Knowledge Search & Extraction
Project 02: Python Test Generator

Fabian Gobet

December 12, 2024

Repo: https://github.com/kamclassroom2022/p2-python-test-generator-FabianGobet
Commit: e547054

Section 1 - Instrumentation

The following section presents an overview of the code's structure by detailing the number of files, the functions contained within each file, and the number of comparisons performed by these functions. This breakdown is summarized in the accompanying table, which highlights the key metrics for each file in the codebase.

File Name	# Functions	Function Name	# Compares	total # compares
`anagram_check.py`	1	anagram_check	4	4
`caesar_cipher.py`	2	encrypt decrypt	1 1	2
`check_armstrong.py`	1	check_armstrong	5	5
`common_divisor_count.py`	1	cd_count	7	7
`exponentiation.py`	1	exponentiation	4	4
`gcd.py`	1	gcd	5	5
`longest_substring.py`	1	longest_sorted_substr	2	2
`rabin_karp.py`	1	rabin_karp_search	5	5
`railfence_cipher.py`	2	railencrypt raildecrypt	4 8	12
`zellers_birthday.py`	1	zeller	8	8

Table 1: Functions and number of compares per function, per file.

Section 2: Fuzzer test generator

Overview of Supporting Classes for Fuzzer Test Generator

In order to explain how the Fuzzer test generator works, we first need to describe four supporting classes: `TestCase`, `Archive`, `EventGenerator`, and `TestPool`.

TestCase (archive_testcase.py)

The `TestCase` class is designed to represent a single test case for a function. It stores information about the arguments of the test case, including their types, names, and values, as well as the expected or actual output of the function being tested. This class has attributes to handle these details: `arg_types` keeps track of the types of the input arguments, `arg_names` holds the names of the arguments, and `arg_values` stores the actual input values. It also includes an `output` attribute for storing the test case result and an `input_size` attribute for tracking the number of arguments.

The class provides methods to retrieve argument types, names, and values, and to set or get the test case output. It ensures consistent formatting of input values as tuples when necessary. Additionally, it includes functionality to compare two `TestCase` objects for equality by comparing their arguments and types. Overall, the `TestCase` class encapsulates all relevant details of a test case and supports comparison, storage, and retrieval of test data.

Archive (archive_testcase.py)

The `Archive` class manages and tracks information about functions, their branch coverage, and associated test cases during the testing process. It maintains data about the tested functions, including branch structures, test cases, coverage status, and evaluation results. Attributes include mappings of function names to the number of branches, distances of current true and false branches, and lists of test cases. It also tracks disregarded test cases (e.g., those causing exceptions) and whether a function achieves full branch coverage. A reference to the script's global namespace allows dynamic execution of functions.

The `add_fn` method adds a function to the archive and initializes its tracking structures, raising errors for inconsistencies in branch count. The `_is_fully_covered` method evaluates branch coverage by checking distances. The `_add_testcase` method adds a test case to the archive, updating branch distances and checking for full coverage. The `_evaluate_test_case` method assesses test cases for branch coverage and decides whether to store or disregard them. The `consider_fn_testcase` method integrates this process, executing functions, updating records, and handling redundant cases.

The `dump` method generates Python `unittest`-compatible strings for all test cases, enabling easy integration into test suites. Additional methods include retrieving test cases for a function and determining the number of branches for a function.

EventGenerator (event_generator.py)

The `EventGenerator` class generates and manipulates test inputs of specific types, such as integers, strings, and key-value pairs. It provides random input generation, mutation, and crossover functionality. The class initializes with input types and optional constraints for integers (minimum and maximum values) and strings (maximum length). These constraints default to predefined constants if not provided. Input types are validated to be homogeneous or a singleton containing the special type `"kv"`.

Random inputs are generated using the `generate_random_input` method, which iterates over input types to create corresponding values. Integer inputs are randomly selected within a range, strings are constructed with random lowercase letters, and key-value pairs are tuples of a random string and integer. Mutation modifies existing inputs, with strings being altered by random character substitution, and key-value pairs undergoing changes to their key or value. The `mutate_input` method applies mutations to all elements of an input based on type.

Crossover combines elements from two inputs to create new ones. For strings, the `_crossover_strings` method swaps parts of two strings to form hybrids. The `crossover_inputs` method combines elements of integers, strings, or key-value pairs based on input type.

