# Assignment 2

*Software Tools and Techniques for CSE*

**Akshat Shah & Rahul Singal**

Roll Number:23110293 & 23110265

Git Hub Repository for this Assessment
https://github.com/Akshat13shah/STT_Ass2.git

# Lab 6

## Introduction

This Lab was conducted on 8 September 2025.

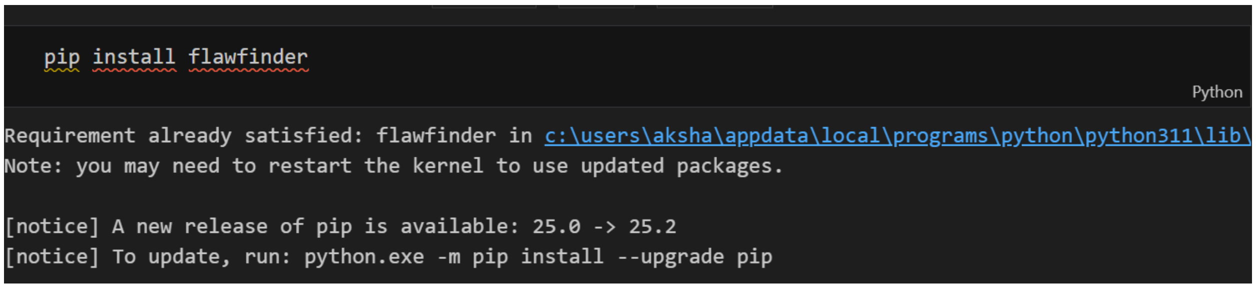
The objective of this lab was to evaluate and compare multiple vulnerability analysis tools based on their detection of software weaknesses categorized by Common Weakness Enumeration (CWE). The experiment focused on analyzing CWE coverage and pairwise similarity between tools using the Intersection over Union (IoU) metric.

## Tools

- Cppcheck – Static analysis tool for C/C++ code quality and potential runtime issues
- Flawfinder – Security-oriented analyzer focusing on function-level vulnerabilities and buffer overflows.
- Semgrep – Pattern-based vulnerability scanner with rules for security, correctness, and style checks.
- Python 3.11+ – For automating tool execution, parsing JSON/CSV reports, and computing IoU values.
- Pandas, Matplotlib, Seaborn – Used for data processing, visualization, and statistical analysis of CWE overlaps.
- GitHub SEART Engine – for searching and selecting suitable repositories.

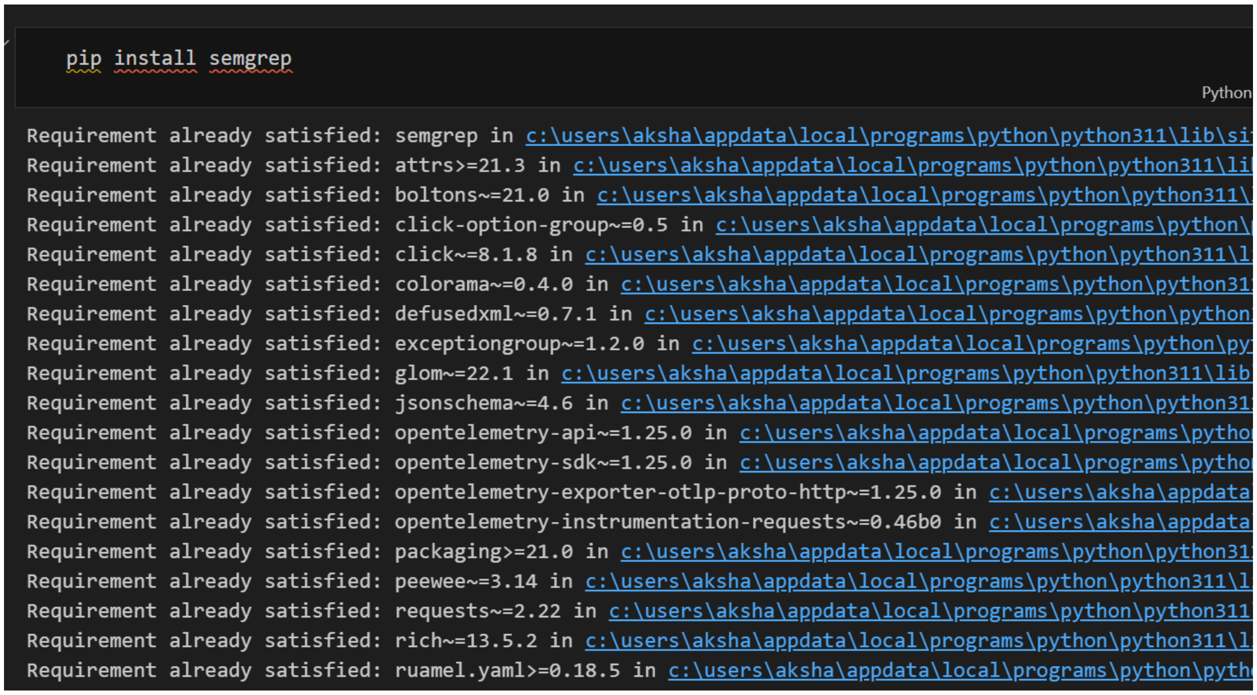
## Set-Up

- Have to install all the tools



```
pip install flawfinder
```

```
Requirement already satisfied: flawfinder in c:\users\aksha\appdata\local\programs\python\python311\lib\
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 25.0 -> 25.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

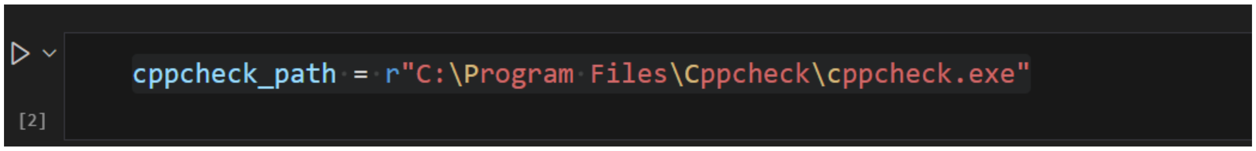


```
pip install semgrep
```

Python

```
Requirement already satisfied: semgrep in c:\users\aksha\appdata\local\programs\python\python311\lib\si
Requirement already satisfied: attrs>=21.3 in c:\users\aksha\appdata\local\programs\python\python311\li
Requirement already satisfied: boltons~=21.0 in c:\users\aksha\appdata\local\programs\python\python311\
Requirement already satisfied: click-option-group~=0.5 in c:\users\aksha\appdata\local\programs\python\
Requirement already satisfied: click~=8.1.8 in c:\users\aksha\appdata\local\programs\python\python311\l
Requirement already satisfied: colorama~=0.4.0 in c:\users\aksha\appdata\local\programs\python\python31
Requirement already satisfied: defusedxml~=0.7.1 in c:\users\aksha\appdata\local\programs\python\python
Requirement already satisfied: exceptiongroup~=1.2.0 in c:\users\aksha\appdata\local\programs\python\py
Requirement already satisfied: glom~=22.1 in c:\users\aksha\appdata\local\programs\python\python311\lib
Requirement already satisfied: jsonschema~=4.6 in c:\users\aksha\appdata\local\programs\python\python31
Requirement already satisfied: opentelemetry-api~=1.25.0 in c:\users\aksha\appdata\local\programs\pytho
Requirement already satisfied: opentelemetry-sdk~=1.25.0 in c:\users\aksha\appdata\local\programs\pytho
Requirement already satisfied: opentelemetry-exporter-otlp-proto-http~=1.25.0 in c:\users\aksha\appdata
Requirement already satisfied: opentelemetry-instrumentation-requests~=0.46b0 in c:\users\aksha\appdata
Requirement already satisfied: packaging>=21.0 in c:\users\aksha\appdata\local\programs\python\python31
Requirement already satisfied: peewee~=3.14 in c:\users\aksha\appdata\local\programs\python\python311\l
Requirement already satisfied: requests~=2.22 in c:\users\aksha\appdata\local\programs\python\python311
Requirement already satisfied: rich~=13.5.2 in c:\users\aksha\appdata\local\programs\python\python311\l
Requirement already satisfied: ruamel.yaml>=0.18.5 in c:\users\aksha\appdata\local\programs\python\pyth
```

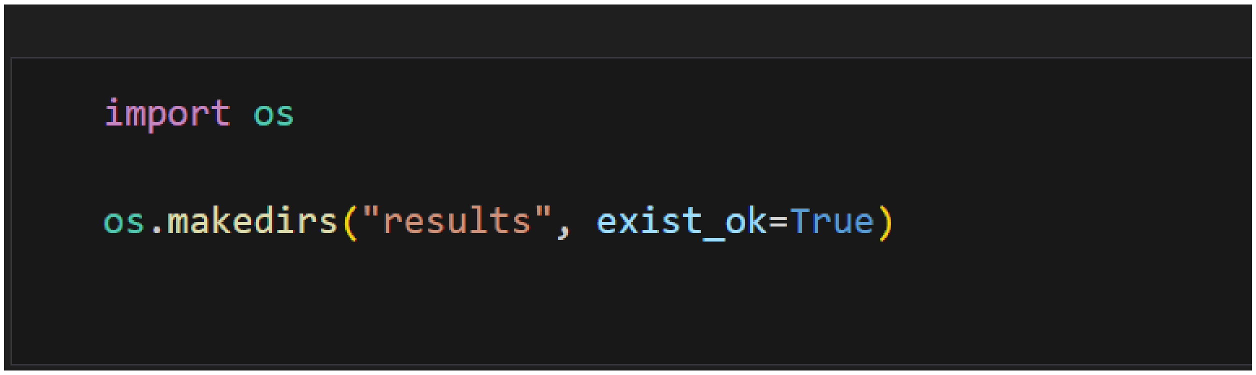
And got the path for cppchecker



```
cppcheck_path = r"C:\Program Files\Cppcheck\cppcheck.exe"
```

[2]

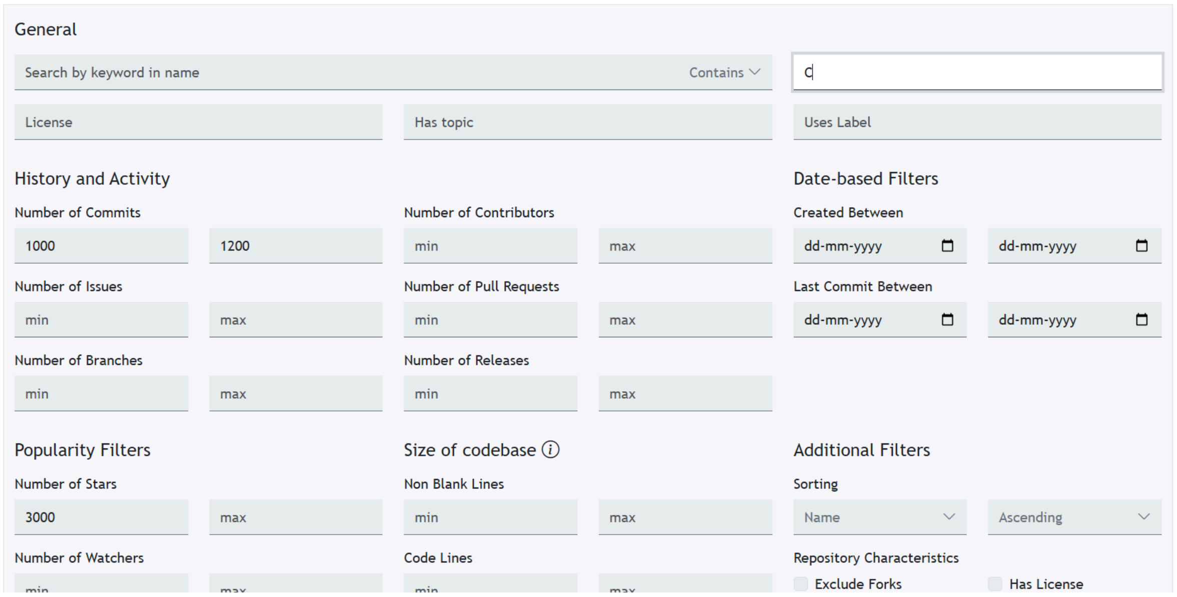
- Created a result folder to store the execution of each tool on each repository



