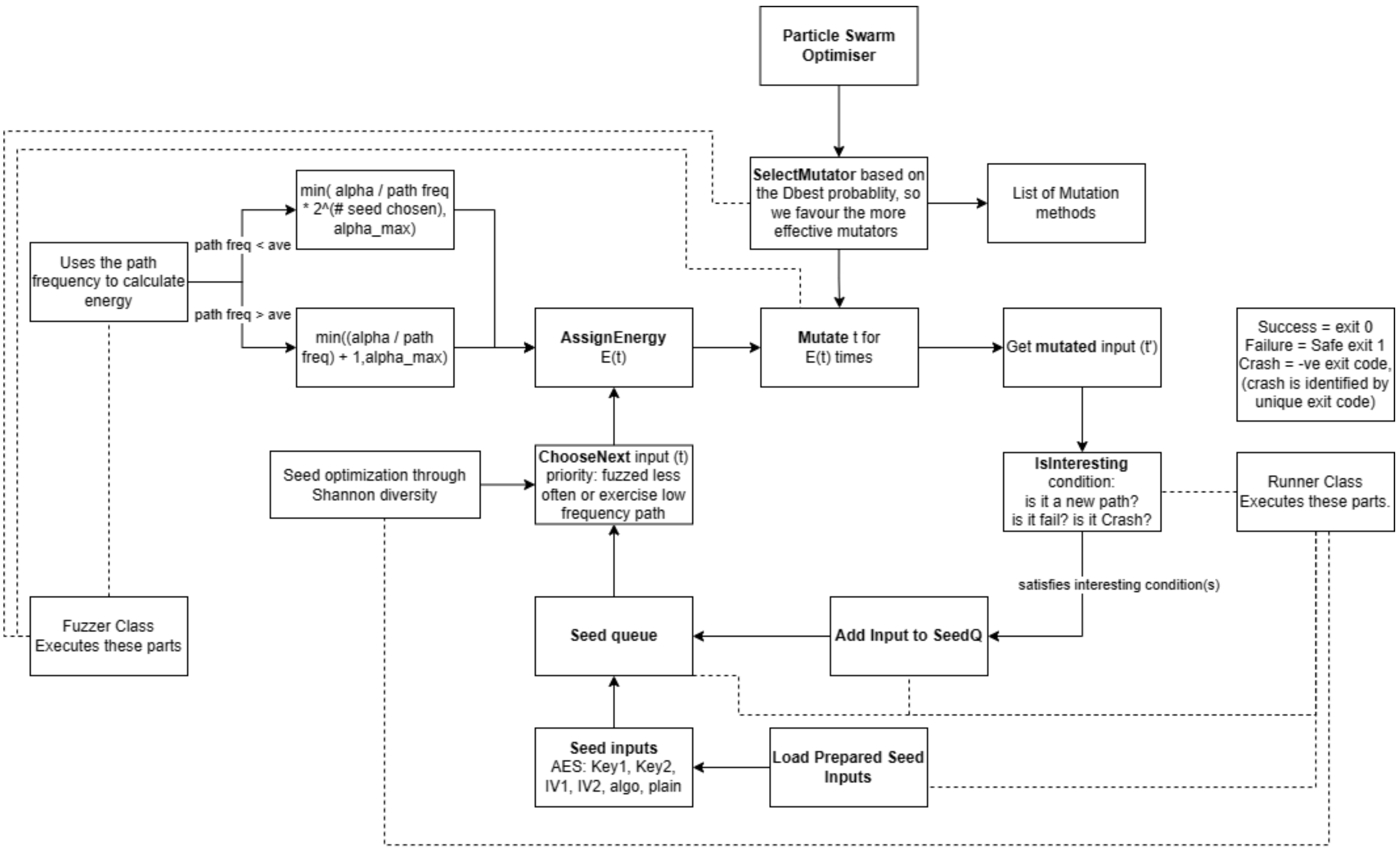
Final Report
Group 3: John, Jayati, Eunice, Brenda, Cheng Xin

