

Lab 5: Code Coverage Analysis and Test Generation

Birudugadda Srivibhav 22110050

March 25, 2025

Introduction, Setup, and Tools

Overview

In this lab, I will analyze and measure different types of code coverage while generating unit test cases using automated testing tools. I will work with a provided dataset of Python programs and evaluate key testing metrics, including:

- Line (statement) coverage
- Branch coverage
- Function coverage

Through this process, I will gain hands-on experience with automated test generation tools, exploring their benefits and limitations.

Objectives

- Understand and distinguish between different types of code coverage.
- Utilize automated tools to measure coverage on a given dataset of Python programs.
- Write or generate effective unit tests to achieve maximum coverage.
- Visualize coverage reports and analyze the effectiveness of tests.

Environment Setup and Tools Used

- Operating System (Ubuntu 24.04 on WSL running on Windows 11)
- Visual Studio Code
- Python 3.10.16
- Libraries:
 - pytest (for running tests)
 - pytest-cov (for line/branch coverage analysis)
 - pytest-func-cov (for function coverage analysis)
 - coverage (for detailed coverage metrics)
 - penguin (for automated unit test generation)
- Some other tools (genhtml, lcov)

Methodology and Execution

Run Existing Test Cases (Test Suite A)

Firstly, lets create a conda environment to do our tasks:

```
(base) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ conda create --name=stt-labs python=3.10
```

Now let us clone our target repo (<https://github.com/keon/algorithms>):

```
(base) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ git clone https://github.com/keon/algorithms
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 | 10.35 MiB/s, done.
Resolving deltas: 100% (3242/3242), done.
```

The current commit hash of our repo is **cad4754bc71742c2d6fcbd3b92ae74834d359844**

There is a txt file named **test_requirements.txt** which contains the dependencies required to run tests.

```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5/algorithms$ cat test_requirements.txt
flake8
python-coveralls
coverage
nose
pytest
tox
black
```

Let's install it using pip.

```
Successfully installed PyYAML-6.0.2 black-25.1.0 cachetools-5.5.2 chardet-5.2.0 coverage-7.6.12 distlib-0.3.9
filelock-3.17.0 flake8-7.1.2 iniconfig-2.0.0 mccabe-0.7.0 mypy-extensions-1.0.0 nose-1.3.7 packaging-24.2 plat
formdirs-4.3.6 pycodestyle-2.12.1 pyflakes-3.2.0 pyproject-api-1.9.0 pytest-8.3.4 python-coveralls-2.9.3 tox-4
.24.1 virtualenv-20.29.2
```

In the above installation, **pytest-cov** is missing so we install it also using pip.

The repo might already have test cases. To execute them and measure coverage for the algorithms folder, run:

```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ coverage run --source=algorithms/algorithm
s --branch --include=* -m pytest algorithms/tests/
```

Ok so when i initially ran the above, i got some errors because the repository was not installed, we we install it using:

```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5/algorithms$ pip install -e .
Obtaining file:///home/birud_ubuntu/STT-Labs/Assignment-2/Lab-5/algorithms
  Preparing metadata (setup.py) ... done
Installing collected packages: algorithms
  Attempting uninstall: algorithms
    Found existing installation: algorithms 0.1.4
    Uninstalling algorithms-0.1.4:
      Successfully uninstalled algorithms-0.1.4
  DEPRECATION: Legacy editable install of algorithms==0.1.4 from file:///home/birud_ubuntu/STT-Labs/Assignment
-2/Lab-5/algorithms (setup.py develop) is deprecated. pip 25.1 will enforce this behaviour change. A possible
replacement is to add a pyproject.toml or enable --use-pep517, and use setuptools >= 64. If the resulting inst
allation is not behaving as expected, try using --config-settings editable_mode=compat. Please consult the set
uptools documentation for more information. Discussion can be found at https://github.com/pypa/pip/issues/1145
  Running setup.py develop for algorithms
Successfully installed algorithms
```

Now we rerun the pytest command. I got an error stating that there is a syntax error in one of the test files namely `test_array.py`

```
===== short test summary info =====
ERROR tests/test_array.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!
===== 2 warnings, 1 error in 3.15s =====

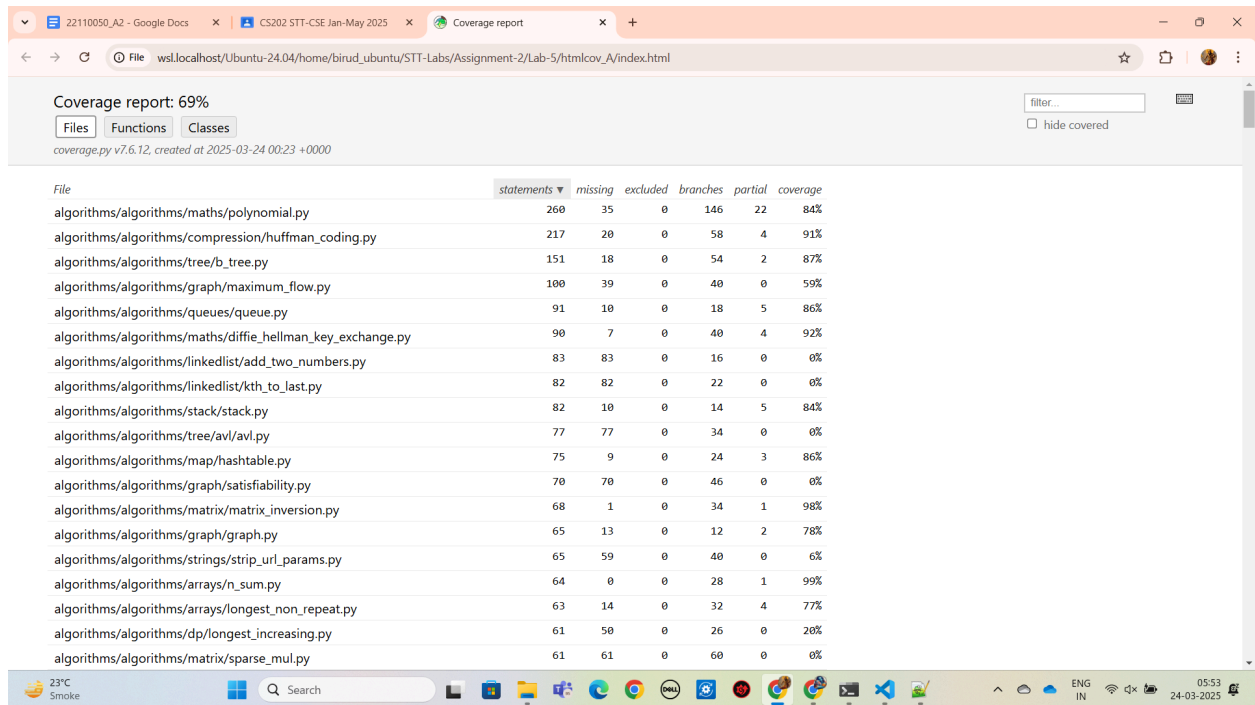
----- ERROR collecting tests/test_array.py -----
../miniconda3/envs/stt-labs/lib/python3.10/site-packages/_pytest/python.py:493: in importtestmodule
    mod = import_path(
../miniconda3/envs/stt-labs/lib/python3.10/site-packages/_pytest/pathlib.py:587: in import_path
    importlib.import_module(module_name)
../miniconda3/envs/stt-labs/lib/python3.10/importlib/__init__.py:126: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
<frozen importlib._bootstrap>:1050: in _gcd_import
    ???
<frozen importlib._bootstrap>:1027: in _find_and_load
    ???
<frozen importlib._bootstrap>:1006: in _find_and_load_unlocked
    ???
<frozen importlib._bootstrap>:688: in _load_unlocked
    ???
../miniconda3/envs/stt-labs/lib/python3.10/site-packages/_pytest/assertion/rewrite.py:175: in exec_module
    source_stat, co = _rewrite_test(fn, self.config)
../miniconda3/envs/stt-labs/lib/python3.10/site-packages/_pytest/assertion/rewrite.py:355: in _rewrite_test
    tree = ast.parse(source, filename=strfn)
../miniconda3/envs/stt-labs/lib/python3.10/ast.py:50: in parse
    return compile(source, filename, mode, flags,
E     File "/home/birud_ubuntu/STT-Labs/Assignment-2/Lab-5/algorithms/tests/test_array.py", line 13
E         rotate_v1, rotate_v2, rotate_v3,
E         ^^^^^^^^^
E SyntaxError: invalid syntax
```

The syntax error is simply a missing comma:

```
from algorithms.arrays import (
    delete_nth, delete_nth_naive,
    flatten_iter, flatten,
    garage,
    josephus,
    longest_non_repeat_v1, longest_non_repeat_v2,
    get_longest_non_repeat_v1, get_longest_non_repeat_v2,
    Interval, merge_intervals,
    missing_ranges,
    move_zeros,
    plus_one_v1, plus_one_v2, plus_one_v3,
    remove_duplicates
    rotate_v1, rotate_v2, rotate_v3,
    summarize_ranges,
    three sum,
    two sum,
    max ones index,
    trimmean,
    top 1,
    limit,
    n sum
)
```

After correcting it, we will rerun the cmd for the coverage report.

This time i got a coverage of 69 %



Coverage report: 69%

Files Functions Classes

coverage.py v7.6.12, created at 2025-03-24 00:23 +0000

File	statements	missing	excluded	branches	partial	coverage
algorithms/algorithms/maths/polynomial.py	260	35	0	146	22	84%
algorithms/algorithms/compression/huffman_coding.py	217	20	0	58	4	91%
algorithms/algorithms/tree/b_tree.py	151	18	0	54	2	87%
algorithms/algorithms/graph/maximum_flow.py	100	39	0	40	0	59%
algorithms/algorithms/queues/queue.py	91	10	0	18	5	86%
algorithms/algorithms/maths/diffie_hellman_key_exchange.py	90	7	0	40	4	92%
algorithms/algorithms/linkedlist/add_two_numbers.py	83	83	0	16	0	0%
algorithms/algorithms/linkedlist/kth_to_last.py	82	82	0	22	0	0%
algorithms/algorithms/stack/stack.py	82	10	0	14	5	84%
algorithms/algorithms/tree/avl/avl.py	77	77	0	34	0	0%
algorithms/algorithms/map/hashable.py	75	9	0	24	3	86%
algorithms/algorithms/graph/satisfiability.py	70	70	0	46	0	0%
algorithms/algorithms/matrix/matrix_inversion.py	68	1	0	34	1	98%
algorithms/algorithms/graph/graph.py	65	13	0	12	2	78%
algorithms/algorithms/strings/strip_url_params.py	65	59	0	40	0	6%
algorithms/algorithms/arrays/n_sum.py	64	0	0	28	1	99%
algorithms/algorithms/arrays/longest_non_repeat.py	63	14	0	32	4	77%
algorithms/algorithms/dp/longest_increasing.py	61	50	0	26	0	20%
algorithms/algorithms/matrix/sparse_mul.py	61	61	0	60	0	0%

Now this time, i have got two failed cases:

```
===== short test summary info =====
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - TypeError: TestCase.assertEqual() 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 9.48s =====
```

The reason for their failures:

```
===== FAILURES =====
_____ TestRemoveDuplicate.test_remove_duplicates _____

self = <test_array.TestRemoveDuplicate testMethod=test_remove_duplicates>

    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]))
E     TypeError: TestCase.assertEqual() missing 1 required positional argument: 'list2'

tests/test_array.py:305: TypeError
_____ TestSummaryRanges.test_summarize_ranges _____

self = <test_array.TestSummaryRanges testMethod=test_summarize_ranges>

    def test_summarize_ranges(self):
>     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
```

Now I will correct these also to ensure max coverage.

