0k, 0/29.8k              │ own finds : 20         │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 19/46.0M, 0/20.7k                     │ stability : 100.00%    │
│        trim : 4.86%/267, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu:999%]

[-] PROGRAM ABORT : Fork server is misbehaving (OOM?)
         Location : run_target(), afl-fuzz.c:2416

ab4349@secure:~/Sec-Project/aflfast$
```

Fig. 8. Output for AFL Fast Average

B. *BUFFER OVERFLOW FUNCTIONS*

We first tried to run a program that has a buffer overflow problem, to perform our experiment on the three schedules.

Fig. 9 shows the number of unique crashes they find and the graph in Fig. 10 shows the number of total paths they explore. The result adheres to our speculation that average may be slower in finding crashes at the initial stage, but will outperform AFL after it discovers more new paths. According to the total number of paths showing in the second graph, the AFL fast and AFL overlap each other, which means they find the same number of paths. AFL fast and AFL reach a point and stay at that level for the rest of the running time. However, the Average schedule explores much more paths than the other two modes and keeps on exploring further while the other two have reached a stable stage. At later time, the average is at least as good as AFL, and at the end it finds more crashes than AFL.

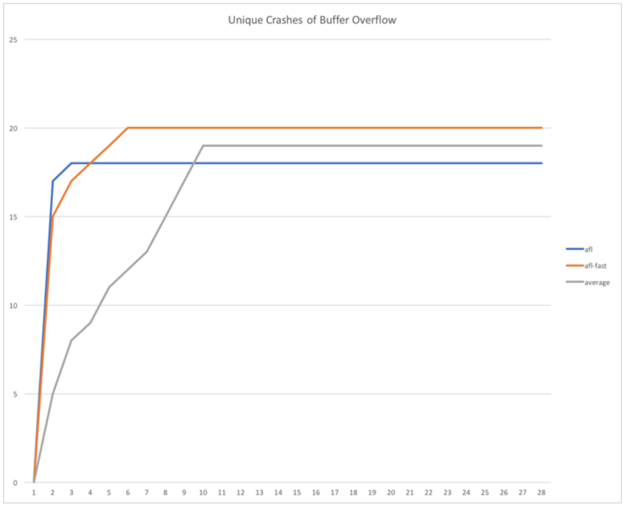


Fig. 9. Unique Crashes for Buffer Overflow

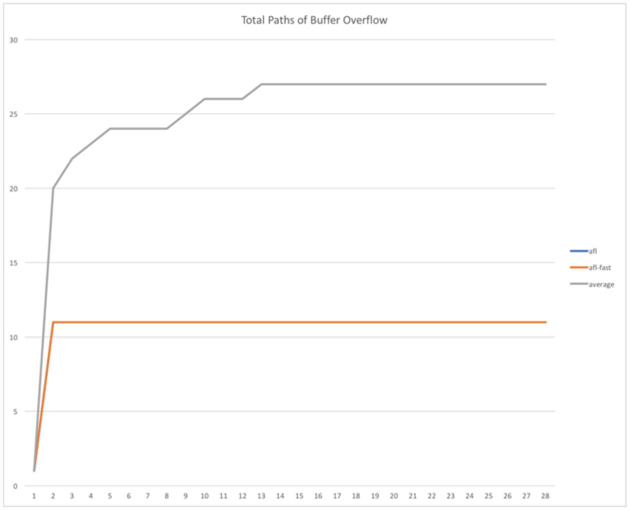


Fig. 10. Total Paths for Buffer Overflow

C. RECURSIVE FUNCTIONS

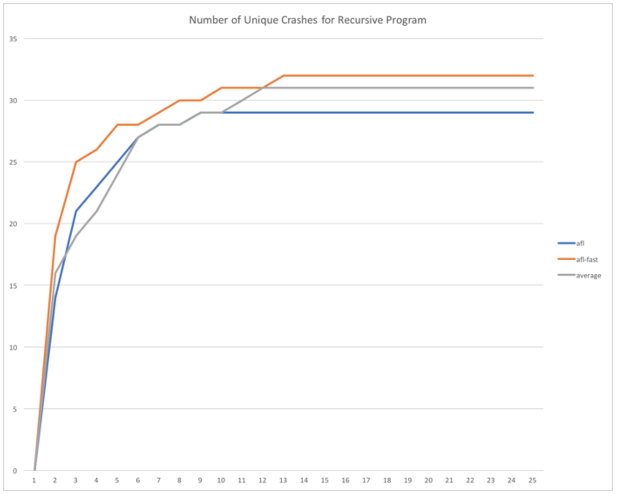


Fig. 11. Unique Crashes for Recursive Functions

The graph in Fig. 11 indicates the number of unique crashes found on fuzzing a recursive c function. We did not include a graph with the number of paths for three schedules, since the main part of a CFG graph for a recursive function is essentially only one path. The program enters the body and exits at the end of the recursive function, and re-enters the looping path if the function is called again. Hence, their average number of paths are the same (only 1 path). Although the number of paths is 1, the number of unique crashes differ as it reaches crashes with different inputs and different number of calls of a recursive function. In this case, the Average schedule requires more time to explore the crashes, but it outperforms AFL after 1/3 of the running time, and generates the same number of crashes like AFL Fast. We could conclude that our Average schedule may find more crashes than AFL and be as good as AFL fast for recursive programs.