```
import os

os.makedirs("results", exist_ok=True)
```

## Methodology and Execution

1) Selected 3 repositories with the help of the SEART GitHub Search Engine with the following criteria, and then cloned them.



General

Search by keyword in name — Contains — C

License | Has topic | Uses Label

History and Activity

Number of Commits: 1000 – 1200

Number of Contributors: min – max

Number of Issues: min – max

Number of Pull Requests: min – max

Number of Branches: min – max

Number of Releases: min – max

Date-based Filters

Created Between: dd-mm-yyyy – dd-mm-yyyy

Last Commit Between: dd-mm-yyyy – dd-mm-yyyy

Popularity Filters

Number of Stars: 3000 – max

Number of Watchers

Size of codebase

Non Blank Lines: min – max

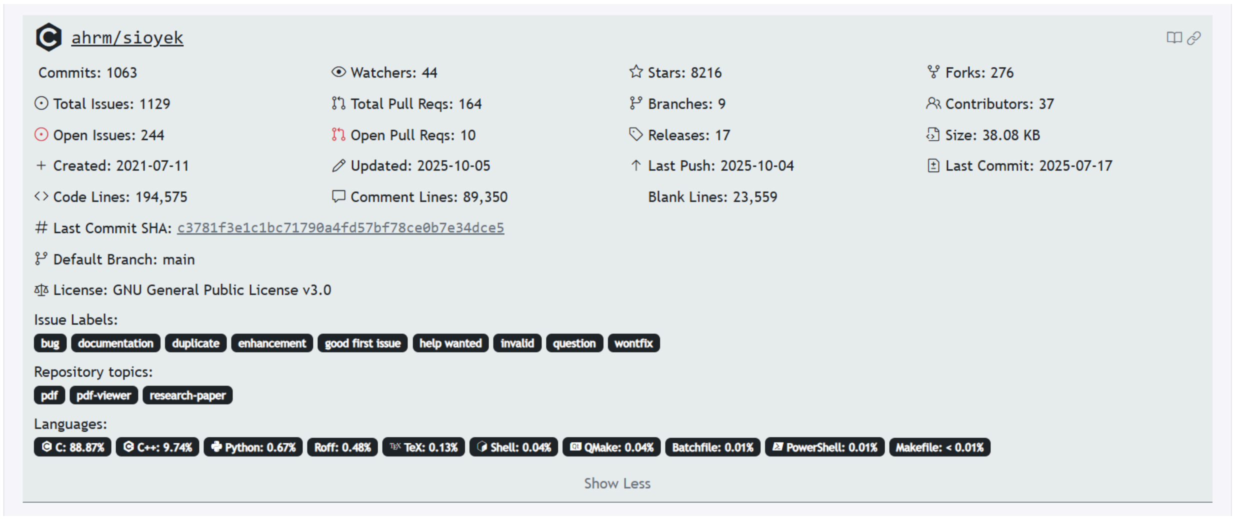
Code Lines

Additional Filters

Sorting: Name — Ascending

Repository Characteristics

Exclude Forks | Has License



ahrm/sioyek

Commits: 1063 | Watchers: 44 | Stars: 8216 | Forks: 276

Total Issues: 1129 | Total Pull Reqs: 164 | Branches: 9 | Contributors: 37

Open Issues: 244 | Open Pull Reqs: 10 | Releases: 17 | Size: 38.08 KB

Created: 2021-07-11 | Updated: 2025-10-05 | Last Push: 2025-10-04 | Last Commit: 2025-07-17

Code Lines: 194,575 | Comment Lines: 89,350 | Blank Lines: 23,559

# Last Commit SHA: c3781f3e1c1bc71790a4fd57bf78ce0b7e34dce5

Default Branch: main

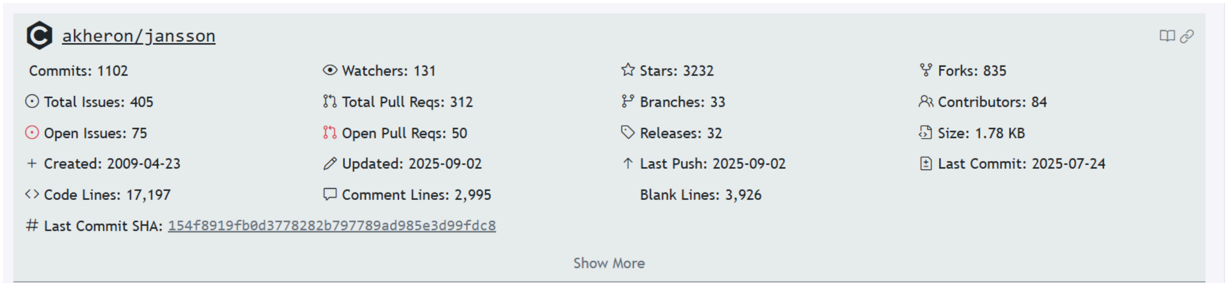
License: GNU General Public License v3.0

Issue Labels:
bug | documentation | duplicate | enhancement | good first issue | help wanted | invalid | question | wontfix

Repository topics:
pdf | pdf-viewer | research-paper

Languages:
C: 88.87% | C++: 9.74% | Python: 0.67% | Roff: 0.48% | TeX: 0.13% | Shell: 0.04% | QMake: 0.04% | Batchfile: 0.01% | PowerShell: 0.01% | Makefile: < 0.01%

Show Less



akheron/jansson

Commits: 1102 | Watchers: 131 | Stars: 3232 | Forks: 835

Total Issues: 405 | Total Pull Reqs: 312 | Branches: 33 | Contributors: 84

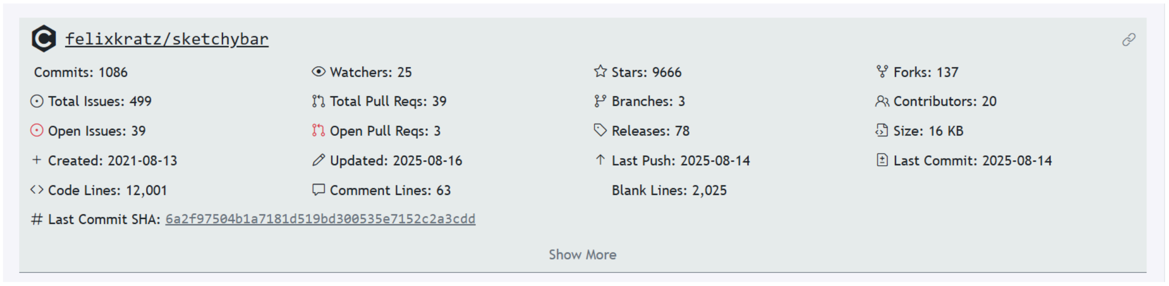
Open Issues: 75 | Open Pull Reqs: 50 | Releases: 32 | Size: 1.78 KB

Created: 2009-04-23 | Updated: 2025-09-02 | Last Push: 2025-09-02 | Last Commit: 2025-07-24

Code Lines: 17,197 | Comment Lines: 2,995 | Blank Lines: 3,926

# Last Commit SHA: 154f8919fb0d3778282b797789ad985e3d99fdc8

Show More



felixkratz/sketchybar

|  |  |  |  |
|---|---|---|---|
| Commits: 1086 | Watchers: 25 | Stars: 9666 | Forks: 137 |
| Total Issues: 499 | Total Pull Reqs: 39 | Branches: 3 | Contributors: 20 |
| Open Issues: 39 | Open Pull Reqs: 3 | Releases: 78 | Size: 16 KB |
| Created: 2021-08-13 | Updated: 2025-08-16 | Last Push: 2025-08-14 | Last Commit: 2025-08-14 |
| Code Lines: 12,001 | Comment Lines: 63 | Blank Lines: 2,025 |  |

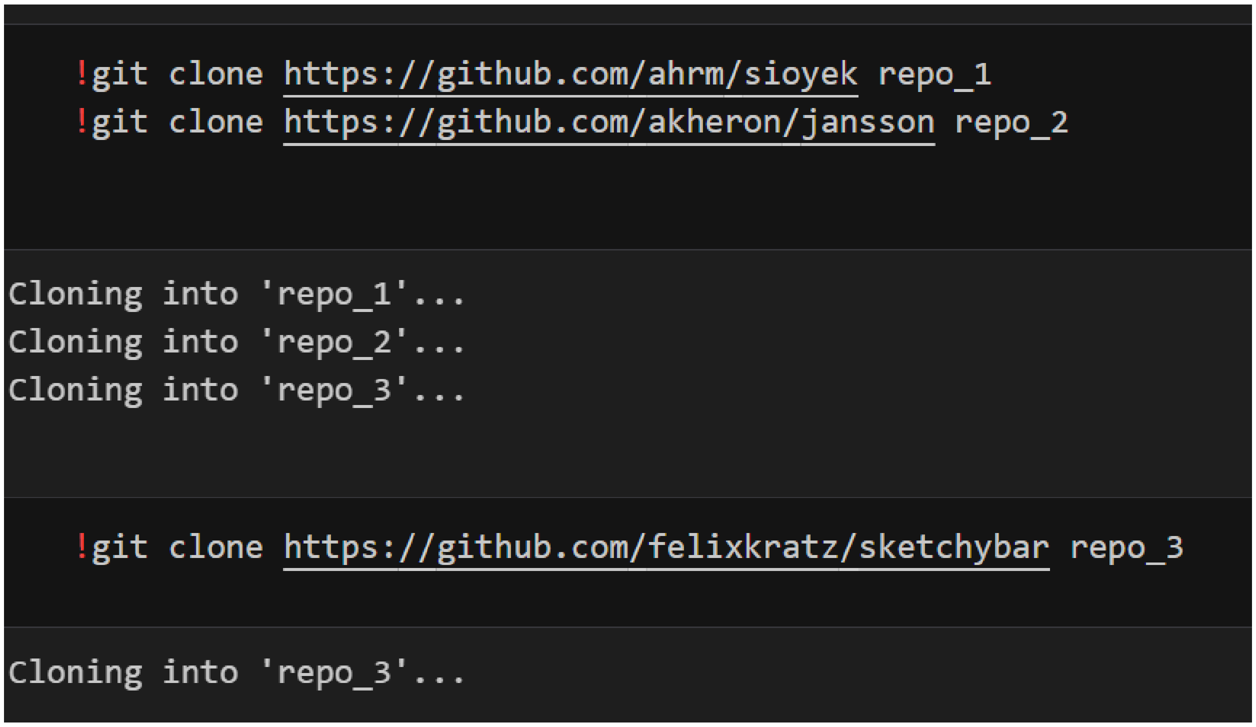
# Last Commit SHA: 6a2f97504b1a7181d519bd300535e7152c2a3cdd

Show More

Repo 1:-https://github.com/ahrm/sioyek

Repo 2:-https://github.com/akheron/jansson

Repo 3:-https://github.com/felixkratz/sketchybar



```
!git clone https://github.com/ahrm/sioyek repo_1
!git clone https://github.com/akheron/jansson repo_2
```

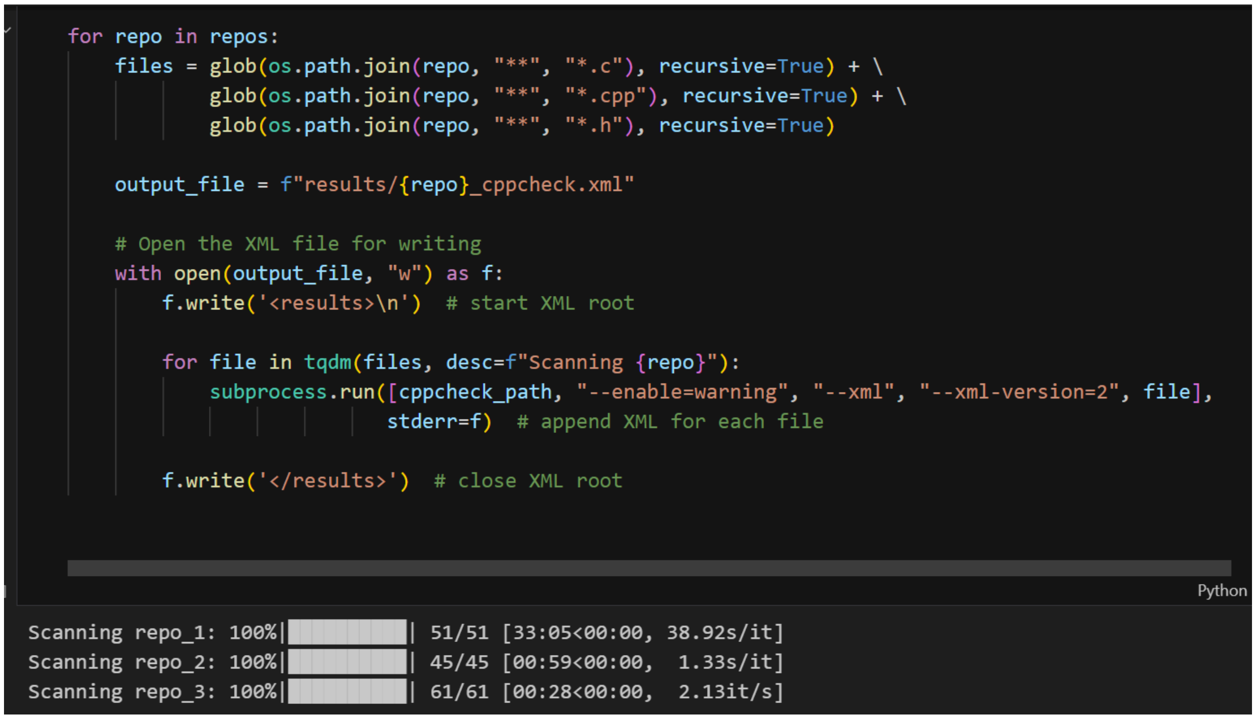
```
Cloning into 'repo_1'...
Cloning into 'repo_2'...
Cloning into 'repo_3'...
```