So like these failures can mean two things, they indicate issues with either the test cases or the functions being tested. They are in fact the issues with the test cases themselves and not the parent functions.

Let us correct them one by one, in the first case, it says that there is a missing argument called list2. We can clearly see that below too:

```
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]))
```

So we add those missing arguments:

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

Now the above failed case passed, let's move on to the next one:

```
self = <test_array.TestSummaryRanges testMethod=test_summarize_ranges>

def test_summarize_ranges(self):
> 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
```

The second one is simply an assertion error, a mismatch between what the function outputs and what the test expects, so i have corrected this too:

```
def test_summarize_ranges(self):
    self.assertEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                     ["0-2", "4-5", "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"])
```

Now we rerun the **pytest** command again and this time all the test cases passed, but there is no improvement in the coverage, which is still 69%.

Now we store the coverage report in json using the below command to perform some visualization plots later, use the below command to get the report in form of json:

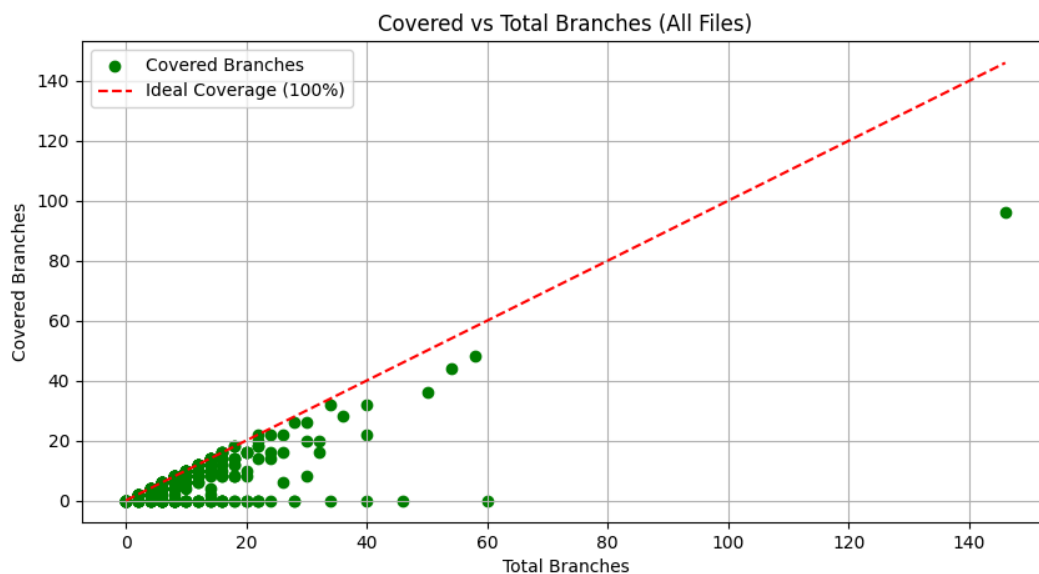
```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ coverage json
Wrote JSON report to coverage.json
```

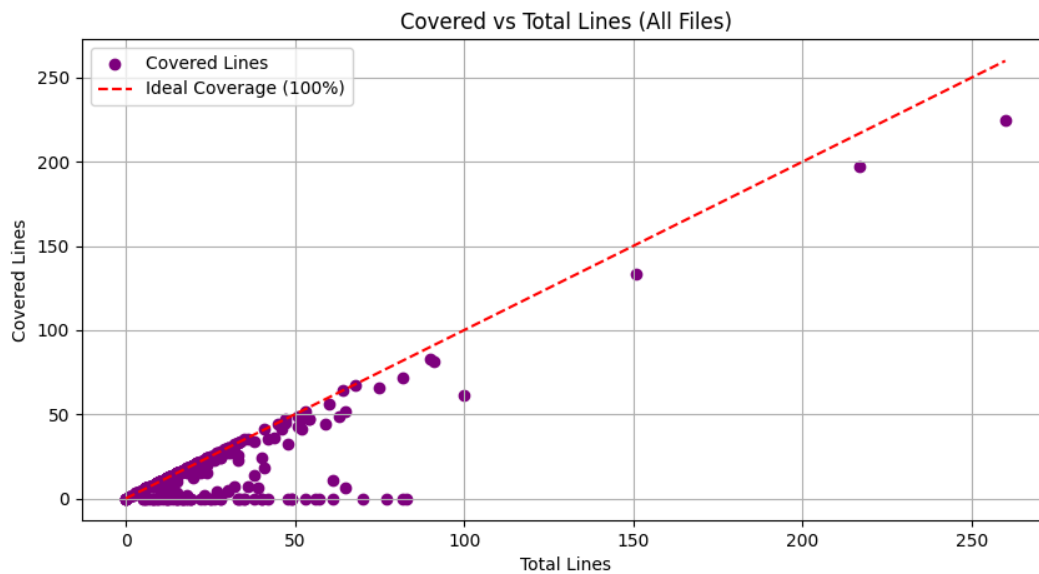
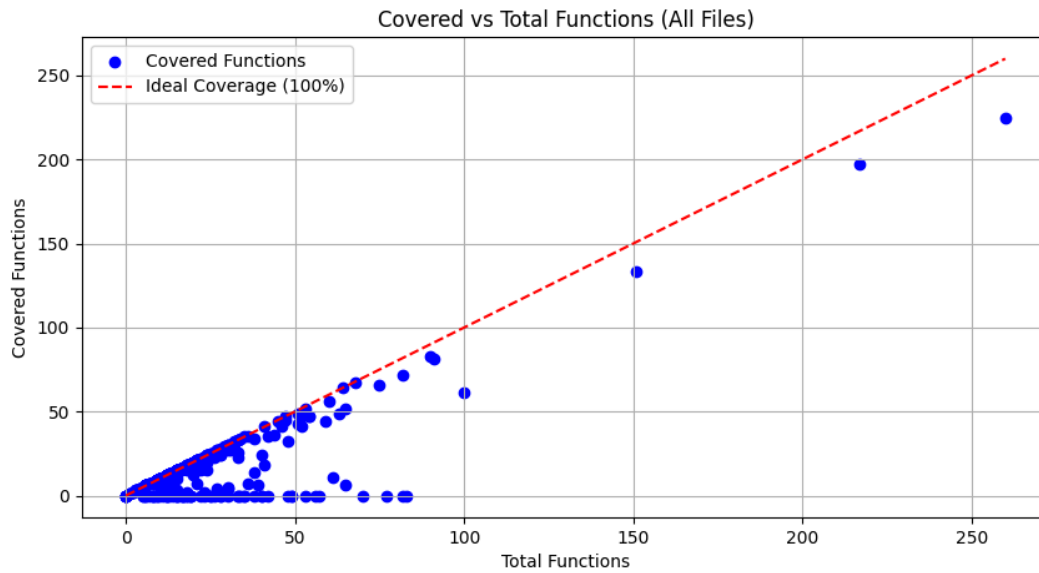
Now i have written a **coverage_stats.py** which reads the json file and then reports the coverage percentage

```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ python coverage_stats.py
Total Statements: 8234
Covered Lines: 5764
Missing Lines: 2470
Excluded Lines: 0
Total Branches: 3780
Covered Branches: 2518
Missing Branches: 1262
Partial Branches: 252
Coverage Percentage: 68.93624105210587
```

Now, we use the **final_visualize.py** script to plot scatter plots for the coverage of lines, branches, and functions for files in the repository, where each dot represents a file.

Below are the plots:



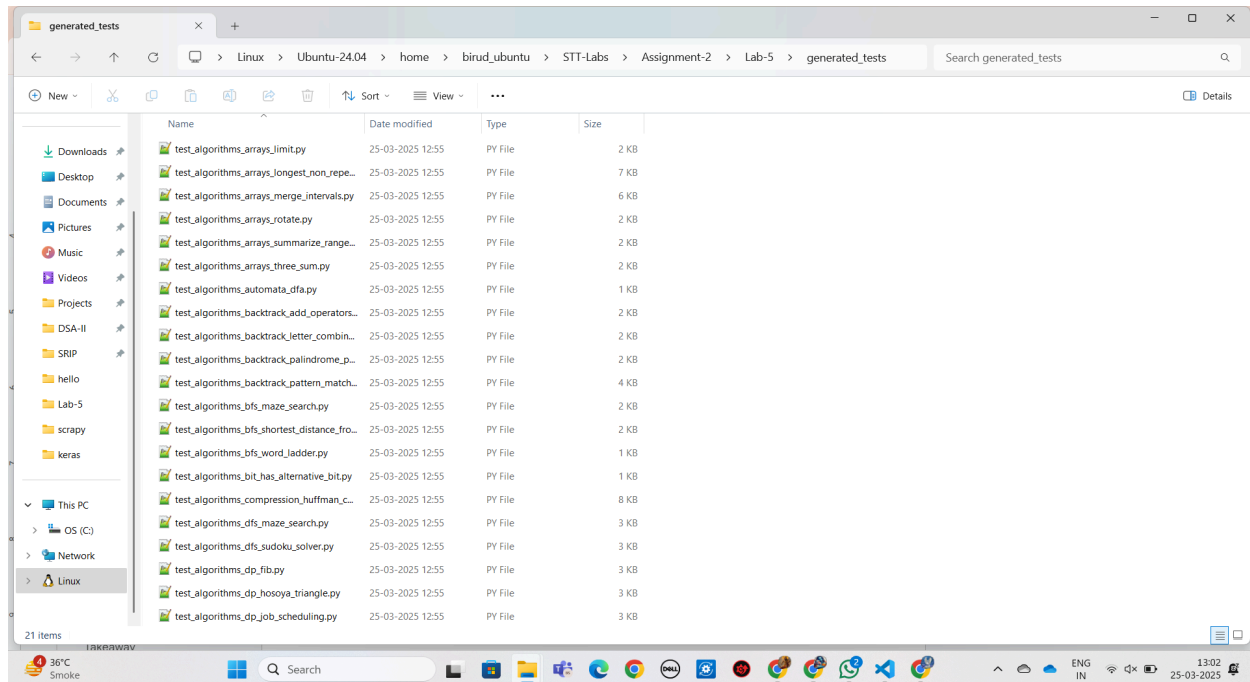


Now we take the files which lie below the ideal coverage line and then generate unit test cases for those files using *pynguin*.

