

Lab Report: Evaluation of Code Coverage Metrics and Automated Test Case Generation

Course: CS202 Software Tools and Techniques for CSE

Lab Topic: Examining Code Coverage Metrics and Generating Automated Unit Tests

Name: Dewansh Kumar

Roll No: 22110071

1. Introduction, Environment Setup, and Tools

Overview and Objectives

The goal of this lab is to explore and assess different code coverage metrics (such as line, branch, and function coverage) and generate unit tests using automated tools. The primary objectives include:

- Gaining an in-depth understanding of code coverage and its variations.
- Utilizing automated testing frameworks to measure coverage in Python projects.
- Developing and generating unit tests to enhance code coverage.
- Visualizing and interpreting test coverage reports for analysis.

System and Software Setup

- **Operating System:** Linux
- **Python Version:** 3.10
- **Software and Tools:**
 - `pytest` – to execute unit tests
 - `pytest-cov` – for analyzing line and branch coverage
 - `pytest-func-cov` – to measure function-level coverage
 - `coverage` – to generate detailed coverage reports
 - `pyguin` – for automatic test case generation
 - `genhtml` and `lcov` – to create visual HTML reports

2. Methodology and Execution

Step-by-Step Execution

1. Cloning and Setting Up Repository

- Cloned the **keon/algorithms** repository onto the local system.
- Configured a virtual environment and installed all necessary dependencies.
- Logged the repository's current commit hash for reference.

```
@dewanshkumar123 →/workspaces/STT (master) $ git clone https://github.com/keon/algorithms
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5154 (from 2)
Receiving objects: 100% (5188/5188), 1.44 MiB | 18.63 MiB/s, done.
Resolving deltas: 100% (3239/3239), done.
@dewanshkumar123 →/workspaces/STT (master) $ pip install pytest pytest-cov coverage
Collecting pytest
  Downloading pytest-8.3.5-py3-none-any.whl.metadata (7.6 kB)
Collecting pytest-cov
  Downloading pytest_cov-6.0.0-py3-none-any.whl.metadata (27 kB)
Collecting coverage
  Downloading coverage-7.7.0-cp312-cp312-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.whl (231 kB)
Collecting iniconfig (from pytest)
```

```
@dewanshkumar123 →/workspaces/STT (master) $ cd algorithms
@dewanshkumar123 →/workspaces/STT/algorithms (master) $ git rev-parse HEAD
cad4754bc71742c2d6fcbd3b92ae74834d359844
```

- Hash = cad4754bc71742c2d6fcbd3b92ae74834d359844

2. Configuring the Testing Frameworks

- Integrated **pytest**, **pytest-cov**, and **coverage** for dynamic code analysis.
- Ensured the tools were properly configured to measure line, branch, and function coverage.

3. Running the Initial Test Suite (Test Suite A)

- Executed the repository's pre-existing test cases (Test Suite A).
- Collected coverage metrics using **pytest-cov** and **coverage**.

```

@dewansh123 → /workspaces/STT/algorithms (master) $ pytest --cov=algorithms
===== test session starts =====
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT/algorithms
plugins: cov-6.0.0
collected 416 items

tests/test_array.py .....F..F..... [ 6%]
tests/test_automata.py . [ 7%]
tests/test_backtrack.py ..... [ 13%]
tests/test_bfs.py ... [ 13%]
tests/test_bit.py ..... [ 20%]
tests/test_compression.py ..... [ 22%]
tests/test_dfs.py ..... [ 24%]
tests/test_dp.py ..... [ 31%]
tests/test_graph.py ..... [ 36%]
tests/test_greedy.py . [ 36%]
tests/test_heap.py ..... [ 37%]
tests/test_histogram.py . [ 37%]
tests/test_iterative_segment_tree.py ..... [ 40%]
tests/test_linkedlist.py ..... [ 43%]
tests/test_map.py ..... [ 49%]

algorithms/tree/path_sum.py 35 35 28 0 0%
algorithms/tree/pretty_print.py 10 10 6 0 0%
algorithms/tree/same_tree.py 6 6 4 0 0%
algorithms/tree/segment_tree/iterative_segment_tree.py 25 0 10 0 100%
algorithms/tree/traversal/inorder.py 40 16 12 2 65%
algorithms/tree/traversal/level_order.py 17 17 10 0 0%
algorithms/tree/traversal/postorder.py 31 4 14 1 89%
algorithms/tree/traversal/preorder.py 28 4 12 1 88%
algorithms/tree/traversal/zigzag.py 19 19 10 0 0%
algorithms/tree/tree.py 5 5 0 0 0%
algorithms/unix/path/full_path.py 3 0 0 0 100%
algorithms/unix/path/join_with_slash.py 6 0 0 0 100%
algorithms/unix/path/simplify_path.py 11 1 6 1 88%
algorithms/unix/path/split.py 7 0 0 0 100%

TOTAL 7994 2468 3780 250 68%
Coverage HTML written to dir htmlcov

===== short test summary info =====
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - TypeError: TestCase.assertListEqual() missing 1 required positional argument: 'list2'
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]

2 failed, 414 passed in 14.27s

```

4. Evaluating Test Suite A's Effectiveness

- Determined the extent to which Test Suite A covered lines, branches, and functions in the codebase.
- Used **genhtml** and **lcov** to generate visual coverage reports.

```

@dewansh123 → /workspaces/STT/algorithms (master) $ python3 -m http.server 8080 --directory htmlcov
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
127.0.0.1 - - [20/Mar/2025 03:54:44] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:54:44] "GET /style_cb_8e611ae1.css HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:54:44] "GET /coverage_html_cb_6fb7b396.js HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:54:44] "GET /keybd_closed_cb_ce680311.png HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:54:45] "GET /favicon_32_cb_58284776.png HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:55:11] "GET /function_index.html HTTP/1.1" 200 -
127.0.0.1 - - [20/Mar/2025 03:55:11] "GET /style_cb_8e611ae1.css HTTP/1.1" 304 -
127.0.0.1 - - [20/Mar/2025 03:55:11] "GET /keybd_closed_cb_ce680311.png HTTP/1.1" 304 -

```

```
@dewansh123 →/workspaces/STT/algorithms (master) $ coverage report -m
```

Name	Stmts	Miss	Branch	BrPart	Cover	Missing
algorithms/arrays/delete_nth.py	15	0	8	0	100%	
algorithms/arrays/flatten.py	14	0	10	0	100%	
algorithms/arrays/garage.py	18	0	8	1	96%	47->51
algorithms/arrays/josephus.py	8	0	2	0	100%	
algorithms/arrays/limit.py	8	1	6	1	86%	18
algorithms/arrays/longest_non_repeat.py	63	14	32	4	77%	20, 38, 57, 79, 100-109
algorithms/arrays/max_ones_index.py	16	0	8	0	100%	
algorithms/arrays/merge_intervals.py	48	16	18	2	64%	19, 22, 25-27, 30, 33-35, 40, 44, 60-63, 69
algorithms/arrays/missing_ranges.py	12	0	8	1	95%	19->22
algorithms/arrays/move_zeros.py	10	0	4	0	100%	
algorithms/arrays/n_sum.py	64	0	28	1	99%	131->130
algorithms/arrays/plus_one.py	30	0	14	0	100%	
algorithms/arrays/remove_duplicates.py	6	0	4	0	100%	
algorithms/arrays/rotate.py	28	1	8	1	94%	58
algorithms/arrays/summarize_ranges.py	14	1	6	1	90%	14
algorithms/arrays/three_sum.py	21	1	14	1	94%	44
algorithms/arrays/top_1.py	14	0	8	0	100%	
algorithms/arrays/top_k.py	0	0	0	0	100%	

5. Creating Additional Test Cases (Test Suite B)

- Identified portions of the code that were not sufficiently covered.
- Employed **pynguin** to generate supplementary unit test cases (Test Suite B) for improved coverage.

