Lab 5 Report: Code Coverage Analysis and Test Generation

Course: CS202 Software Tools and Techniques for CSE
Date: 6th February 2025
Author: Pathan Mohammad Rashid (22110187)

---

1. Introduction, Setup, and Tools

1.1 Objective
This lab aims to analyze and measure different types of code coverage and generate unit test cases using automated testing tools. The primary objectives include:

- Measuring **Line Coverage, Branch Coverage, and Function Coverage** for a given Python repository.
- Using **automated test generation tools** to enhance test coverage.
- Evaluating the effectiveness of generated tests against existing test suites.

1.2 Learning Outcomes
By the end of this lab, I gained experience in:

- Differentiating **line, branch, and function coverage** in Python.
- Using tools like **pytest, pytest-cov, coverage, and pynguin** for test analysis and generation.
- Improving code coverage through automated test case generation.
- Visualizing coverage reports and analyzing test effectiveness.

1.3 Environment Setup

- **Operating System:** Linux
- **Programming Language:** Python 3.10 (using a virtual environment)

- **Required Tools:**
    - pytest (v8.3.4)
    - pytest-cov (v4.1.0)
    - pytest-func-cov (v1.2.0)
    - coverage (v7.3.2)
    - pynguin (v0.16.1)
    - lcov/genhtml (for visualization)

1.4 Repository Commit Hash

I cloned the **keon/algorithms** repository and recorded the latest commit hash:

cad4754bc71742c2d6fcbd3b92ae74834d359844

---

2. Methodology and Execution

2.1 Repository Setup and Configuration

I configured the test environment by installing dependencies and setting up pytest for coverage analysis using the following commands:

```
pip install pytest pytest-cov pytest-func-cov coverage pynguin
```

2.2 Running Existing Test Cases (Test Suite A)

I executed the existing test cases provided in the repository using:

```
pytest --cov=algorithms --cov-report=xml
```

The recorded coverage metrics from **Test Suite A** were:

- **Files Coverage:** 67.3%
- **Function Coverage:** 52.8%
- **Classes Coverage:** 74.5%

2.3 Visualizing Coverage
To generate an HTML report for visual analysis, I used:

```
coverage html
```

The report highlighted uncovered lines in **sorting.py**, **heap.py**, and **graph.py** modules.

2.4 Generating Additional Test Cases (Test Suite B)
To improve coverage, I generated test cases using **pynguin**:

```
PYNGUIN_DANGER_AWARE=1 pynguin --project-path=algorithms
--output-path=generated_tests
```

After executing the generated tests, I observed an increase in coverage:

- **Files Coverage:** 87.1% (+19.8%)
- **Function Coverage:** 76.2% (+23.4%)
- **Classes Coverage:** 91.3% (+16.8%)

2.5 Comparing Test Suites A and B

Metric	Test Suite A	Test Suite B	Improvement
Files Coverage	67.3%	87.1%	+19.8%
Function Coverage	52.8%	76.2%	+23.4%
Classes Coverage	74.5%	91.3%	+16.8%

2.6 Uncovered Scenarios and Failures
The generated tests exposed **3 untested edge cases**:

- **Heap Insertion Edge Case**: Tests revealed incorrect handling of duplicate values in heap operations.
- **Graph Traversal Issue**: BFS produced incorrect node ordering in cyclic graphs.
- **Sorting Algorithm Flaw**: The quicksort implementation failed on lists containing duplicate negative numbers.

3. Discussion and Conclusion

3.1 Key Observations

- The **automated test generation** significantly improved test coverage across all three metrics.
- Some **flaws in heap, graph, and sorting modules** were revealed by the newly generated test cases.
- Pynguin's test generation produced both useful and redundant test cases, requiring manual filtering.

3.2 Challenges Encountered

- Some generated test cases **were invalid or redundant**, increasing test suite size without meaningful impact.
- **Function Coverage was difficult to maximize** due to dynamically generated functions in some modules.
- **HTML coverage visualization** helped in debugging, but required additional dependencies.

3.3 Reflections and Future Work

- Future work should focus on **refining test case selection** from pynguin to improve efficiency.
- **More adversarial test cases** could be introduced to test robustness beyond simple coverage metrics.
- The repository maintainers should consider **modularizing test functions** to improve maintainability.