TestPool (testgen_random.py)

The `TestPool` class extends `EventGenerator` to manage a dynamic pool of test inputs. It generates, mutates, and combines inputs to create diverse and effective test cases. The class initializes with input types and optional constraints, populating the pool using the `_generate_pools` method. The number of inputs is controlled by the `pool_size` parameter.

New inputs are added using the `_add_new_input` method, while input generation occurs through random generation, mutation, or crossover. Random generation creates fresh inputs using the parent class's `generate_random_input` method. Mutation modifies existing inputs using the parent class's mutation methods. Crossover combines elements of two randomly selected inputs using the parent class's `crossover_inputs` method.

The `stochastic_generation` method randomly selects one of the three strategies to generate new inputs, ensuring diversity in test case creation. This method can also log details about the generation process, such as the strategy used, inputs generated, and current pool size.

fuzz_N_times() (testgen_random.py)

The `fuzz_N_times` function generates test inputs, executes test cases, and evaluates their contribution to branch coverage. It works closely with the `Archive`, `TestPool`, and `TestCase` classes to manage test inputs, test results, and branch coverage metrics. The function begins by initializing its parameters. The `Archive` instance stores all information about branch coverage and test cases for the function under test. The `fn_name` parameter specifies the name of the function being tested, while `arg_types` and `arg_names` define the argument types and their corresponding names. The `num_branches` parameter represents the number of branches within the function, and `N` is the maximum number of test cases that will be generated and evaluated. The `full_condition_coverage_stop` parameter determines whether the function should terminate early if full branch coverage is achieved, while any additional constraints for input generation are passed via `kwargs`.

If the argument types for the function are `['str', 'int']`, they are replaced with `['kv']`, treating the input as a key-value pair. A `TestPool` instance is then initialized with the adjusted argument types and any additional constraints, such as ranges for integers or string lengths. The `TestPool` is responsible for dynamically generating diverse test inputs using random generation, mutation, and crossover techniques. The `Archive` instance is updated to include the function under test by invoking its `add_fn` method, which initializes branch distance mappings, test case lists, and coverage tracking structures for the function.

The testing process begins in a loop that continues until either the maximum number of test cases (`N`) has been generated or full branch coverage has been achieved, depending on the value of the `full_condition_coverage_stop` flag. Within each iteration, the `stochastic_generation` method of the `TestPool` generates new test inputs. Each generated input is used to create a `TestCase` instance, encapsulating the input types, argument names, and the input values. The test case is then evaluated by the `Archive` through its `consider_fn_testcase` method, which executes the function under test, calculates branch distances, and updates coverage metrics. If the `full_condition_coverage_stop` flag is enabled, the function checks whether full branch coverage has been achieved during this iteration, and if so, exits the loop early. The number of remaining iterations (`N`) decreases with each input processed.

The function continues to generate and evaluate test cases until it either exhausts the maximum allowed iterations or achieves full branch coverage. Once the loop concludes, the updated `Archive` instance is returned. This instance contains all the test cases generated, including those that contributed to coverage improvement and those disregarded for redundancy or Exception errors raise, as well as the final coverage metrics for the function under test.

Hyperparameters	**Value**
POOL_SIZE	1000
N	2000
MIN_INT	-100
MAX_INT	100
MAX_STRING_LENGTH	20

Table 2: Hyperparameters for Fuzzer

Section 3: Genetic Algorithm test generator

The code implements a genetic algorithm (GA) designed to generate test cases aimed at achieving branch coverage for a specified function. The process involves leveraging classes like `Archive`, `EventGenerator`, and `TestCase` to manage test case generation, coverage evaluation, and optimization. The core functionality is encapsulated in the `rep_ga_generation` function, supported by auxiliary functions like `normalize`, `get_fitness_fn`, `crossover_out`, and `mutate_out`. These components collectively define the process of initializing populations, evaluating their fitness, performing genetic operations (crossover and mutation), and selecting individuals for subsequent generations.

The `HYPERPARAMETERS` dictionary defines key settings for the genetic algorithm, such as the number of individuals in the population (`NPOP`), the number of generations (`NGEN`), mutation and crossover probabilities (`INDMUPROB`, `MUPROB`, `CXPROB`), tournament selection size (`TOURNSIZE`), and the number of repetitions for the GA run (`REPS`).