D. LONG JUMP TEST CASE:

The graphs in Fig. 12 and Fig. 13 indicate the number of unique crashes and the number of paths respectively for a program using set jump (setjmp) and long jump (longjmp).

It can be seen that AFL Fast has a better performance than average and AFL at the initial stage, but our schedule (Average) eventually caught up and found more unique crashes than AFL Fast. The Average schedule needs more time to explore paths than AFL Fast and will adjust its way of choosing the next input seed in order to find more crashes in later runs. After certain amount of time, AFL Fast reaches a stable number of paths and is not able to explore further new paths for a long time, but the Average schedule keeps on finding new paths in a short amount of time and increases the number of total paths for the whole run. This helps the Average schedule to find new crashes, as it indeed keeps on exploring new paths at a comparable stable speed. We ran the program three times and used its average data to plot the graph, and the result was found to be consistent. So we can conclude that for programs that use operations such as setjmp and longjmp, Average is a good schedule to detect bugs, as its overall performance is better than AFL and equally good as AFL Fast schedule.

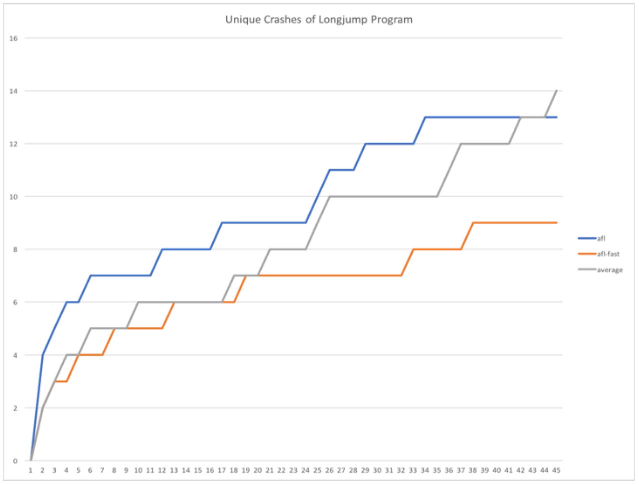


Fig. 12. Unique Crashes for Long Jump Program

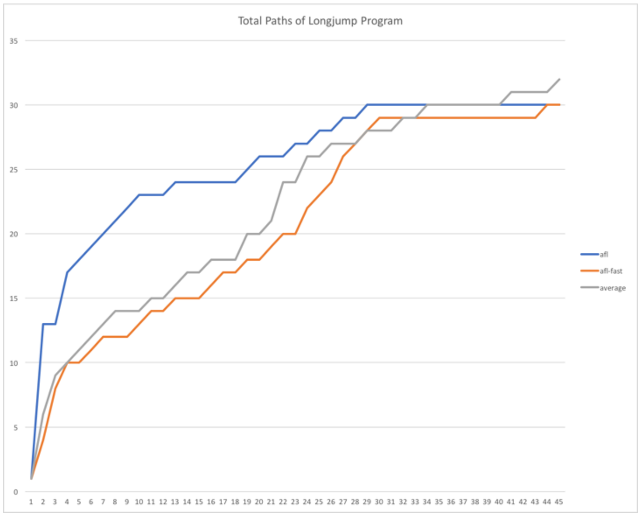


Fig. 13. Total Paths for Long Jump Program

E. GRAVITY COMPILER

Running Gravity Compiler gave interesting results. As can be seen from Fig. 14, the number of unique crashes over the time for the eight different schedules, the average schedule clearly outperforms all other schedules, including the fast schedule. Its fuzzing efficiency starts to become better after 2.5 hours of fuzzing. In the end, it finds 50% more unique crashes than the fast schedule. One thing to note from the graph is that the Trialb schedule actually performs better than the fast schedule and is the second best schedule in terms of finding the number of unique crashes over a fuzzing period of 10 hours. This corroborates our analytical proposal that the fast schedule does assign too much energy to the low-density seeds and the Markov Model proposed by AFL Fast team is overfitting.

