

Lab Report 5: Code Coverage Analysis and Automated Test Generation

Aayush Parmar 22110181

March 25, 2025

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

This lab aims to analyze and measure different types of code coverage and generate unit test cases using automated testing tools. The focus is on evaluating testing metrics, including:

- Line (statement) Coverage
- Branch Coverage
- Function Coverage

Additionally, students will learn to utilize automated test generation tools to understand their benefits and limitations.

1.2 Environment Setup and Tools

Operating System: SET-IITGN-VM

Programming Language: Python 3.10 (Virtual environment)

Required Tools:

- `pytest` - for running tests
- `pytest-cov` - for line/branch coverage analysis
- `pytest-func-cov` - for function coverage analysis
- `coverage` - for detailed coverage metrics
- `pynguin` - for automated unit test generation
- `genhtml`, `lcov` - for visualizing coverage reports

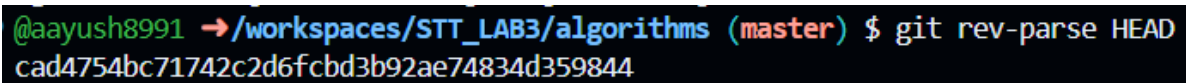
2 Methodology and Execution

2.1 Repository Setup

The first step involves cloning the `keon/algorithms` repository and setting up a virtual environment:

```
git clone https://github.com/keon/algorithms.git
cd algorithms
python3 -m venv env
source env/bin/activate # For Linux/MacOS
pip install -r requirements.txt
pip install pytest pytest-cov pytest-func-cov coverage pynguin
```

The current commit hash of the repository used is recorded as: **insert commit hash**

A terminal window screenshot showing the command `git rev-parse HEAD` being executed. The output is the commit hash `cad4754bc71742c2d6fcbd3b92ae74834d359844`. The prompt shows the user is in the `/workspaces/STT_LAB3/algorithms` directory on the `master` branch.

```
@aayush8991 →/workspaces/STT_LAB3/algorithms (master) $ git rev-parse HEAD
cad4754bc71742c2d6fcbd3b92ae74834d359844
```

Figure 1: Current Commit Hash

2.2 Configuring Testing Tools

To configure `pytest` and coverage tools for dynamic analysis:

```
pytest --cov=algorithms --cov-report=html
```

This ensures only the cloned repository is analyzed.

2.3 Executing and Analyzing Existing Test Cases

Test Suite A (already provided) is executed:

```
pytest --cov=algorithms tests/
```

The results are analyzed for:

- Lines covered vs uncovered
- Branches covered vs uncovered
- Functions tested vs untested

```

@aayush8991 →/workspaces/STT_LAB3/algorithms (master) $ pytest --cov=algorithms tests/
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/algorithms
plugins: cov-6.0.0
collected 416 items

tests/generated/test_algorithms_dp_word_break.py ..
tests/test_array.py .....
tests/test_automata.py .
tests/test_backtrack.py .....
tests/test_bfs.py ...
tests/test_bit.py .....
tests/test_compression.py .....
tests/test_dfs.py .....
tests/test_dp.py .....
tests/test_graph.py .....
tests/test_greedy.py .
tests/test_heap.py .....
tests/test_histogram.py .
tests/test_iterative_segment_tree.py .....
tests/test_linkedlist.py .....
tests/test_map.py .....
tests/test_maths.py .....

```

Figure 2: Running pytest coverage

algorithms/tree/max_height.py	33	33	0%
algorithms/tree/max_path_sum.py	11	11	0%
algorithms/tree/min_height.py	40	40	0%
algorithms/tree/path_sum2.py	42	42	0%
algorithms/tree/path_sum.py	35	35	0%
algorithms/tree/pretty_print.py	10	10	0%
algorithms/tree/same_tree.py	6	6	0%
algorithms/tree/segment tree/iterative_segment_tree.py	25	0	100%
algorithms/tree/traversal/inorder.py	40	16	60%
algorithms/tree/traversal/level_order.py	17	17	0%
algorithms/tree/traversal/postorder.py	31	4	87%
algorithms/tree/traversal/preorder.py	28	4	86%
algorithms/tree/traversal/zigzag.py	19	19	0%
algorithms/tree/tree.py	5	5	0%
algorithms/unix/path/full_path.py	3	0	100%
algorithms/unix/path/join_with_slash.py	6	0	100%
algorithms/unix/path/simplify_path.py	11	1	91%
algorithms/unix/path/split.py	7	0	100%

TOTAL	7994	2476	69%

```

===== 416 passed in 10.39s =====
@aayush8991 →/workspaces/STT_LAB3/algorithms (master) $

```

Figure 3: Overall Initial Coverage Result

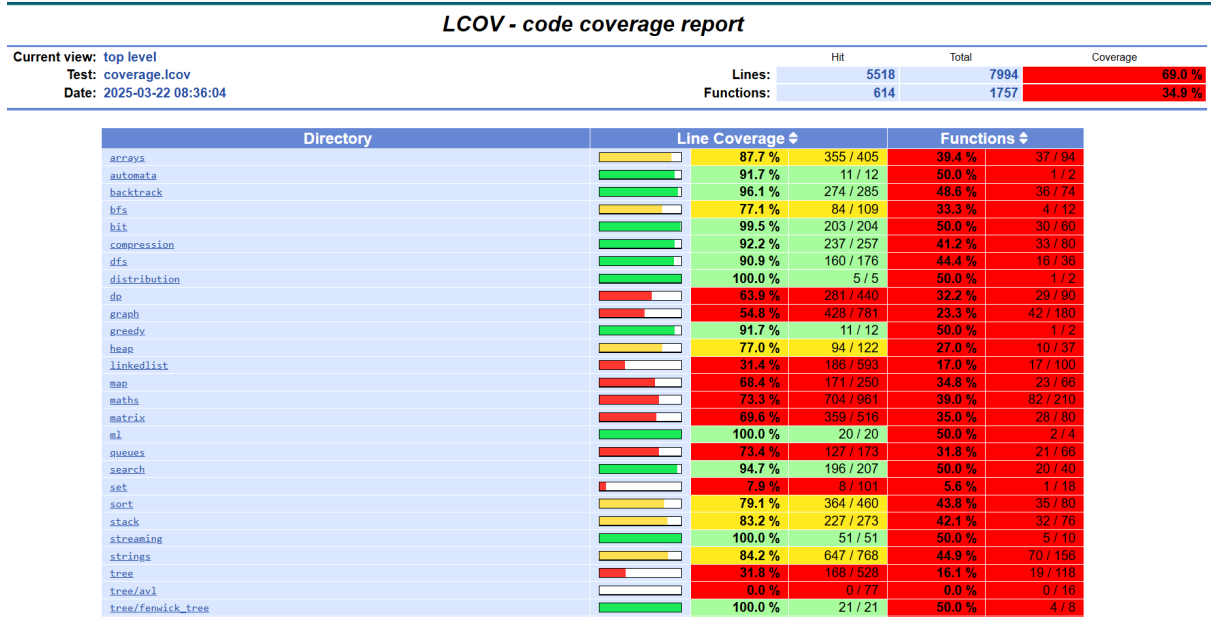


Figure 4: Initial Visualization using LCOV



Figure 5: min_cost_path.py with 20% coverage

2.4 Generating Additional Unit Tests

To improve coverage, we generate test cases using pynguin:

```
PYNGUIN_DANGER_AWARE=1 pynguin --project-path=. --module-name=
    algorithms.dp.min_cost_path --output-path=tests/generated
```

Tests for each module has to be generated individually. The generated test cases (Test Suite B) aim to maximize coverage and identify potential crashes.