Run Generated Test Cases (Test Suite B)

To generate a few test cases using *Pynguin*, I have written a Python script named *generate_tests.py*, which can be found [here](#). We have generated test cases only for the files that fall below the ideal coverage threshold.

Below is the screenshot of some of the generated test case files by *Pynguin*.



Now, we execute coverage on these files using the same command we used earlier for test suite A. However, note that *Pynguin* has generated only a few test cases so far because some tests are taking too long and getting stuck during generation. So far, it has been able to generate **21 test cases** for uncovered files. Therefore, we only take the corresponding test cases from test suite A and compare them with those in test suite B.

Code Coverage Comparison

```
(stt-labs) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-5$ python comparative_analysis.py
Comparative Coverage Analysis (Common Files Only):
```

File	Coverage A (%)	Coverage B (%)
algorithms/algorithms/backtrack/palindrome_partitioning.py	55.55555555555556	100.0
algorithms/algorithms/dp/hosoya_triangle.py	78.37837837837837	100.0
algorithms/algorithms/automata/dfa.py	90.0	91.66666666666667
algorithms/algorithms/dp/fib.py	93.54838709677419	100.0
algorithms/algorithms/dp/num_decodings.py	5.405405405405405	8.695652173913043
algorithms/algorithms/dp/min_cost_path.py	19.047619047619047	20.0
algorithms/algorithms/dp/max_product_subarray.py	9.523809523809524	13.333333333333334
algorithms/algorithms/dp/job_scheduling.py	78.94736842105263	100.0
algorithms/algorithms/backtrack/letter_combination.py	90.0	100.0
algorithms/algorithms/bfs/shortest_distance_from_all_buildings.py	8.51063829787234	81.48148148148148
algorithms/algorithms/dfs/maze_search.py	94.11764705882354	100.0
algorithms/algorithms/arrays/longest_non_repeat.py	76.84210526315789	100.0
algorithms/algorithms/dp/word_break.py	14.285714285714286	15.384615384615385
algorithms/algorithms/dp/matrix_chain_order.py	14.285714285714286	16.666666666666668
algorithms/algorithms/dp/coin_change.py	100.0	16.666666666666668
algorithms/algorithms/bfs/word_ladder.py	96.42857142857143	87.5
algorithms/algorithms/arrays/rotate.py	94.44444444444444	92.85714285714286
algorithms/algorithms/arrays/flatten.py	100.0	21.428571428571427
algorithms/algorithms/dp/egg_drop.py	100.0	12.5
algorithms/algorithms/bit/power_of_two.py	100.0	50.0
algorithms/algorithms/distribution/__init__.py	100.0	100.0

You can clearly see that the coverage percentage has improved for some tests in test suite B compared to test suite A. However, in some cases, test suite A provides better coverage.

The following analysis highlights the differences in coverage percentages between test-suites A and B for selected algorithm files:

- **Higher Coverage in Test-Suite B:**
 - `algorithms/algorithms/bfs/shortest_distance_from_all_buildings.py`: Increased from **8.51% (A)** to **81.48% (B)**.
 - `algorithms/algorithms/dp/hosoya_triangle.py`: Increased from **78.37% (A)** to **100% (B)**.
 - `algorithms/algorithms/dp/job_scheduling.py`: Increased from **78.95% (A)** to **100% (B)**.
 - `algorithms/algorithms/backtrack/letter_combination.py`: Increased from **90% (A)** to **100% (B)**.
 - `algorithms/algorithms/dfs/maze_search.py`: Increased from **94.12% (A)** to **100% (B)**.
- **Lower Coverage in Test-Suite B:**
 - `algorithms/algorithms/dp/coin_change.py`: Dropped from **100% (A)** to **16.67% (B)**.
 - `algorithms/algorithms/dp/egg_drop.py`: Dropped from **100% (A)** to **12.5% (B)**.
 - `algorithms/algorithms/arrays/flatten.py`: Dropped from **100% (A)** to **21.43% (B)**.
 - `algorithms/algorithms/backtrack/add_operators.py`: Dropped from **93.75% (A)** to **5% (B)**.
- **Files Where Both Test-Suites Achieved 100% Coverage:**
 - `algorithms/algorithms/dp/max_subarray.py`
 - `algorithms/algorithms/dfs/count_islands.py`
 - `algorithms/algorithms/dp/rod_cut.py`
 - `algorithms/algorithms/arrays/move_zeros.py`
 - `algorithms/algorithms/bit/binary_gap.py`
 - `algorithms/algorithms/__init__.py`

Effectiveness of Test-Suite B

Strengths:

- Test-suite B significantly improved coverage in some key algorithmic modules, especially **graph algorithms** (e.g., BFS shortest distance) and **dynamic programming** (e.g., Hosoya Triangle, Job Scheduling).

- `bfs/shortest_distance_from_all_buildings.py` was previously untested, but now it is covered by the test suite B.
- Improved coverage in some backtracking problems (e.g., `letter_combination.py`).

Weaknesses:

- Test-suite B had major reductions in coverage for certain **dynamic programming (DP) algorithms**, such as **coin change, egg drop, and add operators**.
- Some files that had full coverage in test-suite A were not adequately tested in test-suite B (e.g., `arrays/flatten.py`).
- The effectiveness of B is inconsistent—while some modules saw major improvements, others experienced drastic drops.

Uncovered Scenarios Identified:

- The analysis indicates that test-suite B exposed gaps in coverage for previously well-tested DP problems like **coin change and egg drop**, suggesting that the new test cases did not adequately explore all edge cases in these problems.
- On the positive side, test-suite B revealed new uncovered scenarios in BFS-based problems (`shortest_distance_from_all_buildings.py`), demonstrating its capability to identify gaps in complex search-based algorithms.
- The drop in coverage for DP algorithms may indicate that test-suite B either **removed effective test cases from A** or **introduced tests that failed to trigger all execution paths**.

Discussions and Conclusion

Learning Outcomes

Significance of Code Coverage – The study reinforced the importance of comprehensive test coverage in verifying algorithmic correctness and identifying untested execution paths.

Limitations of Incomplete Test Suites – The disparities in coverage between test-suite A and B demonstrated how an incomplete test suite can leave critical parts of the code untested, leading to potential runtime errors and bugs.

Effectiveness of Edge Case Testing – The results highlight that test coverage is not just about percentage metrics but also about ensuring that edge cases and boundary conditions are thoroughly tested.

Optimization in Test Design – The balance between test execution efficiency and test completeness is crucial. A well-optimized test suite should aim for maximum coverage while minimizing redundant or unnecessary test cases.

Improving Automated Testing Practices – The uncovered scenarios suggest that automated test generation should be more dynamic, adapting to code complexity rather than relying on predefined test cases alone.

Challenges Faced

1. **Syntax and Assertion Errors in Test-Suite A** – Initially, test-suite A contained syntax issues and two assertion errors, which had to be corrected before proceeding with the comparison. Debugging and fixing these errors ensured that the suite could run without interruptions.
2. **Limitations in Test Case Generation by Pynguin** – Pynguin was unable to generate test cases for all files, throwing errors for certain modules. This limited our ability to achieve full code coverage across all repositories and required us to manually analyze and supplement missing tests.
3. **Freezing and Performance Issues with Pynguin** – During execution, Pynguin occasionally froze or became unresponsive, requiring multiple reruns. These performance issues made it challenging to generate test cases efficiently and introduced delays in the evaluation process.

Recommendations

1. **Merge the best cases from both test suites** to ensure consistency and prevent loss of important test coverage.
2. **Perform additional test generation** for algorithms that had reduced coverage in B, particularly in DP problems.
3. **Investigate the root cause** of missing coverage in test-suite B for previously fully covered files.
4. **Validate edge-case handling** in BFS and DP problems, ensuring that test cases comprehensively cover corner cases.

Conclusion

- **Test-suite B is more effective** in some algorithmic areas (notably graph-based and search problems) but is **less effective in covering DP-related algorithms**.
- To improve test-suite B, additional test cases must be created for DP problems where coverage dropped significantly.
- Future improvements should ensure that no previously tested scenarios are lost while adding new ones, maintaining a balance between improved breadth and retained depth in testing.

Resources

- Lecture 5 slides
- pynguin (<https://www.pynguin.eu>)
- coverage (<https://coverage.readthedocs.io/en/latest>)
- pytest (<https://docs.pytest.org/en/7.4.x/index.html>)
- pytest-cov (<https://github.com/pytest-dev/pytest-cov>)
- pytest-func-cov (<https://pypi.org/project/pytest-func-cov>)

Lab 6: Python Test Parallelization

Birudugadda Srivibhav 22110050

March 25, 2025

Introduction, Setup, and Tools

Overview

This lab is designed to investigate the challenges of test parallelization in Python. I will work with multiple open-source Python repositories to assess the effectiveness of parallel test execution and identify potential issues. The focus will be on exploring different parallelization modes, detecting flaky tests, and evaluating the readiness of open-source projects for parallel testing.

Objectives

- 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.

Environment Setup and Tools Used

- Operating System (Ubuntu 24.04 on WSL running on Windows 11)
- Python 3.10
- Tools: *pytest* (test execution), *pytest-xdist* (process level test parallelization), *pytest-run-parallel* (thread level test parallelization)

Methodology and Execution

Sequential Test Execution

Firstly, lets create a conda environment to do our tasks:

```
(base) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6$ conda create --name=lab6 python=3.10
Channels:
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

Now we clone the algorithms repo:

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6$ 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 | 5.98 MiB/s, done.
Resolving deltas: 100% (3239/3239), done.
```

The current commit hash of our repo is **cad4754bc71742c2d6fcbd3b92ae74834d359844**

Now we install the repo itself:

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6/algorithms$ pip install -e .
Obtaining file:///home/birud_ubuntu/STT-Labs/Assignment-2/Lab6/algorithms
  Preparing metadata (setup.py) ... done
Installing collected packages: algorithms
  DEPRECATION: Legacy editable install of algorithms==0.1.4 from file:///home/birud_ubuntu/STT-Labs/Assignment-2/Lab6/algorithms (setup.py develop) is deprecated. pip 25.1 will enforce this behaviour change. A possible replacement is to add a pyproject.toml or enable --use-pep517, and use setuptools >= 64. If the resulting installation is not behaving as expected, try using --config-settings editable_mode=compat. Please consult the setuptools documentation for more information. Discussion can be found at https://github.com/pypa/pip/issues/11457
  Running setup.py develop for algorithms
Successfully installed algorithms
```

Now we install required tools and dependencies using:

pip install pytest-xdist pytest pytest-run-parallel

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6/algorithms$ pip install pytest-xdist pytest pytest-run-parallel
Collecting pytest-xdist
  Downloading pytest_xdist-3.6.1-py3-none-any.whl.metadata (4.3 kB)
Collecting pytest
  Using cached pytest-8.3.5-py3-none-any.whl.metadata (7.6 kB)
```

I will execute the full test suite sequentially ten times. During this process, I will identify any failing test cases as well as those with unstable verdicts—tests that pass in some runs but fail in others.