Design

The overall design of our fuzzer follows the afl fuzzing loop, where we repeat for a given energy and selected input, fuzzing the input through mutations and comparing outputs to find interesting/unique paths to add to the seed queue. These are the important parts of our implementation:

- assignEnergy
- chooseNext seed
- seedOptimisation
- Mutators
- PSO - mutator optimisation



Particle Swarm Optimiser
SelectMutator based on the Dbest probablity, so we favour the more effective mutators
List of Mutation methods
Uses the path frequency to calculate energy
path freq < ave
path freq > ave
min(alpha / path freq * 2^(# seed chosen), alpha_max)
min((alpha / path freq) + 1,alpha_max)
AssignEnergy E(t)
Mutate t for E(t) times
Get mutated input (t')
Success = exit 0 Failure = Safe exit 1 Crash = -ve exit code, (crash is identified by unique exit code)
Seed optimization through Shannon diversity
ChooseNext input (t) priority: fuzzed less often or exercise low frequency path
IsInteresting condition: is it a new path? is it fail? is it Crash?
Runner Class Executes these parts.
satisfies interesting condition(s)
Fuzzer Class Executes these parts
Seed queue
Add Input to SeedQ
Seed inputs AES: Key1, Key2, IV1, IV2, algo, plain
Load Prepared Seed Inputs

Figure 1: Fuzzer Implementation

Implementation

The overall fuzzer implementation is summarised in Figure 1 above.

There are the important parts of our implementation:

- assignEnergy
- chooseNext seed
- seedOptimisation
- Mutators
- PSO - mutator optimisation

Target

One of the challenges faced during implementation was how to interact with mbedtls and run it. We overcame this challenge by researching and looking up mbedtls documentation and choosing our targets to be AES. Symmetric encryption methods in the AES library of mbedtls were implemented. As such 4 algorithms were tested: CBC, ECB, CFB128, CTR

Inputs to fuzz:

- Plaintext
- Encrypting(All), Decrypting Key (CBC, ECB)
- Initiation Vectors

The inputs were prepared in a txt file.