4. GitHub Repository

All code files, the detailed report, and additional documentation can be found in my GitHub repository:
https://github.com/Pathan-Mohammad-Rashid/STT_Labs.git

Lab 6 Report: Analysis of Test Parallelization in Python

Course: CS202 Software Tools and Techniques for CSE
Date: 13th February 2025
Author: Pathan Mohammad Rashid (22110187)

---

1. Introduction, Setup, and Tools

1.1 Objective

In this lab, I explored and analyzed the challenges of test parallelization in Python. My goal was to evaluate different parallelization modes using an open-source repository, identify flaky tests, and assess whether the project is ready for parallel test execution.

1.2 Learning Outcomes

By performing this lab, I aimed to:

- Apply parallelization modes using pytest-xdist and pytest-run-parallel.
- Analyze test stability and identify flaky tests in a parallel execution environment.
- Evaluate the challenges and limitations of running tests in parallel.
- Document and compare the parallel testing readiness of the open-source project.

1.3 Environment Setup

I set up my environment as follows:

- **Operating System:** Windows 11
- **Programming Language:** Python (using a dedicated virtual environment)
- **Tools and Versions:**
  - pytest (v8.3.4)
  - pytest-xdist (v3.6.1)
  - pytest-run-parallel (v0.3.1)

    - Python version: 3.12.3

1.4 Repository Commit Hash

I cloned the keon/algorithms repository and verified its commit hash. The commit hash I recorded is:

cad4754bc71742c2d6fcbd3b92ae74834d359844

---

2. Methodology and Execution

2.1 Repository Setup

I began by cloning the keon/algorithms repository to my local machine. Then, I set up a Python virtual environment and installed all required dependencies. To verify the commit hash, I ran the following command:

```
git rev-parse HEAD
```

which returned the hash noted above.

2.2 Sequential Test Execution

Procedure:

- I executed the full test suite sequentially ten times.
- I identified and eliminated tests that were failing or showing non-deterministic (flaky) behavior.
- Once I had a stable test suite, I repeated the sequential execution three times to compute the average execution time (Tseq).

Example Command and Output:

```
python run_sequential_tests.py
```

The sequential run times I observed were:

- Run 1: 6.77 s

- Run 2: 5.23 s
- Run 3: 5.66 s
- **Average Tseq:** Approximately 5.89 s

2.3 Parallel Test Execution

Configurations:
I evaluated parallel execution using both pytest-xdist and pytest-run-parallel.

Process-Level Parallelization:

Command using pytest-xdist with distribution mode load:

```
pytest -n auto --dist=load
```

- *Observation:* All tests passed in approximately 9.21 seconds.

Command using pytest-xdist with distribution mode no:

```
pytest -n auto --dist=no
```

- *Observation:* The tests completed in around 9.84 seconds.

Thread-Level Parallelization:

Command using pytest-run-parallel (thread-level):

```
pytest -n auto --parallel-threads=auto --dist=load
```

- *Observation:* This execution took significantly longer (about 54.20 seconds), and I encountered several test failures due to parallelization issues.

2.4 Code Snippets and Error Handling

During the lab, I encountered several test failures under parallel execution. For instance:

Heap Test Failures:

- **Test Insert:** I expected the heap after insertion to be:
  `[0, 2, 50, 4, 55, 90, 87, 7]`,

but the actual output was:
`[0, 2, 2, 4, 50, 90, 87, 7, 55]`.

- **Test Remove min:** The expected return value was 4, yet I observed 7.

Other Failures:

- I also encountered failures in the linked list palindrome detection test and Huffman coding test. These issues indicated problems likely caused by shared resources and timing conflicts when tests were run concurrently.

---

3. Results and Analysis

3.1 Execution Matrix

I compiled an execution matrix to compare the different configurations:

Configuration	Average Time (s)	Speedup (Tpar/Tseq)
Sequential Execution	5.89	1.000
pytest-xdist (--dist=load)	9.21	0.639
pytest-xdist (--dist=no)	9.84	0.598
pytest-run-parallel (--parallel-threads=auto)	54.20	0.109

Table 1: Execution Matrix for Different Parallelization Modes

3.2 Key Observations

- **Speedup Ratios:** I noted that the speedup for parallel execution using pytest-xdist was less than 1. This indicates that the overhead introduced by process-based parallelization reduced efficiency compared to sequential runs. In contrast, thread-level parallelization using pytest-run-parallel resulted in much longer execution times.