The `normalize` function calculates a normalized value for a given branch distance, transforming it into a floating-point number between 0 and 1. This is used in fitness evaluation to ensure distances are treated comparably across branches.

The `get_fitness_fn` function computes the fitness of an individual in the population. It executes the function under test using the individual's input values, represented as a `TestCase`. If full branch coverage is not achieved, the function calculates the normalized branch distances for all uncovered branches, summing these values to determine the fitness score. A lower score indicates better fitness, as it implies closer proximity to covering all branches.

The `crossover_out` function implements the crossover operation for the GA. It takes two parent individuals, retrieves their inputs, and uses the `EventGenerator`'s `crossover_inputs` method to produce offspring. The resulting offspring are assigned back to the respective parents.

The `mutate_out` function defines the mutation operation for the GA. It retrieves the input from an individual, applies the `EventGenerator`'s `mutate_input` method to produce a mutated version, and updates the individual with the mutated input.

The `rep_ga_generation` function is the main driver of the genetic algorithm. It initializes the `EventGenerator` with the argument types of the function under test and updates the `Archive` with details about the function and its branches using the `add_fn` method. The function then defines custom genetic operators (crossover and mutation) and a fitness function based on the branch coverage status maintained by the `Archive`.

A `toolbox` is set up using DEAP (Distributed Evolutionary Algorithms in Python) to manage the GA pipeline. The toolbox defines how individuals and populations are initialized, how fitness is evaluated, and which genetic operators are used. The population is initialized with random inputs generated by the `EventGenerator`.

For each repetition (`REPS`), the algorithm resets the branch coverage data in the `Archive` and initializes a new population. The `eaSimple` algorithm runs the GA for the specified number of generations (`NGEN`), applying crossover and mutation with probabilities defined in the hyperparameters. After each run, the branch coverage achieved is calculated based on zero-distance branches. If the coverage improves compared to previous runs, the corresponding test cases are retained.

Finally, the function updates the `Archive` with the best test cases and resets the branch coverage data. The updated `Archive`, containing the best test cases and their associated coverage metrics, is returned. This framework integrates the GA's ability to explore and optimize inputs with the detailed coverage tracking provided by the `Archive`. It ensures that test case generation is both systematic and adaptive, improving the likelihood of achieving comprehensive branch coverage.

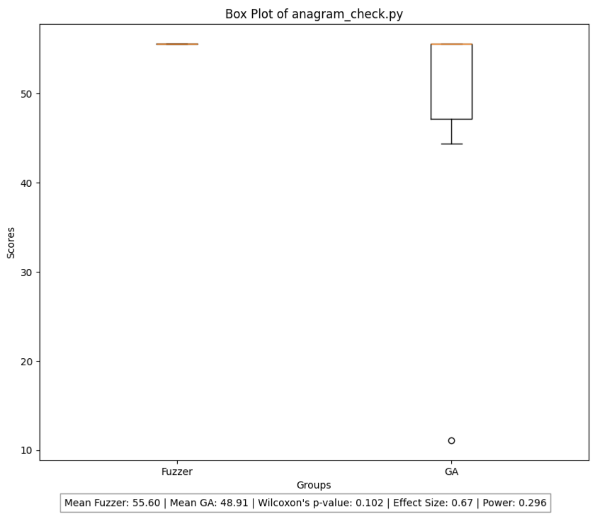
Hyperparameter	**Value**
MIN_INT	-100
MAX_INT	100
MAX_STRING_LENGTH	20
NPOP	200
NGEN	20
INDMUPROB	0.05
MUPROB	0.1
CXPROB	0.5
TOURNSIZE	3
REPS	3

Table 3: Hyperparameters for Genetic Algorithm

Section 4: Statistical comparison of test generators

This section presents the results of the experimental procedure comparing the Fuzzer and GA generators across ten benchmark programs. Each comparison includes the mutation scores, their visualization using box plots, and statistical analysis, including Cohen's d effect size, Wilcoxon's p-value, and power, and a table containing the summary of these results.