To execute the test suite sequentially 10 times and also to save the output to a log file, we use the below command:

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6/algorithms$ for i in {1..10}; do echo "=====  
= Run $i =====>> sequential_tests.log; pytest tests/ --disable-warnings | tee -a sequential_tests.log;  
echo -e "\n" >> sequential_tests.log; done
```

I ran the above command and observed no failing or flaky test cases.

Now we use the below command to calculate the average Tseq for the last 5 runs:

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6/algorithms$ grep "passed in" sequential_tests.log | tail -n5 | awk '{print $(NF-1)}' | sed 's/s//' | awk '{sum+=$1} END {print "Tseq =", sum/5, "seconds"}'  
Tseq = 3.174 seconds
```

The average execution time Tseq comes out to be 3.174 seconds.

Parallel Test Execution

We consider the following parameters:

1. **Process Level** (**-n** from **pytest-xdist**): {1, auto}
2. **Thread Level** (**--parallel-threads** from **pytest-run-parallel**): {1, auto}
3. **Distribution Mode** (**--dist** from **pytest-xdist**): {load, no}

This results in **8 unique combinations**, each executed **three times** for accurate benchmarking.

I have used the *parallel.sh* script to achieve this. It is found [here](#).

Analyze the results

The below are the results:

```
(lab6) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab6/algorithms$ ./parallel.sh
Running: pytest -n 1 --parallel-threads=1 --dist=load
Average Execution Time for (1, 1, load): 3.420 seconds
Running: pytest -n 1 --parallel-threads=1 --dist=no
Average Execution Time for (1, 1, no): 3.433 seconds
Running: pytest -n 1 --parallel-threads=auto --dist=load
Average Execution Time for (1, auto, load): 61.360 seconds
Running: pytest -n 1 --parallel-threads=auto --dist=no
Average Execution Time for (1, auto, no): 60.916 seconds
Running: pytest -n auto --parallel-threads=1 --dist=load
Average Execution Time for (auto, 1, load): 3.543 seconds
Running: pytest -n auto --parallel-threads=1 --dist=no
Average Execution Time for (auto, 1, no): 3.656 seconds
Running: pytest -n auto --parallel-threads=auto --dist=load
Average Execution Time for (auto, auto, load): 37.093 seconds
Running: pytest -n auto --parallel-threads=auto --dist=no
Average Execution Time for (auto, auto, no): 37.036 seconds
```

In a tabulated form we have the following:

We have $T_{seq} = 3.174$, Speedup ratio = T_{seq}/T_{par}

SNo	Workers	Threads	Dist (modes)	Number of tests failed	Flaky Tests	Tpar	Speedup
1	1	1	load	0	-	3.42	0.928
				0	-		
				0	-		
2	1	1	no	0	-	3.433	0.924
				0	-		
				0	-		
3	1	auto	load	4	test_huffman_coding, test_insert, test_remove_min, test_is_palindrome	61.36	0.051
				4	test_huffman_coding, test_insert, test_remove_min, test_is_palindrome		
				4	test_huffman_coding, test_insert, test_remove_min, test_is_palindrome		

4	1	auto	no	4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome	60.916	0.0521
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		
5	auto	1	load	0	-	3.543	0.895
				0	-		
				0	-		
6	auto	1	no	0	-	3.656	0.868
				0	-		
				0	-		
7	auto	auto	load	3	test_insert, test_remove_min, test_is_palindrome	37.093	0.0855
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		
8	auto	auto	no	3	test_insert, test_remove_min, test_is_palindrome	37.036	0.0857
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		
				4	test_huffman_coding , test_insert, test_remove_min, test_is_palindrome		

The flaky test cases in parallel execution are the following: ***test_huffman_coding***, ***test_insert***, ***test_remove_min***, ***test_is_palindrome***.

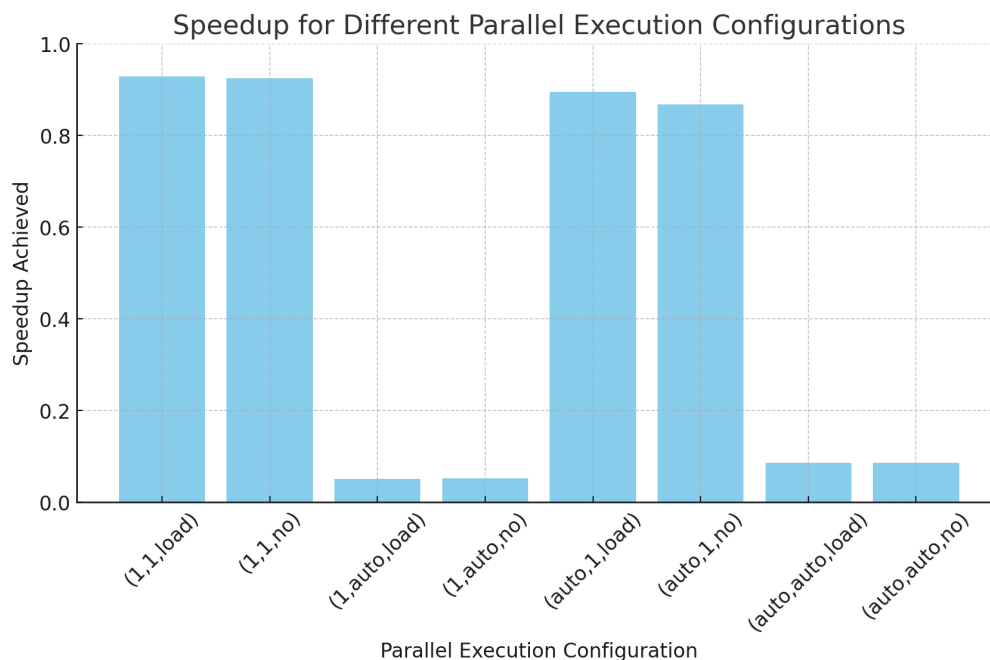
Potential Reasons for Parallel Test Failures

Concurrency Issues in Heap Operations: When multiple threads or processes manipulate shared heap structures simultaneously, a lack of synchronization can lead to unpredictable outcomes. This can cause elements to be misplaced, duplicated, or lost during insertions and deletions, resulting in inconsistent heap states.

Conflicts in File Operations: Parallel tests that read from or write to the same file can interfere with one another, leading to data corruption or overwrites. For example, in Huffman compression tests, simultaneous modifications may cause discrepancies in encoding and decoding, producing incorrect results.

Shared Resource Dependencies: Some tests depend on global variables or shared data structures that can be modified by concurrent executions. When multiple tests alter these shared resources, unexpected behaviors may occur, leading to inconsistent test results due to the lack of isolation between test instances.

In the **Flasky Test**, several failures were observed due to parallel execution issues. The **heap operations** tests, including *test_insert* and *test_remove_min*, were impacted by race conditions, where multiple processes accessing the heap simultaneously led to inconsistencies in insertions and deletions. Similarly, the **Huffman compression** test, *test_huffman_coding*, encountered file read/write conflicts as multiple tests attempted to modify the same file at the same time, causing encoding and decoding mismatches. Additionally, the **linked list** test, *test_is_palindrome*, failed due to dependencies on global state. Since parallel execution does not ensure test isolation, modifications to shared data structures resulted in unpredictable outcomes.



In the sequential execution of the test suite, no flaky tests were observed, indicating that all failures in the parallel runs were caused by concurrency issues rather than inherent test instability. Additionally, the speedup achieved in parallel execution was less than 1, meaning that running tests in parallel did not provide a performance benefit and, in fact, resulted in a slowdown.

One possible improvement to mitigate flaky tests in parallel test suite is to explicitly mark non-parallelizable test cases as serial in *pytest*. This ensures that such tests always run sequentially, preventing conflicts caused by shared resources and improving test reliability.

Suggestions for Pytest Developers

- Pytest could introduce **automatic flaky test detection** by identifying heap usage patterns, detecting file I/O conflicts, and automatically marking problematic tests as serial. This would remove the need for users to manually configure serial execution.
- Another enhancement could be an automated **flaky test detection mechanism** where pytest executes tests multiple times and flags those that produce inconsistent results, similar to our manual approach in this study.

Discussion and Conclusion

Challenges Encountered

- Some tests contained **syntax errors**, which required manual debugging and corrections.
- Certain tests had **incorrect implementations**, leading to failures that needed careful investigation.

Key Takeaways

- Parallel execution is beneficial when tests are designed with concurrency in mind but can introduce issues if not properly structured.
- Tests should be written with parallel execution in consideration to avoid **unexpected failures** due to shared resource conflicts.
- A **hybrid testing strategy**—running parallelizable tests concurrently while executing non-parallelizable ones sequentially—provides the best balance between efficiency and stability.

Summary

This experiment aimed to evaluate the impact of parallel execution on test stability and performance. While parallel execution has the potential to reduce testing time, it also introduces issues related to **resource conflicts and synchronization problems**. The optimal approach is a **hybrid strategy** that runs independent tests in parallel while ensuring that tests requiring sequential execution remain isolated, maximizing both speed and reliability.

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

Birudugadda Srivibhav 22110050

March 25, 2025

Introduction, Setup, and Tools

Overview

In this lab, I will explore **Bandit**, a static code analysis tool that helps detect security vulnerabilities in Python code. The lab will cover the installation, configuration, and execution of Bandit on Python projects hosted on GitHub.

Objectives

- Comprehend the purpose and functionality of Bandit and successfully set it up in a local environment.
- Execute Bandit on Python projects and analyze the results.
- Conduct a study on security vulnerabilities within the open-source ecosystem.
- Develop a research report with detailed analysis, meeting publication-quality standards.