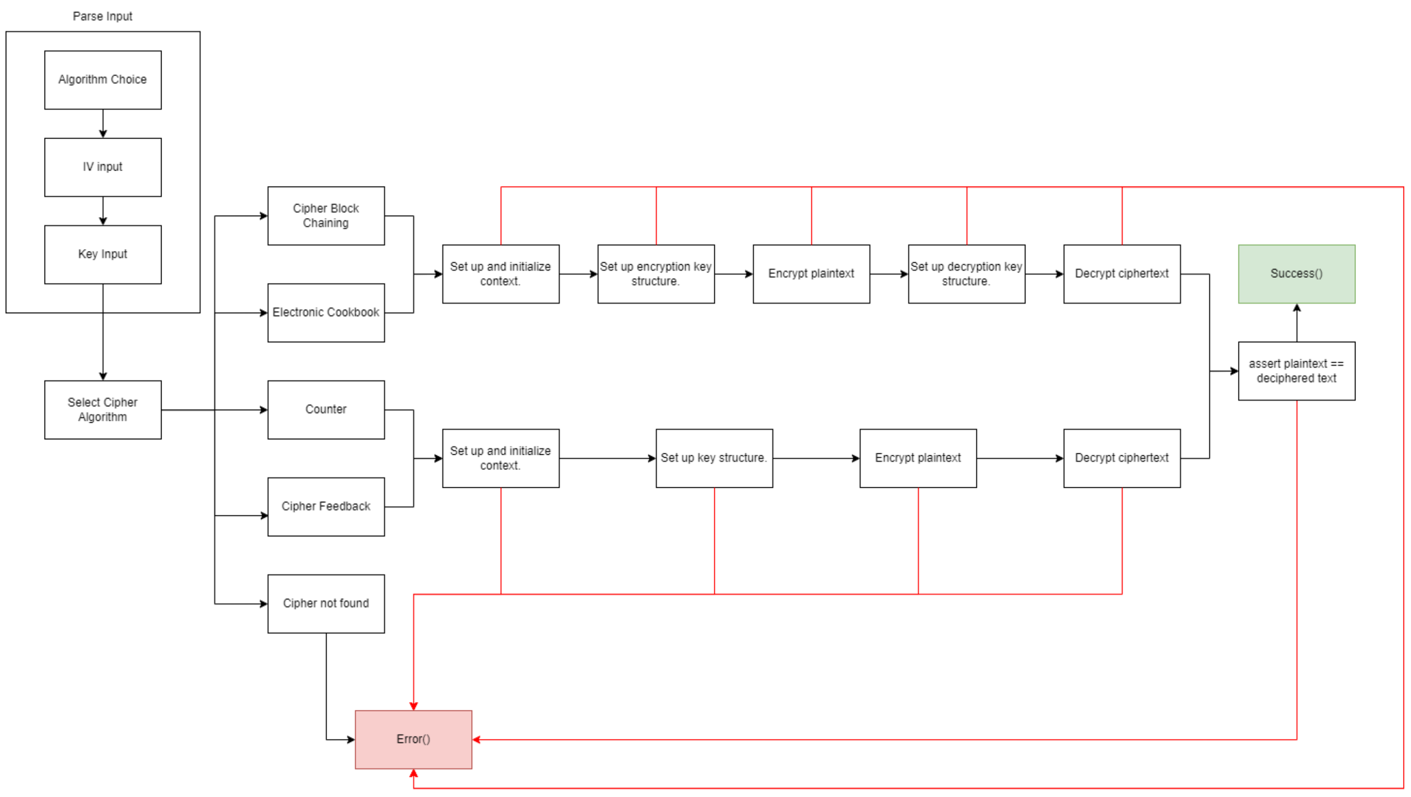


Figure 2: Mbedtls AES Components

Gcov was used to check the coverage of the test driver and code. More than 2000 lines of code have been analysed. We used python programming language to code the mutators, etc.

Assign Energy

Inspired from the AFL Fast we decided to use a similar algorithm and logic to assign energy.

```
if $f(i) > mean$
    $E(t_i) = 0$
if $f(i) <= mean$
    $E(t_i) = \min(\alpha(i)/\rho * 2^{s(i)}, M)$
```

- M = 2000
- alpha(i) = 15 (initial value)
- p = number of times path executed / total path iterations
- s(i) = number of times given seed is selected
- Initially we grew the energy exponentially. But once a path is chosen a greater number of times i.e a high frequency path, the path is no longer assigned any energy
- Energy assignment is dynamic based on the mean number of times a seed is executed, which changes throughout the iterations so each seed may have different energy assignment throughout the fuzzer iterations
- Our implementation If f(i) > mean : alpha(i)/f(i) +1
- Design decision we took was to add 1 to avoid any seeds with 0 energy.
- Our design of assigning energy is inspired by AFL Fast.

isInteresting

The aim of isInteresting function is to find unique paths that could be explored given a seed. We used the gcov tool in order to find the most interesting paths for our test driver analysis.

- gcov is used to analyse code coverage for each new path explored.
- The gcov file was parsed into a dictionary mapping lines of code to frequency of execution
- A path is considered interesting if it has executed new lines or executed existing lines a different number of times than any previous execution.
- Challenge and Future Scope: We did not use bucketing as the number of times each line is executed is less than 20. Majority of lines are executed in the same bucket.

PSO

The PSO is run on swarms representing categorical probabilities of each possible mutator function. The swarms are initialised with each value randomly chosen from 0-100 (however, when using it with the fuzzer, we normalise the values so they add to 1). At each iteration, a fuzzing cycle is run for each swarm, returning a dictionary containing how many interesting paths each function found. The fitness function is then defined using the Shannon Diversity Index (used also in Seed Optimisation:

$$H = -\sum_{j=1}^{S} p_i \ln p_i$$

Where p is defined as the total interesting paths for that function / total interesting paths. Since the fuzzer is not very fast, the PSO runs for a fixed amount of iterations instead of running until convergence. After the PSO ends, it returns the optimal categorical probability to use for the mutator selector function in the fuzzer.