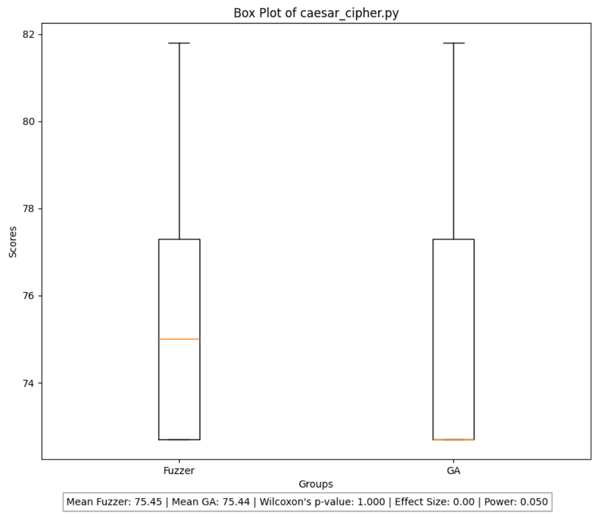
anagram_check.py



Mean Fuzzer: 55.60 | Mean GA: 48.91 | Wilcoxon's p-value: 0.102 | Effect Size: 0.67 | Power: 0.296

The Fuzzer achieved a higher mean mutation score (55.60) compared to the GA (48.91). The Wilcoxon's p-value of 0.102 indicates no statistically significant difference at the 0.05 level. However, the effect size ($d = 0.67$) suggests a moderate-to-large practical difference favoring the Fuzzer. The power of the test (0.296) is relatively low, indicating a limited ability to detect true differences.

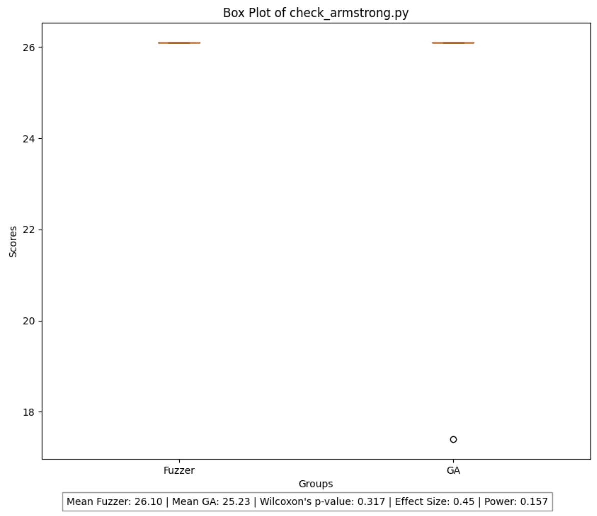
caesar_cipher.py



Mean Fuzzer: 75.45 | Mean GA: 75.44 | Wilcoxon's p-value: 1.000 | Effect Size: 0.00 | Power: 0.050

The mutation scores for the Fuzzer and GA are nearly identical, with means of 75.45 and 75.44, respectively. The Wilcoxon p-value of 1.000 confirms no statistically significant difference, while the effect size ($d = 0.00$) suggests no practical difference. The power (0.050) is extremely low, as expected in such scenarios with no difference.

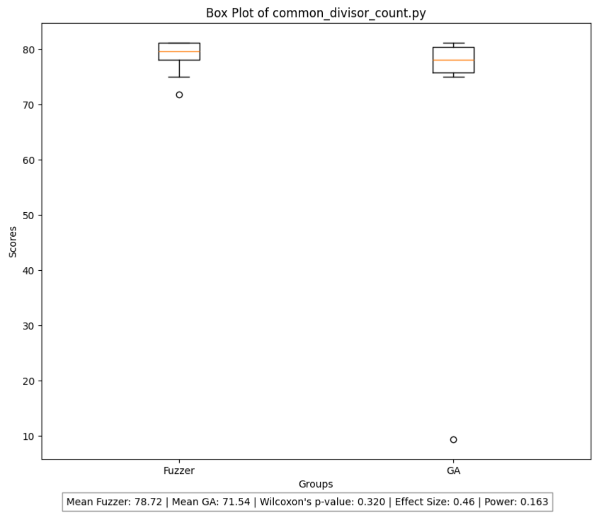
check_armstrong.py



Mean Fuzzer: 26.10 | Mean GA: 25.23 | Wilcoxon's p-value: 0.317 | Effect Size: 0.45 | Power: 0.157

The Fuzzer slightly outperforms the GA with a mean score of 26.10 compared to 25.23. The Wilcoxon p-value of 0.317 indicates no statistically significant difference, while the effect size ($d = 0.45$) suggests a small-to-moderate practical advantage for the Fuzzer. The test's power (0.157) is low, limiting confidence in detecting a true difference.