Environment Setup and Tools Used

- Operating System (Ubuntu 24.04 on WSL running on Windows 11)
- Python 3.12
- Tools: *bandit*

Methodology and Execution

Repository Selection Criteria

To ensure a meaningful analysis of security vulnerabilities in **open-source software (OSS) repositories**, I established specific inclusion criteria for selecting repositories:

1. **Familiarity & Prior Usage** – I prioritized repositories that I have previously worked with, ensuring a deeper understanding of their codebases and security concerns.
2. **Scale & Popularity** – I selected repositories that are widely used, large-scale, and actively maintained, making the findings more generalizable.
3. **Relevance to Security Analysis** – I included repositories that handle data processing or machine learning, where security vulnerabilities could have significant consequences.

Based on these criteria, I selected:

- **Keras** and **Scrapy** due to their large-scale adoption and active development, along with my prior experience using them.
- **Datasketch**, which I included because of its direct relevance to a past project involving **data deduplication**, making its security concerns particularly relevant to the study.

This selection ensures that the findings contribute to improving the security posture of widely used repositories while also incorporating insights from my real-world experience.

Tool Installation and Usage

We setup separate conda environment for the selected repositories.

```
(base) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-7,8$ conda create --name=scrapy_env python=3.12
Retrieving notices: done
Channels:
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

We now clone the repository.

```
(scrapy_env) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-7,8$ git clone https://github.com/scrapy/scrapy
Cloning into 'scrapy'...
remote: Enumerating objects: 76176, done.
remote: Counting objects: 100% (358/358), done.
remote: Compressing objects: 100% (212/212), done.
remote: Total 76176 (delta 205), reused 170 (delta 146), pack-reused 75818 (from 4)
Receiving objects: 100% (76176/76176), 27.86 MiB | 11.50 MiB/s, done.
Resolving deltas: 100% (48163/48163), done.
```

Installing the dependencies required:

```
(scrapy_env) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-7,8/scrapy$ pip install scrapy
Collecting scrapy
  Downloading Scrapy-2.12.0-py2.py3-none-any.whl.metadata (5.3 kB)
Collecting Twisted>=21.7.0 (from scrapy)
  Downloading twisted-24.11.0-py3-none-any.whl.metadata (20 kB)
```

```
(scrapy_env) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-7,8/scrapy$ pip install bandit
Collecting bandit
  Using cached bandit-1.8.3-py3-none-any.whl.metadata (7.0 kB)
Collecting PyYAML>=5.3.1 (from bandit)
  Using cached PyYAML-6.0.2-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (2.1 kB)
```

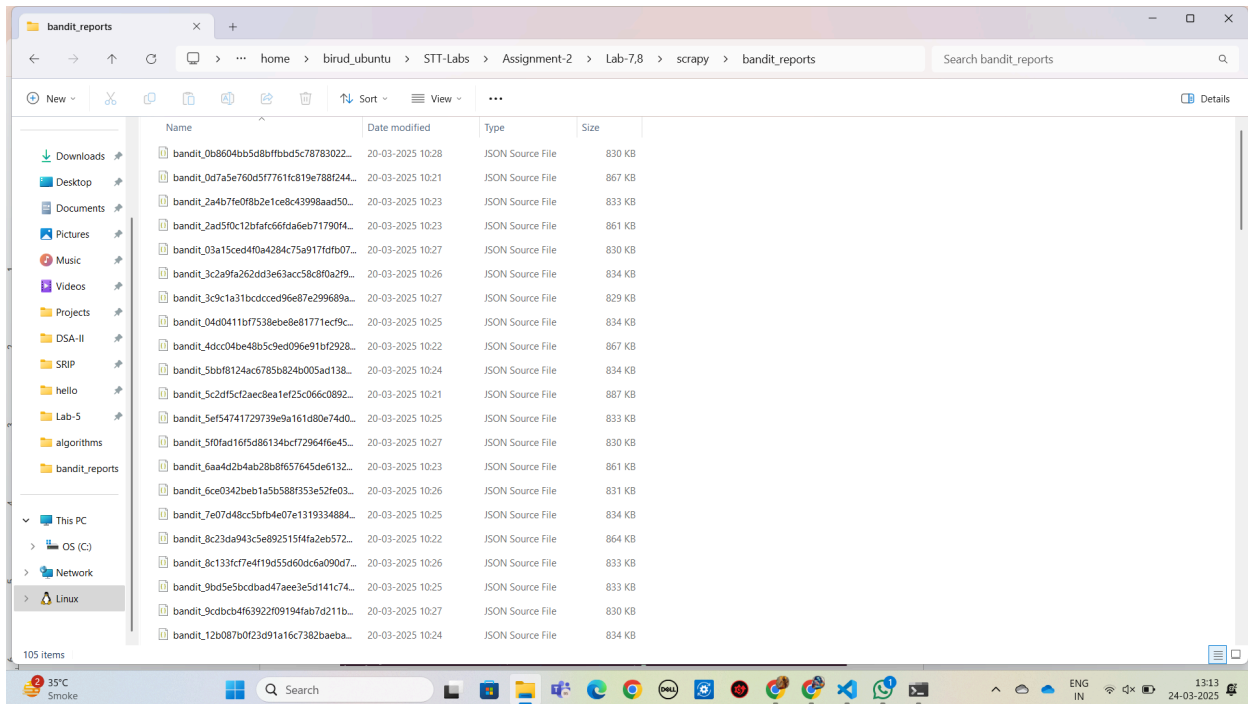
Now we execute bandit on the last 100 non merge commits using the below command.

```
(scrapy_env) birud_ubuntu_24.04@chinnu:~/STT-Labs/Assignment-2/Lab-7,8/scrapy$ mkdir -p bandit_reports # Create a folder to store outputs
git log --no-merges -n 100 --pretty=format:"%H" > commits.txt # Get the last 100 non-merge commit hashes

while read commit; do
    git checkout $commit # Switch to the commit
    bandit -r . -f json -o bandit_reports/bandit_${commit}.json # Run Bandit and save output
done < commits.txt