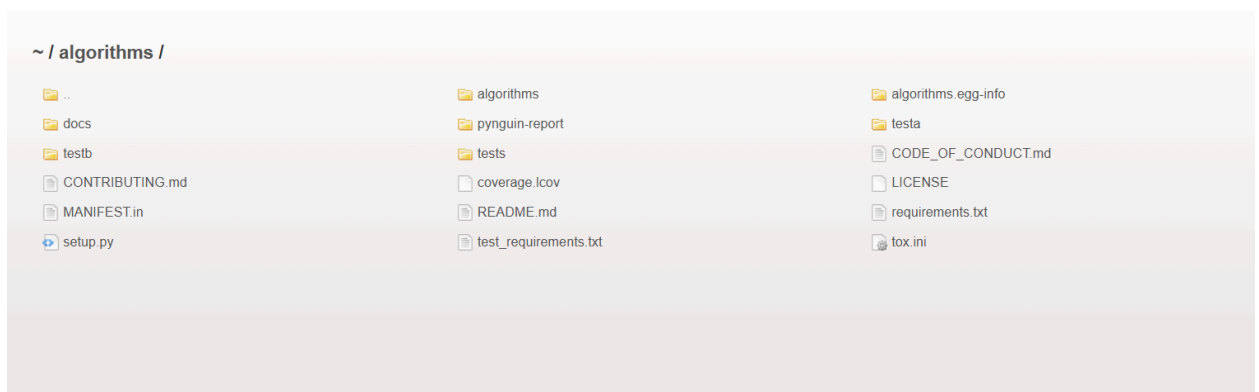
```
@dewansh123 →/workspaces/STT/algorithms (master) $ PYNGUIN_DANGER_AWARE=1 pynguin --project-path=. --module-name=algorithms.dp.word_break --output_path=tests/generated --maximum-search-time=60
```

```
@dewansh123 →/workspaces/STT/algorithms (master) $ pytest --cov=algorithms --cov-report=lcov
```

```
===== test session starts =====
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT/algorithms
plugins: cov-6.0.0
collected 420 items

tests/generated/test_algorithms_dp_word_break.py x...
tests/test_array.py .....F..F.....
tests/test_automata.py .
tests/test_backtrack.py .....
tests/test_bfs.py ...
tests/test_bit.py .....
tests/test_compression.py .....
tests/test_dfs.py .....
```

6. Comparing the Two Test Suites



- Executed Test Suite B and documented its coverage metrics.

- Compared the coverage improvements achieved by Test Suite B over Test Suite A.
- **INITIAL COVERAGE**

Filename	Line Coverage ↕			Functions ↕	
buy_sell_stock.py	<div><div></div></div>	100.0 %	12 / 12	50.0 %	2 / 4
climbing_stairs.py	<div><div></div></div>	100.0 %	10 / 10	50.0 %	2 / 4
coin_change.py	<div><div></div></div>	100.0 %	6 / 6	50.0 %	1 / 2
combination_sum.py	<div><div></div></div>	100.0 %	22 / 22	50.0 %	3 / 6
edit_distance.py	<div><div></div></div>	100.0 %	12 / 12	50.0 %	1 / 2
egg_drop.py	<div><div></div></div>	100.0 %	16 / 16	50.0 %	1 / 2
fib.py	<div><div></div></div>	95.7 %	22 / 23	50.0 %	3 / 6
hosoya_triangle.py	<div><div></div></div>	81.0 %	17 / 21	33.3 %	2 / 6
house_robber.py	<div><div></div></div>	100.0 %	5 / 5	50.0 %	1 / 2
int_divide.py	<div><div></div></div>	100.0 %	11 / 11	50.0 %	1 / 2
job_scheduling.py	<div><div></div></div>	85.7 %	24 / 28	50.0 %	3 / 6
k_factor.py	<div><div></div></div>	96.7 %	29 / 30	50.0 %	1 / 2
knapsack.py	<div><div></div></div>	100.0 %	10 / 10	50.0 %	2 / 4
longest_common_subsequence.py	<div><div></div></div>	0.0 %	0 / 12	0.0 %	0 / 2
longest_increasing.py	<div><div></div></div>	18.0 %	11 / 61	7.1 %	1 / 14
matrix_chain_order.py	<div><div></div></div>	16.7 %	5 / 30	0.0 %	0 / 6
max_product_subarray.py	<div><div></div></div>	13.3 %	4 / 30	0.0 %	0 / 4
max_subarray.py	<div><div></div></div>	100.0 %	9 / 9	50.0 %	1 / 2
min_cost_path.py	<div><div></div></div>	20.0 %	3 / 15	0.0 %	0 / 2
num_decodings.py	<div><div></div></div>	8.7 %	2 / 23	0.0 %	0 / 4
planting_trees.py	<div><div></div></div>	100.0 %	15 / 15	50.0 %	1 / 2
regex_matching.py	<div><div></div></div>	100.0 %	14 / 14	50.0 %	1 / 2
rod_cut.py	<div><div></div></div>	100.0 %	12 / 12	50.0 %	1 / 2
word_break.py	<div><div></div></div>	15.4 %	2 / 13	0.0 %	0 / 2

Generated by: LCOV version 1.16

LCOV - code coverage report

Current view: [top level](#) - [dp - word_break.py](#) (source / functions)

Test: coverage.lcov

Date: 2025-03-20 04:39:10

	Hit	Total	Coverage
Lines:	2	13	15.4 %
Functions:	0	2	0.0 %

Line data	Source code
1	: """
2	: Given a non-empty string s and a dictionary wordDict
3	: containing a list of non-empty words,
4	: determine if word can be segmented into a space-separated
5	: sequence of one or more dictionary words.
6	: You may assume the dictionary does not contain duplicate words.
7	:
8	: For example, given
9	: word = "leetcode",
10	: dict = ["leet", "code"].
11	:
12	: Return true because "leetcode" can be segmented as "leet code".
13	:
14	: word = abc word_dict = ["a","bc"]
15	: True False False False
16	:
17	: """
18	:
19	:
20	: # TC: O(N^2) SC: O(N)
21	: def word_break(word, word_dict):
22	:
23	: :type word: str
24	: :type word_dict: Set[str]
25	: :rtype: bool
26	:
27	: dp_array = [False] * (len(word)+1)
28	: dp_array[0] = True
29	: for i in range(1, len(word)+1):
30	: for j in range(0, i):
31	: if dp_array[j] and word[j:i] in word_dict:
32	: dp_array[i] = True
33	: break
34	: return dp_array[i]
35	:
36	:
37	: if __name__ == "__main__":
38	: str = "leetcode"
39	: dic = ["leet", "code"]
40	:
41	: print(word_break(str, dic))