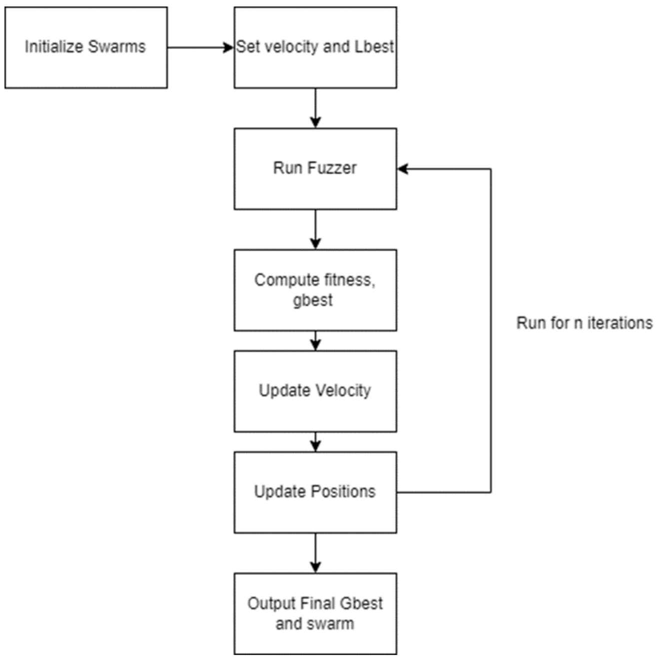


Figure 3: Overall PSO Structure

Mutator Methods

The following mutator methods were used. From our seed file, we randomly choose one of the inputs from

- Plaintext
- Encrypting (All), Decrypting Key (CBC, ECB)
- Initiation Vectors

Note for bit based mutation, the ascii value of the character was used to increment/decrement byte and then replace the randomly chosen character with the specified character with higher/lower ASCII value. (The flowchart is a guide to names of the functions to refer to in our fuzzer implementation)

To implement Encryption Algorithm Mutation, we randomly choose from a list of working algorithms or a ‘fail’ case where we aim to fuzz and run the driver without a valid algorithm employed through an empty list.