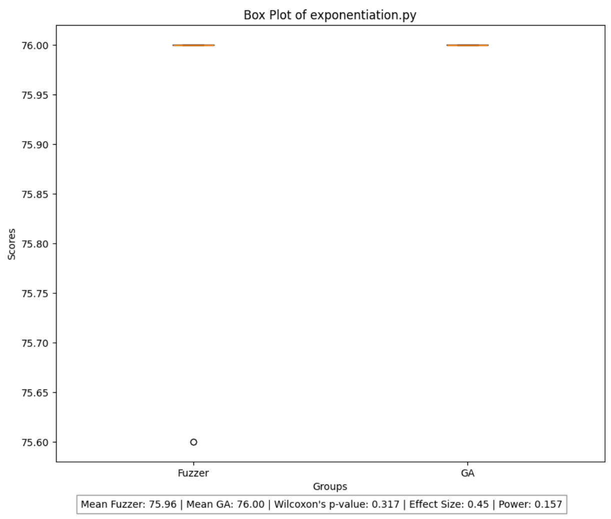
common_divisor_count.py



Mean Fuzzer: 78.72 | Mean GA: 71.54 | Wilcoxon's p-value: 0.320 | Effect Size: 0.46 | Power: 0.163

The Fuzzer outperformed the GA with a mean mutation score of 78.72 compared to 71.54. The Wilcoxon p-value of 0.320 shows no statistically significant difference. The effect size ($d = 0.46$) indicates a small-to-moderate practical advantage for the Fuzzer. The power (0.163) remains low, reducing the confidence in this result.

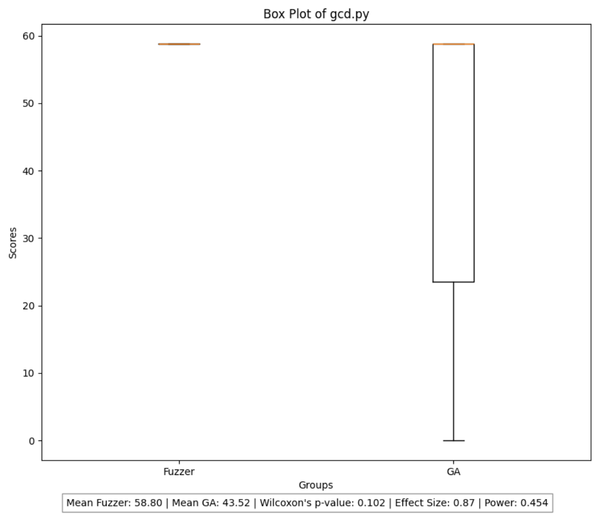
exponentiation.py



Mean Fuzzer: 75.96 | Mean GA: 76.00 | Wilcoxon's p-value: 0.317 | Effect Size: 0.45 | Power: 0.157

The GA and Fuzzer achieved very similar mean scores (76.00 and 75.96, respectively). The Wilcoxon p-value of 0.317 indicates no statistically significant difference. The effect size ($d = 0.45$) suggests a small practical difference, while the power (0.157) is low.

gcd.py

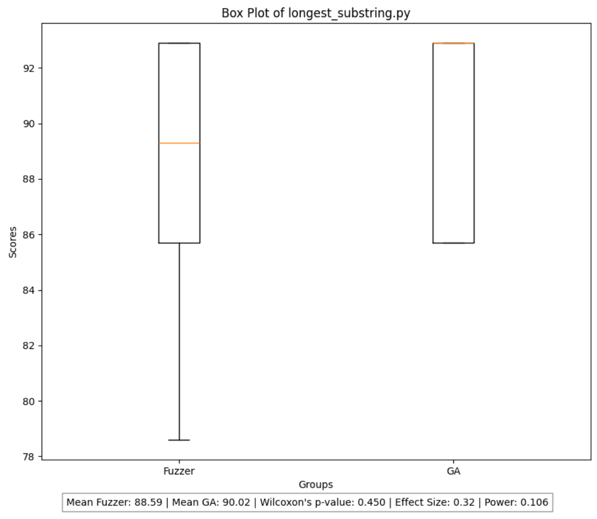


Mean Fuzzer: 58.80 | Mean GA: 43.52 | Wilcoxon's p-value: 0.102 | Effect Size: 0.87 | Power: 0.454