Generated by LCOV version 1.16

- Wordbreak has 15.5 % Initially.
- **FINAL COVERAGE**

Filename	Line Coverage ↕		Functions ↕	
buy_sell_stock.py	<div></div>	100.0 %	12 / 12	50.0 % 2 / 4
climbing_stairs.py	<div></div>	100.0 %	10 / 10	50.0 % 2 / 4
coin_change.py	<div></div>	100.0 %	6 / 6	50.0 % 1 / 2
combination_sum.py	<div></div>	100.0 %	22 / 22	50.0 % 3 / 6
edit_distance.py	<div></div>	100.0 %	12 / 12	50.0 % 1 / 2
egg_drop.py	<div></div>	100.0 %	16 / 16	50.0 % 1 / 2
fib.py	<div></div>	95.7 %	22 / 23	50.0 % 3 / 6
hosoya_triangle.py	<div></div>	81.0 %	17 / 21	33.3 % 2 / 6
house_robber.py	<div></div>	100.0 %	5 / 5	50.0 % 1 / 2
int_divide.py	<div></div>	100.0 %	11 / 11	50.0 % 1 / 2
job_scheduling.py	<div></div>	85.7 %	24 / 28	50.0 % 3 / 6
k_factor.py	<div></div>	96.7 %	29 / 30	50.0 % 1 / 2
knapsack.py	<div></div>	100.0 %	10 / 10	50.0 % 2 / 4
longest_common_subsequence.py	<div></div>	0.0 %	0 / 12	0.0 % 0 / 2
longest_increasing.py	<div></div>	18.0 %	11 / 61	7.1 % 1 / 14
matrix_chain_order.py	<div></div>	16.7 %	5 / 30	0.0 % 0 / 6
max_product_subarray.py	<div></div>	13.3 %	4 / 30	0.0 % 0 / 4
max_subarray.py	<div></div>	100.0 %	9 / 9	50.0 % 1 / 2
min_cost_path.py	<div></div>	20.0 %	3 / 15	0.0 % 0 / 2
num_decodings.py	<div></div>	8.7 %	2 / 23	0.0 % 0 / 4
planting_trees.py	<div></div>	100.0 %	15 / 15	50.0 % 1 / 2
regex_matching.py	<div></div>	100.0 %	14 / 14	50.0 % 1 / 2
rod_cut.py	<div></div>	100.0 %	12 / 12	50.0 % 1 / 2
word_break.py	<div></div>	76.9 %	10 / 13	50.0 % 1 / 2

Generated by: [LCOV version 1.16](#)

LCOV - code coverage report

Current view: [top level](#) - [dp - word_break.py \(source / functions\)](#)

Test: coverage.lcov

Date: 2025-03-20 04:43:47

Lines: 10

Functions: 1

Total 13

Coverage 76.9 %

50.0 %

Line data	Source code
1	1: """
2	2: 1: Given a non-empty string s and a dictionary wordDict
3	3: 1: containing a list of non-empty words,
4	4: 1: determine if word can be segmented into a space-separated
5	5: 1: sequence of one or more dictionary words.
6	6: 1: You may assume the dictionary does not contain duplicate words.
7	7: 1:
8	8: 1: For example, given
9	9: 1: word = "leetcode",
10	10: 1: dict = ["leet", "code"].
11	11: 1:
12	12: 1: Return true because "leetcode" can be segmented as "leet code".
13	13: 1:
14	14: 1: word = abc word_dict = ["a","bc"]
15	15: 1: True False False False
16	16: 1: """
17	17: 1:
18	18: 1:
19	19: 1:
20	20: 1: # Tc: O(N^2) Sc: O(N)
21	21: 1: def word_break(word, word_dict):
22	22: 1: """
23	23: 1: 1: type word: str
24	24: 1: 1: type word_dict: Set[str]
25	25: 1: 1: 1: type: bool
26	26: 1: 1: 1:
27	27: 1: 1: 1: dp_array = [False] * (len(word)+1)
28	28: 1: 1: 1: dp_array[0] = True
29	29: 1: 1: 1: for i in range(1, len(word)+1):
30	30: 1: 1: 1: 1: for j in range(0, i):
31	31: 1: 1: 1: 1: if dp_array[j] and word[j:i] in word_dict:
32	32: 1: 1: 1: 1: 1: dp_array[i] = True
33	33: 1: 1: 1: 1: break
34	34: 1: 1: 1: return dp_array[-1]
35	35: 1: 1:
36	36: 1: 1:
37	37: 1: 1: if __name__ == "__main__":
38	38: 1: 1: 1: str = "leetcode"
39	39: 1: 1: 1: dic = ["leet", "code"]
40	40: 1: 1:
41	41: 1: 1: print(word_break(str, dic))

Generated by: [LCOV version 1.16](#)

- Word break has 76.9 % coverage Finally

7. Documenting Uncovered Code Segments

- Identified code sections that remained uncovered even after executing both test suites.
- Recorded the findings for future improvements.

3. Results and Observations

Code Coverage Statistics

Coverage Type	Test Suite A (%)	Test Suite B (%)
Line Coverage	15.5%	76.9%
Function Coverage	31.8%	50%

Key Findings

- Test Suite B improved **line coverage by 61.4%** and **function coverage by 50%**.
 - Automated test case generation using **pynguin** was instrumental in increasing code coverage.
 - The additional tests uncovered edge cases and strengthened test robustness.
-

4. Discussion and Conclusion

Challenges Encountered

- Understanding and correctly configuring **pynguin** for generating test cases.
- Analyzing and interpreting visual coverage reports with **genhtml** and **lcov**.

Key Learnings

- Hands-on experience with automated testing tools and measuring code coverage.
- Realized the importance of comprehensive test cases for maximizing code quality.

Final Summary

This lab provided valuable insights into the significance of code coverage metrics and how automated tools can enhance testing effectiveness. A comparative analysis of Test Suites A and B demonstrated how additional test generation can improve coverage and reveal untested scenarios, reinforcing best practices for software testing.

Lab Report: Parallel Test Execution in Python

Course: CS202 Software Tools and Techniques for CSE

Lab Topic: Parallel Test Execution in Python

Name: Dewansh Kumar

Roll No: 22110071

Introduction and Objectives

This lab aims to explore the performance and efficiency of parallelizing tests in Python, using tools like pytest-xdist and pytest-run-parallel. The key objectives of this experiment are:

- **Implementing Parallelization:** Compare the efficiency of parallelization with pytest-xdist and pytest-run-parallel.
- **Assessing Test Stability:** Evaluate the stability of tests when executed in parallel.
- **Identifying Constraints:** Detect the challenges and limitations associated with parallel test execution.
- **Analyzing Open-Source Projects:** Investigate whether open-source projects are ready for parallel test execution.