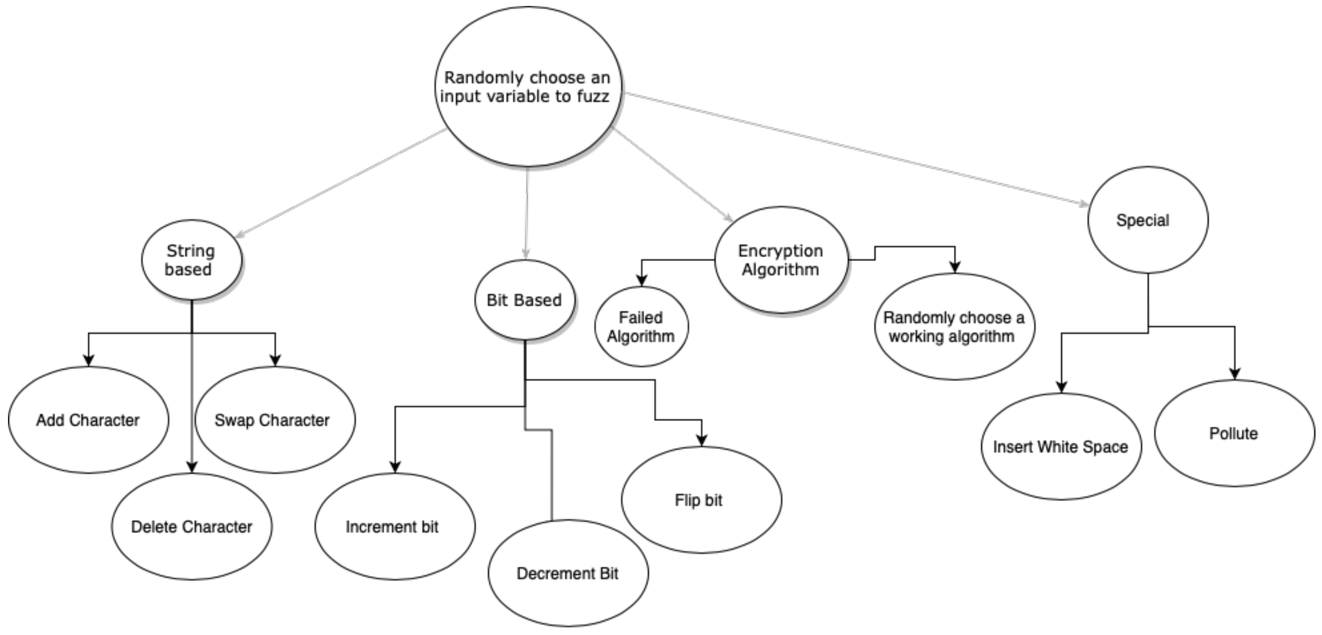


Figure 4: Types of Mutations

Seed optimisation

Multiple algorithms were tried and tested in order to implement a method to optimise seed selection. These included choosing seeds with the most coverage or the least coverage in the gcov analysis, random set, hotset etc. In the end, one of the best seed optimisations we could build was inspired from Shannon Diversity. Shannon Diversity measures diversity of species in a community. Using this concept, we aimed to choose the seeds with the most diverse paths.

Following are the steps for calculation:

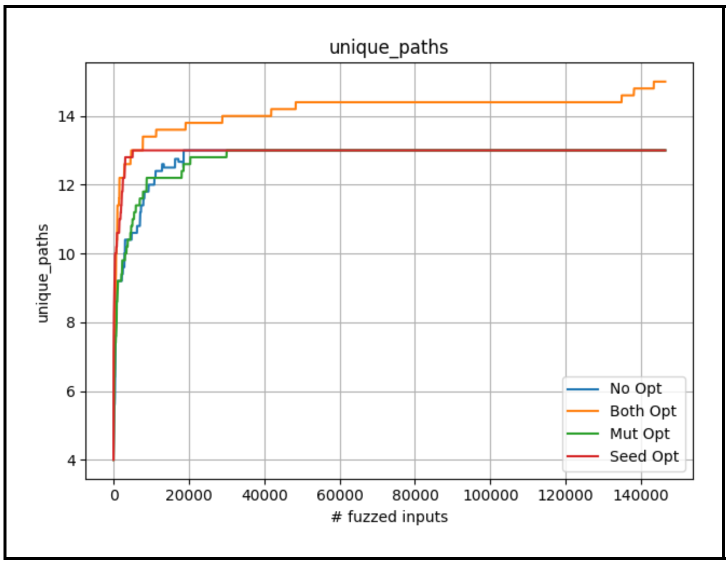
$$H = -\sum_{j=1}^{S} p_i \ln p_i$$

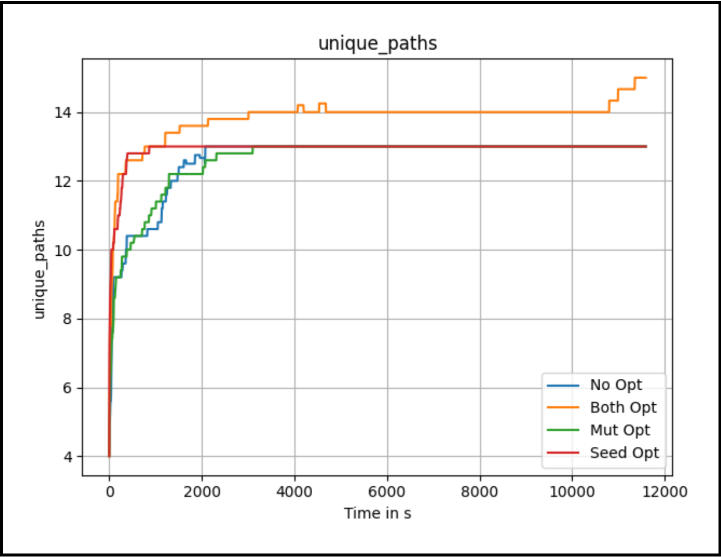
- Probability Score is calculated as a ratio based on the frequency of interesting paths / total interesting paths
- With current probability distribution, we calculate the Shannon diversity score
- For each seed, we increase its frequency of interesting paths by one, and compute the new Shannon Diversity.
- The seed with the greatest increase in Shannon Diversity is chosen.
- Balancing paths with higher probability and exploring new seeds in the queue.