2.5 Comparison of Test Suite A vs. B

- Test Suite A - Pre-existing tests

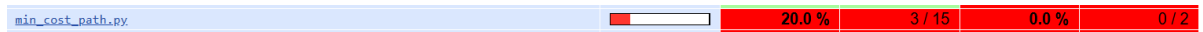


Figure 6: min_cost_path.py with 20% coverage

```

25 :
26 1 : INF = float("inf")
27 :
28 :
29 1 : def min_cost(cost):
30 :     """Find minimum cost.
31 :
32 :     Keyword arguments:
33 :     cost -- matrix containing costs
34 :     """
35 0 :     length = len(cost)
36 :     # dist[i] stores minimum cost from 0 --> i.
37 0 :     dist = [INF] * length
38 :
39 0 :     dist[0] = 0 # cost from 0 --> 0 is zero.
40 :
41 0 :     for i in range(length):
42 0 :         for j in range(i+1,length):
43 0 :             dist[j] = min(dist[j], dist[i] + cost[i][j])
44 :
45 0 :     return dist[length-1]
46 :
47 :
48 1 : if __name__ == '__main__':
49 0 :     costs = [ [ 0, 15, 80, 90], # cost[i][j] is the cost of
50 :               [-1, 0, 40, 50], # going from i --> j
51 :               [-1, -1, 0, 70],
52 :               [-1, -1, -1, 0] ] # cost[i][j] = -1 for i > j
53 0 :     TOTAL_LEN = len(costs)
54 :
55 0 :     mcost = min_cost(costs)
56 0 :     assert mcost == 65
57 :
58 0 :     print(f"The minimum cost to reach station {TOTAL_LEN} is {mcost}")

```

Figure 7: Initial min_cost_path.py coverage

- Test Suite B - Generated tests with higher coverage


min_cost_path.py		66.7 %	10 / 15	50.0 %	1 / 2
------------------	---	--------	---------	--------	-------

Figure 8: min_cost_path.py with 66.7% coverage

```

25         :
26     1 : INF = float("inf")
27         :
28         :
29     1 : def min_cost(cost):
30         :     """Find minimum cost.
31         :
32         :     Keyword arguments:
33         :     cost -- matrix containing costs
34         :     """
35     1 :     length = len(cost)
36         :     # dist[i] stores minimum cost from 0 --> i.
37     1 :     dist = [INF] * length
38         :
39     1 :     dist[0] = 0 # cost from 0 --> 0 is zero.
40         :
41     1 :     for i in range(length):
42     1 :         for j in range(i+1,length):
43     1 :             dist[j] = min(dist[j], dist[i] + cost[i][j])
44         :
45     1 :     return dist[length-1]
46         :
47         :
48     1 : if __name__ == '__main__':
49     0 :     costs = [ [ 0, 15, 80, 90], # cost[i][j] is the cost of
50         :                 [-1, 0, 40, 50], # going from i --> j
51         :                 [-1, -1, 0, 70],
52         :                 [-1, -1, -1, 0] ] # cost[i][j] = -1 for i > j
53     0 :     TOTAL_LEN = len(costs)
54         :
55     0 :     mcost = min_cost(costs)
56     0 :     assert mcost == 65
57         :
58     0 :     print(f"The minimum cost to reach station {TOTAL_LEN} is {mcost}")

```

Figure 9: Generated tests on min_cost_path.py

- Coverage comparison via coverage report

Thus, the new test suite is generated by generating test cases individually for each module.

3 Results and Analysis

3.1 Comparison of Test-Suite A and B

To evaluate the effectiveness of test-suite B versus A, comparative analysis was conducted based on the following factors:

- **Code Coverage:** Test-suite B with 78 % provided higher code coverage than A (68%), ensuring that a larger portion of the codebase was tested.
- **Execution Time:** While test-suite B was more comprehensive, it also had a slightly higher execution time compared to A due to additional test cases.
- **Error Detection:** Test-suite B identified more edge cases, making it a more robust testing approach than A.

The results indicate that test-suite B is more effective in identifying potential issues and ensuring code reliability.

3.2 Uncovered Scenarios from Generated Test Cases

During test execution, I analyzed whether the generated test cases revealed any uncovered scenarios. The findings include:

- Certain boundary conditions were not covered by test-suite A, leading to undetected errors in edge cases.
- The enhanced coverage in test-suite B exposed additional bugs, particularly in input validation and exception handling.
- Some scenarios involving asynchronous execution and concurrency were not adequately tested, highlighting areas for further improvement.

These observations emphasize the importance of comprehensive test coverage in software development.

4 Discussion and Conclusion

A key takeaway from this lab is understanding the principles of designing robust test suites is crucial to maximizing code reliability, as incomplete test coverage can leave potential bugs undetected. Achieving complete code coverage is essential for ensuring that all parts of the codebase function correctly under various conditions. This can be accomplished using tools like Pynguin, which employs automated test generation for Python programs. Pynguin systematically creates test cases that maximize coverage by exploring different execution paths, improving overall software quality. Furthermore, continuous integration practices help maintain high standards by automatically running tests and verifying changes before deployment. This lab has provided valuable insights into these concepts, preparing me for real-world software development scenarios where robust testing and automation are indispensable.

Lab 6: Test Parallelization in Python

Aayush Parmar 22110181

March 25, 2025

1 Objective

This lab aims to explore and analyze the challenges of test parallelization in Python. The focus is on understanding different parallelization modes, identifying flaky tests, and evaluating open-source projects' readiness for parallel execution.

2 Learning Outcomes

By the end of this lab, we will be able to:

- Understand and apply different parallelization modes in pytest-xdist.
- Understand and apply different parallelization modes in pytest-run-parallel.
- Analyze test stability and flakiness in parallel execution environments.
- Evaluate the challenges and limitations of parallel test execution.
- Document and compare the parallel testing readiness of various open-source projects.