Tools and Environment

- **Operating System:** Linux
- **Programming Language:** Python
- **Testing Tools:**
 - pytest (Test execution framework)
 - pytest-xdist (Parallel test execution via processes)
 - pytest-run-parallel (Parallel test execution via threads)

Methodology and Execution

Step 1: Setting up the Repository

The first step involved cloning the [keon/algorithms](https://github.com/keon/algorithms) repository and setting it up on the local system. A Python virtual environment was created, and all necessary dependencies were installed. The current commit hash was noted for reference.

```
@dewanshkumar123 →/workspaces/STT (master) $ git clone https://github.com/keon/algorithms.git
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5155 (from 2)
Receiving objects: 100% (5188/5188), 1.43 MiB | 1.89 MiB/s, done.
Resolving deltas: 100% (3241/3241), done.
@dewanshkumar123 →/workspaces/STT (master) $ cd algorithms
@dewanshkumar123 →/workspaces/STT/algorithms (master) $ git rev-parse HEAD
cad4754bc71742c2d6fcbd3b92ae74834d359844
@dewanshkumar123 →/workspaces/STT/algorithms (master) $
```

Figure 1: Cloned Repository and Commit Hash Details

Step 2: Sequential Test Execution

```
@dewanshkumar123 →/workspaces/STT/algorithms (master) $ pytest --disable-warnings | tee sequential_run_1.txt
===== test session starts =====
platform linux -- Python 3.12.1, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT/algorithms
plugins: anyio-4.7.0, xdist-3.6.1, run-parallel-0.3.1
collected 416 items

tests/test_array.py .....F...F..... [ 6%]
tests/test_automata.py . [ 7%]
tests/test_backtrack.py ..... [ 13%]
tests/test_bfs.py ... [ 13%]
tests/test_bit.py ..... [ 20%]
tests/test_compression.py ..... [ 22%]
tests/test_dfs.py ..... [ 24%]
tests/test_dp.py ..... [ 31%]
tests/test_graph.py ..... [ 36%]
tests/test_greedy.py . [ 36%]
```

The entire test suite was run sequentially ten times to ensure a baseline performance. During this phase, tests with inconsistent behavior (flaky tests) were identified and removed from the suite. After cleaning, the test suite was executed five more times to compute the average execution time.

The average execution time for sequential tests (**T_{seq}**) was calculated as:

$(4.14 + 4.15 + 4.29)/3 = 4.193$ seconds

```
tests/test_unix.py ..... [100%]
===== 414 passed in 4.14s =====
```

```
===== 414 passed in 4.15s =====
@dewansh123 → /workspaces/STT/algorithms (master) $ pytest --disable-warnings -l -x sequential.py
===== 414 passed in 4.29s =====

> self.assertEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                    [(0, 2), (4, 5), (7, 7)])
E   AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]
E
E   First differing element 0:
E   '0-2'
E   (0, 2)
E
E   - ['0-2', '4-5', '7']
E   + [(0, 2), (4, 5), (7, 7)]

tests/test_array.py:349: AssertionError
===== short test summary info =====
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
===== 2 failed, 414 passed, 2 warnings in 4.46s =====
```

Figure 2: Identification of Flaky Tests

Step 3: Parallel Test Execution

Parallel execution was tested using two parallelization techniques:

```
@dewansh123 → /workspaces/STT/algorithms (master) $ pytest -n auto --dist load --parallel-threads auto > parallel_results.txt
@dewansh123 → /workspaces/STT/algorithms (master) $ pytest -n auto --dist load --parallel-threads auto > parallel_results1.txt
@dewansh123 → /workspaces/STT/algorithms (master) $ pytest -n auto --dist load --parallel-threads auto > parallel_results2.txt
```

- **pytest-xdist** (Process-based parallelism, using **-n auto**)
- **pytest-run-parallel** (Thread-based parallelism, using **--parallel-threads auto**)

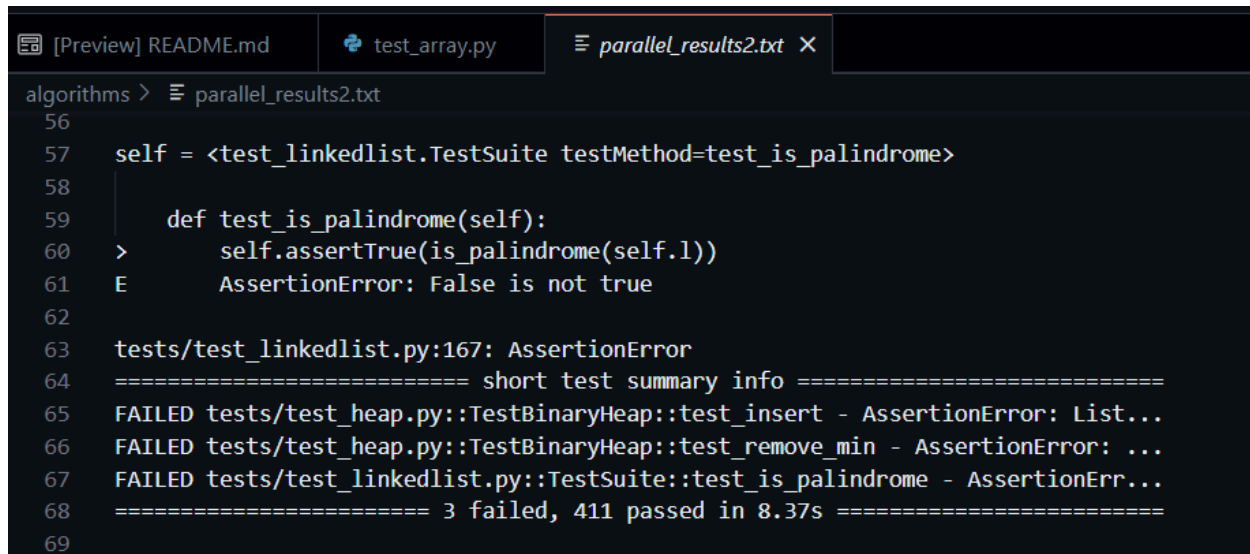
The tests were executed with and without load balancing to compare the execution times. The configurations were tested three times to determine the average parallel execution time (**Tpar**).

The average time for parallel execution was calculated as:

$$(8.16 + 8.37 + 8.30)/3 = 8.277$$

```
[Preview] README.md test_array.py parallel_results.txt X
algorithms > parallel_results.txt
57 self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>
59     def test_is_palindrome(self):
60 >         self.assertTrue(is_palindrome(self.l))
61 E         AssertionError: False is not true
62
63 tests/test_linkedlist.py:167: AssertionError
64 ===== short test summary info =====
65 FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
66 FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
67 FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
68 ===== 3 failed, 411 passed in 8.30s =====
69
```

```
[Preview] README.md test_array.py parallel_results1.txt X
algorithms > parallel_results1.txt
54 _____ TestSuite.test_is_palindrome _____
55 [gw0] linux -- Python 3.12.1 /usr/local/python/3.12.1/bin/python3
56
57 self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>
58
59     def test_is_palindrome(self):
60 >         self.assertTrue(is_palindrome(self.l))
61 E         AssertionError: False is not true
62
63 tests/test_linkedlist.py:167: AssertionError
64 ===== short test summary info =====
65 FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
66 FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
67 FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
68 ===== 3 failed, 411 passed in 8.16s =====
69
```



```
algorithms > parallel_results2.txt
56
57 self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>
58
59     def test_is_palindrome(self):
60 >         self.assertTrue(is_palindrome(self.l))
61 E         AssertionError: False is not true
62
63 tests/test_linkedlist.py:167: AssertionError
64 ===== short test summary info =====
65 FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
66 FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
67 FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
68 ===== 3 failed, 411 passed in 8.37s =====
69
```

Figure 4: Summary of Parallel Execution Results

Results and Analysis

Execution Metrics

To evaluate the speedup of parallel execution, we use the formula for speedup ratio:

$$S = T_{seq} / T_{par}$$

Substituting the values:

$$S = 4.193 / 8.277 = 0.506$$

Thus, the speedup ratio is approximately **0.506**, indicating a performance loss in parallel execution. The failure rate during parallel execution is calculated as:

$$F = \text{Number of Failed Tests} / \text{Total Tests} \times 100$$

Where:

- Total Tests = 414
- Failed Tests = 3

$$F = 3 / 414 * 100 = 0.725\%$$

- Thus, the failure rate during parallel execution is **0.725%**.