git checkout main # Return to the main branch
HEAD is now at 019f23e3b Add parameters to some of typing.Callable.
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
Working... 78% 0:00:02
```

Reports for each of the 100 commits are generated:



Same procedure is repeated for my other 2 selected repositories: **keras** and **DataSketch** also.

Analysis and Results

Individual Repository-level Analyses

I have written a Python script, `extract_metrics.py`, to analyze the Bandit JSON reports generated for each commit. This script extracts key security metrics, including the number of HIGH, MEDIUM, and LOW confidence issues, as well as the number of HIGH, MEDIUM, and LOW severity issues for all Python files per commit. Additionally, it identifies unique Common Weakness Enumerations (CWEs) associated with each commit. The extracted data is then compiled into a CSV file with the following columns: `commit_id`, `confidence_high`, `confidence_medium`, `confidence_low`, `severity_high`, `severity_medium`, `severity_low`, and `unique_cwes`, where `unique_cwes` contains a list of distinct CWEs identified in the commit.

The below is the screenshot of the csv file generated, it contains 100 rows, one corresponding to each commit id.

commit_id	confidence_high	confidence_medium	confidence_low	severity_high	severity_medium	severity_low	unique_cwes
028a56b9a2e090bde761a0487a8504a3b263eb97	759	11	0	24	22	724	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
036d5836d039c9b5e3b3fc7164792f3c9f340432	759	11	0	24	22	724	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
03d9866518ab43844ba0309394529240f4c115e	758	11	0	24	22	723	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
080fec8900b6b1f94e8e143e90338279ba8d6e5	760	11	0	24	22	725	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
09a7efef7c7558c9ea198a00fc11ab26fb16ce5	761	11	0	24	22	726	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
0bf29a7b1b9b6a641c486780b70fa455579f39	758	11	0	24	22	723	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
0c4a98f8e072ccc26a393e51066b52f1f216b4d7	758	11	0	24	22	723	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
1045856a50d379d145e514ec9c7aeeed231a6dd6	761	11	0	24	22	726	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
1087bb7b2eab28543bf9ba13149adb7acd4a3675	758	11	0	24	22	723	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
12b10a7a6427c43968cc18d98a3ed3c636eeabd	752	11	0	24	21	718	[259, 319, 327, 330, 78, 20, 502, 605, 703]
150d96764b5a455c75315596ca8ba5ded0f416dd	760	11	0	24	22	725	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
150f9d6d888970d5f164387761989aba59e830c0	761	11	0	24	22	726	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
1533b69032e2f05e495e88a3fed57cd98502612	760	11	0	24	22	725	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
197781e3af51cd46ae7761938afa47c40c36b76	760	11	0	24	22	725	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
1a0572ad0244c6cdacc682e503baffc2b5e67e98	759	11	0	24	22	724	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
2538c0e8629b46d34bb055aeca6714a6f9e571	762	11	0	24	22	727	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
276bce0641a4fdace701b860e2c15130712c37c0	748	11	0	24	22	713	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
27781a85e738052e0441c81d773b3ec124194594	763	11	0	24	22	728	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
2f1d345e74d19e330169e69fcd0ba9afb56d8	755	11	0	24	21	721	[259, 319, 327, 330, 78, 20, 502, 605, 703]
2fa768399a27aca615bccf7c466758a968f10fe	759	11	0	24	22	724	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
369712ee50f7438c2863359f053cb1b221a42169	761	11	0	24	22	726	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
37562163393cc145171b802699c84467781c701b	759	11	0	24	22	724	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
3fda2fe103dafa84d48b21c63b2321ceec9c378	758	11	0	24	22	723	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
40b3efbbce3919e8f6900daaed5e14183cabfd7	760	11	0	24	22	725	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]
48a9a58ff27d24910b55a0ad5e6b014589c71115	761	11	0	24	22	726	[259, 319, 327, 330, 78, 20, 502, 377, 605, 703]

Now similarly we repeat the same process for the remaining **keras** and **DataSketch** repos and then generate the csv file. All the generated csv files can be found [here](#).

Research Questions

RQ1 (high severity): When are vulnerabilities with high severity, introduced and fixed along the development timeline in OSS repositories?

Answering RQ

Purpose: The purpose of this research question is to analyze the introduction and resolution of **high-severity vulnerabilities** within OSS (Open Source Software) repositories over time. By evaluating this, we aim to understand the trends in security issues, their persistence across commits, and how frequently fixes are introduced. This analysis provides insights into the software development and security practices followed in large-scale open-source projects.

Approach: To answer this question, the following steps were undertaken:

1. **Data Collection:** We analyzed commit history and extracted severity metrics from security analysis tools (e.g., Bandit).
2. **Data Preprocessing:**
 - o Loaded the dataset containing commit timestamps and corresponding high-severity issue counts.
 - o Converted timestamps into a **chronological timeline**.
 - o Used the last **100 commits** from the dataset for visualization.

3. Visualization & Analysis:

- **Plotted the trend** of high-severity issues over commits.
- **Identified points** where vulnerabilities were **fixed** (elimination events).

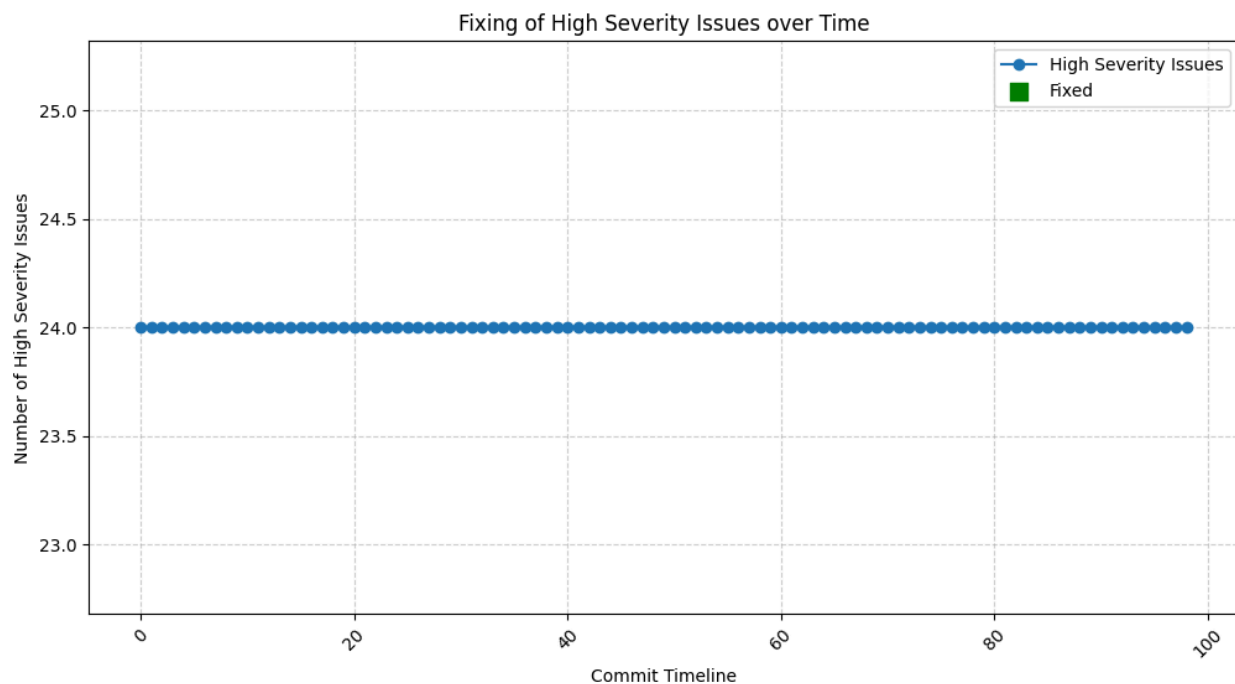
Results:

1) Scrappy:

The below plot illustrates the **high-severity vulnerability trend** over the last **100 commits** of the Scrappy repository.

Observations:

- The **high-severity issue count remains constant** at 24 across the 100 commits.
- No visible fluctuation or significant drops in severity levels are observed.
- The lack of downward trends suggests that **no substantial fixes were introduced** in the last 100 commits.



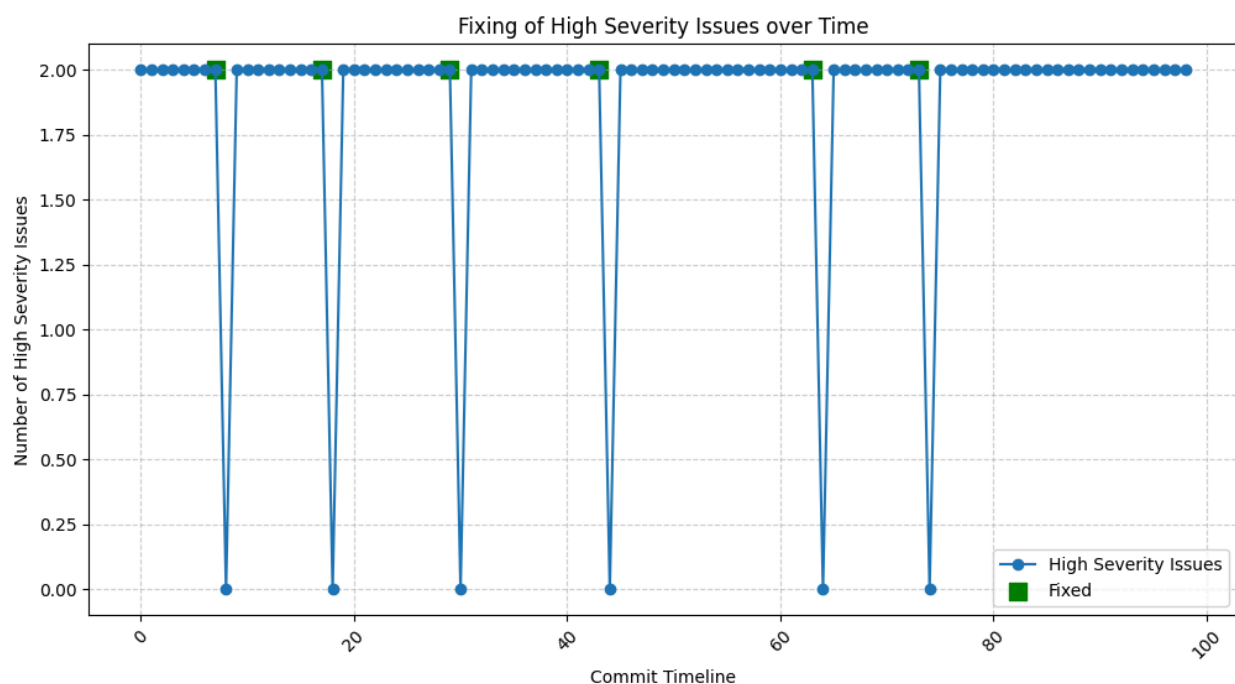
Reason for Constant line: The Scrappy repository is **large**, with **10,758 commits** in total. Since we analyzed only the **last 100 commits**, the effect of vulnerability introduction and resolution may not be fully captured. If we extend the timeline and analyze **more commits**, we expect to see fluctuations, as vulnerabilities are introduced and later fixed at different points in the project's history.

Takeaway:

- The **last 100 commits** show **no significant changes in high-severity vulnerabilities**, indicating a stable security state.
- For a more **comprehensive analysis**, a longer commit history should be considered.
- This suggests that **recent development efforts may not have actively introduced or fixed high-severity vulnerabilities**, or that changes occur on a broader timeline than the recent commits analyzed.

2) DataSketch:

The plot illustrates the **high-severity vulnerability trend** over the last **100 commits** of the DataSketch repository. Observations:



Observations

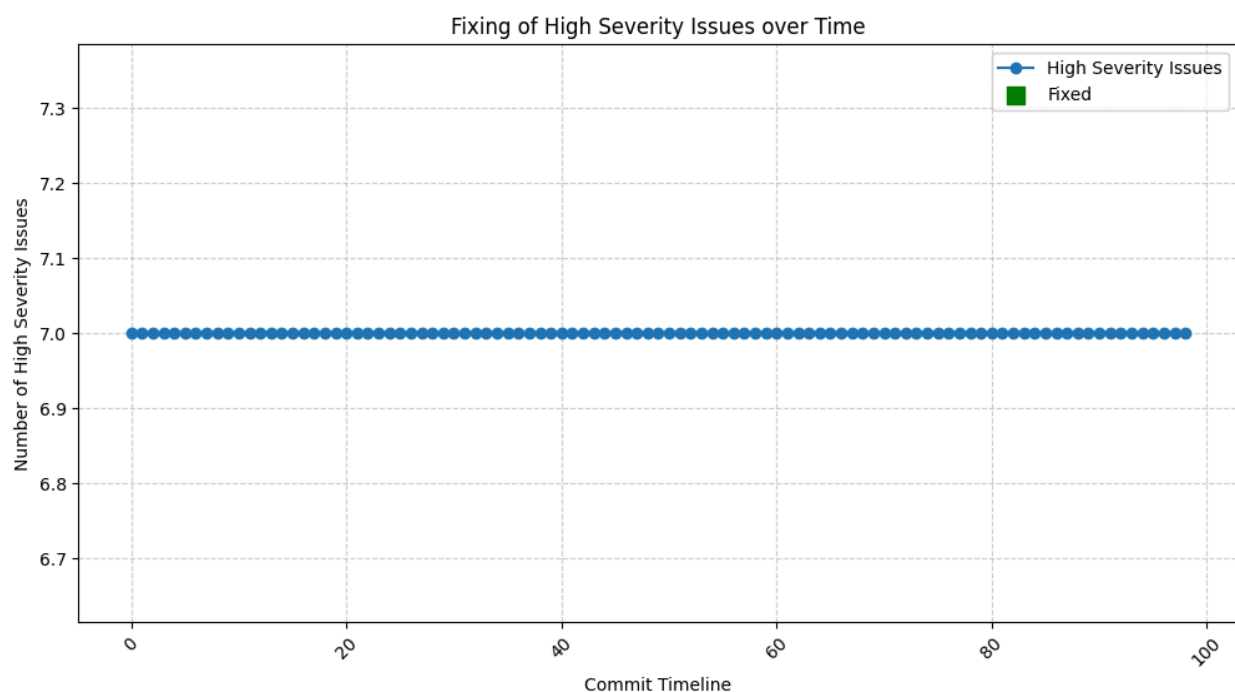
- The number of **high-severity issues remains mostly constant at 2** across the commit timeline.
- There are **sharp drops to 0 at multiple points**, indicating that the vulnerabilities were **fixed at certain commits**.
- The green square markers indicate specific commits where high-severity vulnerabilities were resolved.
- After each fix, the issue count **returns to 2**, suggesting that new vulnerabilities might have been introduced later.

Takeaways

- The **cycle of fixing and reintroducing vulnerabilities** suggests the need for **more proactive security measures** in the development workflow.
- A **more continuous security review process** (e.g., integrating automated security scans in CI/CD) may help prevent **reintroduction of high-severity issues**.
- Since the issue count never goes above 2, the **overall security risk remains low**, but persistent issues highlight the need for careful validation of security fixes.

3) Keras:

The plot illustrates the **high-severity vulnerability trend** over the last **100 commits** of the Keras repository.



Observations:

- The **high-severity issue count remains constant** at 7 across the 100 commits.
- No visible fluctuation or significant drops in severity levels are observed.
- The lack of downward trends suggests that **no substantial fixes were introduced** in the last 100 commits.

