



Indian Institute of Technology Gandhinagar

---

*CS202 - Software Tools and Techniques for CSE*

# ASSESSMENT 2

---

Name - Bhavya Parmar & Surriya Gokul

Roll No. - 23110059 & 23110324

Date of Submission - 19 Oct 2025

---

# Table of Contents

---

<b>Lab 6 - Evaluation of Vulnerability Analysis Tools using CWE-based Comparison.....</b>	<b>3</b>
Introduction.....	3
Methodology & Execution.....	4
Results & Analysis.....	12
Discussions & Conclusion.....	15
References.....	16
<b>Lab 7 – Reaching Definitions Analyser for C Programs.....</b>	<b>17</b>
Introduction.....	17
Methodology & Execution.....	18
Interpretation of Results.....	22
How I Checked.....	22
Findings.....	23
Reasoning / Intuition.....	23
Overview.....	23
Groups Identified.....	23
Group {D1, D9} — Counter/Position Variable.....	23
Group {D2, D3, D7, D15, D21} — Word/Character Tracker.....	23
Group {D4, D6} — Secondary Counter / Temporary Variable.....	23
General Intuition.....	24
Overview.....	24
Groups Identified.....	24
Group {D1, D6, D11, D16} — Main Loop/Index Variable.....	24
Group {D2, D5, D12, D15} — Secondary Index/Counter.....	24
Group {D20, D25, D30, D37} — Path-Length / Step Counter.....	24
Group {D29, D36} — Flag / State Variable.....	24
Takeaway.....	25
Results & Analysis.....	25
Discussions & Conclusion.....	25
References.....	26

# *Lab 6 - Evaluation of Vulnerability Analysis Tools using CWE-based Comparison*

---

## **Introduction**

### Objective

The objective of this lab is to evaluate and compare the effectiveness of selected Static Application Security Testing (SAST) tools in identifying common software weaknesses within real-world projects. This comparative analysis will focus on detection capabilities across specific Common Weakness Enumeration (CWE) categories and utilize metrics, such as Intersection over Union (IoU), to quantify and analyze the pairwise agreement and disagreement between the tools.

### Tools

- Python (version 3.13.6)
- VS Code (version 1.103.2)
- Kaggle Notebook (Python version 3.11.13)
- Snyk CLI (version 1.1300.0)
- Semgrep CLI (version 1.139.0)
- CodeQL CLI (version 2.23.2)
- Pandas (version 2.3.1)
- Matplotlib (version 3.10.5)
- Seaborn (version 0.13.2)

### Setup

For this lab, I used both VS Code and Kaggle Python notebook. The SAST tools used were: CodeQL, Semgrep, Snyk.

### **Setting up CodeQL**

I followed the instructions given in the [CodeQL CLI documentation](#)

*NOTE - Add <extraction-root>/codeql to your PATH, so that you can run the executable as just codeql.*

Then I installed the Python packages required using:

*codeql pack download codeql/python-queries*

### **Setting up Semgrep**

I installed Semgrep CLI using the below command in the terminal:

*pip install semgrep*

Then I authorized using:

*semgrep login*

I was then redirected to Semgrep's website's login page, where I set up an account and allowed permissions to access my repositories.

## Setting up Snyk

I installed Snyk CLI using the following command:

*npm install -g snyk*

Then I authorized using:

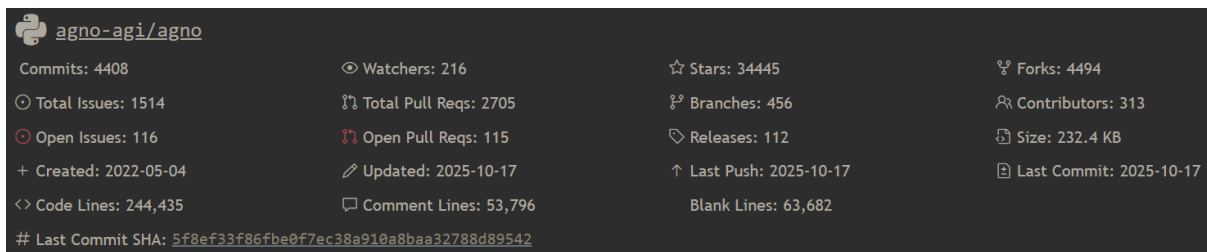
*snyk auth*

And again, I was redirected to Snyk's login page, where I set up an account and allowed permissions to access my repositories.