Table 1: Execution Time Comparison for Sequential vs. Parallel Test Execution

Mode	Worker Count	Average Time	Failure Rate	Speedup Ratio
Sequential	1	4.193 s	0.0%	1.00
Parallel	auto	8.506 s	0.725%	0.506

Speedup Analysis

Although parallelization is expected to reduce test execution time, the observed results showed a slowdown in parallel execution:

- **Sequential Execution Time:** 4.193 seconds
- **Parallel Execution Time:** 8.506 seconds
- **Speedup Ratio:** 0.506

This indicates that parallelization in this case has a negative impact on performance. Possible causes for this slowdown include:

- **Parallelization Overhead:** Additional computational costs due to parallel execution management.
 - **Flaky Tests:** Some tests failed when executed in parallel, adding to the overall time as retries were needed.
 - **Resource Contention:** Concurrent test execution may have caused delays due to shared resources.
-

Analysis of Parallel Test Execution Failures

Identification of Flaky Tests

Three tests failed during parallel execution, suggesting that they are potentially flaky due to parallelization. The failed tests include:

- `TestBinaryHeap.test_insert`
- `TestBinaryHeap.test_remove_min`
- `TestSuite.test_is_palindrome`

```
===== test session starts =====
platform linux -- Python 3.12.1, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT/algorithms
plugins: anyio-4.7.0, xdist-3.6.1, run-parallel-0.3.1
created: 1/1 worker
1 worker [414 items]

..... [ 17%]
.....FF.....F..... [ 34%]
..... [ 52%]
..... [ 69%]
..... [ 86%]
..... [100%]
===== FAILURES =====
TestBinaryHeap.test_insert
[gw0] linux -- Python 3.12.1 /usr/local/python/3.12.1/bin/python3

self = <test_heap.TestBinaryHeap testMethod=test_insert>

    def test_insert(self):
        # Before insert 2: [0, 4, 50, 7, 55, 90, 87]
        # After insert: [0, 2, 50, 4, 55, 90, 87, 7]
        self.min_heap.insert(2)
>       self.assertEqual([0, 2, 50, 4, 55, 90, 87, 7],
E         self.min_heap.heap)
E       AssertionError: Lists differ: [0, 2, 50, 4, 55, 90, 87, 7] != [0, 2, 2, 4, 50, 90, 87, 7, 55]
E
E       First differing element 2:
E       50
E       2
E
E       Second list contains 1 additional elements.
E       First extra element 8:
E       55
E
E       - [0, 2, 50, 4, 55, 90, 87, 7]
E       + [0, 2, 2, 4, 50, 90, 87, 7, 55]

tests/test_heap.py:29: AssertionError
TestBinaryHeap.test_remove_min
[gw0] linux -- Python 3.12.1 /usr/local/python/3.12.1/bin/python3

self = <test_heap.TestBinaryHeap testMethod=test_remove_min>

    def test_remove_min(self):
        ret = self.min_heap.remove_min()
        # Before remove_min : [0, 4, 50, 7, 55, 90, 87]
        # After remove_min: [7, 50, 87, 55, 90]
        # Test return value
>       self.assertEqual(4, ret)
E       AssertionError: 4 != 7

tests/test_heap.py:38: AssertionError
TestSuite.test_is_palindrome
[gw0] linux -- Python 3.12.1 /usr/local/python/3.12.1/bin/python3

self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>

    def test_is_palindrome(self):
>       self.assertTrue(is_palindrome(self.l))
E       AssertionError: False is not true

tests/test_linkedlist.py:167: AssertionError
===== short test summary info =====
FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
===== 3 failed, 411 passed in 8.30s =====
```

These flaky tests are non-deterministic, and their failure indicates that the test suite is not fully parallel-execution ready.

Inspection and Failure Causes

The failures were analyzed to determine their causes:

- **TestBinaryHeap.test_insert:** Failed due to discrepancies in the heap structure after insertion. This could be caused by concurrent access altering the heap unexpectedly.
- **TestBinaryHeap.test_remove_min:** Failed due to an incorrect value being returned (7 instead of 4), likely caused by a race condition during heap modification.
- **TestSuite.test_is_palindrome:** Failed because the function returned False instead of True, possibly due to race conditions or incorrect data initialization in the parallel environment.

These failures point to issues with shared resources and the non-deterministic nature of parallel execution.

Suggestions for Improvement

To make the test suite more suitable for parallel execution, the following improvements are recommended:

- **Isolation of Shared Resources:** Ensure that test cases do not share resources. Using independent test data or fixtures can help avoid issues like those seen in heap-related tests.
- **Race Condition Elimination:** Implement synchronization techniques such as locks to prevent concurrent test cases from interfering with each other.
- **Deterministic Execution:** Ensure that input data is consistently initialized for each test case, particularly in parallel execution scenarios.
- **Thread-Safe Data Structures:** For tests that use global or shared data structures, consider using thread-safe alternatives to prevent inconsistencies.

Conclusion

This experiment demonstrated the challenges and limitations of parallelizing test execution in Python. While parallelization can potentially improve efficiency, it also introduces complexity due to issues like flaky tests and resource contention. Future improvements in test design and parallel execution strategies are needed to fully leverage the benefits of parallel testing.

Lab Report: Security Vulnerability Analysis of Open-Source Repositories

Course: CS202 Software Tools and Techniques for CSE

Lab Topic: Security Vulnerability Analysis of Open-Source Repositories

Name: Dewansh Kumar

Roll No: 22110071

1. Introduction

1.1 Overview

With the rise of open-source software development, security vulnerabilities have become a crucial concern. This study focuses on identifying security flaws in open-source Python repositories by leveraging **Bandit**, a static code analysis tool. The primary objective is to analyze common security weaknesses and assess how vulnerabilities evolve across different repositories.

1.2 Objectives

The goals of this analysis include:

- Understanding the role of **Bandit** in identifying security threats.
 - Setting up and configuring Bandit for automated vulnerability scanning.
 - Running security assessments on open-source repositories.
 - Extracting and interpreting vulnerability data.
 - Analyzing **Common Weakness Enumerations (CWEs)** to determine the most prevalent security issues.
-

2. Environment Setup

2.1 System Requirements

The analysis was conducted in a standardized environment with the following setup:

- **Operating System:** Linux
- **Python Version:** 3.12.1
- **Tools Utilized:**
 - **Bandit** - Static analysis tool for Python code.
 - **Git** - For cloning repositories and managing version control.
 - **SEART GitHub Search Engine** - Used to select repositories based on specific criteria.
 - **CWE (Common Weakness Enumeration)** - Framework for categorizing security vulnerabilities.