```
!git clone https://github.com/felixkratz/sketchybar repo_3
```

```
Cloning into 'repo_3'...
```

2) Now I have selected 3 tools for vulnerability detection.

1. Cppcheck:- A general-purpose static analyzer designed to detect bug-prone code patterns in C and C++. Detects logical errors, null pointer dereferences, memory leaks, and uninitialized variables. Generates detailed XML reports that make automated analysis easy.

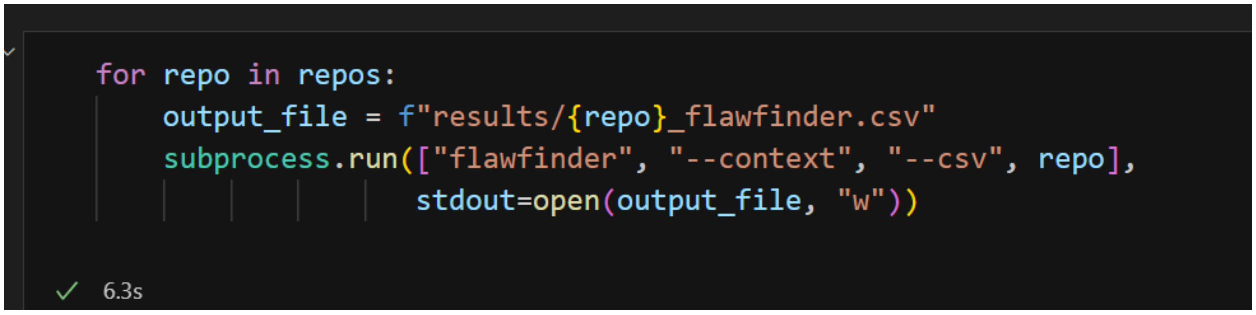


```python
for repo in repos:
    files = glob(os.path.join(repo, "**", "*.c"), recursive=True) + \
            glob(os.path.join(repo, "**", "*.cpp"), recursive=True) + \
            glob(os.path.join(repo, "**", "*.h"), recursive=True)

    output_file = f"results/{repo}_cppcheck.xml"

    # Open the XML file for writing
    with open(output_file, "w") as f:
        f.write('<results>\n')  # start XML root

        for file in tqdm(files, desc=f"Scanning {repo}"):
            subprocess.run([cppcheck_path, "--enable=warning", "--xml", "--xml-version=2", file],
                           stderr=f)  # append XML for each file

        f.write('</results>')  # close XML root
```

Python

```
Scanning repo_1: 100%|██████████| 51/51 [33:05<00:00, 38.92s/it]
Scanning repo_2: 100%|██████████| 45/45 [00:59<00:00,  1.33s/it]
Scanning repo_3: 100%|██████████| 61/61 [00:28<00:00,  2.13it/s]
```

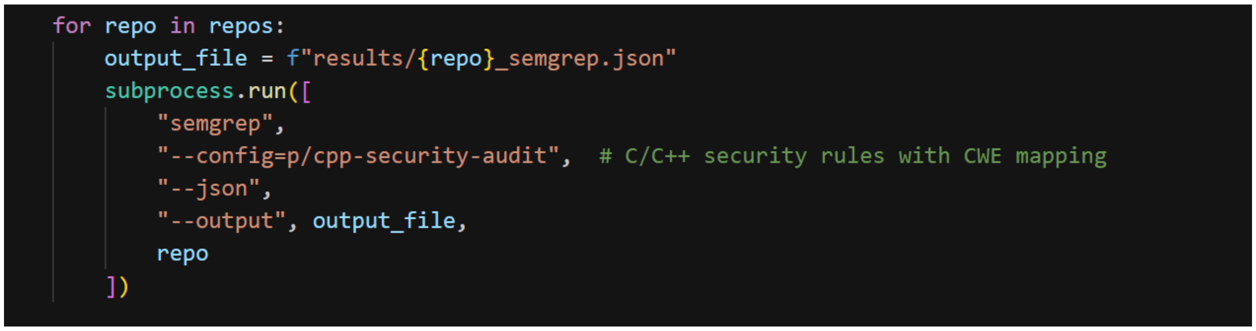
2. Flawfinder:- Security-oriented static analysis tool specialized for identifying dangerous C/C++ functions. Maps risky functions (e.g., gets, strcpy, sprintf) to CWE IDs. Focuses on memory safety and input handling vulnerabilities, and created CSV for that.



```python
for repo in repos:
    output_file = f"results/{repo}_flawfinder.csv"
    subprocess.run(["flawfinder", "--context", "--csv", repo],
                   stdout=open(output_file, "w"))
```

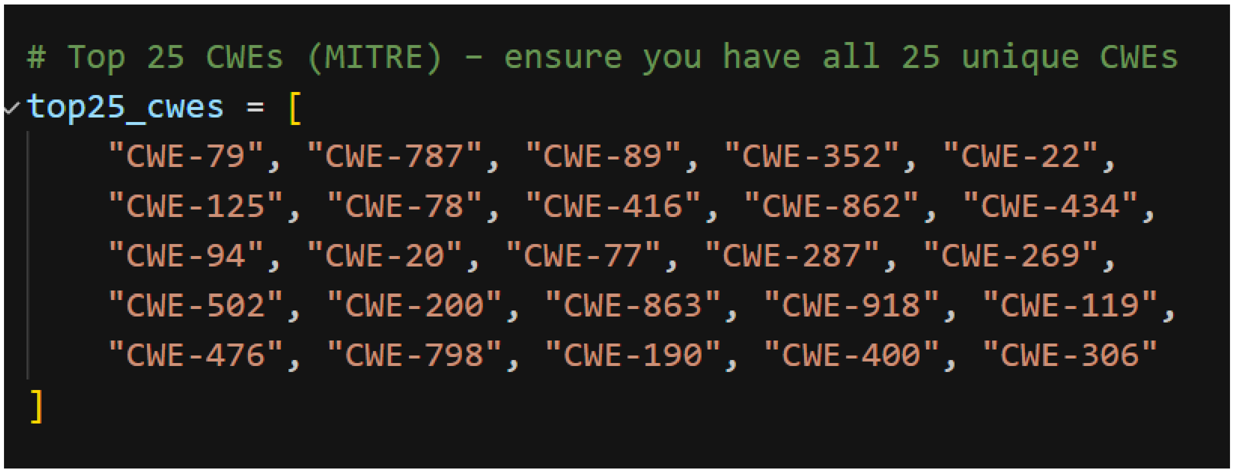
✓ 6.3s

3. Sempreg:- A rule-based static analysis tool that scans code using customizable security and correctness patterns. Supports user-defined or community-driven rules for flexible vulnerability detection. It can detect code injection, insecure API usage, and misconfigurations, and it creates a JSON format file.

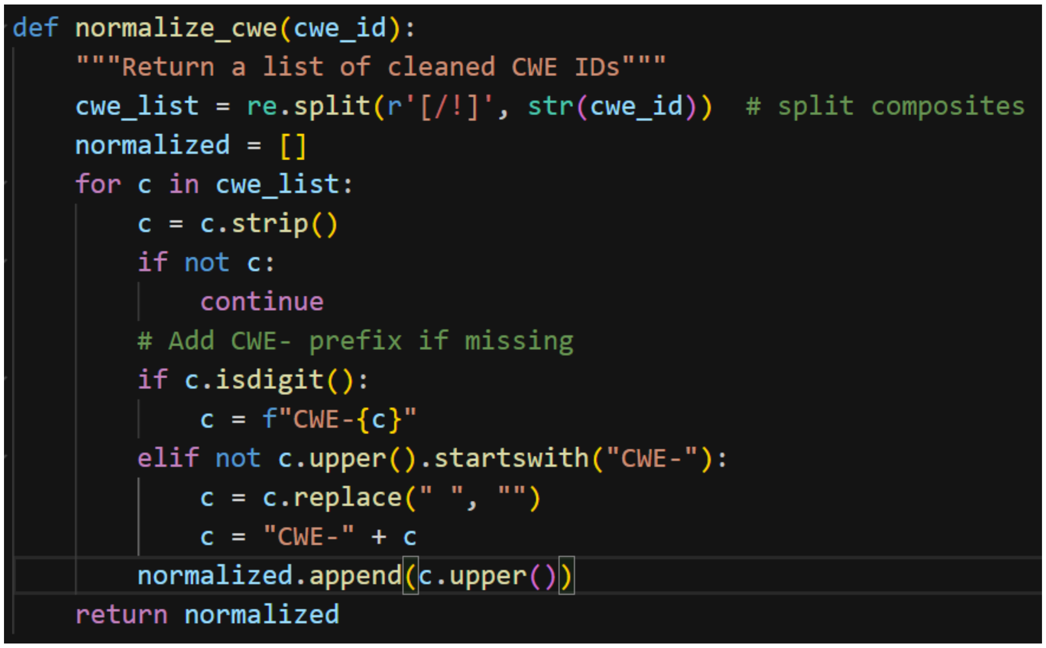

```python
for repo in repos:
    output_file = f"results/{repo}_semgrep.json"
    subprocess.run([
        "semgrep",
        "--config=p/cpp-security-audit",  # C/C++ security rules with CWE mapping
        "--json",
        "--output", output_file,
        repo
    ])
```

Each tool was executed independently on all three repositories. And the result is stored in the results folder. Each tool output contained CWE identifiers for vulnerabilities found in the source code.

3) Use the Top 25 CWEs


```python
# Top 25 CWEs (MITRE) - ensure you have all 25 unique CWEs
top25_cwes = [
    "CWE-79", "CWE-787", "CWE-89", "CWE-352", "CWE-22",
    "CWE-125", "CWE-78", "CWE-416", "CWE-862", "CWE-434",
    "CWE-94", "CWE-20", "CWE-77", "CWE-287", "CWE-269",
    "CWE-502", "CWE-200", "CWE-863", "CWE-918", "CWE-119",
    "CWE-476", "CWE-798", "CWE-190", "CWE-400", "CWE-306"
]
```

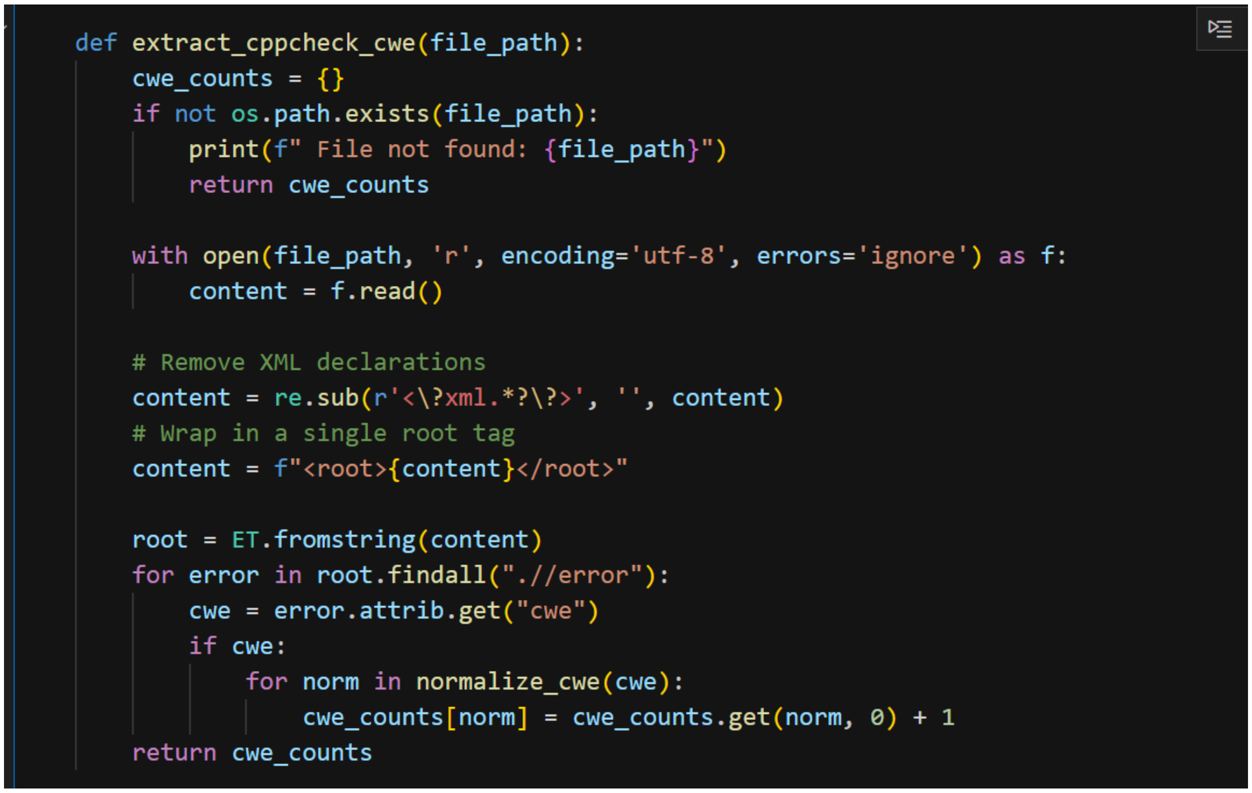
4) Before analyzing the outputs from different tools, it was essential to ensure that all CWE identifiers followed a consistent format. Different tools often report CWEs inconsistently — for example: "79", "CWE 79", "cwe-79", "CWE-079", or "CWE/79" may all refer to the same vulnerability category.
To handle such inconsistencies, a custom helper function normalize_cwe() was implemented.