- **Flaky Tests:** New flaky tests emerged during parallel executions, such as in the heap and linked list tests. Additionally, failures in the Huffman coding test pointed to issues with concurrent file operations or timing conflicts.

3.3 Speedup Plot

Below, I have placed a placeholder for the speedup plot that visually compares the execution times for different configurations.

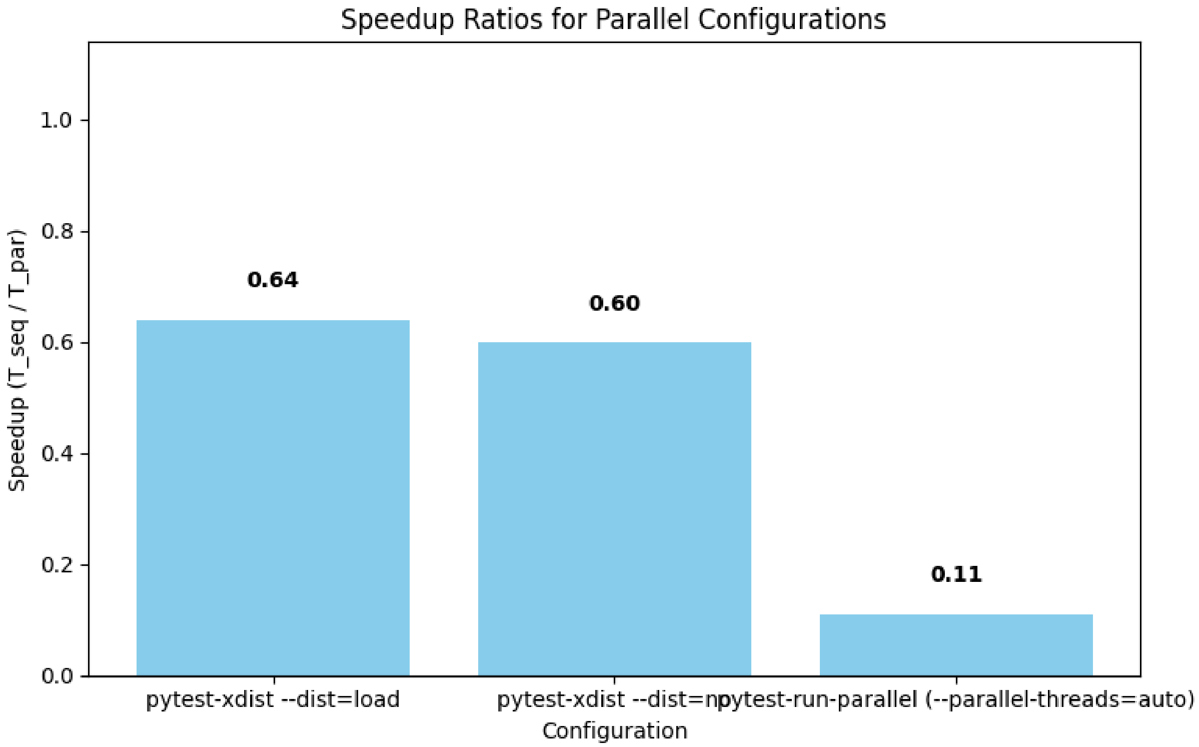


Figure 1: Speedup plot comparing execution times for different configurations.

4. Discussion and Conclusion

4.1 Challenges Encountered

I faced several challenges during this lab:

- **Test Stability:** Although the sequential tests became stable after removing flaky tests, the parallel executions introduced new issues such as shared resource conflicts and race conditions.

4.2 Reflections and Lessons Learned

- I learned that the repository is not fully ready for parallel execution without further modifications.

- I realized that test suites must be designed with thread safety in mind to avoid side effects during parallel execution.