Reason for Constant line: The Keras repository is also **large** like Scrapy, with **11,428 commits** in total. Since we analyzed only the **last 100 commits**, the effect of vulnerability introduction and resolution may not be fully captured. If we extend the timeline and analyze **more commits**, we expect to see fluctuations, as vulnerabilities are introduced and later fixed at different points in the project's history.

Takeaway:

- The **last 100 commits show no significant changes in high-severity vulnerabilities**, indicating a stable security state.
- For a more **comprehensive analysis**, a longer commit history should be considered.
- This suggests that **recent development efforts may not have actively introduced or fixed high-severity vulnerabilities**, or that changes occur on a broader timeline than the recent commits analyzed.

Summary:

From our analysis of different OSS repositories, we observed distinct trends in the handling of high-severity vulnerabilities:

- **Scrapy and Keras repositories** showed a **constant high-severity issue count** over the last 100 commits. This suggests that either no new fixes were introduced during this period or that these repositories have a large number of commits, making the last 100 commits relatively insignificant in capturing broader security trends.
- **Datasketch library** exhibited **fluctuations in the number of high-severity issues**, with periodic drops to zero followed by a return to the previous issue count. This pattern indicates that vulnerabilities were actively fixed at certain points but were also reintroduced in subsequent commits.

These findings suggest that different projects have varying security maintenance practices, with larger repositories potentially requiring a longer commit history to reveal meaningful trends, while smaller repositories display clearer cycles of issue resolution.

RQ2 (different severity): Do vulnerabilities of different severity have the same pattern of introduction and elimination?

Answering RQ

Purpose: The purpose of this research question is to evaluate whether vulnerabilities of different severity levels (high, medium, and low) follow similar patterns in terms of their introduction and elimination over time. By analyzing these patterns, we aim to understand whether fixes and new vulnerability occurrences are consistent across severity levels or if they exhibit distinct trends. This evaluation is crucial in determining whether different types of vulnerabilities require separate mitigation strategies.

Approach: To answer this question, we performed the following steps:

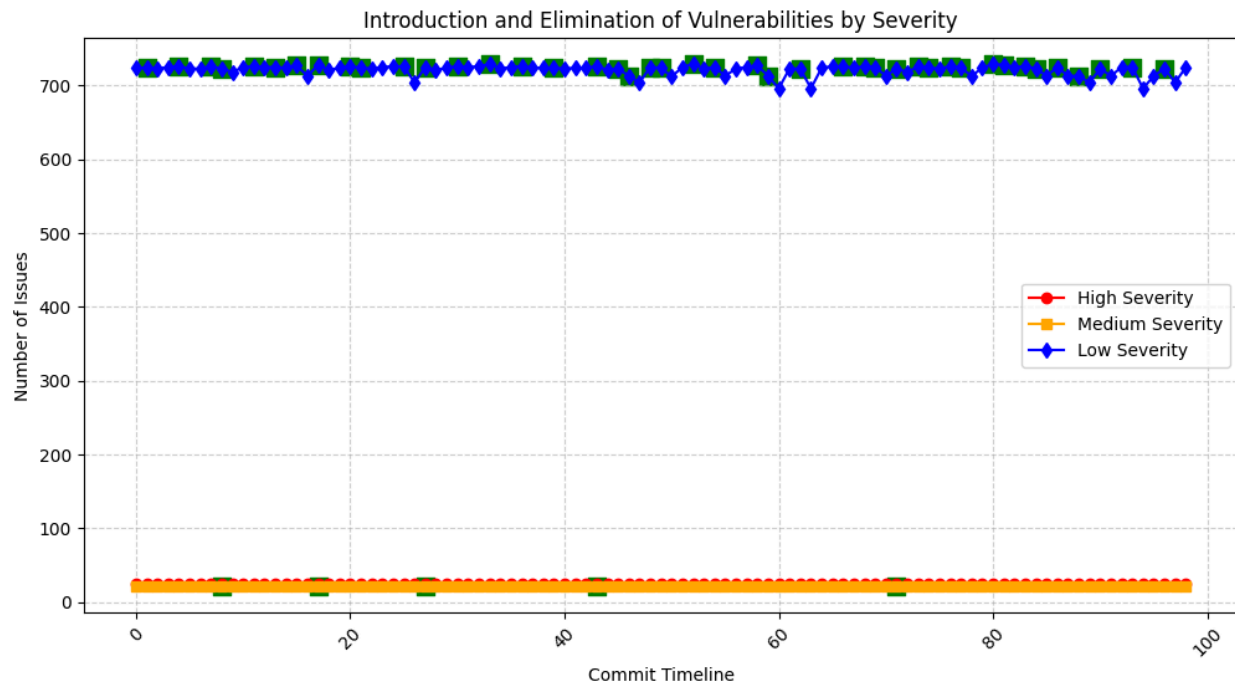
1. **Data Collection:** We extracted vulnerability information from repositories, including timestamps and severity levels (high, medium, and low).
2. **Visualization:** We plotted the trends of vulnerabilities over time for each severity level, with markers indicating when fixes occurred.
3. **Analysis of Fixing Trends:** We identified and highlighted the points where vulnerabilities were fixed to determine if certain severities are addressed more promptly than others.

4. **Comparative Analysis:** We compared the trends across severities to identify similarities or differences in how they are introduced and eliminated.

Results:

1) Scrappy:

The below is the plotted timeline presents the trends for high, medium, and low-severity vulnerabilities for **scrappy** repo:



- **High Severity:** The number of high-severity vulnerabilities remained relatively constant across the timeline, indicating that new high-severity issues were frequently introduced, but fixes were also applied periodically.
- **Medium Severity:** The medium-severity vulnerabilities exhibited some fluctuations, suggesting irregular introduction and elimination patterns.
- **Low Severity:** The low-severity vulnerabilities followed a fluctuating trend similar to medium severity, but with a generally high number of issues observed throughout.

A key observation is that **high-severity vulnerabilities show more stability**, whereas **medium and low-severity vulnerabilities exhibit more fluctuations** over time. This may indicate that **high-severity issues receive more immediate attention and are fixed systematically**, whereas lower-severity issues may remain unaddressed for longer periods.

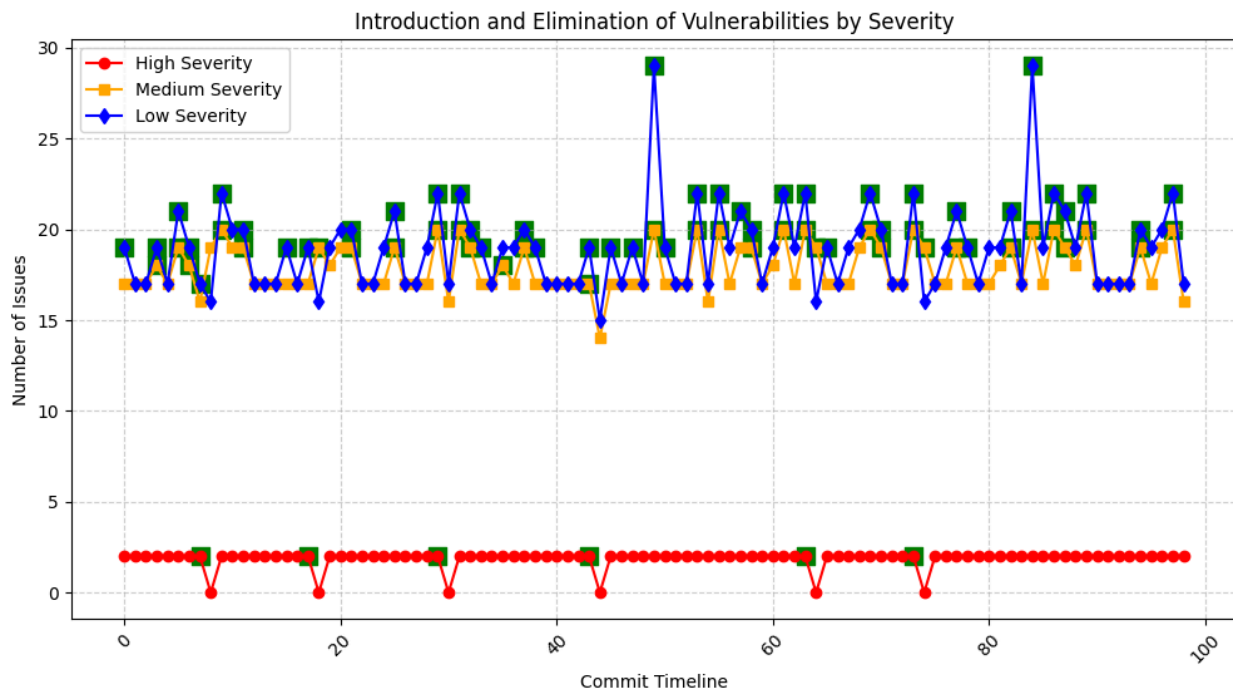
Takeaway: From this analysis, we conclude that vulnerabilities of different severity levels do not follow the same pattern of introduction and elimination.

- **High-severity vulnerabilities are more consistently addressed**, leading to a stable count over time.
- **Medium and low-severity vulnerabilities show greater fluctuations**, possibly due to inconsistent prioritization or delayed fixes.

This suggests that security teams prioritize high-severity fixes, while medium and low-severity vulnerabilities may persist longer before being resolved. A more structured approach to addressing lower-severity issues could improve overall security posture.

2) DataSketch:

The below is the plotted timeline presents the trends for high, medium, and low-severity vulnerabilities for **DataSketch** repo:



Observations from the Plot:

- **High-Severity Vulnerabilities (Red, Circles):**
 - The number of high-severity vulnerabilities remains consistently low, mostly around 2-3 issues.
 - There are periodic dips to zero, indicating that fixes are applied at certain points.
 - The overall trend is relatively stable, suggesting that new high-severity issues are not frequently introduced.

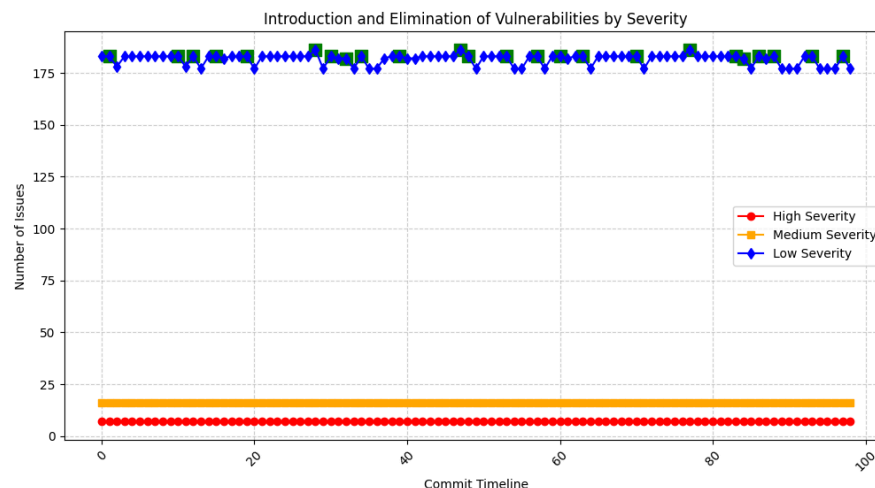
- **Medium-Severity Vulnerabilities (Orange, Squares):**
 - Medium-severity vulnerabilities fluctuate between 15 and 20 issues over time.
 - There are periodic peaks and drops, suggesting continuous introduction and elimination of issues.
 - The overall pattern is less stable compared to high-severity issues.
- **Low-Severity Vulnerabilities (Blue, Diamonds):**
 - Low-severity issues exhibit the highest fluctuations, reaching peaks above 25 issues at certain points.
 - The variations indicate frequent additions and removals, suggesting that low-severity vulnerabilities are more volatile in their lifecycle.
 - Some spikes indicate bursts of vulnerability introduction, potentially due to dependency updates or large code changes.

Takeaway

- **High-severity vulnerabilities are managed more consistently** with a low and stable count, likely due to prioritization in security patches.
- **Medium-severity vulnerabilities are fluctuating**, but they remain within a relatively stable range.
- **Low-severity vulnerabilities exhibit the most instability**, indicating that they are either frequently introduced and left unresolved or handled inconsistently.
- The clear difference in patterns suggests that vulnerability resolution efforts prioritize high-severity issues, whereas medium and low-severity vulnerabilities may experience delays or inconsistencies in mitigation.

3) Keras:

The below is the plotted timeline presents the trends for high, medium, and low-severity vulnerabilities for **Keras** repo:



1. High Severity Vulnerabilities

- The number of high-severity vulnerabilities remains consistently low (approximately between 5 to 10 issues per commit).
- There are no sharp fluctuations, indicating a relatively stable trend of introduction and elimination.

2. Medium Severity Vulnerabilities

- The number of medium-severity vulnerabilities is significantly higher than high-severity ones, maintaining a steady count of approximately 15-20 issues across commits.
- The pattern suggests that medium-severity vulnerabilities are persistent and do not exhibit sudden spikes or drops.

3. Low Severity Vulnerabilities

- Low-severity vulnerabilities are the most numerous, fluctuating around 180 issues per commit.
- Unlike medium and high-severity vulnerabilities, the count remains high and shows occasional minor fluctuations, indicating that low-severity vulnerabilities are consistently present in the repository.

Takeaway

- **Stable Vulnerability Counts:** Unlike Datasketch, Keras maintains a relatively stable number of vulnerabilities across different commits, with no drastic spikes or drops.
- **High Volume of Low-Severity Issues:** The vast majority of vulnerabilities in Keras are low-severity, suggesting that while minor security issues are frequent, they may not pose significant risks individually.