Experimentation

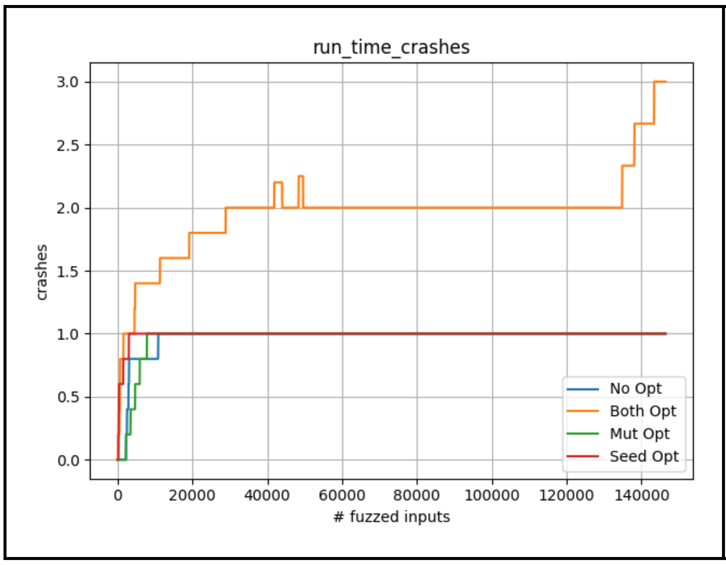
Effectiveness

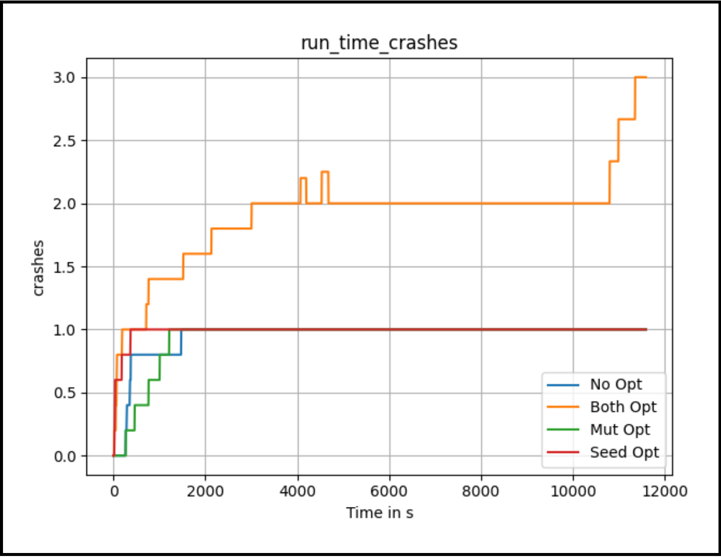
Across the iterations or time, we measured effectiveness based on three values, unique paths found, failure, runtime crashes. We observe that having both our mutator and seed selection optimisation was able to find two more unique paths compared to those with only one or no optimization.





These new paths were runtime crashes. According to the logger, they are segmentation faults, occurring when trying to access freed variables. So, it appears that our optimisations have improved the overall effectiveness in finding bugs.