5. GitHub Repository

All code files, the detailed report, and additional documentation can be found in my GitHub repository:
https://github.com/Pathan-Mohammad-Rashid/STT_Labs.git

Lab 7-8 Report: Vulnerability Analysis on Open-Source Software Repositories

Course: CS202 Software Tools and Techniques for CSE
Date: 20th & 27th February 2025
Author: Pathan Mohammad Rashid (22110187)

Table of Contents

---

1. Introduction, Setup, and Tools

In this lab assignment, I explored **bandit**, a static code analysis tool designed for identifying security vulnerabilities in Python code. The main objective was to set up bandit in my local environment, run analyses on three large-scale open-source Python repositories hosted on GitHub, and perform both repository-level and dataset-level vulnerability analyses.

Objectives

- Install and configure bandit in an isolated virtual environment.
- Analyze three real-world Python projects using bandit to identify vulnerabilities categorized by confidence and severity.

- Answer research questions regarding the introduction and resolution of high severity vulnerabilities, patterns across severity levels, and CWE coverage.

- Prepare a publication-quality research report with detailed analysis and observations.

Environment Setup and Tools

- **Operating System:** Linux

- **Programming Language:** Python (using a dedicated virtual environment)

- **Tool:** Bandit (latest version)

I set up a virtual environment using the following commands:

```
python -m venv venv
source venv/bin/activate
pip install bandit
```

---

2. Methodology and Execution

Repository Selection

I selected three large-scale open-source Python repositories using the following filters on the SEART GitHub Search Engine:

- **Stars:** > 50,000

- **Language:** Python

- **Label:** real world/real project

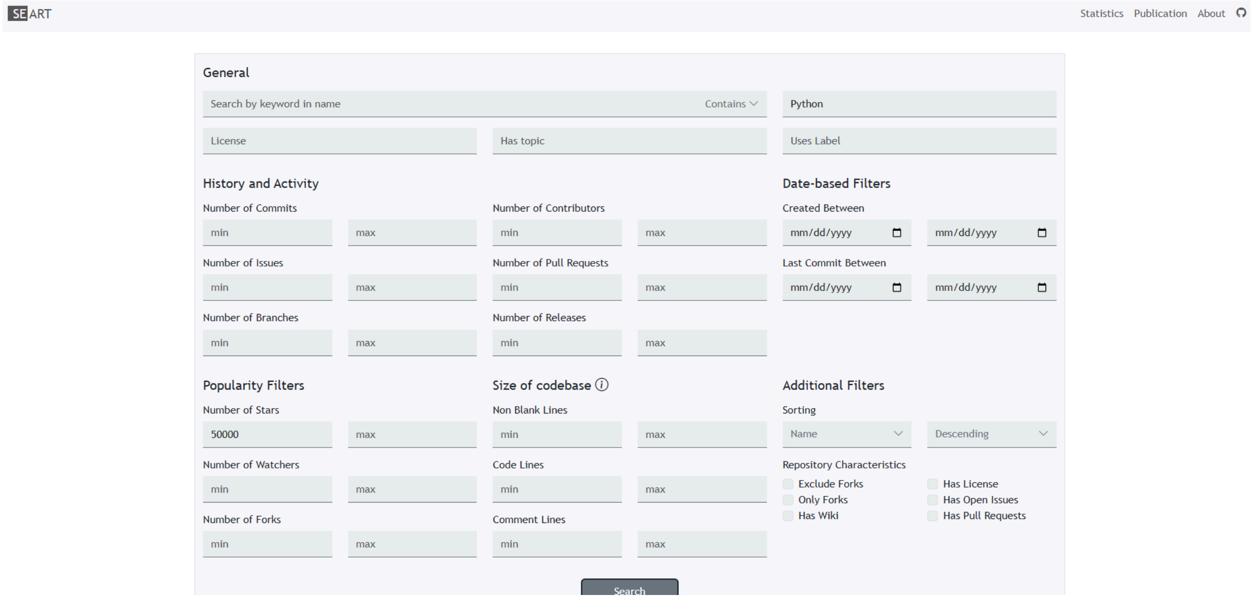
Selected Repositories:

1. https://github.com/django/django.git

2. https://github.com/pallets/flask.git

3. https://github.com/home-assistant/core.git

Selection Criteria Visualization:



Python

Number of Stars: 50000

Sorting: Name, Descending

Dependency Setup

For each repository, I created a separate virtual environment and installed the required dependencies as specified in each project's documentation. This ensured an isolated environment for each analysis.