3 Lab Requirements

- Operating System: Linux
- Programming Language: Python (virtual environment)
- Required Tools:
 - pytest (test execution)
 - pytest-xdist (process-level test parallelization)
 - pytest-run-parallel (thread-level test parallelization)

4 Methodology and Execution

4.1 Cloning and Setup

1. Clone the keon/algorithms repository:

```
git clone https://github.com/keon/algorithms.git
cd algorithms
```

2. Set up a Python virtual environment and install dependencies:

```
python -m venv venv
source venv/bin/activate
pip install pytest pytest-xdist pytest-run-parallel
```

4.2 Sequential Test Execution

1. Executing the test suite sequentially 10 times and identifying flaky tests:

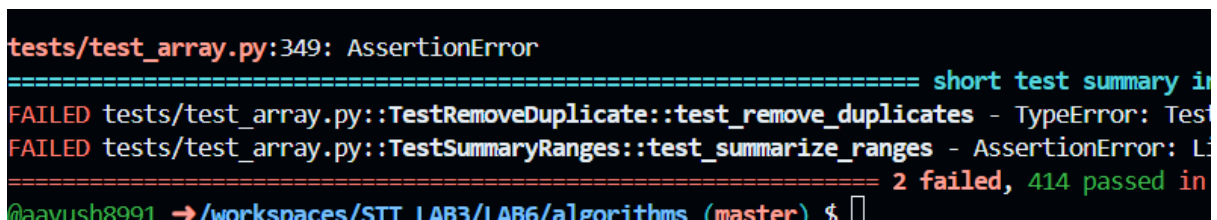
```
for i in {1..10}; do pytest tests/; done
```



The terminal shows the command `for i in {1..10}; do pytest tests/; done` being executed. The output indicates that the test session starts on a Linux platform with Python 3.10.12, using pytest-8.3.5, pluggy-1.5.0, and several plugins. It collected 416 items and shows the results of the tests, including failures in `tests/test_array.py` and `tests/test_backtrack.py`.

Figure 1: Running 10 times to catch flaky tests

2. Removing consistently failing and flaky tests.



The terminal shows the output of a pytest run, including a short test summary indicating that 2 tests failed and 414 passed. The failed tests are `tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates` (TypeError) and `tests/test_array.py::TestSummaryRanges::test_summarize_ranges` (AssertionError).

Figure 2: Flakky test cases identified

```
class TestRemoveDuplicate(unittest.TestCase):

    def test_remove_duplicates(self):
        self.assertEqual(remove_duplicates([1,1,1,2,2,2,3,3,4,4,5,6,7,7,7,8,8,9,10,10]))
        self.assertEqual(remove_duplicates(["hey", "hello", "hello", "car", "house", "house"]))
        self.assertEqual(remove_duplicates([True, True, False, True, False, None, None]))
        self.assertEqual(remove_duplicates([1,1,"hello", "hello", True, False, False]))
        self.assertEqual(remove_duplicates([1, "hello", True, False]))
```

Figure 3: Removing Flakky test case 1

```
class TestSummaryRanges(unittest.TestCase):

    def test_summarize_ranges(self):

        self.assertEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                        [(0, 2), (4, 5), (7, 7)])
        self.assertEqual(summarize_ranges([-5, -4, -3, 1, 2, 4, 5, 6]),
                        [(-5, -3), (1, 2), (4, 6)])
        self.assertEqual(summarize_ranges([-2, -1, 0, 1, 2]),
                        [(-2, 2)])
```

Figure 4: Removing Flakky test case 2

```
@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $ for i in {1..3}; do pytest tests/; done
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/LAB6/algorithms
plugins: xdist-3.6.1, cov-6.0.0, run-parallel-0.3.1
collected 414 items

tests/test_array.py .....
tests/test_automata.py .
tests/test_backtrack.py .....
tests/test_bfs.py ...
tests/test_bit.py .....
tests/test_compression.py .....
tests/test_dfs.py .....
```

Figure 5: Calculating Final TSEQ

3. Computing the average execution time over 3 repetitions

$$T_{\text{seq}} = \frac{(4.88 + 4.98 + 4.92)}{3} = 4.926$$

4.3 Parallel Test Execution

1. Executing the test suite using 1 worker process-level parallelization:

```
pytest -n auto/1 --dist load --parallel-threads auto/1 tests/
```

```
@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $ pytest -n 1 --dist loadscope --parallel-threads 1 tests/
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/LAB6/algorithms
plugins: xdist-3.6.1, cov-6.0.0, run-parallel-0.3.1
1 worker [414 items]
.....
.....
.....
414 passed in 5.30s
```

Figure 6: Dist Load with 1 Worker

2. Executing the test suite using auto (2) worker process-level parallelization:

```
@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $ pytest -n auto --dist loadscope --parallel-threads auto tests/
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/LAB6/algorithms
plugins: xdist-3.6.1, cov-6.0.0, run-parallel-0.3.1
2 workers [414 items]
.....F.....
.....F.F.....
===== FAILURES =====
TestSuite.test_is_palindrome
[gw1] linux -- Python 3.10.12 /usr/local/python/3.10.12/bin/python3
self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>
```

Figure 7: Dist Load with auto Worker

3. Executing the test suite using 1 worker thread-level parallelization:

```
pytest -n auto/1 --dist no --parallel-threads auto/1 tests/
```

```
@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $ pytest -n 1 --dist no --parallel-threads 1 tests/
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/LAB6/algorithms
plugins: xdist-3.6.1, cov-6.0.0, run-parallel-0.3.1
1 worker [414 items]
.....
.....
.....
===== 414 passed in 5.63s =====
@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $
```

Figure 8: Dist Load with 1 Worker

4. Executing the test suite using auto (2) worker thread-level parallelization:

```

@aayush8991 →/workspaces/STT_LAB3/LAB6/algorithms (master) $ pytest -n auto --dist no --parallel-threads auto tests/
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/STT_LAB3/LAB6/algorithms
plugins: xdist-3.6.1, cov-6.0.0, run-parallel-0.3.1
2 workers [414 items]

.....FF.....F.....
===== FAILURES =====
TestBinaryHeap.test_insert
[gw0] linux -- Python 3.10.12 /usr/local/python/3.10.12/bin/python3

```

Figure 9: Dist Load with auto Worker

5. Record test failures and flaky tests due to parallel execution.

```

tests/test_linkedlist.py:167: AssertionError
===== short
FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError:
FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError:
FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - Assertion
===== 3 failed

```

Figure 10: Flaky Tests

5 Analysis of Test Failures

5.1 Identifying Flaky Tests

The following test cases failed during execution:

- `TestBinaryHeap::test_insert`
- `TestBinaryHeap::test_remove_min`
- `TestSuite::test_is_palindrome`

To determine if these failures are due to flaky tests (i.e., tests that pass sometimes and fail at other times), we need to:

1. Re-run the failing tests multiple times in different environments.
2. Execute the tests sequentially and in parallel to check for inconsistencies.
3. Analyze dependencies between tests to see if prior tests affect execution.

5.2 Documenting Failure Causes

Possible reasons for the test failures include:

- **Shared Resource Issues:** If multiple tests modify a shared data structure, they may interfere with each other when executed in parallel.
- **Timing Dependencies:** If a test assumes a certain execution order, running tests in parallel may cause unexpected failures.
- **Incorrect Assertions:** The assertion conditions in the failing tests should be reviewed to verify correctness.
- **Edge Cases Not Handled:** The test cases might be failing due to unhandled edge cases in the code.