In Fig. 15 the number of total explored paths versus time, all the eight schedules are shown to be active in exploring paths, with Average and Trialb schedules performing better than the rest of the schedules.

From Fig. 16, we can see that the total number of unique hangs follow the same pattern and the results in these three graphs support each other. It is important to point out that all the result findings are average of three ten-hour runs to minimize inherent randomness of fuzzing. Another interesting point to note is that the difference between AFL Fast schedule and AFL native schedule (Exploit) is not as large as the one shown in the AFL Fast paper.[1] In our fuzzing result of gravity C compiler, the Fast schedule is more efficient than AFL native schedule and finds roughly 30% more unique crashes, which are not as drastic as the one (10X fold more crashes) shown in the AFL Fast paper. But this could be due to the specific gravity compiler program, on which the Fast schedule is not able to outperform the AFL native schedule that much.

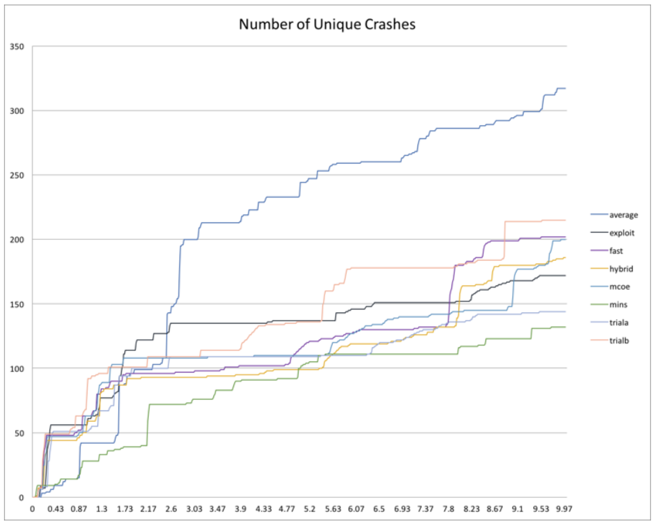


Fig. 14. Unique Crashes for Gravity Compiler

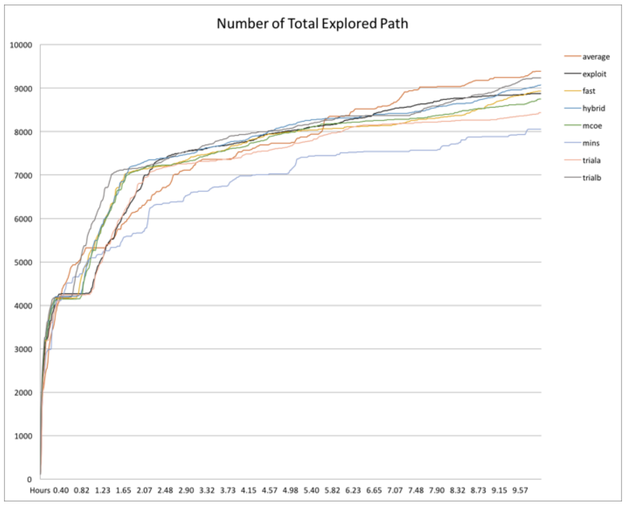


Fig. 15. Total Paths for Gravity Compiler

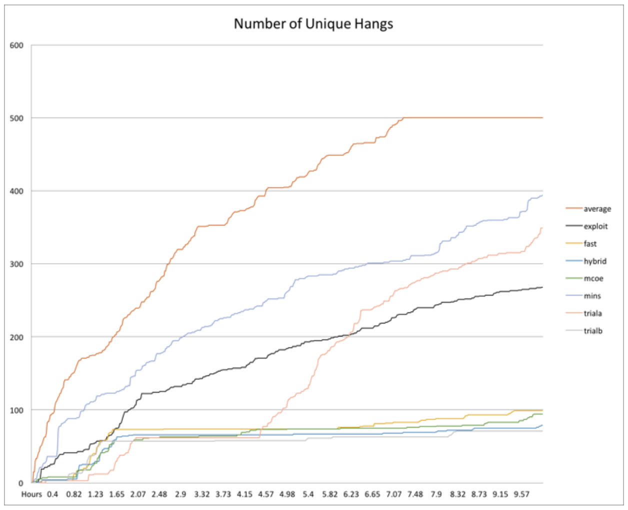


Fig. 16. Unique Hangs for Gravity Compiler

F. DEADBEEF TEST CASE:

The program DeadBeef runs and aborts if the seed fuzzes and reaches the string deadbeef. We ran the mentioned program with three fuzzers namely, AFL, AFL Fast and the Average schedule. The Average schedule was found to better in terms of performance than the other two.