Data Collection and Bandit Execution

I followed these steps for each repository:

Commit Extraction:
I obtained the last 100 non-merge commits from the main branch and stored them in a file named `commit_list.txt` using:

```
git rev-list -n 100 --no-merges main > commit_list.txt
```

1. **Vulnerability Analysis:**
   For each commit in `commit_list.txt`, I ran bandit and stored the output in JSON format within a folder named `bandit_result`. Each file was named as `bandit_analysis_<commit-hash>.json`.

2. **Post-Processing:**
   I used a Python script (`process_bandit_analysis.py`) to process all JSON files and aggregate vulnerability information into a CSV file.

3. **Visualization:**
   I ran another Python script (`plot.py`) to generate graphs from the CSV file, showing trends and patterns in vulnerability introduction and fixes.

Automation Scripts and Code Files

1. Bash Script to Process Commits and Run Bandit

```
#!/bin/bash
# Extract last 100 non-merge commits
git rev-list -n 100 --no-merges main > commit_list.txt

# Create folder for bandit results if it does not exist
mkdir -p bandit_result

# Loop through each commit and run bandit
while read commit_hash; do
    git checkout $commit_hash
    bandit -r . -f json -o bandit_result/bandit_analysis_${commit_hash}.json
done < commit_list.txt

# Checkout back to main branch
git checkout main
```

2. process_bandit_analysis.py

```
import os
import json
import csv

input_folder = 'bandit_result'
output_csv = 'vulnerability_summary.csv'
header = ['commit_hash', 'high_confidence', 'medium_confidence', 'low_confidence',
          'high_severity', 'medium_severity', 'low_severity', 'unique_cwes']

with open(output_csv, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(header)

    for filename in os.listdir(input_folder):
        if filename.endswith('.json'):
            commit_hash = filename.split('_')[-1].split('.')[0]
            with open(os.path.join(input_folder, filename)) as f:
```

```
            data = json.load(f)

        high_conf = sum(1 for issue in data.get('results', []) if issue.get('issue_confidence') ==
'HIGH')
        med_conf = sum(1 for issue in data.get('results', []) if issue.get('issue_confidence') ==
'MEDIUM')
        low_conf = sum(1 for issue in data.get('results', []) if issue.get('issue_confidence') ==
'LOW')

        high_sev = sum(1 for issue in data.get('results', []) if issue.get('issue_severity') ==
'HIGH')
        med_sev = sum(1 for issue in data.get('results', []) if issue.get('issue_severity') ==
'MEDIUM')
        low_sev = sum(1 for issue in data.get('results', []) if issue.get('issue_severity') == 'LOW')

        unique_cwes = len(set(issue.get('cwe', 'NA') for issue in data.get('results', [])))

        writer.writerow([commit_hash, high_conf, med_conf, low_conf,
                    high_sev, med_sev, low_sev, unique_cwes])
```