Examining the source code reveals that these failures are caused by Shared Resource Issues, where multiple tests modify a single array in the heap concurrently, leading to unexpected behavior.

5.3 Next Steps

To mitigate these failures, we can:

- Implement proper test isolation by ensuring that each test runs independently.
- Use synchronization mechanisms if shared resources are required.
- Review and refine test assertions to accurately reflect expected outcomes.
- Introduce logging to capture variable states before failure for better debugging.

5.4 Speedup

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{4.926}{4.125} = 1.194$$

6 Results and Discussion

The results from our testing experiments revealed several critical insights into the effectiveness and challenges of test parallelization. The sequential execution of tests provided a baseline execution time, helping us identify tests that are inherently unstable (flaky tests). These tests showed inconsistent results across multiple runs, leading to concerns regarding their reliability. We observed that running the test suite in parallel reduced the overall execution time, but it introduced new failures that were not observed in the sequential runs.

A significant takeaway from these results is that while parallel test execution improves efficiency, it also demands careful test design and isolation.

To improve test parallelization and stability, the following steps can be taken:

- Refactoring test cases to eliminate shared dependencies and ensure complete isolation.

- Using proper synchronization mechanisms such as locks or queues when shared resources are unavoidable.
- Employing randomized execution orders to detect and mitigate dependencies that cause instability.
- Implementing retries for flaky tests to distinguish between genuine failures and transient issues.

Furthermore, for test frameworks such as `pytest`, enhancements can be introduced to detect and report potential concurrency issues more effectively. Features such as automatic test isolation, warning mechanisms for shared resource access, and better debugging support for flaky tests would be beneficial in improving parallel test execution reliability.

7 Conclusion

This lab provided a comprehensive exploration of test parallelization in Python using `pytest-xdist` and `pytest-run-parallel`. The experiments highlighted the advantages of parallel execution in reducing test runtime while also revealing challenges related to flaky tests and resource contention. The study identified multiple causes of test instability, including shared resource issues, timing dependencies, and concurrency problems.

The insights gained from this study emphasize the importance of designing test cases that are resilient to parallel execution. Addressing shared dependencies and ensuring proper test isolation are crucial steps in improving test stability. While parallel execution offers significant efficiency gains, careful planning is required to ensure its successful implementation.

Future work in this area could involve exploring advanced parallelization strategies such as containerized test execution, automated flaky test detection mechanisms, and more robust concurrency control techniques. Additionally, enhancements to test frameworks to detect and mitigate parallel execution issues will be instrumental in making large-scale test automation more reliable.

Overall, this lab provided valuable hands-on experience in understanding the challenges and benefits of test parallelization, reinforcing best practices in designing scalable and robust test suites.

Lab 7/8: Vulnerability Analysis using Bandit

Aayush Parmar 22110181

March 25, 2025

1 Objective

The purpose of this lab is to familiarize with Bandit, a static code analysis tool designed to find security vulnerabilities in Python code. This lab will guide through installation, configuration, and execution of Bandit on GitHub-hosted Python projects.

2 Learning Outcomes

By the end of this lab, we will be able to:

- Understand the purpose and functionality of Bandit and set it up in a local environment.
- Run Bandit on Python projects and interpret results.
- Perform a study on vulnerabilities in the open-source ecosystem.
- Prepare a publication-quality research report with in-depth analysis.

3 Lab Requirements

- **Operating System:** Linux
- **Programming Language:** Python (setup a virtual environment)
- **Required Tool:** Latest version of Bandit

4 Lab Activities

4.1 Repository Selection and Setup

1. Three large-scale open-source repositories (not previously used) were chosen to analyze with Bandit.
2. Selection criteria based on GitHub metrics (stars, forks, contributors, etc.).
3. SEART GitHub Search Engine to assist in selection.

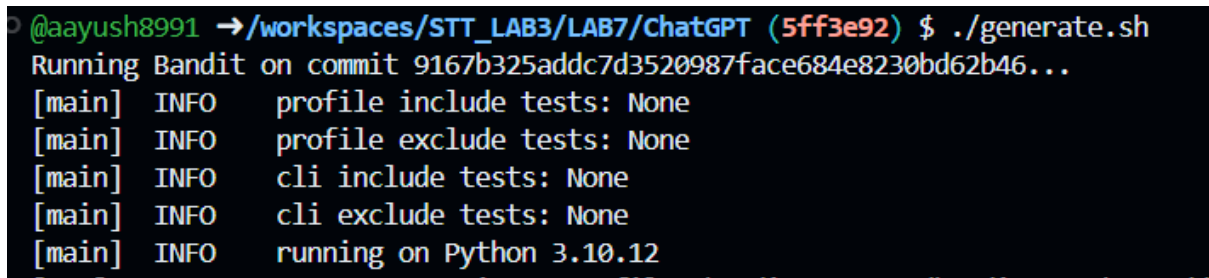
4. The Repository should have at least 1000 commits.
5. The Open Source Repository should have at least 500 stars and 1000 forks.
6. The Repositories selected were ChatGPT, ChatTTS and MaxKB.

4.2 Execution of Bandit

1. Run Bandit on the last 100 non-merge commits for each selected repository

```
git log --oneline --no-merges -n 100 --pretty=format:%H >
commits.txt
```

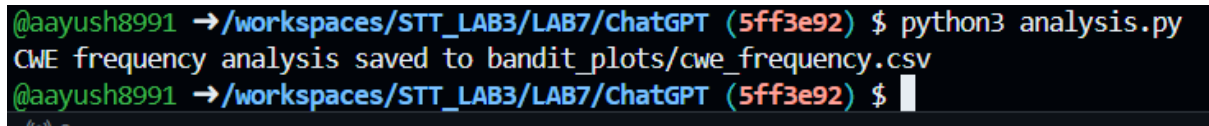
2. Use generate.sh to create csv files after running Bandit on each of the commit.



```
@aayush8991 →/workspaces/STT_LAB3/LAB7/ChatGPT (5ff3e92) $ ./generate.sh
Running Bandit on commit 9167b325addc7d3520987face684e8230bd62b46...
[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 3.10.12
```

Figure 1: Running Bandit on each of the commit

3. Running analysis.py to get the final plots.



```
@aayush8991 →/workspaces/STT_LAB3/LAB7/ChatGPT (5ff3e92) $ python3 analysis.py
CWE frequency analysis saved to bandit_plots/cwe_frequency.csv
@aayush8991 →/workspaces/STT_LAB3/LAB7/ChatGPT (5ff3e92) $
```