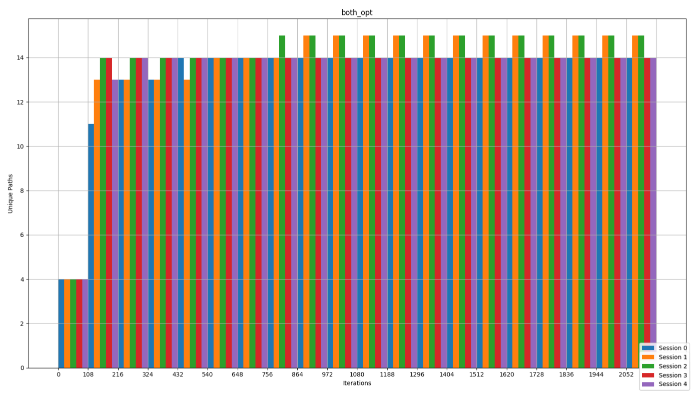


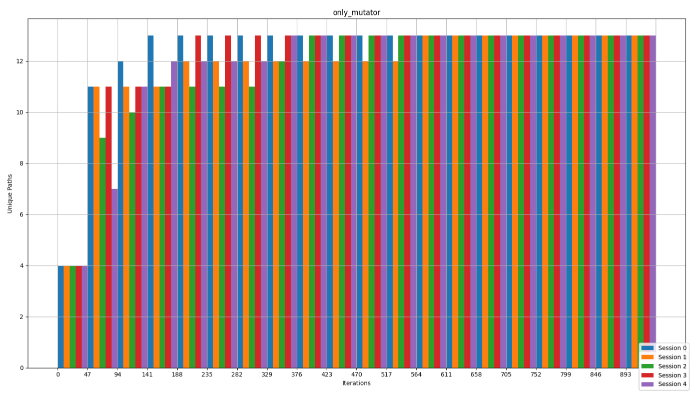
Efficiency

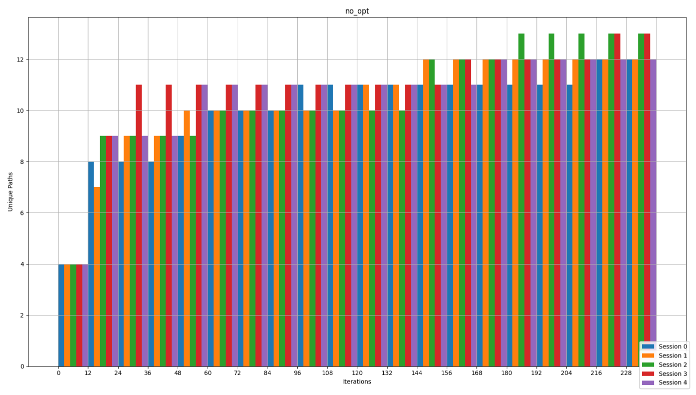
	Both Optimisation
Time (s) to first crash	135.00s
Iterations / Time (s)	9.62/s
Time taken (s) per fuzz iteration Average of 1000 calls of fuzzInput()	0.131s

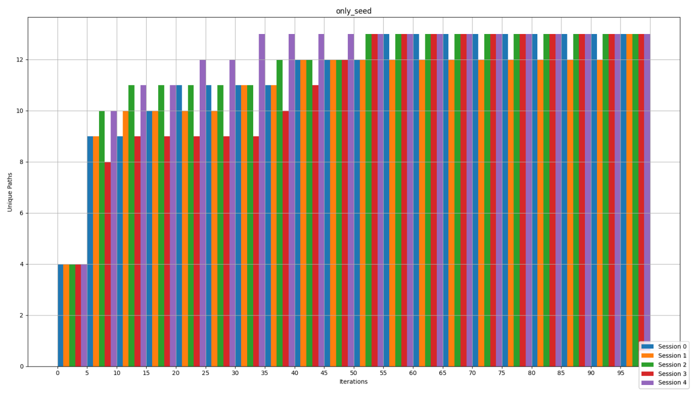
Stability

We observe that the values converge to a common endpoint across the different sessions in each campaign. It can be said the results are stable.









Limitations/Reflection

The main challenge is with the quality of the pso results. Due to the time taken to run the pso and the fuzzer, we only had about three opportunities to test parameters. There are several parameters:

- Energy
- Population size
- Number of generations

In order to reliably test the parameters, we need to run a large PSO population and then run one fuzzer campaign to find out its suitability. On reflection, we need to work on optimising the speed of the pso fuzzer in order to be able to run more of these tests efficiently.

This optimization run we had 30 swarms for each mutator and had a generation of 5. Generation means the number of times we select a new input from seedQ. So, we had a total of 30*8 (# of mutators) swarms that we track. We felt comfortable with it as it had the most stable results across the different campaigns we ran. This was from our testing from checkoff 3 and checkoff 4.

Github repo: johnlim2019/mbedtls-afl (github.com)