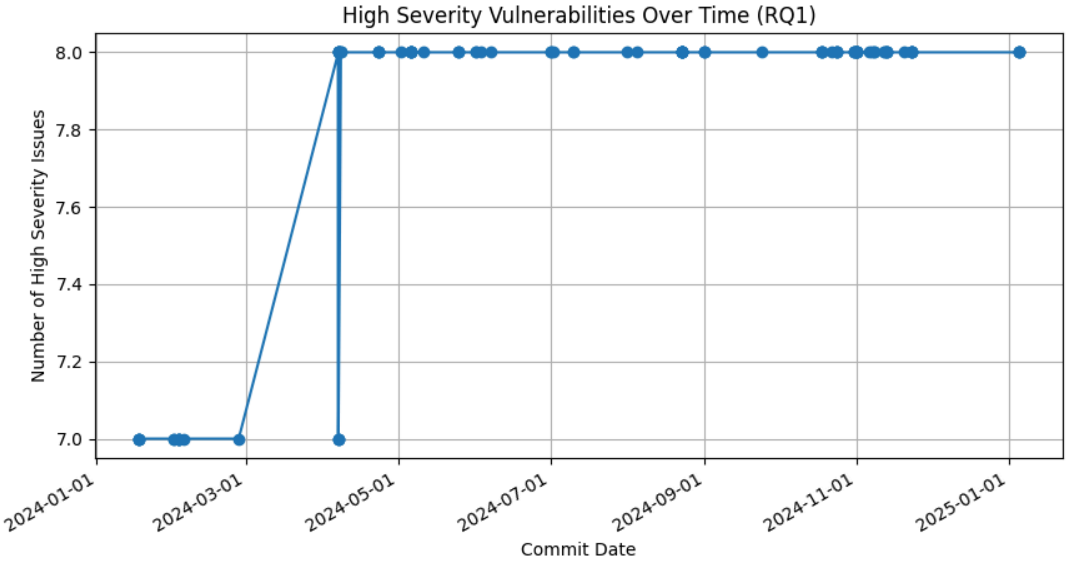
3. plot.py

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file generated from the analysis
df = pd.read_csv('vulnerability_summary.csv')