Figure 2: Running analysis.py

4.3 Individual Repository Level Analysis

4.3.1 ChatGPT

- Confidence level Issues per Commit

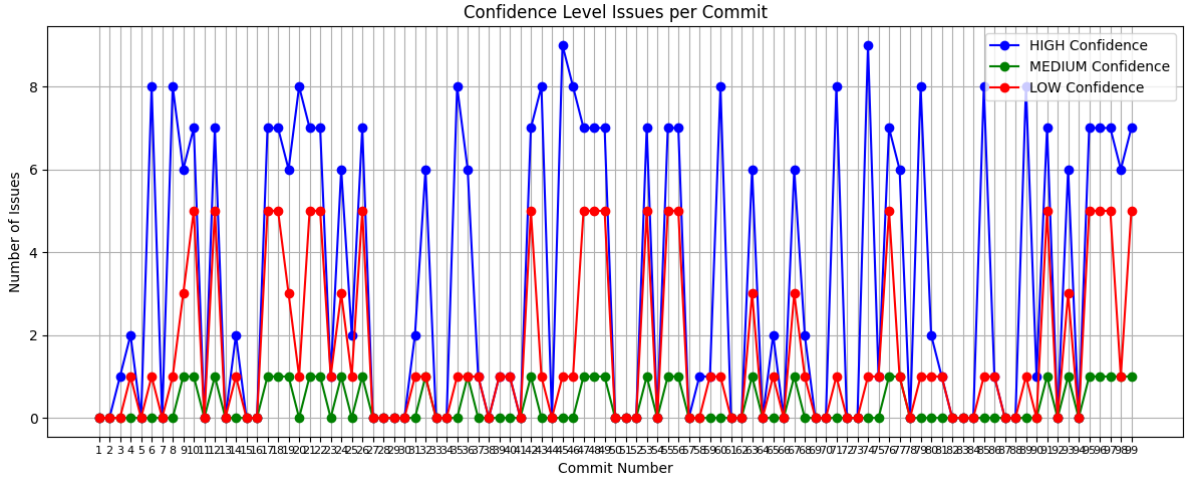


Figure 3: Confidence Level Issues

- Severity level Issues per Commit

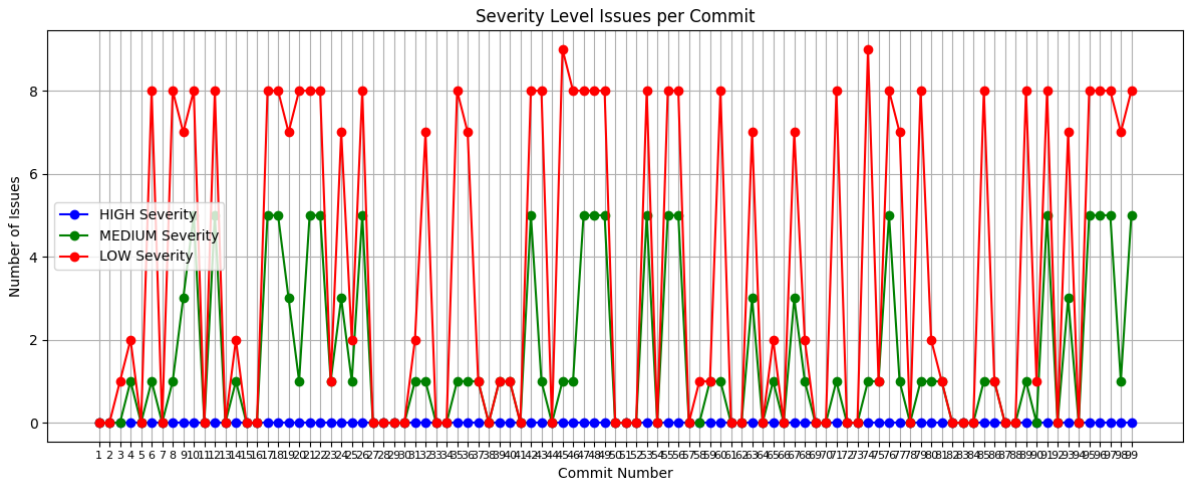


Figure 4: Severity Level Issues

- Unique CWEs identified per commit.

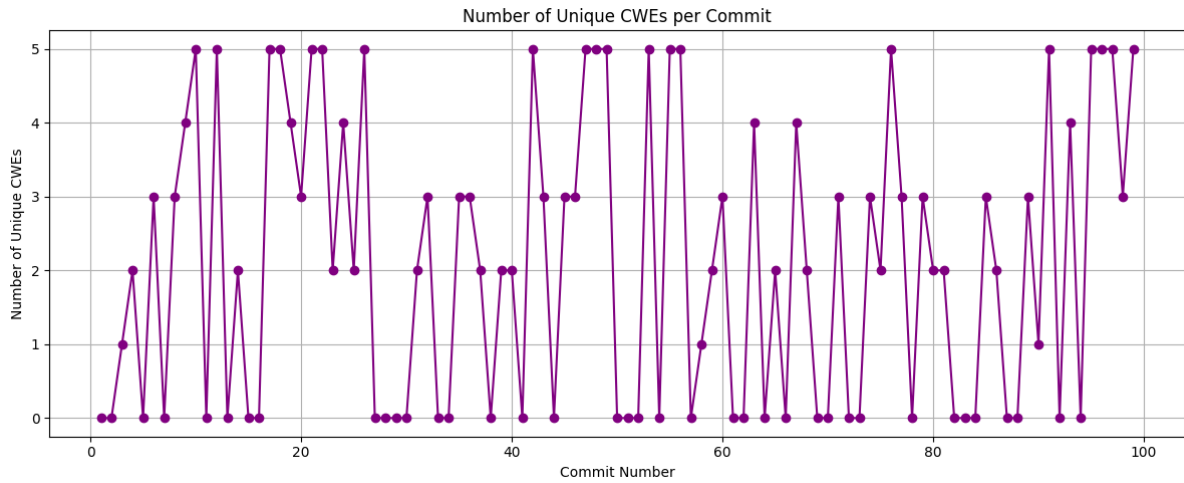


Figure 5: No of Unique CWE per commit

- Total Unique CWEs identified.

CWE	Frequency
https://cwe.mitre.org/data/definitions/78.html	271
https://cwe.mitre.org/data/definitions/400.html	92
https://cwe.mitre.org/data/definitions/703.html	70
https://cwe.mitre.org/data/definitions/89.html	58
https://cwe.mitre.org/data/definitions/330.html	20