```python
def normalize_cwe(cwe_id):
    """Return a list of cleaned CWE IDs"""
    cwe_list = re.split(r'[/!]', str(cwe_id))  # split composites
    normalized = []
    for c in cwe_list:
        c = c.strip()
        if not c:
            continue
        # Add CWE- prefix if missing
        if c.isdigit():
            c = f"CWE-{c}"
        elif not c.upper().startswith("CWE-"):
            c = c.replace(" ", "")
            c = "CWE-" + c
        normalized.append(c.upper())
    return normalized
```

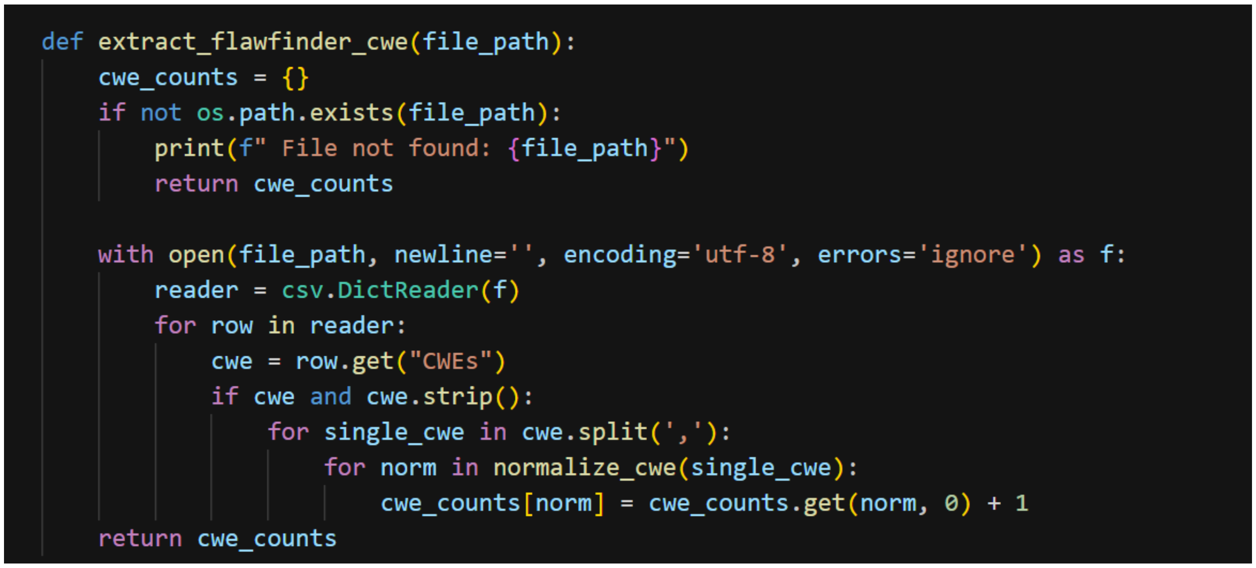
5) The script parsed each file type using dedicated extraction functions:

- extract_cppcheck_cwe() — parses XML to read <error cwe="...">



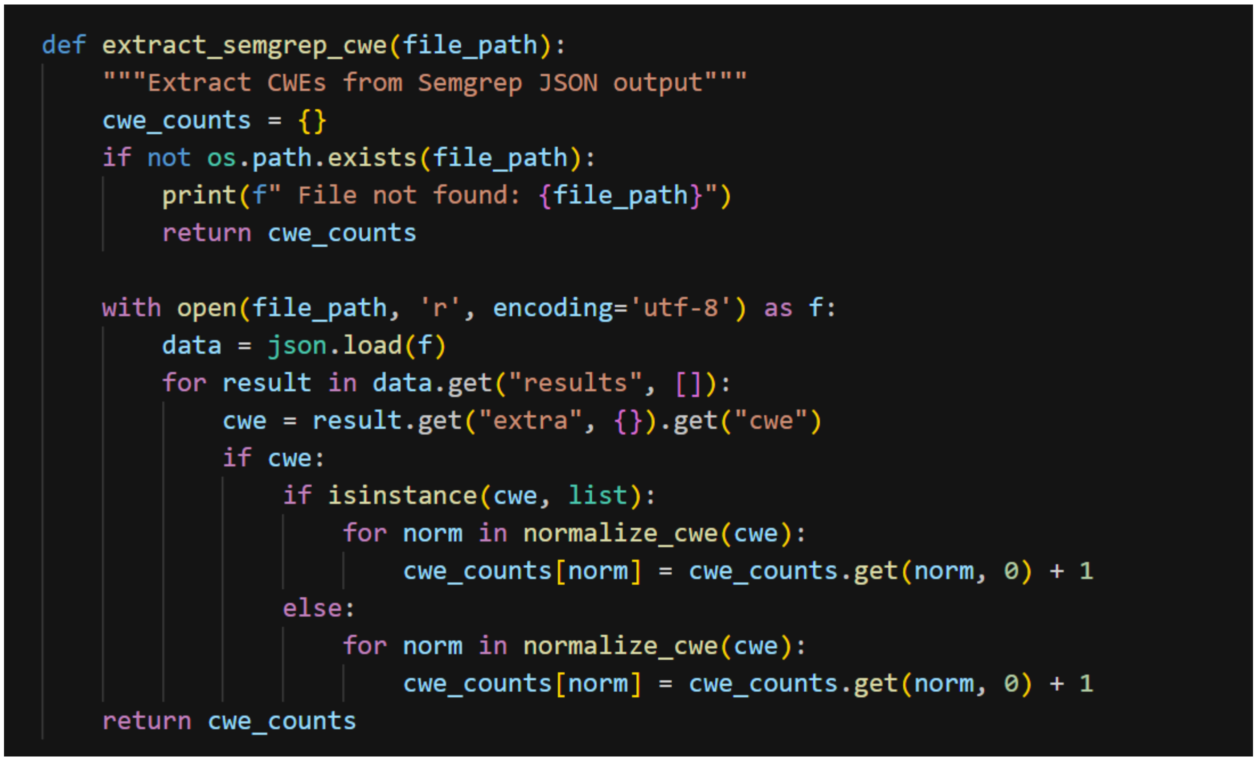
```python
def extract_cppcheck_cwe(file_path):
    cwe_counts = {}
    if not os.path.exists(file_path):
        print(f" File not found: {file_path}")
        return cwe_counts

    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        content = f.read()

    # Remove XML declarations
    content = re.sub(r'<\?xml.*?\?>', '', content)
    # Wrap in a single root tag
    content = f"<root>{content}</root>"

    root = ET.fromstring(content)
    for error in root.findall(".//error"):
        cwe = error.attrib.get("cwe")
        if cwe:
            for norm in normalize_cwe(cwe):
                cwe_counts[norm] = cwe_counts.get(norm, 0) + 1
    return cwe_counts
```

- extract_flawfinder_cwe() — parses CSV and reads CWE column.



```python
def extract_flawfinder_cwe(file_path):
    cwe_counts = {}
    if not os.path.exists(file_path):
        print(f" File not found: {file_path}")
        return cwe_counts

    with open(file_path, newline='', encoding='utf-8', errors='ignore') as f:
        reader = csv.DictReader(f)
        for row in reader:
            cwe = row.get("CWEs")
            if cwe and cwe.strip():
                for single_cwe in cwe.split(','):
                    for norm in normalize_cwe(single_cwe):
                        cwe_counts[norm] = cwe_counts.get(norm, 0) + 1
    return cwe_counts
```

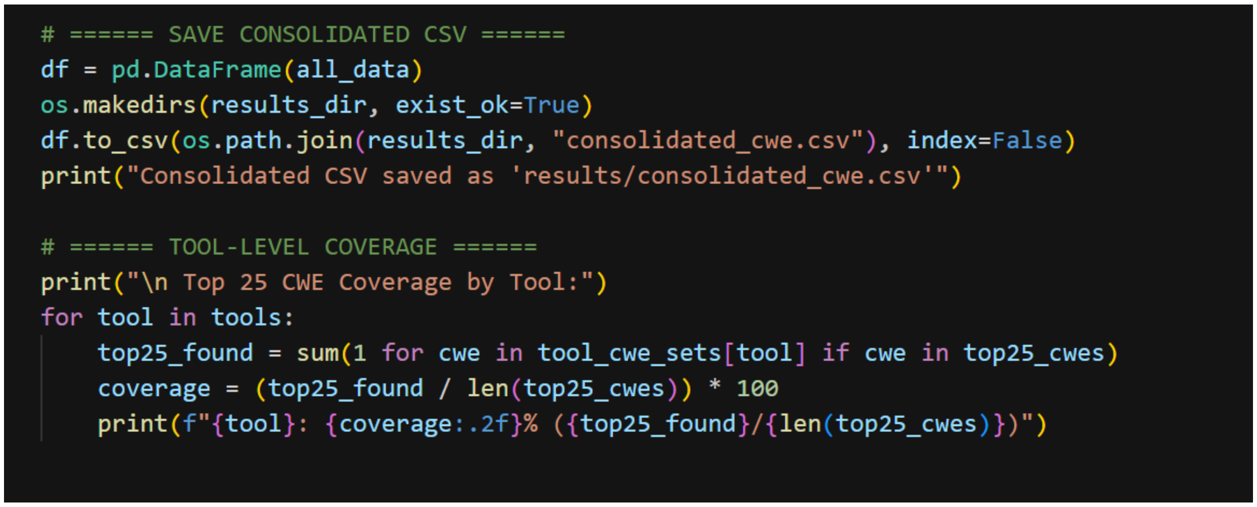
- extract_semgrep_cwe() — reads JSON and extracts extra.CWE field.



```python
def extract_semgrep_cwe(file_path):
    """Extract CWEs from Semgrep JSON output"""
    cwe_counts = {}
    if not os.path.exists(file_path):
        print(f" File not found: {file_path}")
        return cwe_counts

    with open(file_path, 'r', encoding='utf-8') as f:
        data = json.load(f)
        for result in data.get("results", []):
            cwe = result.get("extra", {}).get("cwe")
            if cwe:
                if isinstance(cwe, list):
                    for norm in normalize_cwe(cwe):
                        cwe_counts[norm] = cwe_counts.get(norm, 0) + 1
                else:
                    for norm in normalize_cwe(cwe):
                        cwe_counts[norm] = cwe_counts.get(norm, 0) + 1
    return cwe_counts
```

6) Store all this information in a consolidated CSV and also indicate whether the indicated vulnerability is one of the “top 25 CWE categories”.

Link to file



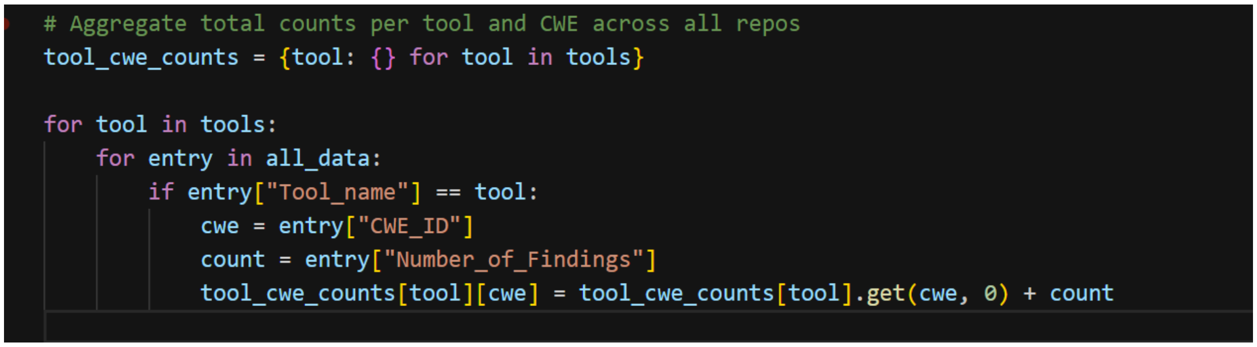
```
# ====== SAVE CONSOLIDATED CSV ======
df = pd.DataFrame(all_data)
os.makedirs(results_dir, exist_ok=True)
df.to_csv(os.path.join(results_dir, "consolidated_cwe.csv"), index=False)
print("Consolidated CSV saved as 'results/consolidated_cwe.csv'")