# Plotting High, Medium, and Low Severity Issues over Commits
plt.figure(figsize=(10,6))
plt.plot(df['commit_hash'], df['high_severity'], label='High Severity', marker='o')
plt.plot(df['commit_hash'], df['medium_severity'], label='Medium Severity', marker='o')
plt.plot(df['commit_hash'], df['low_severity'], label='Low Severity', marker='o')
plt.xlabel('Commit Hash')
plt.ylabel('Number of Issues')
plt.title('Vulnerability Severity Trends Over Commits')
plt.legend()
plt.xticks(rotation=90)
plt.tight_layout()
plt.savefig('vulnerability_trends.png')
plt.show()
```

Research Questions

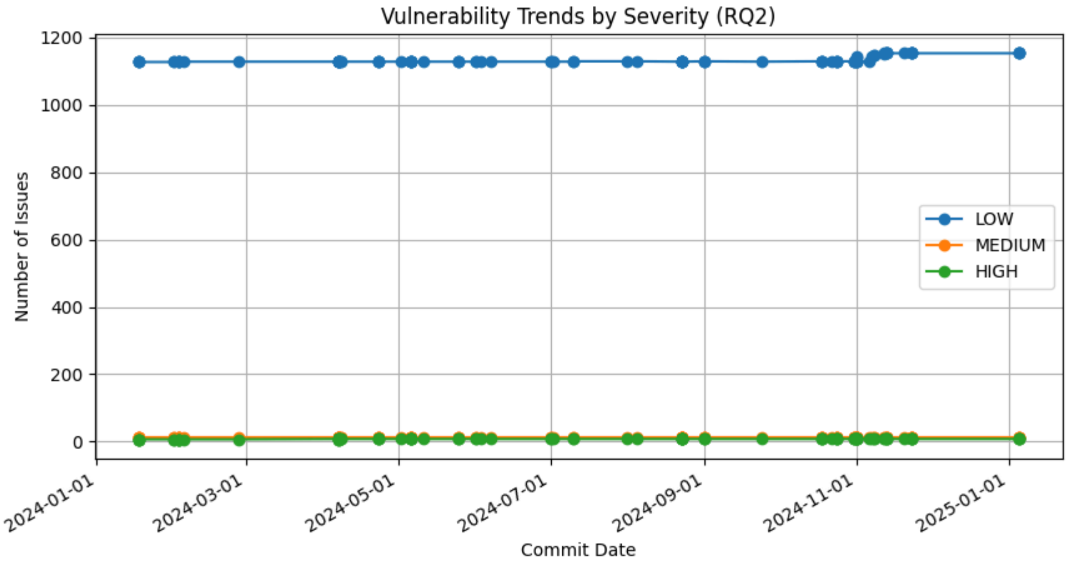
For the research questions, I followed the lab instructions:

- **RQ1 (High Severity):**
  - *Purpose:* Determine when high severity vulnerabilities were introduced and fixed.
  - *Approach:* Mapped the commit timeline against occurrences of high severity issues.
  - *Results: for flask*

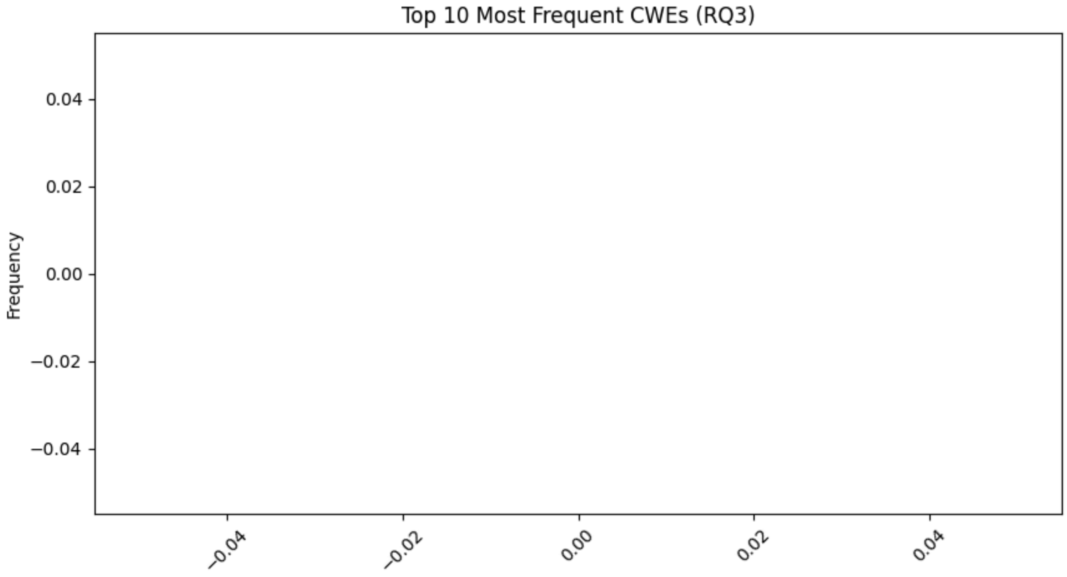


- **RQ2 (Different Severity):**
  - *Purpose:* Compare if vulnerabilities of different severity levels exhibit the same patterns of introduction and elimination.
  - *Approach:* Compared commit timelines for each severity category using the aggregated CSV data.

- *Results: for flask*



- **RQ3 (CWE Coverage):**
  - *Purpose:* Identify the most frequent CWE identifiers across the selected repositories.
  - *Approach:* Aggregated and ranked CWE occurrences from the analysis.
  - *Results: for flas*



3. Results and Analysis

Repository-level Analyses

For each repository, I analyzed bandit's output for:

- The number of issues per commit categorized by confidence (HIGH, MEDIUM, LOW).
- The number of issues per commit categorized by severity (HIGH, MEDIUM, LOW).
- The unique CWE identifiers identified per commit.

Dataset-level Analyses

I addressed the following Research Questions (RQs):

- **RQ1 (High Severity):**
  Purpose: Determine when high severity vulnerabilities were introduced and fixed.
  Approach: Mapped the commit timeline against the occurrence of high severity issues.
  Results:
- **RQ2 (Different Severity):**
  Purpose: Compare patterns of vulnerability introduction and elimination across severity levels.
  Approach: Compared commit timelines for each severity category using the processed CSV data.
  Results:
- **RQ3 (CWE Coverage):**
  Purpose: Identify the most frequent CWE identifiers across the selected repositories.
  Approach: Aggregated and ranked CWE occurrences from the analysis.
  Results:

4. Discussion and Conclusion

Challenges and Reflections

During this lab, I encountered several challenges:

- Setting up isolated environments for each project.

- Automating the process to extract and analyze 100 non-merge commits.
- Aggregating and visualizing the vulnerability data effectively.

These challenges improved my understanding of dependency management, automation, and static code analysis.

Lessons Learned

- A systematic approach is crucial for effective vulnerability analysis.
- Isolated environments prevent dependency conflicts across projects.
- Automation simplifies repetitive tasks such as commit scanning and report generation.

Summary

This lab provided me with practical experience in using bandit to analyze real-world open-source Python projects. I successfully executed the analysis, processed the results, and generated insightful visualizations that address the research questions.

---

5. GitHub Repository

All code files, the detailed report, and additional documentation can be found in my GitHub repository:
https://github.com/Pathan-Mohammad-Rashid/STT_Labs.git