Figure 6: All Uniquely Identified CWE

4.3.2 ChatTTS

- Confidence level Issues per Commit

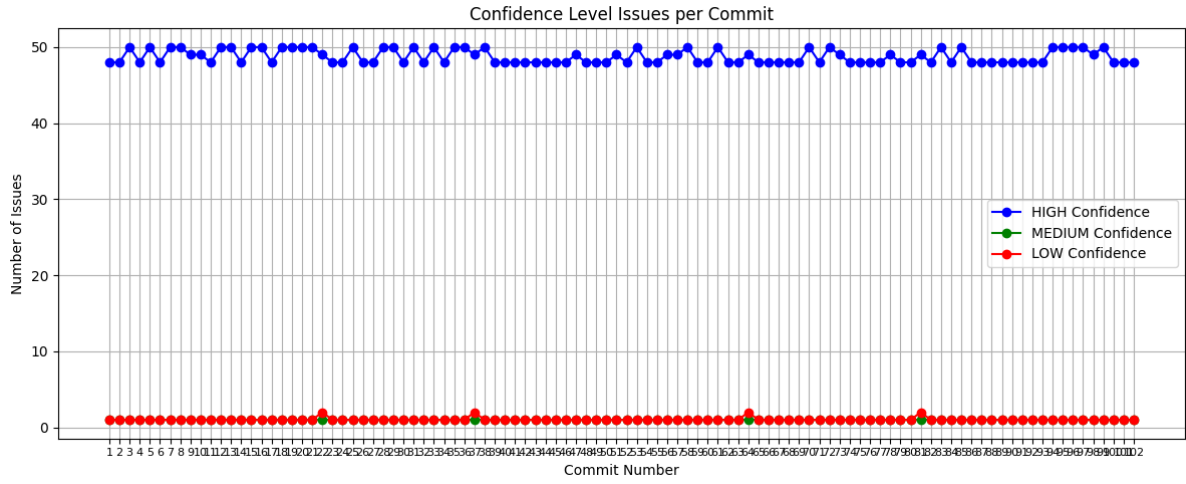


Figure 7: Confidence Level Issues

- Severity level Issues per Commit

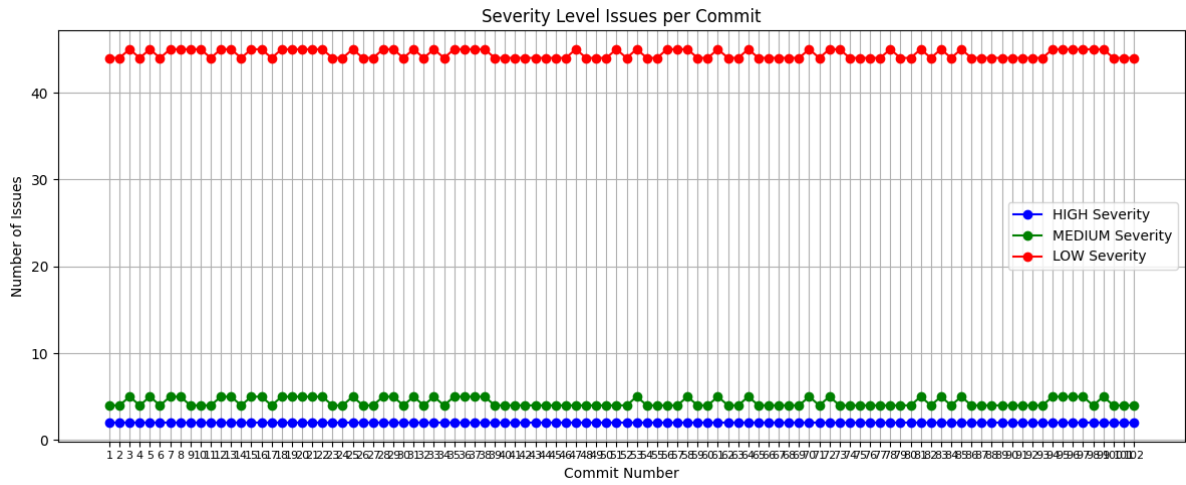


Figure 8: Severity Level Issues

- Unique CWEs identified per commit.

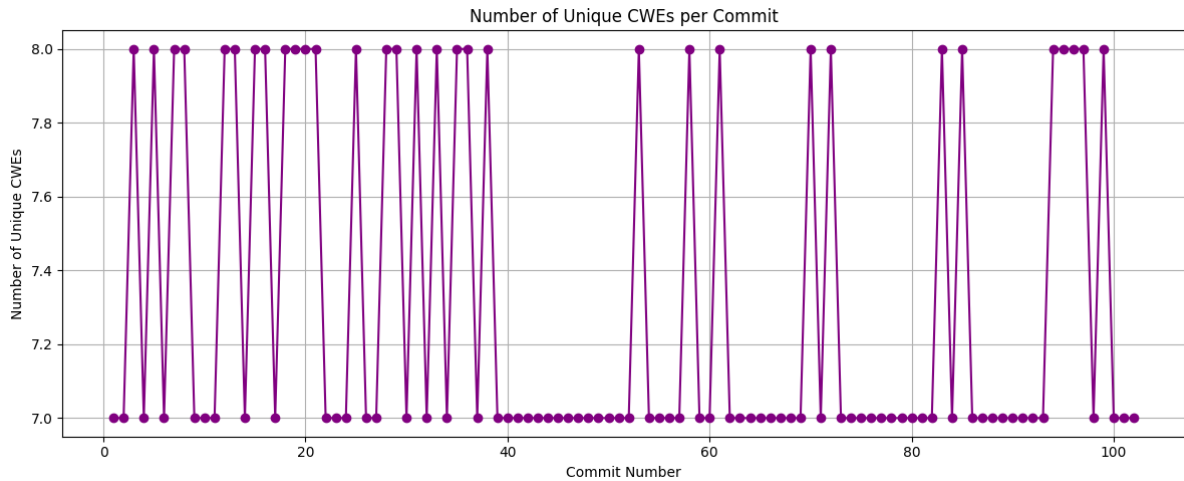


Figure 9: No of Unique CWE per commit

- Total Unique CWEs identified.

```
CWE, Frequency
https://cwe.mitre.org/data/definitions/703.html, 4023
https://cwe.mitre.org/data/definitions/78.html, 306
https://cwe.mitre.org/data/definitions/22.html, 204
https://cwe.mitre.org/data/definitions/732.html, 204
https://cwe.mitre.org/data/definitions/330.html, 204
https://cwe.mitre.org/data/definitions/400.html, 106
https://cwe.mitre.org/data/definitions/605.html, 102
https://cwe.mitre.org/data/definitions/502.html, 32
```