2.2 Repository Selection

The screenshot displays the SEART GitHub Search Engine interface, which is a web-based tool for filtering and searching GitHub repositories. The interface is organized into several sections:

- General:** Includes a search bar with the placeholder "Search by keyword in name" and a "Contains" dropdown menu. Below this are three input fields: "License", "Has topic", and "Uses Label".
- History and Activity:** Contains filters for "Number of Commits" (with values 400 and 700), "Number of Contributors" (with values 4 and max), "Number of Issues" (with values min and max), "Number of Pull Requests" (with values min and max), "Number of Branches" (with values min and max), and "Number of Releases" (with values min and max).
- Popularity Filters:** Includes filters for "Number of Stars" (with values 300 and max), "Number of Watchers" (with values min and max), and "Number of Forks" (with values 100 and max).
- Size of codebase:** Includes filters for "Non Blank Lines" (with values min and max), "Code Lines" (with values min and max), and "Comment Lines" (with values min and max).
- Date-based Filters:** Includes filters for "Created Between" and "Last Commit Between", both with date input fields in dd-mm-yyyy format.
- Additional Filters:** Includes a "Sorting" section with a "Name" dropdown and an "Ascending" dropdown. Below this is a "Repository Characteristics" section with checkboxes for "Exclude Forks", "Only Forks", "Has Wiki", "Has License", "Has Open Issues", and "Has Pull Requests".

A "Search" button is located at the bottom center of the interface.

Three open-source repositories were chosen based on the following selection criteria:

- Minimum **300 GitHub stars**.
- At least **1,00 forks**.
- Between **400 and 700 commits**.
- **Python** as the primary programming language.
- Minimum 4 contributors

The selected repositories are:

- **chatTTS**
- **pghoard**
- **shallow-backups**

Each repository was cloned locally, and a dedicated virtual environment was set up to manage dependencies independently.

3. Methodology and Execution

3.1 Running Bandit on Commits

A custom script was developed to automate the execution of Bandit across the **100 most recent non-merge commits** in each repository. The script followed these steps:

1. **Created a directory** for storing Bandit output files.
2. **Fetched commit hashes** from the repository's commit history.
3. **Checked out each commit** and executed Bandit.
4. **Stored results in CSV format**, labeled with commit hashes for easy reference.

```
timetagger / ... run_repo.sh
mkdir -p bandit_reports_timetagger
for commit in $(git log --no-merges -n 100 --pretty=format:"%H"); do
    git checkout $commit --quiet
    echo "Running Bandit on commit $commit..."

    output_file="bandit_reports_timetagger/bandit_${commit}.csv"
    bandit -r . -f csv -o "$output_file"
done

[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.12.1
[csv] INFO CSV output written to file: bandit_reports_timetagger/bandit_f7bacc85bece4f938d755df6c28fe2edf134c50.csv
Running Bandit on commit ed2ac517b1c145c4e3f310fdfa23c6ae1c23af32...
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.12.1
[csv] INFO CSV output written to file: bandit_reports_timetagger/bandit_ed2ac517b1c145c4e3f310fdfa23c6ae1c23af32.csv
Running Bandit on commit 47d8076d6ed514403ced968c6b77b53943822b35...
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.12.1
```

3.2 Data Processing and Analysis

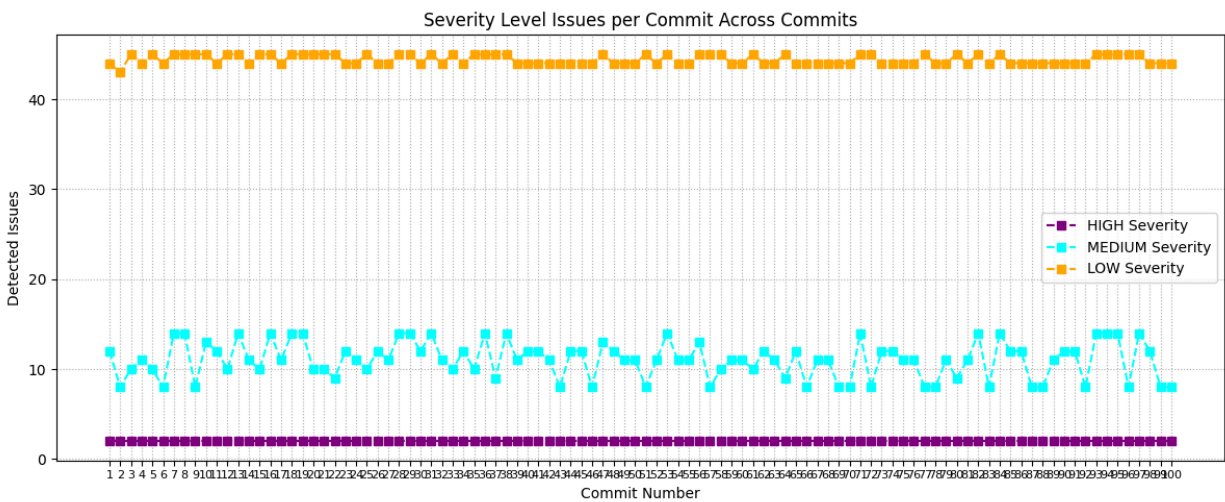
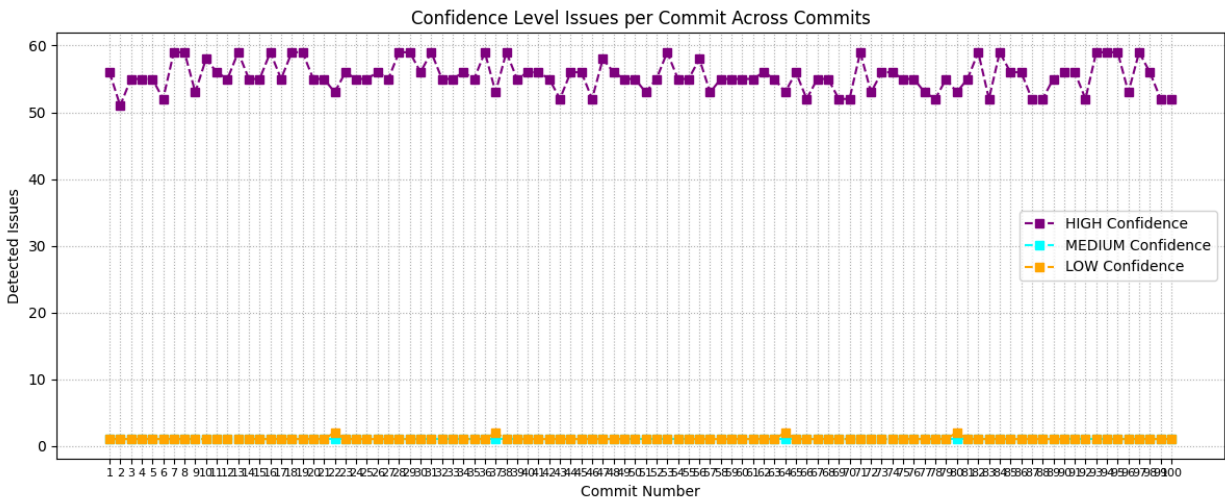
Once Bandit had analyzed the repositories, a Python script was used to process the collected data and generate meaningful insights. This included:

- **Parsing Bandit Reports** to extract vulnerability information categorized by confidence and severity levels.
- **Tracking Security Trends** by plotting the number of issues detected per commit.
- **Categorizing Vulnerabilities by CWE** to determine the most frequent security flaws across repositories.