The Fuzzer performed significantly better than the GA, with mean scores of 58.80 and 43.52, respectively. Although the Wilcoxon p-value (0.102) does not indicate statistical significance, the large effect size

($d = 0.87$) highlights a substantial practical advantage for the Fuzzer. The power (0.454) is higher than in most other cases.

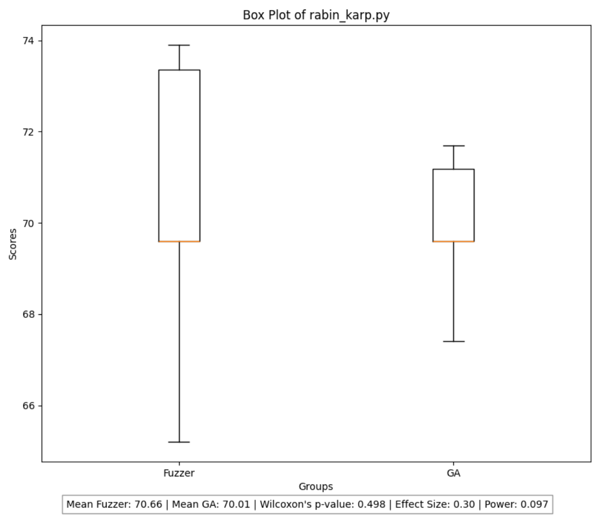
longest_substring.py



Mean Fuzzer: 88.59 | Mean GA: 90.02 | Wilcoxon's p-value: 0.450 | Effect Size: 0.32 | Power: 0.106

The GA slightly outperformed the Fuzzer with mean scores of 90.02 and 88.59, respectively. The Wilcoxon p-value of 0.450 indicates no statistically significant difference. The small effect size ($d = 0.32$) suggests minimal practical differences, and the power (0.106) is low.

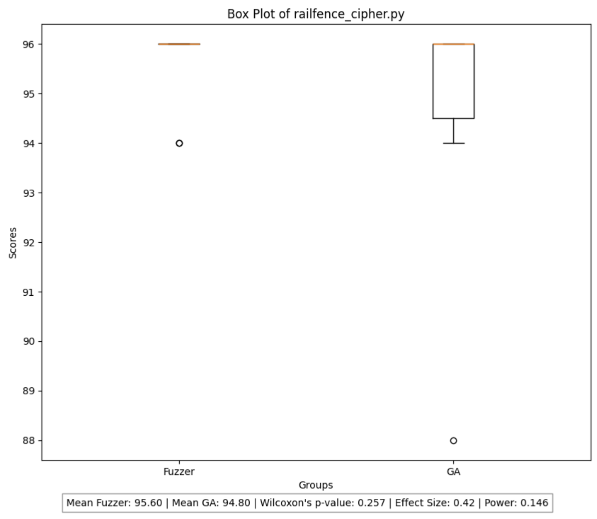
rabin_karp.py



Mean Fuzzer: 70.66 | Mean GA: 70.01 | Wilcoxon's p-value: 0.498 | Effect Size: 0.30 | Power: 0.097

The Fuzzer and GA performed nearly identically, with mean scores of 70.66 and 70.01. The Wilcoxon p-value of 0.498 confirms no statistically significant difference, and the effect size ($d = 0.30$) suggests a negligible practical difference. The power (0.097) remains low.