Figure 10: All Uniquely Identified CWE

4.3.3 MaxKB

- Confidence level Issues per Commit

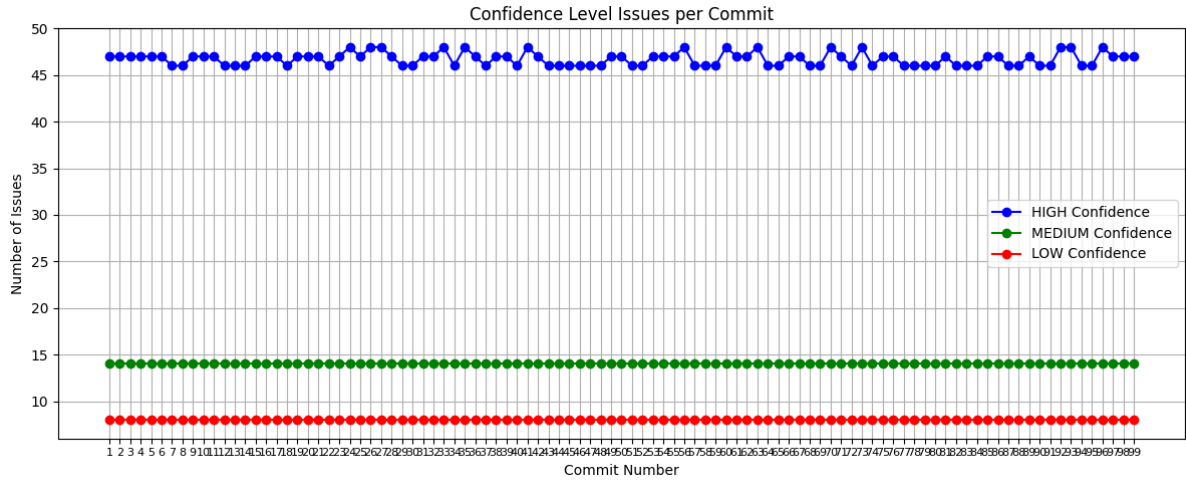


Figure 11: Confidence Level Issues

- Severity level Issues per Commit

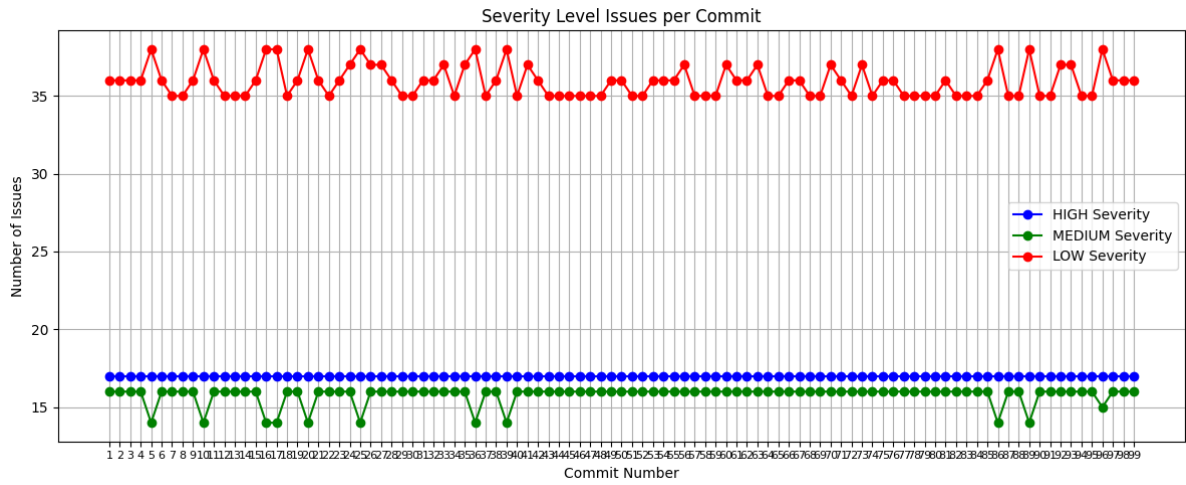


Figure 12: Severity Level Issues

- Unique CWEs identified per commit.

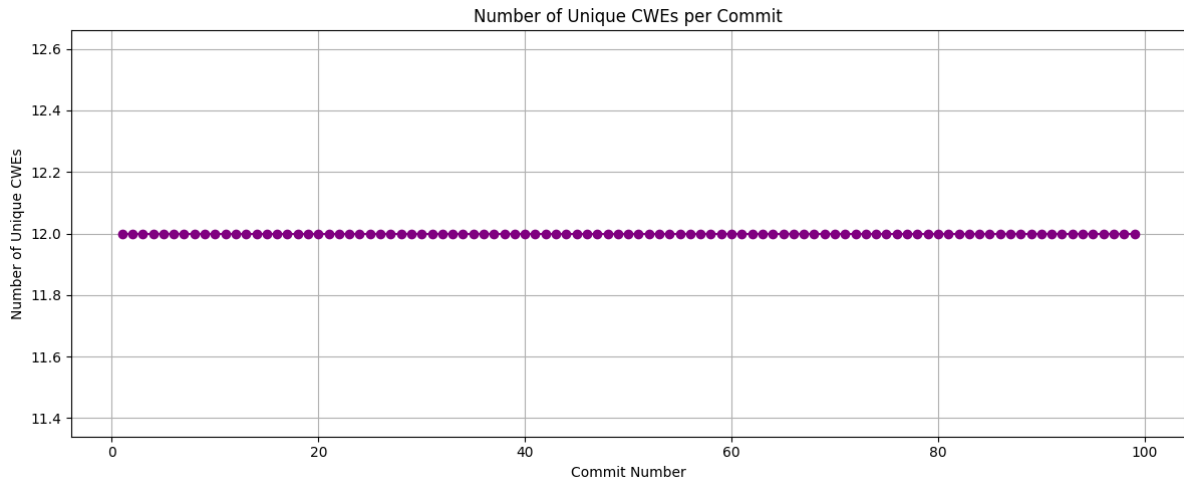


Figure 13: No of Unique CWE per commit

- Total Unique CWEs identified.

```
CWE, Frequency
https://cwe.mitre.org/data/definitions/327.html, 1386
https://cwe.mitre.org/data/definitions/703.html, 1368
https://cwe.mitre.org/data/definitions/78.html, 990
https://cwe.mitre.org/data/definitions/259.html, 990
https://cwe.mitre.org/data/definitions/400.html, 693
https://cwe.mitre.org/data/definitions/502.html, 386
https://cwe.mitre.org/data/definitions/20.html, 297
https://cwe.mitre.org/data/definitions/377.html, 297
https://cwe.mitre.org/data/definitions/89.html, 99
https://cwe.mitre.org/data/definitions/605.html, 99
https://cwe.mitre.org/data/definitions/295.html, 99
https://cwe.mitre.org/data/definitions/330.html, 99
```

Figure 14: All Uniquely Identified CWE

4.4 Research Questions and Analysis on Dataset Level

For ChatGPT, ChatTTS and MaxKB

Answering RQ1

Purpose: This RQ aims to determine when high-severity vulnerabilities are introduced and subsequently fixed in OSS repositories. The goal is to understand patterns in the life cycle of severe vulnerabilities to improve security management strategies.

Approach: The second graph, titled “*Severity Level Issues per Commit*”, is analyzed to track the High Severity (blue points) vulnerabilities over different commits. We observe when these vulnerabilities appear and when they disappear, indicating when they are introduced and subsequently fixed.

The timeline of commits is examined to identify any patterns in the introduction and resolution of high-severity issues.

Results: The graph indicates that High Severity vulnerabilities (blue points) remain consistently low or absent across commits. This suggests that most of the vulnerabilities in the dataset are of medium or low severity.

The persistence of issues of other severities (medium, low) suggests that high-severity issues may be addressed quickly when detected.

Takeaway: High-severity vulnerabilities appear to be introduced less frequently compared to medium and low-severity ones, and they are promptly fixed upon detection, likely due to their critical impact on security. These severities appear then there is a change in the libraries version, Cross-Site Scripting (XSS), XML External Entity (XXE) injection, Local File Inclusion (LFI).