# ====== TOOL-LEVEL COVERAGE ======
print("\n Top 25 CWE Coverage by Tool:")
for tool in tools:
    top25_found = sum(1 for cwe in tool_cwe_sets[tool] if cwe in top25_cwes)
    coverage = (top25_found / len(top25_cwes)) * 100
    print(f"{tool}: {coverage:.2f}% ({top25_found}/{len(top25_cwes)})")
```

7) Tool-level CWE Coverage Analysis Detailed Breakdown

After collecting and normalizing all CWE identifiers, a detailed breakdown was generated to show how many instances of each CWE were detected by each tool.

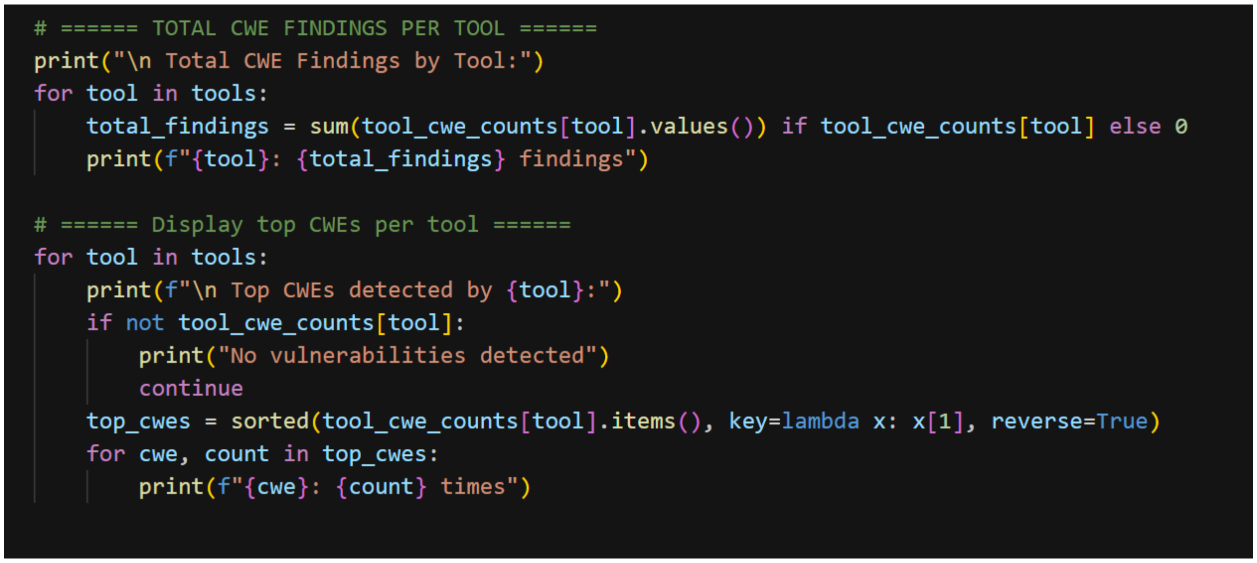
The script aggregated all findings across the three repositories and computed total counts per (Tool, CWE) pair.

Sums up total occurrences



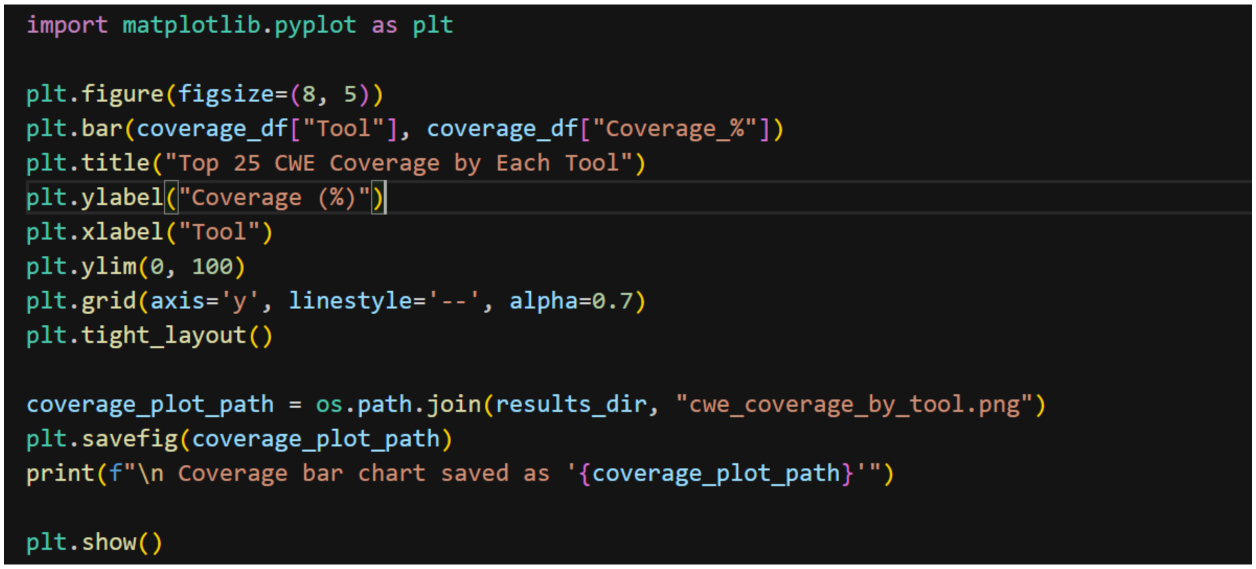
```
# Aggregate total counts per tool and CWE across all repos
tool_cwe_counts = {tool: {} for tool in tools}

for tool in tools:
    for entry in all_data:
        if entry["Tool_name"] == tool:
            cwe = entry["CWE_ID"]
            count = entry["Number_of_Findings"]
            tool_cwe_counts[tool][cwe] = tool_cwe_counts[tool].get(cwe, 0) + count
```

The results were saved in a CSV file named tool_cwe_breakdown.csv



```python
# ====== TOTAL CWE FINDINGS PER TOOL ======
print("\n Total CWE Findings by Tool:")
for tool in tools:
    total_findings = sum(tool_cwe_counts[tool].values()) if tool_cwe_counts[tool] else 0
    print(f"{tool}: {total_findings} findings")

# ====== Display top CWEs per tool ======
for tool in tools:
    print(f"\n Top CWEs detected by {tool}:")
    if not tool_cwe_counts[tool]:
        print("No vulnerabilities detected")
        continue
    top_cwes = sorted(tool_cwe_counts[tool].items(), key=lambda x: x[1], reverse=True)
    for cwe, count in top_cwes:
        print(f"{cwe}: {count} times")
```

Visualize coverage with appropriate graphs and plots as necessary.



```python
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
plt.bar(coverage_df["Tool"], coverage_df["Coverage_%"])
plt.title("Top 25 CWE Coverage by Each Tool")
plt.ylabel("Coverage (%)")
plt.xlabel("Tool")
plt.ylim(0, 100)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

coverage_plot_path = os.path.join(results_dir, "cwe_coverage_by_tool.png")
plt.savefig(coverage_plot_path)
print(f"\n Coverage bar chart saved as '{coverage_plot_path}'")

plt.show()
```

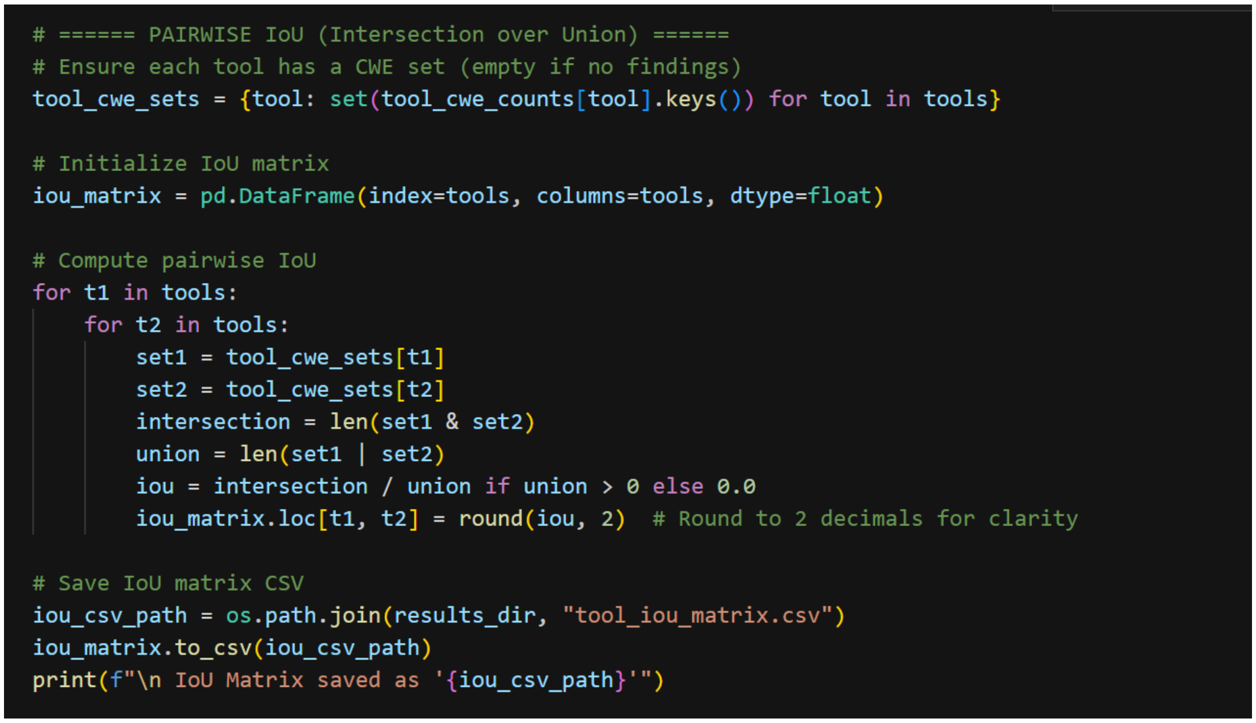
8)Pairwise Agreement (IoU) Analysis

This step was to measure the overlap or agreement between tools based on the CWE identifiers they reported. Different static analysis tools often focus on different types of vulnerabilities; therefore, comparing their outputs helps us understand whether they complement each other or detect similar patterns.

This is being computed by the tool pair using the formula for Jaccard Index (IoU):

IoU (T1, T2) = |{CWE IDs found by both T1 and T2}|/ |{CWE IDs found by T1 or T2}|

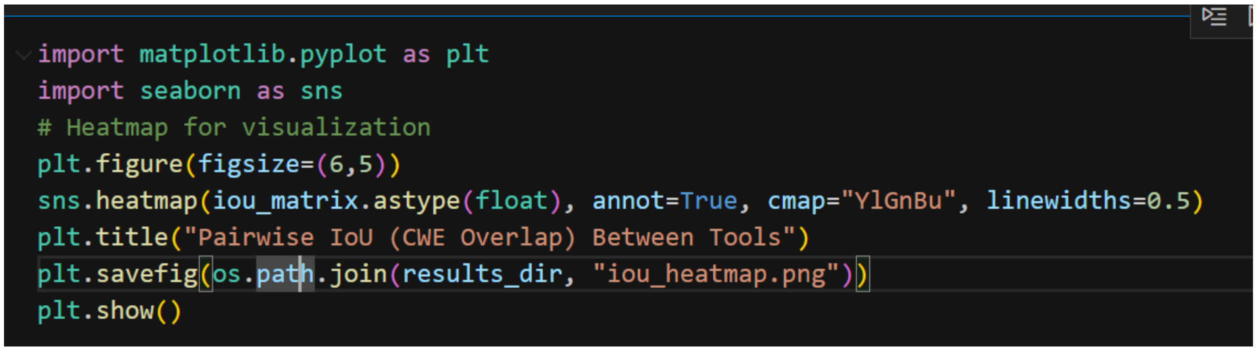
A higher IoU value indicates that the tools found more similar vulnerabilities, while a lower IoU means they detected different types of weaknesses



```python
# ====== PAIRWISE IoU (Intersection over Union) ======
# Ensure each tool has a CWE set (empty if no findings)
tool_cwe_sets = {tool: set(tool_cwe_counts[tool].keys()) for tool in tools}

# Initialize IoU matrix
iou_matrix = pd.DataFrame(index=tools, columns=tools, dtype=float)

# Compute pairwise IoU
for t1 in tools:
    for t2 in tools:
        set1 = tool_cwe_sets[t1]
        set2 = tool_cwe_sets[t2]
        intersection = len(set1 & set2)
        union = len(set1 | set2)
        iou = intersection / union if union > 0 else 0.0
        iou_matrix.loc[t1, t2] = round(iou, 2)  # Round to 2 decimals for clarity

# Save IoU matrix CSV
iou_csv_path = os.path.join(results_dir, "tool_iou_matrix.csv")
iou_matrix.to_csv(iou_csv_path)
print(f"\n IoU Matrix saved as '{iou_csv_path}'")
```

And saved it in [tool_iou_matrix.csv]

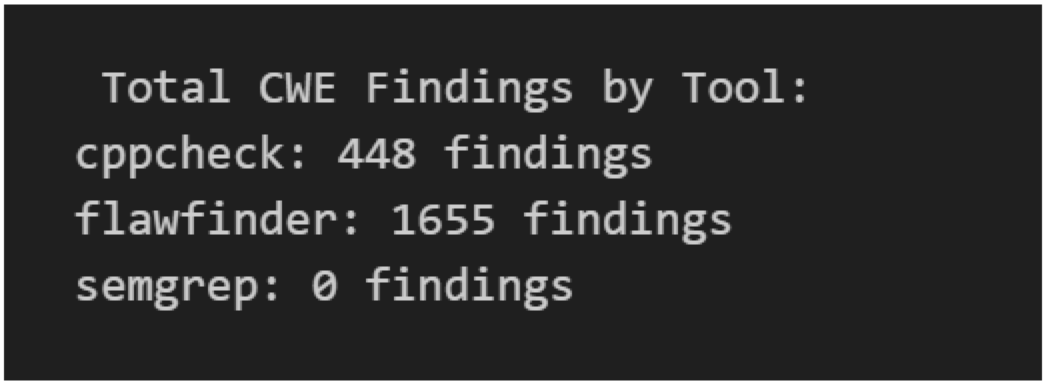
And for visualization, we have created the heatmap of this matrix.