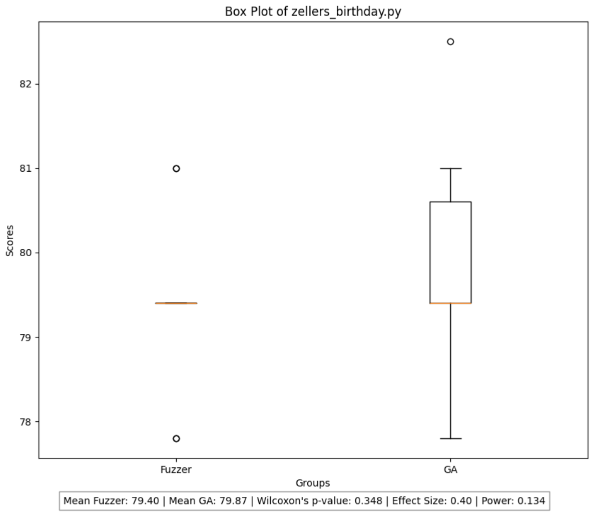
railfence_cipher.py



Mean Fuzzer: 95.60 | Mean GA: 94.80 | Wilcoxon's p-value: 0.257 | Effect Size: 0.42 | Power: 0.146

The Fuzzer slightly outperformed the GA with mean scores of 95.60 and 94.80. The Wilcoxon p-value of 0.257 indicates no statistically significant difference, but the effect size ($d = 0.42$) suggests a small practical advantage for the Fuzzer. The power (0.146) is relatively low.

zellers_birthday.py



Mean Fuzzer: 79.40 | Mean GA: 79.87 | Wilcoxon's p-value: 0.348 | Effect Size: 0.40 | Power: 0.134

The GA marginally outperformed the Fuzzer with mean scores of 79.87 and 79.40. The Wilcoxon p-value of 0.348 indicates no statistically significant difference, and the effect size ($d = 0.40$) suggests only a small practical difference. The power (0.134) remains low.

Summary Table of Metrics

Benchmark	**Mean Fuzzer**	**Mean GA**	**p-value**	**Effect Size**	**Power**
`anagram_check.py`	55.60	48.91	0.102	0.67	0.296
`caesar_cipher.py`	75.45	75.44	1.000	0.00	0.050
`check_armstrong.py`	26.10	25.23	0.317	0.45	0.157
`common_divisor_count.py`	78.72	71.54	0.320	0.46	0.163
`exponentiation.py`	75.96	76.00	0.317	0.45	0.157
`gcd.py`	58.80	43.52	0.102	0.87	0.454
`longest_substring.py`	88.59	90.02	0.450	0.32	0.106
`rabin_karp.py`	70.66	70.01	0.498	0.30	0.097
`railfence_cipher.py`	95.60	94.80	0.257	0.42	0.146
`zellers_birthday.py`	79.40	79.87	0.348	0.40	0.134

Table 4: Summary of performance metrics for Fuzzer and GA across benchmarks.

Summary and Observations

The results indicate that the Fuzzer generally outperformed the Genetic Algorithm (GA) in terms of mean mutation scores across several benchmarks, particularly in `anagram_check.py` and `gcd.py`. For example, in `anagram_check.py`, the Fuzzer achieved a mean score of 55.60 compared to GA's 48.91, with a moderate effect size ($d = 0.67$), suggesting a practical advantage. However, the statistical analysis yielded a p-value of 0.102, indicating that this difference is not statistically significant.

In some cases, such as `caesar_cipher.py` and `exponentiation.py`, the mean scores for both techniques were nearly identical (e.g., 75.45 vs. 75.44 in `caesar_cipher.py`), with p-values of 1.000 and 0.317, respectively. These results suggest no meaningful performance difference between the two methods for these benchmarks.

The most striking difference was observed in `gcd.py`, where the Fuzzer achieved a significantly higher mean score (58.80) compared to GA (43.52). The effect size ($d = 0.87$) indicates a strong practical advantage for the Fuzzer. However, the p-value of 0.102 still falls short of statistical significance, preventing definitive conclusions.

In benchmarks like `longest_substring.py` and `rabin_karp.py`, GA showed marginally higher mean scores than the Fuzzer, but the differences were minimal, as reflected in small effect sizes ($d = 0.32$ and $d = 0.30$, respectively). These results suggest that the observed differences are unlikely to have practical significance. Additionally, low statistical power across benchmarks highlights the difficulty of detecting true performance differences with the current sample size.

Overall, while the Fuzzer displayed practical advantages in some cases, the lack of consistent statistical significance suggests that the observed differences may not generalize across benchmarks. Future work should consider increasing the sample size and repetitions to enhance statistical power and explore additional metrics to better evaluate the strengths and weaknesses of each testing method.