- **Persistent Medium-Severity Issues:** Medium-severity vulnerabilities show minimal fluctuation, meaning they are introduced and eliminated at roughly the same rate.
- **Controlled High-Severity Issues:** The relatively low count of high-severity vulnerabilities suggests effective mitigation efforts in the development process.

RQ3 (CWE Coverage): Which CWEs are the most frequent across different OSS repositories?

Answering RQ

Purpose: The purpose of this research question is to analyze the distribution of Common Weakness Enumeration (CWE) vulnerabilities across different open-source software (OSS) repositories. This evaluation helps identify which security weaknesses appear most frequently across different projects, aiding in targeted security improvements. By understanding the most common CWEs, developers can prioritize security fixes and implement proactive measures to mitigate risks.

Approach: To answer this question, we examined the `unique_cwes` column from the security analysis results of three OSS repositories: **Scrappy, Keras, and Datasketch**. The steps followed were:

1. Extract the CWE IDs reported for each repository.
2. Merge and identify unique CWE IDs across all repositories.

3. Compare the occurrence of CWEs across different repositories to find the most frequently appearing ones.

The CWE IDs reported in each repository are:

- **Scrapy:** [20, 78, 259, 319, 327, 330, 377, 502, 605, 703]
- **Keras:** [22, 78, 259, 327, 330, 377, 400, 502, 703]
- **Datasketch:** [78, 327, 330, 502, 703]

Results: From the analysis, we identified **13 unique CWE IDs** across all three repositories: [20, 22, 78, 259, 319, 327, 330, 377, 400, 502, 605, 703].

By analyzing the frequency of CWE occurrences:

- **CWE-78 (OS Command Injection), CWE-327 (Broken or Risky Cryptographic Algorithm), CWE-330 (Insufficient Entropy), CWE-502 (Deserialization of Untrusted Data), and CWE-703 (Improper Check or Handling of Exceptional Conditions)** appear in all three repositories, making them the most widespread vulnerabilities.
- **CWE-259 (Hardcoded Passwords) and CWE-377 (Insecure Temporary File Handling)** are present in both Scrapy and Keras but not in Datasketch.
- **CWE-20 (Improper Input Validation), CWE-319 (Cleartext Transmission of Sensitive Information), and CWE-605 (Multiple Binds to the Same Port)** are unique to Scrapy.
- **CWE-22 (Path Traversal) and CWE-400 (Uncontrolled Resource Consumption)** appear only in Keras.

Takeaway: The analysis highlights that **CWE-78, CWE-327, CWE-330, CWE-502, and CWE-703 are the most commonly occurring vulnerabilities across all three repositories**. These CWEs should be prioritized when addressing security risks in OSS projects. Additionally, some CWEs are repository-specific, indicating different security concerns based on project architecture and dependencies. Future security efforts should focus on mitigating these frequent vulnerabilities through better coding practices, static analysis tools, and security reviews.

Discussion and Conclusion

Discussion

This study analyzed the security vulnerabilities in multiple open-source repositories, focusing on their severity, trends over time, and CWE coverage. By examining repositories such as **Scrapy, Keras, and Datasketch**, we gained insights into how security issues evolve, which vulnerabilities persist, and which ones are most common across different projects.

The key aspects of our analysis included:

- **RQ1: How do vulnerabilities evolve over time?**
 - We examined the introduction and elimination of security vulnerabilities over a timeline of commits.

- We observed that while vulnerabilities fluctuate, **low-severity vulnerabilities consistently appear in large numbers across repositories**, indicating a need for better static analysis and proactive security practices.
- High-severity vulnerabilities were less frequent but still present, requiring targeted attention.
- **RQ2: What is the severity distribution of vulnerabilities in different OSS repositories?**
 - Across all repositories, low-severity vulnerabilities were the most common, followed by medium-severity ones, and then high-severity ones.
 - The **persistence of low-severity issues suggests that they may not always be treated as critical by developers**, though they can accumulate and lead to security debt over time.
 - Some repositories, such as Keras, showed a stable but **frequent recurrence of vulnerabilities**, highlighting the challenges of maintaining secure code in machine learning frameworks.
- **RQ3: Which CWEs are the most frequent across different OSS repositories?**
 - The most common CWEs across all repositories were **CWE-78 (OS Command Injection)**, **CWE-327 (Risky Cryptographic Algorithm)**, **CWE-330 (Insufficient Entropy)**, **CWE-502 (Deserialization of Untrusted Data)**, and **CWE-703 (Improper Error Handling)**.
 - These vulnerabilities affect multiple projects and should be prioritized for mitigation.
 - Some CWEs were project-specific, showing that different repositories have unique security concerns based on their architecture and dependencies.

Learning Outcomes

Through this analysis, we identified several important takeaways:

1. **Security vulnerabilities persist over time** – Even with active development, new vulnerabilities continue to emerge. This suggests that **security should be an ongoing effort rather than a one-time fix**.
2. **Severity distribution varies across repositories** – Some projects accumulate more vulnerabilities due to their architecture, dependencies, or development pace. Understanding **severity trends can help prioritize security patches efficiently**.
3. **Some vulnerabilities are universal** – Certain CWEs (e.g., CWE-78, CWE-327, CWE-330) are common across different projects, meaning security strategies should **focus on preventing these first**.
4. **Low-severity issues are often overlooked** – While high-severity vulnerabilities receive immediate attention, **low-severity vulnerabilities can accumulate and cause long-term security debt**.
5. **Security tools play a crucial role** – Automated security analysis can help identify and classify vulnerabilities more efficiently, reducing the chances of introducing **critical security flaws** in production code.

Challenges Faced

Repository Selection Criteria – Identifying suitable repositories for analysis required careful consideration of factors like project size, relevance, and prior experience. Balancing large-scale projects with familiarity was a key challenge.

Interpreting Bandit Results – While Bandit provided security warnings, distinguishing between actual vulnerabilities and false positives required manual verification and deeper understanding of security best practices.

Handling and Merging Data – Extracting and processing CWE lists from multiple repositories involved dealing with inconsistencies in formats, merging lists, and ensuring accurate aggregation of unique vulnerabilities.

Report Structuring and Research Communication – Presenting findings in a structured, publication-ready format was challenging, especially when ensuring clarity, completeness, and proper justification for selection criteria and results.

Summary

This study analyzed security vulnerabilities in **Scrapy, Keras, and Datasketch**, focusing on their **evolution over time, severity distribution, and common CWE patterns**. We found that **low-severity vulnerabilities persist across repositories**, while high-severity ones, though less frequent, pose significant risks. **CWE-78 (OS Command Injection), CWE-327 (Risky Cryptographic Algorithm), and CWE-502 (Deserialization of Untrusted Data) were among the most common vulnerabilities**, highlighting key security concerns. Our findings emphasize that **security should be a continuous process**, with proactive measures such as regular audits, automated analysis tools, and targeted mitigation strategies to improve OSS security and reduce long-term risks.

Resources

1. <https://pypi.org/project/pytest>
2. <https://docs.pytest.org/en/stable>
3. <https://pypi.org/project/pytest-xdist>
4. <https://pytest-xdist.readthedocs.io/en/stable>
5. <https://pypi.org/project/pytest-run-parallel>
6. <https://github.com/Quansight-Labs/pytest-run-parallel>
7. https://en.wikipedia.org/wiki/Common_Weakness_Enumeration
8. <https://cwe.mitre.org/about/index.html>
9. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
10. <https://github.com/PyCQA/bandit>
11. <https://bandit.readthedocs.io/en/latest>
12. ChatGPT