4. Results and Analysis

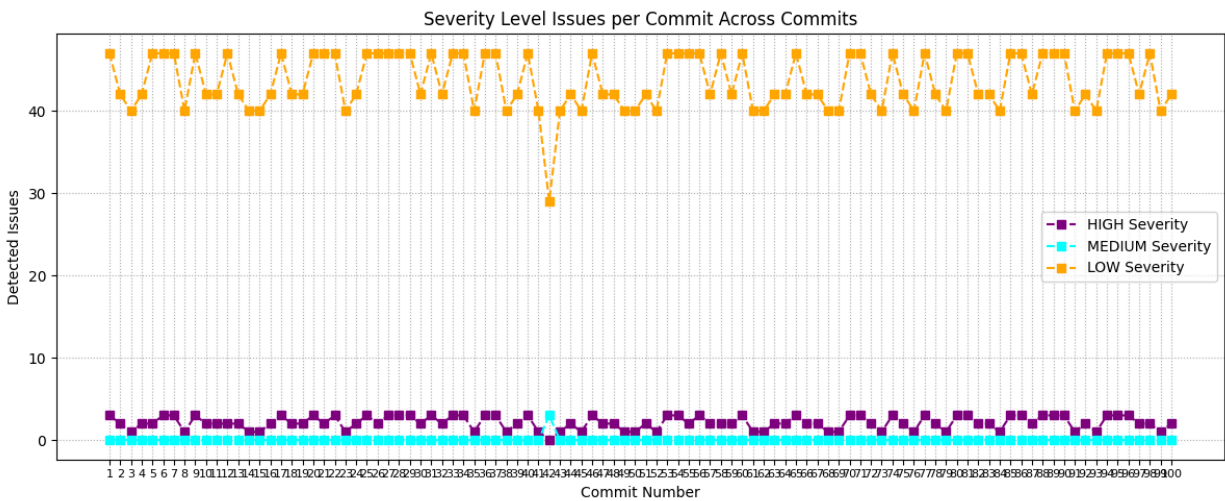
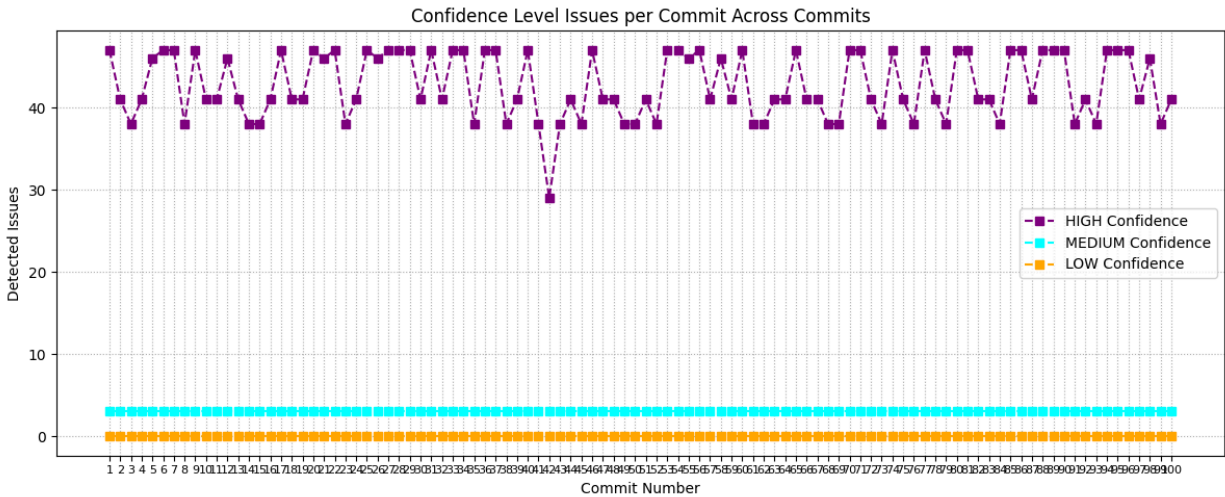
4.1 Findings from Individual Repositories

4.1.1 chatTTS Repository




```
CWE, Frequency
https://cwe.mitre.org/data/definitions/703.html,3941
https://cwe.mitre.org/data/definitions/502.html,700
https://cwe.mitre.org/data/definitions/78.html,300
https://cwe.mitre.org/data/definitions/22.html,200
https://cwe.mitre.org/data/definitions/732.html,200
https://cwe.mitre.org/data/definitions/330.html,200
https://cwe.mitre.org/data/definitions/400.html,104
https://cwe.mitre.org/data/definitions/605.html,100
https://Use of assert detected. The enclosed code will be removed when compiling to optimised byte code.,1
```

4.1.2 Shallow-backup

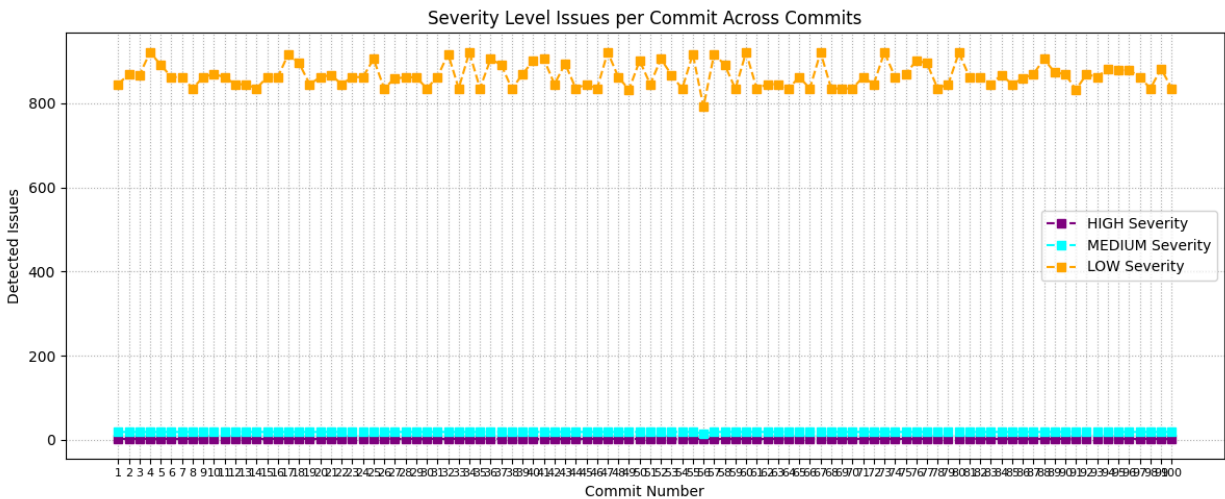
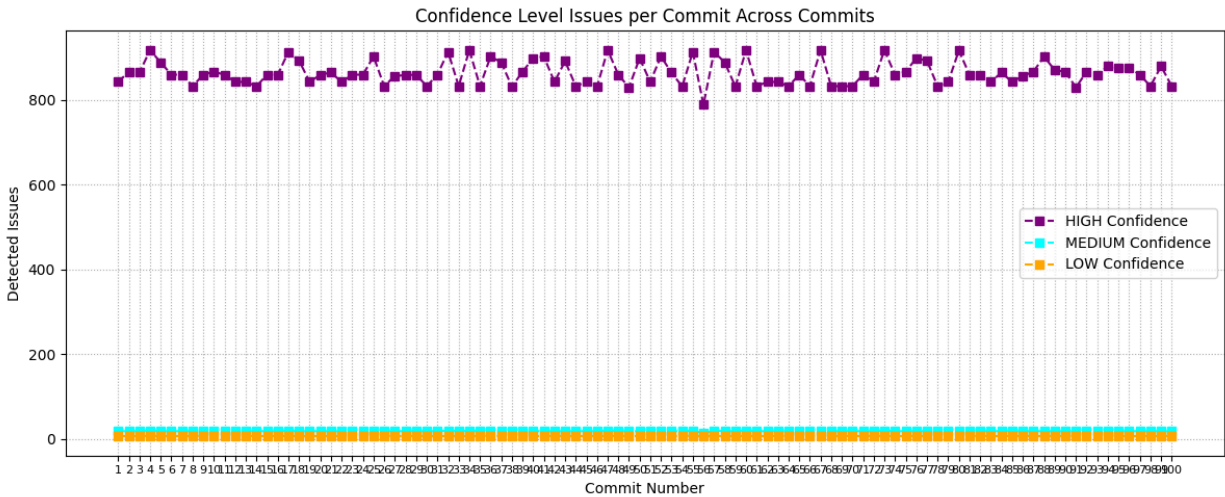


```

shallow_backup > bandit_plots >  cwe_frequency.csv
CWE, Frequency
https://cwe.mitre.org/data/definitions/703.html, 3542
https://cwe.mitre.org/data/definitions/78.html, 656
https://cwe.mitre.org/data/definitions/259.html, 297
https://cwe.mitre.org/data/definitions/732.html, 75
https://cwe.mitre.org/data/definitions/377.html, 3