With AFL and AFL Fast, the program was run averaged over three trials, and it took around the same time of 45 minutes to get the crash. With AFL Fast the same thing achieved the crash in 45 minutes. But with the Average schedule the first crash came in 23 minutes. There is no graph corresponding to the number of crashes found as for this program the same crash that was expected. The time need to find the crash by all three schedules have been reported in Table 1.

TABLE I. TIME TAKEN TO FIND CRASH

AFL	1 hours 5 minutes
AFL Fast	42 minutes
Average Schedule	45 minutes

G. CPYTHON TEST CASE:

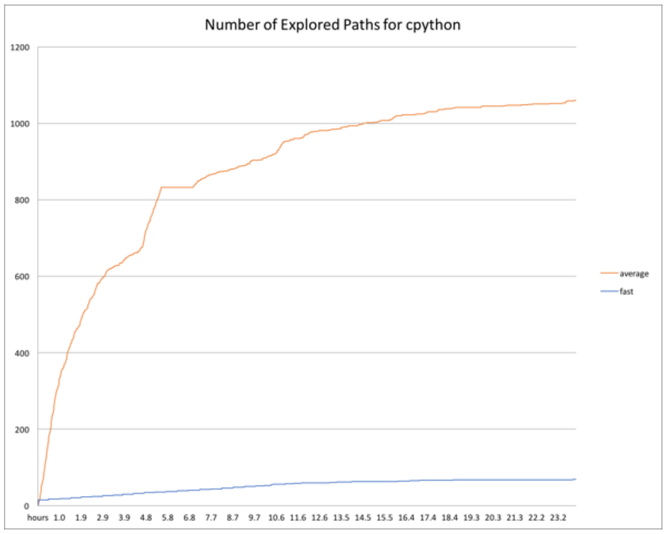


Fig. 17. Paths Explored for CPython

CPython is the original Python implementation of the python language which has been developed in C. So we tried to fuzz the python compiler by providing a sample input program to check if we are able to find any vulnerability. We ran both AFL Fast and AFL Fast Average on the cpython compiler. Even though neither of the two is able to find a single crash in cpython using the sample initial input but the number of paths explored by AFL Fast Average was higher in comparison to the paths explored by AFL Fast. This shows that the Average schedule tries to explore more paths by assigning energy to the seed inputs in a better way. Fig. 17 shows the graph of the number of paths explored by the two schedules over a period of 24 hours.