```python
import matplotlib.pyplot as plt
import seaborn as sns
# Heatmap for visualization
plt.figure(figsize=(6,5))
sns.heatmap(iou_matrix.astype(float), annot=True, cmap="YlGnBu", linewidths=0.5)
plt.title("Pairwise IoU (CWE Overlap) Between Tools")
plt.savefig(os.path.join(results_dir, "iou_heatmap.png"))
plt.show()
```

## Results and Analysis

1) Link to the XML file created by the CPPcheck tool repo1,repo2,repo3
2) Link to the CSV file created by flawfinder tool repo1,repo2,repo3
3) Linl of JSON file created by sempreg tool repo1,repo2,repo3
4) Total number of finding by each tools



```
 Total CWE Findings by Tool:
cppcheck: 448 findings
flawfinder: 1655 findings
semgrep: 0 findings
```

This shows that

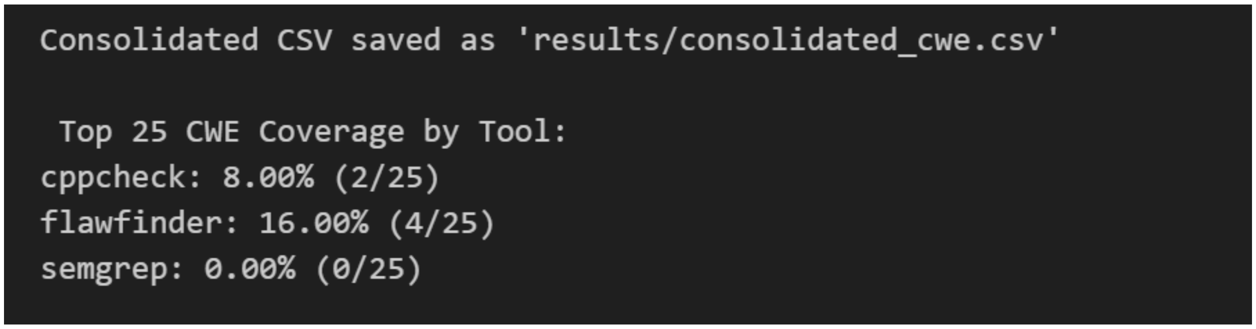
a) Flawfinder gives the maximum number of findings as it is a security-oriented static analysis tool that scans for potentially risky C/C++ functions that may cause buffer overflows, format string vulnerabilities, and memory corruption.
    i) It is pattern-based and flags every possible instance of unsafe usage, even if the vulnerability might be benign or mitigated elsewhere.
    ii) Because of this conservative approach, it often over-reports
    iii) Conclusion: Flawfinder has very high recall but low precision
b) Cppcheck focuses primarily on code correctness, logic errors, and memory safety rather than purely security-specific patterns.
    i) It performs deep static code analysis, examining data flow and control flow to find actual errors rather than potential ones.
    ii) It's less noisy than Flawfinder and typically avoids flagging false positives.
    iii) Cppcheck maps only certain categories of issues to CWEs
    iv) Conclusion: Cppcheck provides a balanced number of findings but generally more accurate and actionable.
c) Semgrep gives 0 findings
    i) Semgrep is a static analysis tool that detects security issues based on custom rule patterns written over the Abstract Syntax Tree (AST) of source code.

ii) Semgrep uses AST-based semantic matching, not just keyword search. If the code doesn't structurally match the vulnerability patterns in the rules, Semgrep will not flag anything, even if function names look suspicious. This makes it precise, but also very strict; the code must exactly match the defined rule pattern. Even if a function name looks risky (like strcpy), it won't be flagged unless it matches a rule's specific condition.

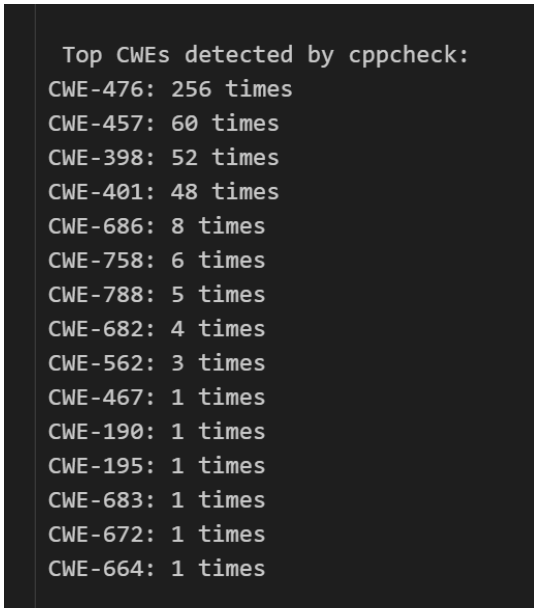
iii) Conclusion it likely produced 0 CWEs because the chosen ruleset lacked CWE mappings, unlike Flawfinder and Cppcheck, which have built-in CWE-based detections.

The observed differences in CWE counts are tool-specific characteristics rather than errors in the analysis workflow. Flawfinder and Cppcheck provide broad CWE coverage automatically, while Semgrep requires targeted rule configuration.

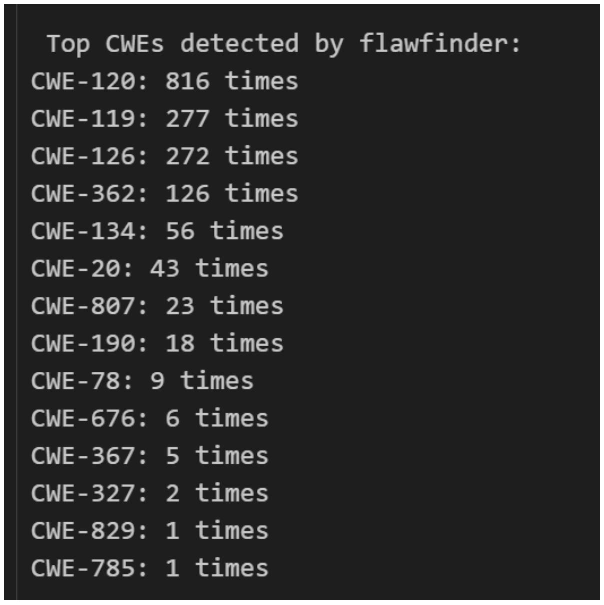
5) Coverage in Top 25 CWE by each tool



```
Consolidated CSV saved as 'results/consolidated_cwe.csv'

 Top 25 CWE Coverage by Tool:
cppcheck: 8.00% (2/25)
flawfinder: 16.00% (4/25)
semgrep: 0.00% (0/25)
```

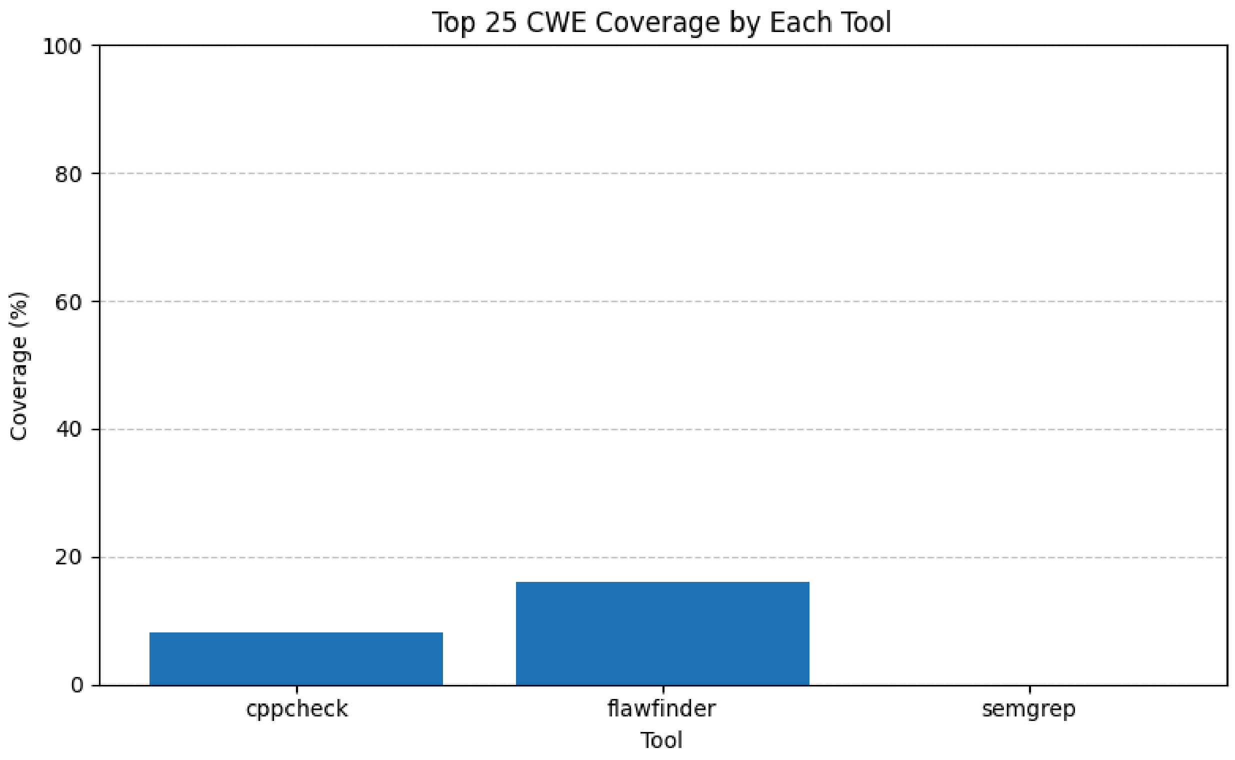


```
 Top CWEs detected by cppcheck:
CWE-476: 256 times
CWE-457: 60 times
CWE-398: 52 times
CWE-401: 48 times
CWE-686: 8 times
CWE-758: 6 times
CWE-788: 5 times
CWE-682: 4 times
CWE-562: 3 times
CWE-467: 1 times
CWE-190: 1 times
CWE-195: 1 times
CWE-683: 1 times
CWE-672: 1 times
CWE-664: 1 times
```

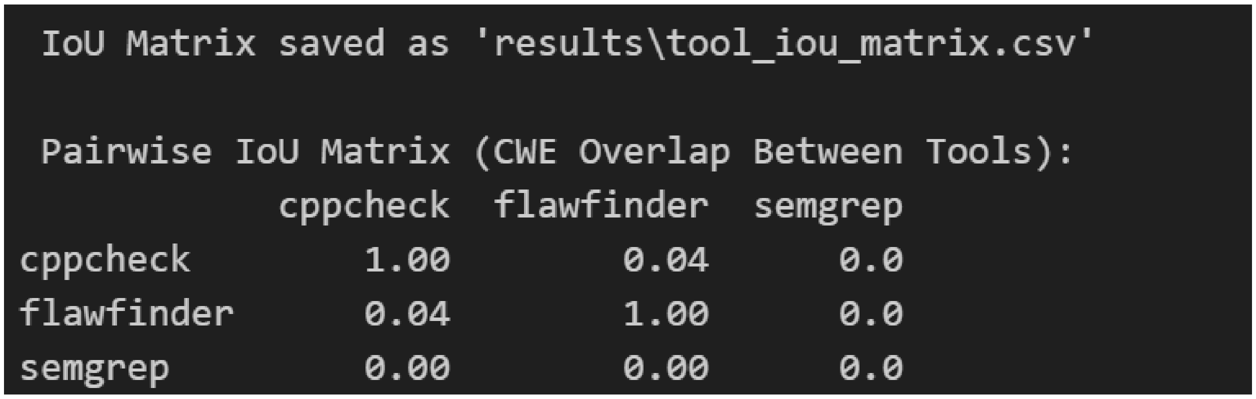


```
 Top CWEs detected by flawfinder:
CWE-120: 816 times
CWE-119: 277 times
CWE-126: 272 times
CWE-362: 126 times
CWE-134: 56 times
CWE-20: 43 times
CWE-807: 23 times
CWE-190: 18 times
CWE-78: 9 times
CWE-676: 6 times
CWE-367: 5 times
CWE-327: 2 times
CWE-829: 1 times
CWE-785: 1 times
```

6) Graph of Coverage of each tool in Top 25

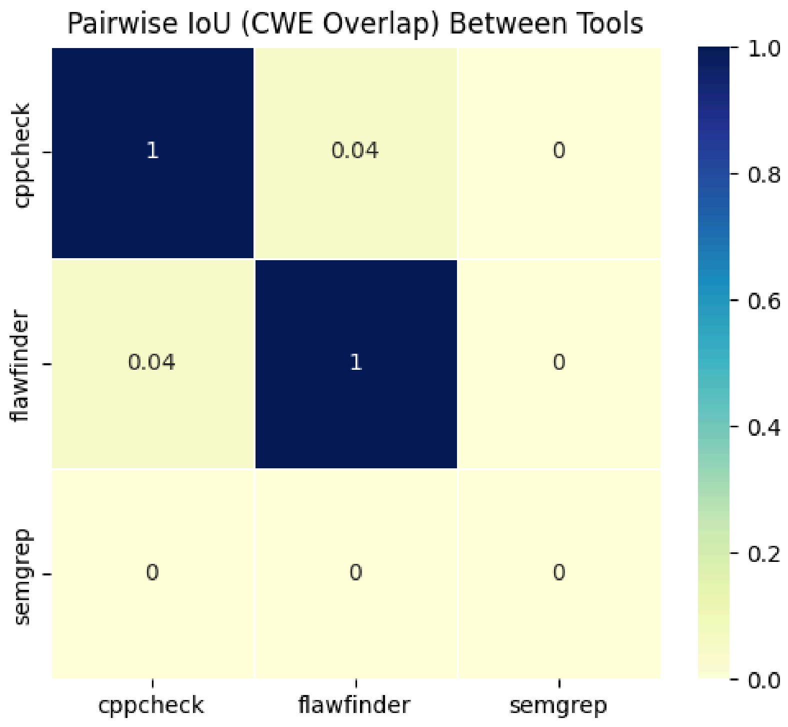


7) Pairwise IoU Matrix