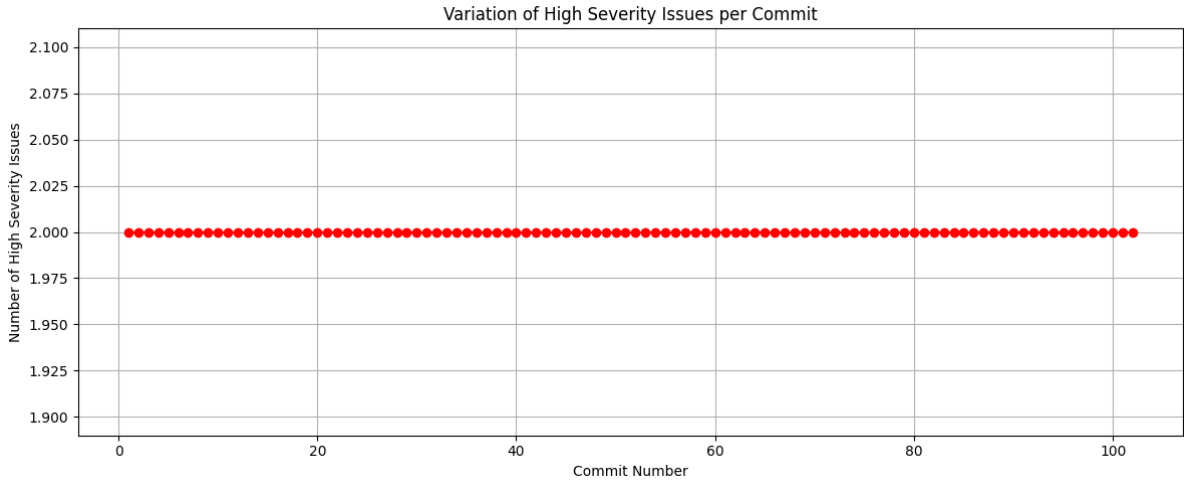


Figure 15: High Level Issues

Answering RQ2

Purpose: This RQ aims to analyze whether vulnerabilities of different severity levels (high, medium, low) exhibit similar patterns in their introduction and elimination in OSS repositories.

Approach: The second graph is used to compare the distribution of high, medium, and low-severity issues across commits.

The frequency and persistence of vulnerabilities of each severity level are analyzed.

Differences in how quickly vulnerabilities are introduced and eliminated are examined.

Results:

- **High Severity (blue points):** Appear infrequently and are promptly resolved when they do occur.
- **Medium Severity (green points):** Introduced sporadically but persist longer than high-severity vulnerabilities.
- **Low Severity (red points):** Appear frequently and persist over multiple commits, indicating that they are either less prioritized for fixing or harder to detect.

Takeaway: The pattern of introduction and elimination differs across severity levels. High-severity vulnerabilities are rare and quickly addressed, while medium and low-severity vulnerabilities appear more frequently and persist longer in the development timeline. These issues mainly occur due to coding errors, oversights, or a lack of adherence to secure coding practices and improperly configured systems, servers, or applications.

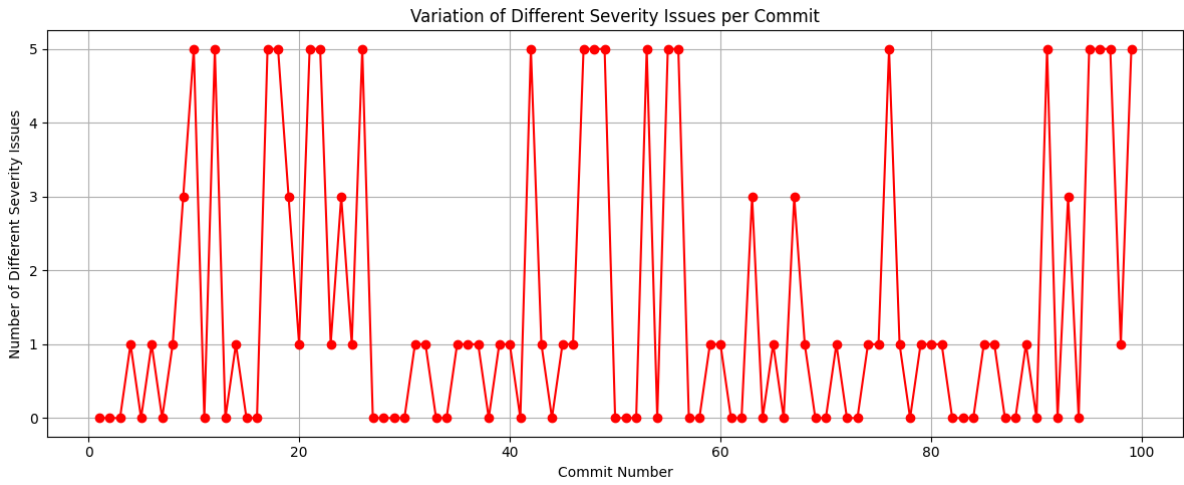


Figure 16: Different Level Issues

Answering RQ3

Purpose: This RQ seeks to identify the most frequently occurring CWEs (Common Weakness Enumerations) in OSS repositories. Identifying these vulnerabilities helps prioritize security measures.

Approach: The provided CWE frequency data is analyzed to determine the most common vulnerabilities.

The CWEs are ranked based on their frequency.

Results: The top five most frequent CWEs observed in the dataset:

Takeaway: The most common vulnerability is CWE-703 (Improper Check or Handling of Exceptional Conditions), which appears significantly more frequently than others, indicating a major security concern in OSS repositories. Other common issues include OS Command Injection (CWE-78) and Improper Path Traversal (CWE-22), which highlight areas requiring stricter security measures.

CWE ID	Description	Frequency
CWE-703	Improper Check or Handling of Exceptional Conditions	4023
CWE-78	Improper Neutralization of Special Elements in Command	306
CWE-22	Path Traversal	204
CWE-732	Incorrect Permission Assignment for Critical Resource	204
CWE-330	Use of Insufficiently Random Values	204
CWE-400	Uncontrolled Resource Consumption	106
CWE-605	Multiple Binds to the Same Port	102
CWE-502	Deserialization of Untrusted Data	32

Table 1: Most frequent CWEs based on the dataset

5 Conclusion and Discussion

In this study, we conducted an extensive analysis of Common Weakness Enumerations (CWEs) and Different Severity Issues in various software repositories to understand the frequency, severity, and patterns of vulnerabilities introduced over time. Our investigation was driven by key research questions that aimed to uncover trends in software security, particularly focusing on how vulnerabilities are distributed across commits, how severe they are, and how they evolve over different repository types.

By categorizing CWEs based on their frequency and impact, we observed distinct patterns that can inform better security practices. For instance, repositories with a higher number of unique CWEs tend to have a more complex codebase, which could benefit from stricter security policies and static analysis tools. Additionally, our study emphasizes the necessity of continuous monitoring and proactive mitigation strategies to reduce the risk of security breaches.

Overall, our research provides valuable insights into the prevalence and distribution of software vulnerabilities. Future work could focus on expanding the dataset to include a wider range of repositories and employing machine learning techniques to predict potential vulnerabilities before they are introduced. By leveraging these insights, developers and security professionals can enhance software security, ultimately leading to more robust and secure applications.