V. ANALYSIS

From the above analysis of the data generated after fuzzing different programs with different schedules, we arrived at the following analytical results about the eight power schedules. The native AFL schedule (Exploit) is slow at some test cases but overall performs in a stable manner. The AFL Fast schedule designed and implemented by AFL Fast team is fast and highly efficient at finding unique crashes and exploring new paths. However, according to the proposed Markov Chain model, it assigns higher energy to the seeds in the low distribution-density paths, that is, it overfits the input seeds and is not at its optimal point. This means that AFL Fast schedule spends too much time fuzzing each low-density seed before moving on to the next one, hence, decreasing the probability of finding a new path over a unit time.

We speculate that the optimal point of fastest speed in finding unique crashes and new paths is between AFL native schedule and the AFL Fast schedule. This idea is corroborated by the fact that Trialb and Average schedule, both of which slightly lower the assigned energy to low-density seeds, performed better than the fast schedule for long duration fuzzing. Unfortunately, the Triala and MINS schedules do not perform as good as AFL native schedule, which is possibly because that both, Triala and MINS schedules lower the assigned energy too much. The Hybrid schedule is able to catch up with the Fast schedule after nine hours but fails to surpass its efficiency at the beginning, which does not meet our expectation.

The Average and Trialb schedules are designed successfully and run faster and more efficient than AFL Fast schedule. The Average and Trialb schedules slightly reduce the assigned energy for both high-density and low-density seeds from the Fast schedule and therefore moves towards the optimal point.

VI. RELATED WORK

Several techniques have been used to improve the efficiency of AFL for automated fuzzing. A wiser seed selection can improve the performance of AFL and AFL Fast tremendously. We leverage the smart seed selection using the concept of power schedules. Since we use the power schedules that give higher weight to low-frequency paths, our approach is more inclined to find unexplored paths and crashes. Operators that perform better in previous fuzzing iterations are chosen with higher probability during fuzzing. Our approach builds upon the Markov Chain model used by AFL Fast to control the time spent on fuzzing an input seed and to select the next input seed to be fuzzed for further analysis.

VII. Conclusion

AFL is a state of the art greybox fuzzer that starts from a random input seed and fuzzes in the direction based on new paths discovered and logs the path identified on the way. AFL Fast schedule exposes much more unique crashes and new paths than the native AFL schedule. Our optimized Average power schedule further enhances the efficiency of AFL Fast and finds close to twice the number of unique crashes explored by AFL Fast. The Average schedule is efficient at exploring new paths at an increased speed than AFL Fast.

In conclusion, the Average schedule exposes substantially more number of bugs faster than AFL Fast. We designed and implemented six new power schedules and the insights behind the design. We tested them against various programs and evaluated their performances in comparison to each other as well as AFL and AFL Fast. A detailed analysis of the six newly designed power schedules is completed. From both the data and our analysis, the Average power schedule performs the best out of the eight total power schedules. We believe the use of the Average schedule will explore many previously unexplored crashes of the C compiled programs within a short span of time.

Acknowledgment

We would like to thank Professor Suman Jana and the Teaching Assistants, Eugene Ang and Plaban Mohanty, for their constant support and encouragement towards the idea and guiding us to bring our thoughts to execution.

References

[1] Marcel Bohme, Van-thuan Pham, Abhik RoyChoudhury , Coverage-based Greybox Fuzzing as Markov Chain. School of Computing, National University of Singapore, Singapore.

[2] M. B¨ohme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 334–344, 2013.

[3] M. B¨ohme and S. Paul. A probabilistic analysis of the efficiency of automated software testing. IEEE Transactions on Software Engineering, 42(4):345–360, April 2016.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI '08, pages 209–224, 2008.

[5] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 474–484, 2009. [10] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 166–176, 2012.

[6] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. Queue, 10(1):20:20–20:27, Jan. 2012.

[7] J. R. Norris. Markov Chains (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge University Press, July 1998.