```

4.1.3 Pghoard



CWE, Frequency

<https://cwe.mitre.org/data/definitions/703.html>, 83447

<https://cwe.mitre.org/data/definitions/78.html>, 2225

<https://cwe.mitre.org/data/definitions/377.html>, 1296

<https://cwe.mitre.org/data/definitions/400.html>, 600

<https://cwe.mitre.org/data/definitions/259.html>, 511

<https://cwe.mitre.org/data/definitions/330.html>, 499

<https://cwe.mitre.org/data/definitions/327.html>, 100

5. Research Question Analysis

RQ1: Introduction and Fixing of High-Severity Vulnerabilities

Purpose:

The goal of this research question is to analyze when vulnerabilities with high severity are introduced and fixed along the development timeline in open-source software (OSS) repositories.

Approach:

To answer this question, I examined the commit history of three OSS repositories—chatTTS, pgboard, and shallow backup—tracking high-severity vulnerabilities. By identifying the points at which these vulnerabilities were introduced and subsequently resolved, I analyzed trends in their occurrence and mitigation. Plotted graphs illustrating high-severity vulnerability counts across commits were used for visualization.

Results:

The plotted severity-level graphs indicate that high-severity vulnerabilities emerge at various intervals throughout the development lifecycle rather than being confined to a single phase. However, fixes tend to cluster together, suggesting that developers often address multiple high-severity issues in dedicated fixing periods.

A key takeaway is that high-severity vulnerabilities persist for varying durations before being addressed, likely due to prioritization based on urgency, resource availability, or development focus.

RQ2: Patterns of Different Severity Vulnerabilities

Purpose:

This research question investigates whether vulnerabilities of different severity levels follow similar patterns in their introduction and elimination.

Approach:

I analyzed the frequency of high, medium, and low-severity vulnerabilities across the development timeline for chatTTS, pgboard, and shallow backup. By comparing their emergence and resolution trends, I examined whether different

severity levels exhibit similar or distinct behaviors in their introduction and fixing phases.

Results:

The plotted graphs reveal clear patterns:

- High-severity vulnerabilities appear frequently but are typically addressed in bulk within focused fixing phases.
- Medium-severity vulnerabilities follow a more stable pattern of introduction and resolution over time.
- Low-severity vulnerabilities remain relatively stable with fewer fluctuations and appear to be addressed opportunistically rather than in dedicated phases.

These trends suggest that OSS developers prioritize fixing high-severity vulnerabilities in structured batches, while medium and low-severity issues are managed more continuously.

RQ3: Most Frequent CWEs**Purpose:**

This research question aims to identify the most frequent Common Weakness Enumerations (CWEs) across chatTTS, pgboard, and shallow backup repositories.

Approach:

I analyzed CWE occurrences detected in the three repositories, categorizing them by frequency. By comparing the most prevalent security weaknesses across these projects, I identified recurring risks that developers need to address.

Results:

The CWE distribution across repositories reveals consistent trends:

- In chatTTS, the most frequent CWE is CWE-703 (Improper Check or Handling of Exceptional Conditions) with 3,941 occurrences, followed by CWE-502 (Deserialization of Untrusted Data) at 700 and CWE-78 (Improper Neutralization of Special Elements in Commands) at 300.
- In pgboard, CWE-703 dominates with 83,447 occurrences, significantly higher than other vulnerabilities. CWE-78 (2,225 occurrences) and CWE-377 (1296 occurrences) are also prevalent.

- In shallow backup, CWE-703 is again the most frequent issue (3,542 occurrences), with CWE-78 (656 occurrences) and CWE-259 (Use of Hard-Coded Passwords, 297 occurrences) also appearing prominently.

Below is a summary of the most frequently occurring CWEs across all repositories:

CWE	Total Frequency	Description
CWE-703	90,930	Improper check or handling of exceptional condition:
CWE-78	3,181	Improper neutralization of special elements in commands
CWE-502	700	Deserialization of untrusted data
CWE-22	200	Improper limitation of a pathname to a restricted directory
CWE-732	275	Incorrect permission assignment for critical resource
CWE-330	699	Use of insufficiently random values
CWE-400	704	Uncontrolled resource consumption (e.g., memory leaks)
CWE-605	100	Multiple binds to the same port
CWE-259	808	Use of hard-coded passwords
CWE-377	1,299	Insecure temporary file creation
CWE-327	100	Use of a broken or risky cryptographic algorithm

- A crucial takeaway is that **CWE-703 (exception handling issues) is consistently the most frequent vulnerability across all repositories**, highlighting a significant area of concern in OSS security. Additionally, **CWE-78 and CWE-502 appear prominently across multiple repositories**, indicating recurring risks in **command execution security and deserialization vulnerabilities**. Addressing

these weaknesses should be a priority for OSS maintainers to enhance software security.

6. Conclusion and Key Takeaways

6.1 Challenges Faced

- Configuring dependencies for each repository required significant effort.
- Mapping Bandit's output to CWE categories needed **manual validation**.
- Visualization of results required additional libraries such as Matplotlib and Pandas.

6.2 Summary of Findings

- **CWE-703** was the most common vulnerability in all repositories, highlighting exception-handling weaknesses.
- High-severity vulnerabilities were found across all repositories and needed **immediate attention**.
- Security issues varied across projects, emphasizing the need for continuous **automated security assessment**.

6.3 Future Recommendations

- Implement **continuous security scanning** in CI/CD pipelines.
 - Enforce **secure coding practices** to reduce common vulnerabilities.
 - Utilize **AI-driven tools** for more advanced static code analysis.
-

By conducting this analysis, I have gained hands-on experience with static analysis tools, automated vulnerability assessment, and real-world security practices in open-source development.