```
 IoU Matrix saved as 'results\tool_iou_matrix.csv'

 Pairwise IoU Matrix (CWE Overlap Between Tools):
            cppcheck  flawfinder  semgrep
cppcheck        1.00        0.04      0.0
flawfinder      0.04        1.00      0.0
semgrep         0.00        0.00      0.0
```

8) Heatmap of IoU



## Discussion and Conclusion

### Challenges Faced

- Running multiple vulnerability analysis tools (Cppcheck, Flawfinder, Semgrep) and consolidating their outputs was challenging due to different output formats (XML, CSV, JSON). Parsing them correctly required dedicated extraction functions for each tool.
- Ensuring consistent CWE identifiers across tools was non-trivial, as tools often report the same CWE in multiple formats (e.g., 79, CWE-79, cwe-79). This required careful normalization to avoid duplicate or missed entries.
- Semgrep’s rule-based scanning produced no findings with default rules, which highlighted the importance of understanding each tool’s coverage scope and limitations.

- Due to which I have tried many other tools like CodeQL, rats, et,c and even tried to change repo language to Python, where I was also able to find only 2 tools, Bandit and sempreg, so I reverted back to C Repos.
- Aggregating results across multiple repositories and computing pairwise IoU and Top 25 CWE coverage required careful bookkeeping to avoid errors in counts or duplicate entries.

**Reflections and Insights**

- The lab provided practical exposure to static vulnerability analysis across multiple tools, demonstrating how each tool has strengths and limitations.
- Automating CWE extraction and normalization ensured reproducible and accurate data, allowing meaningful comparison between tools.
- Visualizations such as the pairwise IoU heatmap helped in understanding overlap and diversity in vulnerability detection across tools.
- Working with multiple repositories and tools highlighted the importance of standardized reporting formats for large-scale software security assessment.

**Lessons Learned**

- Each vulnerability analysis tool has a different detection focus, so combining multiple tools provides broader coverage.
- Normalizing and consolidating outputs is essential for accurate comparative analysis and reporting.
- Automation in data extraction, aggregation, and visualization reduces human error and improves the reliability of results.

**Conclusion**

In conclusion, this lab demonstrated an end-to-end workflow for CWE-based vulnerability analysis across multiple C/C++ repositories. The process—from running tools independently, normalizing outputs, consolidating findings, computing Top 25 CWE coverage, and visualizing pairwise IoU—highlighted both technical challenges and practical solutions in static analysis. The methodology ensured accurate, reproducible results while providing insights into tool coverage, overlaps, and detection limitations. Overall, lab reinforced the value of systematic automation and careful data handling in software security assessment.

# Lab 7

## Introduction

This Lab was conducted on 15 September 2025.

The objective of this lab was to implement program analysis techniques on C programs, including constructing Control Flow Graphs (CFGs) and performing Reaching Definitions Analysis. The lab emphasized structured program analysis beyond toy examples, allowing students to understand how software tools compute program properties behind the scenes.

## Tools

- Python 3.10+ – for implementing CFG construction and dataflow analysis.
- Graphviz – for visualizing CFGs.
- Matplotlib – for plotting graphs and metrics.
- Text Editor/IDE – VS Code or PyCharm for coding and debugging.

## Set-Up

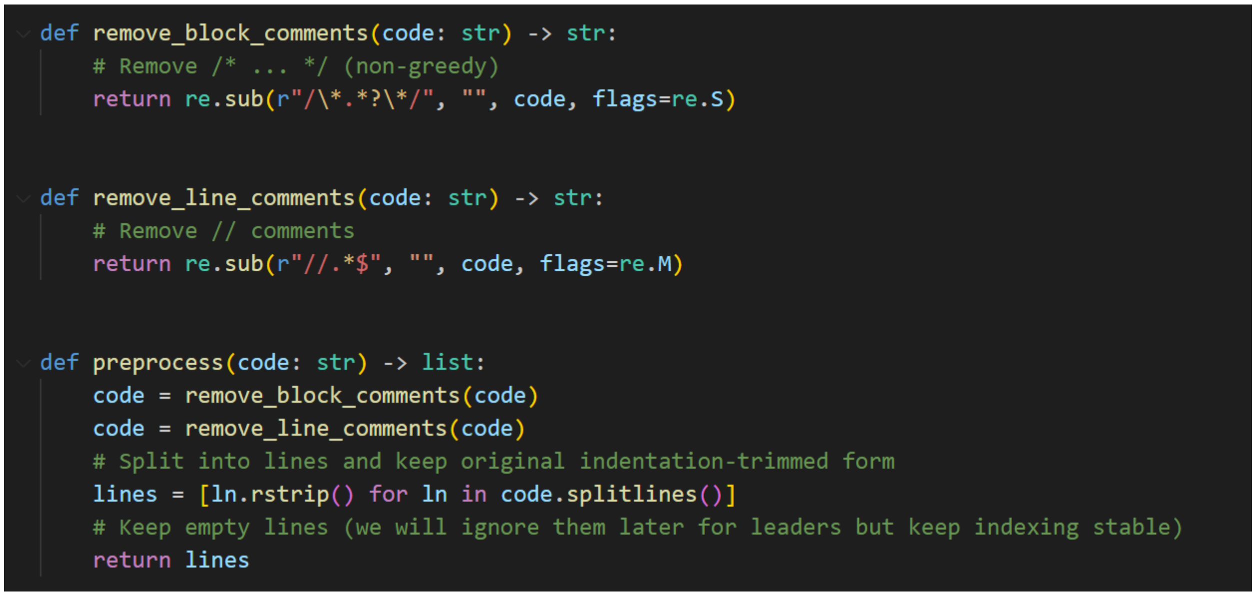
- Installed required Python packages:
  ```
  pip install graphviz matplotlib
  ```
- Selected three C programs (200–300 LOC each) containing:
  - Conditionals (`if/else`)
  - Loops (`for`, `while`)
  - Multiple variable assignments and reassignments
  - Prog1,prog2,prog3

## Methodology and Execution

1) Preprocessing the Input C Code

The first step was to clean the input .c file by removing both single-line (//) and multi-line (/* ... */) comments, and then splitting the program into individual lines while preserving their indices.

This was done to make line-based CFG mapping consistent later.

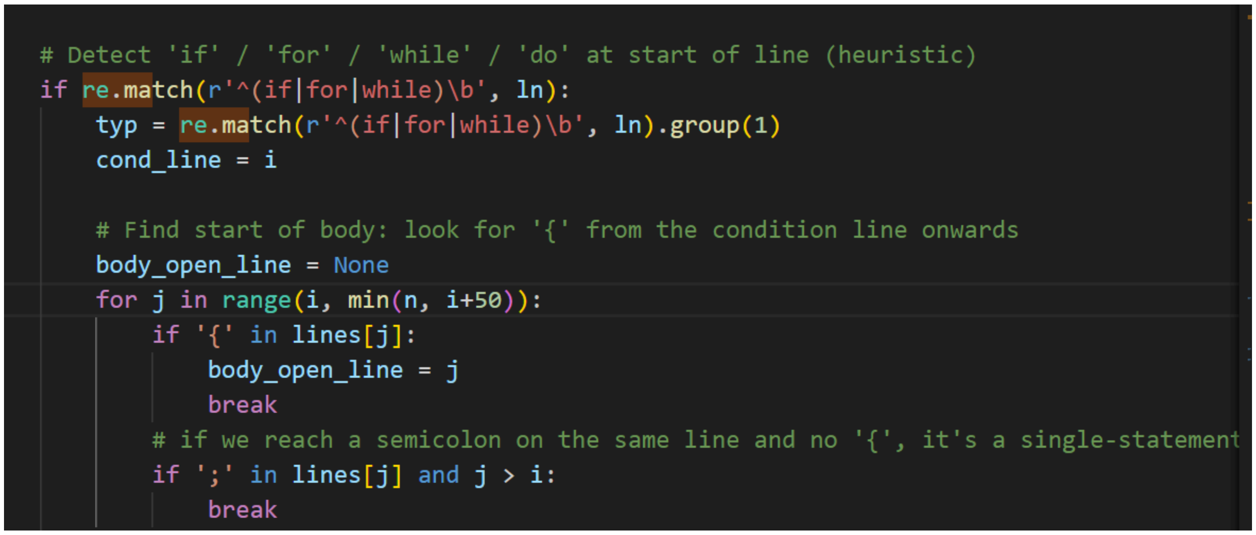


```python
def remove_block_comments(code: str) -> str:
    # Remove /* ... */ (non-greedy)
    return re.sub(r"/\*.*?\*/", "", code, flags=re.S)


def remove_line_comments(code: str) -> str:
    # Remove // comments
    return re.sub(r"//.*$", "", code, flags=re.M)


def preprocess(code: str) -> list:
    code = remove_block_comments(code)
    code = remove_line_comments(code)
    # Split into lines and keep original indentation-trimmed form
    lines = [ln.rstrip() for ln in code.splitlines()]
    # Keep empty lines (we will ignore them later for leaders but keep indexing stable)
    return lines
```

2) Leader Identification

- The first instruction of the program is a leader.
- Any instruction that is the target of a branch/jump/loop is a leader.
- Any instruction that comes immediately after a branch/jump/loop is also a leader.

The following logic was used to detect if, else, for, while, and do-while constructs using regular expressions.



```python
# Detect 'if' / 'for' / 'while' / 'do' at start of line (heuristic)
if re.match(r'^(if|for|while)\b', ln):
    typ = re.match(r'^(if|for|while)\b', ln).group(1)
    cond_line = i

    # Find start of body: look for '{' from the condition line onwards
    body_open_line = None
    for j in range(i, min(n, i+50)):
        if '{' in lines[j]:
            body_open_line = j
            break
        # if we reach a semicolon on the same line and no '{', it's a single-statement
        if ';' in lines[j] and j > i:
            break
```

3) Basic Block Construction

After finding all leaders, the next step was to group consecutive statements between leaders into basic blocks.

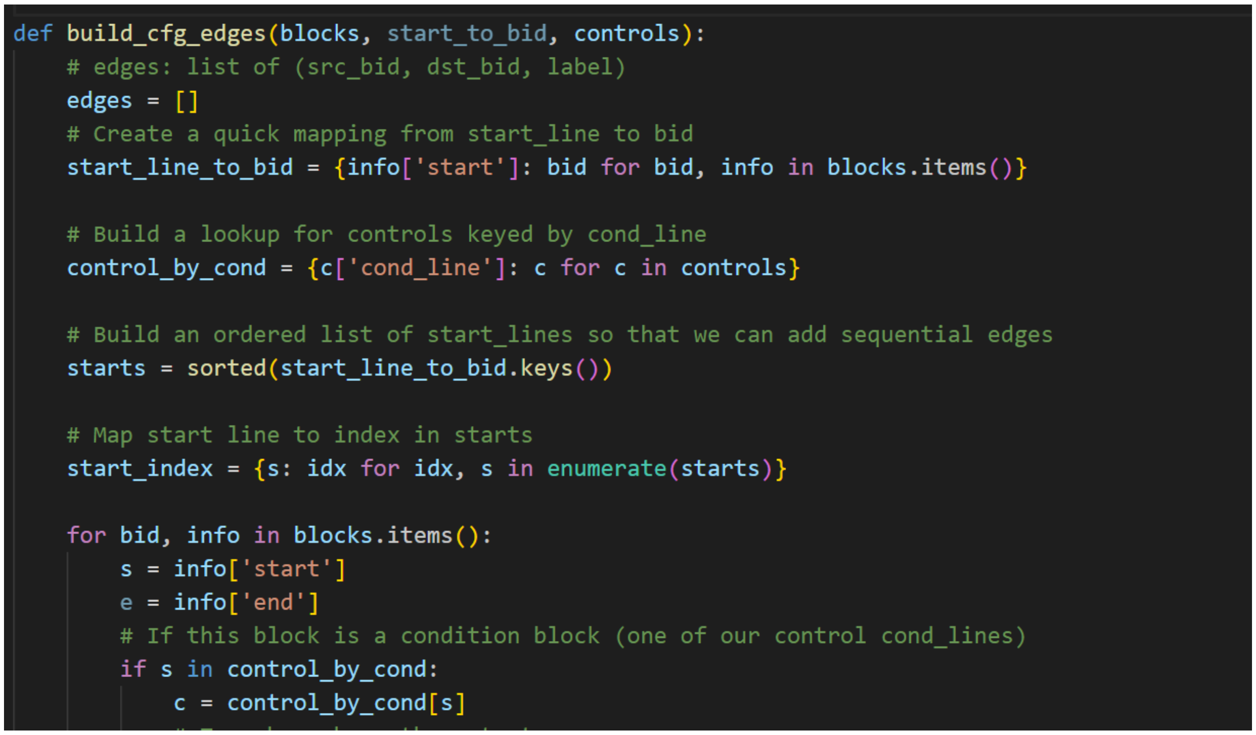
This produced a clean mapping of statements to their corresponding blocks, making it easy to visualize later in the CFG.

4) Building the CFG Edges

Once blocks were identified, edges were created to represent control flow:

- **Sequential edge (seq)** – Normal flow from one block to the next.
- **True/False edges** – From condition blocks to their "then" or "else" blocks.
- **Back edges** – From loop bodies to their loop conditions.
- **Exit edges** – From loops or condition blocks to their join points.

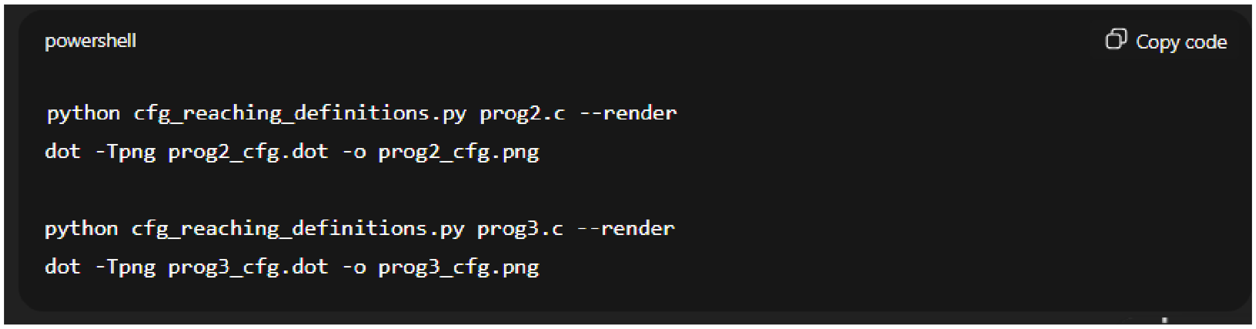
This was done inside the `build_cfg_edges()` function:



```python
def build_cfg_edges(blocks, start_to_bid, controls):
    # edges: list of (src_bid, dst_bid, label)
    edges = []
    # Create a quick mapping from start_line to bid
    start_line_to_bid = {info['start']: bid for bid, info in blocks.items()}

    # Build a lookup for controls keyed by cond_line
    control_by_cond = {c['cond_line']: c for c in controls}

    # Build an ordered list of start_lines so that we can add sequential edges
    starts = sorted(start_line_to_bid.keys())

    # Map start line to index in starts
    start_index = {s: idx for idx, s in enumerate(starts)}

    for bid, info in blocks.items():
        s = info['start']
        e = info['end']
        # If this block is a condition block (one of our control cond_lines)
        if s in control_by_cond:
            c = control_by_cond[s]
```

5) CFG Visualization

The script writes a DOT file that can be visualized using Graphviz. Each block is shown as a node, and directed edges show control flow between them.



```
python cfg_reaching_definitions.py prog2.c --render
dot -Tpng prog2_cfg.dot -o prog2_cfg.png

python cfg_reaching_definitions.py prog3.c --render
dot -Tpng prog3_cfg.dot -o prog3_cfg.png
```

6) Cyclomatic Complexity Metrics

Computed automatically from CFG:

$$C = E - N + 2$$

Where:

- **N** = Number of nodes (basic blocks)
- **E** = Number of edges

7) Reaching Definitions Analysis

To perform dataflow analysis, all assignment statements in each block were treated as definitions.

1. Identify Definitions
   a. Assigned a unique ID to each assignment (D1, D2, …)
2. Compute`gen[B]`and `kill[B]`
   a. `gen[B]` = definitions generated inside block B.
   b. `kill[B]` = definitions of the same variables outside block B.
3. Dataflow Equations
   a. `in[B]`=sum of `out[B]` of all the predecessors(B)
   b. `out[B]= gen[B]∪(in[B]-kill[B])`

4. Iterative Computation
    a. Initialize all `in[B]` and `out[B]` as empty
    b. Repeatedly compute until sets converge

8) Output Files

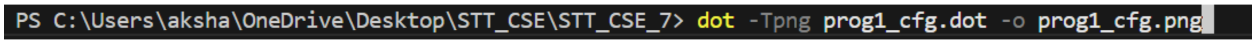
For every input C program (e.g., `prog1.c`), the following files are generated:

- Prog1_cfg.dot:- DOT graph definition of the CFG
- Prog1_cfg.png:- Rendered control flow graph
- Prog1_reaching.txt:- Detailed reaching definitions report

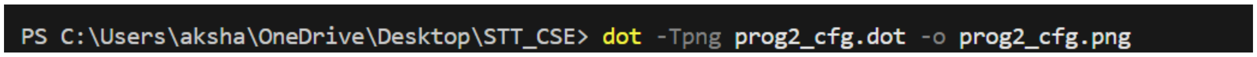
The same is done for each 3 programs. Prog1_cfg.dot, Prog2_cfg.dot, Prog3_cfg.dot

## Results and Analysis

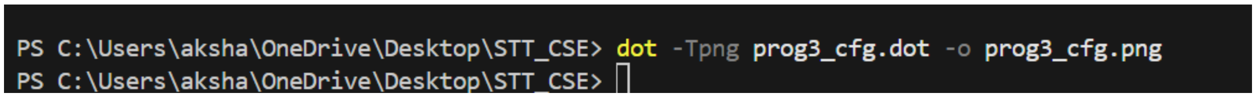
1) CFG diagrams it is in the GitHub repository inside the STT CSE lab 7 folder, its PNG format image is made by



```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE\STT_CSE_7> dot -Tpng prog1_cfg.dot -o prog1_cfg.png
```



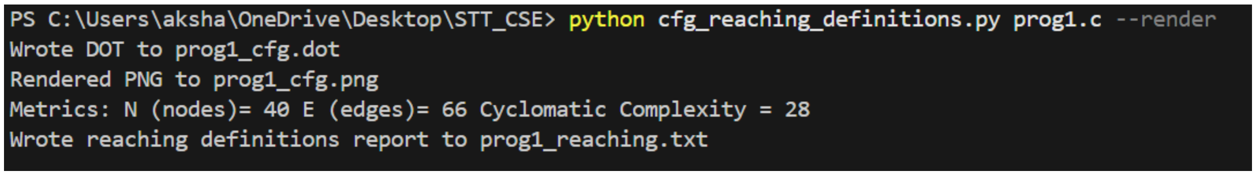
```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE> dot -Tpng prog2_cfg.dot -o prog2_cfg.png
```



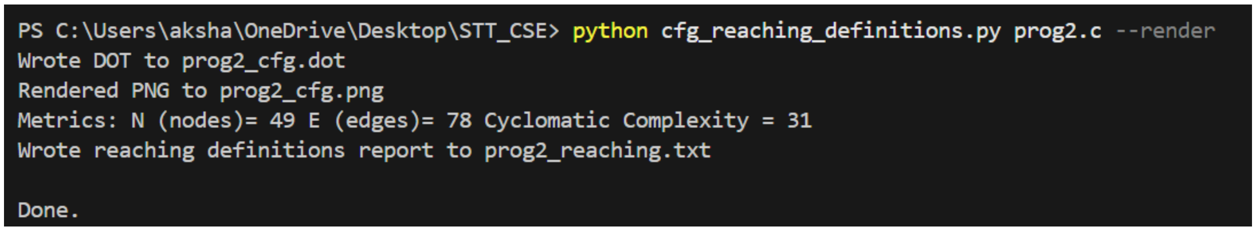
```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE> dot -Tpng prog3_cfg.dot -o prog3_cfg.png
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE>
```

prog1_cfg.png, prog2_cfg.png, prog3_cfg.png

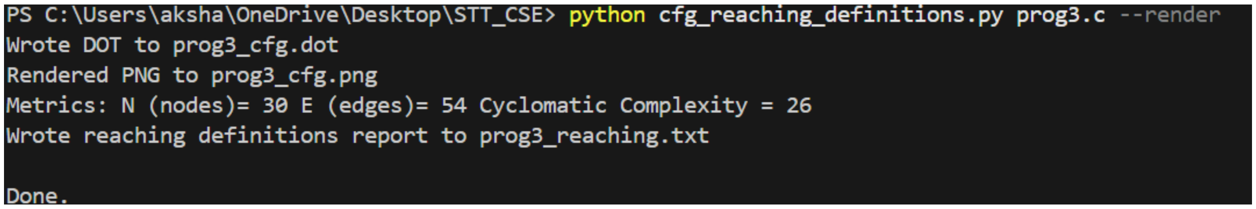
2) Metrics Table



```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE> python cfg_reaching_definitions.py prog1.c --render
Wrote DOT to prog1_cfg.dot
Rendered PNG to prog1_cfg.png
Metrics: N (nodes)= 40 E (edges)= 66 Cyclomatic Complexity = 28
Wrote reaching definitions report to prog1_reaching.txt
```



```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE> python cfg_reaching_definitions.py prog2.c --render
Wrote DOT to prog2_cfg.dot
Rendered PNG to prog2_cfg.png
Metrics: N (nodes)= 49 E (edges)= 78 Cyclomatic Complexity = 31
Wrote reaching definitions report to prog2_reaching.txt

Done.
```



```
PS C:\Users\aksha\OneDrive\Desktop\STT_CSE> python cfg_reaching_definitions.py prog3.c --render
Wrote DOT to prog3_cfg.dot
Rendered PNG to prog3_cfg.png
Metrics: N (nodes)= 30 E (edges)= 54 Cyclomatic Complexity = 26
Wrote reaching definitions report to prog3_reaching.txt

Done.
```

3) Reaching Definitions Table
    a) Tables for `gen[B]`, `kill[B]`, `in[B]`, `out[B]` for all programs and final `in[B]`and `out[B]`for each block is also made inside prog1_reaching.txt, prog2_reaching.txt, and prog3_reaching.txt for respective files.

## Discussion and Conclusion

This lab provided a comprehensive understanding of control flow analysis and dataflow analysis on C programs. The workflow  from preprocessing C code to constructing the Control Flow Graph (CFG), computing basic blocks, and performing Reaching Definitions analysis highlighted both theoretical concepts and practical implementation challenges.

**Challenges Faced**

- One significant challenge was accurately identifying leaders and control structures. C programs often contain complex nested if-else, for, while, and do-while loops, sometimes without braces, which require careful heuristic handling. Ensuring that all leaders were correctly marked for CFG construction demanded multiple iterations and testing against sample programs.
- Another challenge was matching braces and handling single-line blocks. While multi-line blocks were straightforward, single-line statements without {} needed special handling to avoid incorrect block boundaries.
- Managing edges in the CFG also posed difficulties. Correctly adding sequential, conditional, back, and exit edges required mapping block start and end indices carefully. Deduplication of edges was necessary to ensure accurate representation of the control flow.
- Finally, performing Reaching Definitions analysis over loops and nested structures required a worklist-based iterative approach. Ensuring convergence of in and out sets and handling variable redefinitions across multiple blocks demanded careful bookkeeping and testing.

## Reflections and Insights

- This lab reinforced the importance of systematic preprocessing. Removing comments and preserving line indexing was crucial for leader detection, block formation, and accurate mapping of definitions.
- It also highlighted the utility of basic blocks and control structures in understanding program behavior. Representing code in this abstracted form made subsequent dataflow analyses like gen/kill computation and iterative Reaching Definitions feasible and interpretable.
- Visualizing the CFG via Graphviz proved extremely helpful for debugging and validating the block and edge construction. Seeing nodes and labeled edges clarified how conditional and loop statements affected control flow.
- Additionally, the lab emphasized that automation and heuristics have limitations. While most cases were handled correctly, some complex nested or unusual C constructs may require manual inspection or enhanced parsing logic.

## Lessons Learned

- Preprocessing and normalization of code is essential for accurate static analysis.
- Proper leader identification directly affects CFG accuracy and subsequent dataflow analyses.
- Iterative worklist algorithms are reliable for computing reaching definitions but require careful handling of loops and variable scoping.
- Visualization helps bridge the gap between abstract analysis and human comprehension.
- Combining heuristics with systematic bookkeeping allows handling of real-world program constructs effectively.

## Conclusion

In conclusion, this lab provided a hands-on understanding of control flow graph construction, cyclomatic complexity measurement, and Reaching Definitions analysis in C programs. The methodology—from preprocessing code to leader detection, basic block formation, edge construction, and dataflow analysis—highlighted practical challenges in static program analysis, particularly with nested control structures and variable scoping. Overall, the lab offered both technical insights into CFGs and dataflow computation, and practical reflections on accurately modeling program behavior for static analysis and software metrics.

Reference:-

1]https://drive.google.com/file/d/1cMLqFTzRJcpvq8iTEL4XV0K9BclsALzE/view?usp=sharing

2]https://drive.google.com/file/d/195OJgPdnY7ixEyMarA5ZdkL7tVR8deG_/view

3]https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

4]https://seart-ghs.si.usi.ch/