## Methodology & Execution

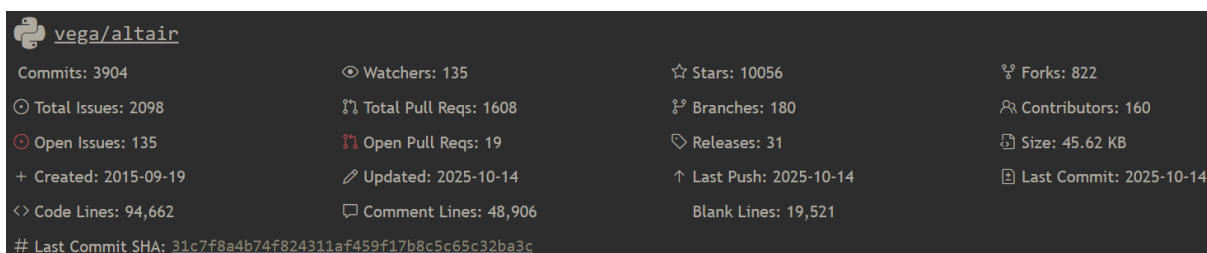
### 1) Repository Selection

The first step was to select 3 appropriate medium-to-large scale open-source repository to analyze. It had to be a real-world project. So, for this, I used the SEART GitHub Search Engine. The metrics I set in order to find such a repository were: >2k commits, >5k stars, >30k lines of code, and Python as the language. These are the repositories that I finally chose: Agno, Altair, and Androguard. I cloned these repositories into my working directory using "git clone <repo\_name>".



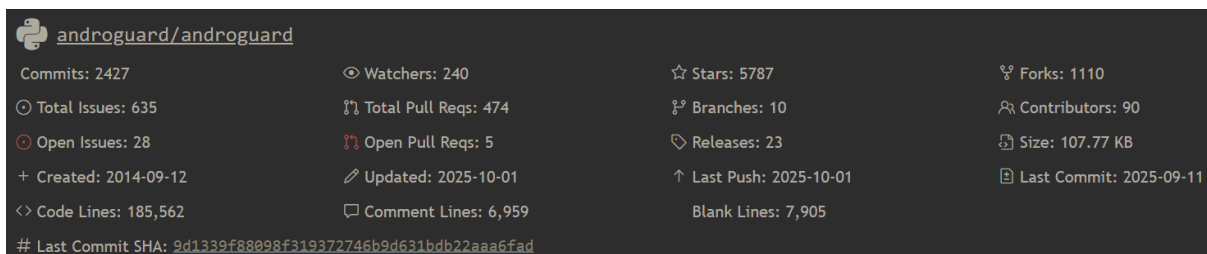
agno-agi/agno

Commits: 4408	👁 Watchers: 216	☆ Stars: 34445	🍴 Forks: 4494
🕒 Total Issues: 1514	🔗 Total Pull Req: 2705	🌿 Branches: 456	👤 Contributors: 313
🔴 Open Issues: 116	🔗 Open Pull Req: 115	📦 Releases: 112	📄 Size: 232.4 KB
+ Created: 2022-05-04	🕒 Updated: 2025-10-17	↑ Last Push: 2025-10-17	📅 Last Commit: 2025-10-17
<> Code Lines: 244,435	💬 Comment Lines: 53,796	Blank Lines: 63,682	
# Last Commit SHA: 5f8ef33f86fba0f7ec38a910a8baa32788d89542			



vega/altair

Commits: 3904	👁 Watchers: 135	☆ Stars: 10056	🍴 Forks: 822
🕒 Total Issues: 2098	🔗 Total Pull Req: 1608	🌿 Branches: 180	👤 Contributors: 160
🔴 Open Issues: 135	🔗 Open Pull Req: 19	📦 Releases: 31	📄 Size: 45.62 KB
+ Created: 2015-09-19	🕒 Updated: 2025-10-14	↑ Last Push: 2025-10-14	📅 Last Commit: 2025-10-14
<> Code Lines: 94,662	💬 Comment Lines: 48,906	Blank Lines: 19,521	
# Last Commit SHA: 31c7f8a4b74f824311af459f17b8c5c65c32ba3c			



androguard/androguard

Commits: 2427	👁 Watchers: 240	☆ Stars: 5787	🍴 Forks: 1110
🕒 Total Issues: 635	🔗 Total Pull Req: 474	🌿 Branches: 10	👤 Contributors: 90
🔴 Open Issues: 28	🔗 Open Pull Req: 5	📦 Releases: 23	📄 Size: 107.77 KB
+ Created: 2014-09-12	🕒 Updated: 2025-10-01	↑ Last Push: 2025-10-01	📅 Last Commit: 2025-09-11
<> Code Lines: 185,562	💬 Comment Lines: 6,959	Blank Lines: 7,905	
# Last Commit SHA: 9d1339f88098f319372746b9d631bdb22aaa6fad			

## 2) Selection of SAST Tools

Three static vulnerability analysis tools were selected that explicitly support CWE-based reporting and can be applied to Python codebases from <https://github.com/analysis-tools-dev/static-analysis>

These tools were:

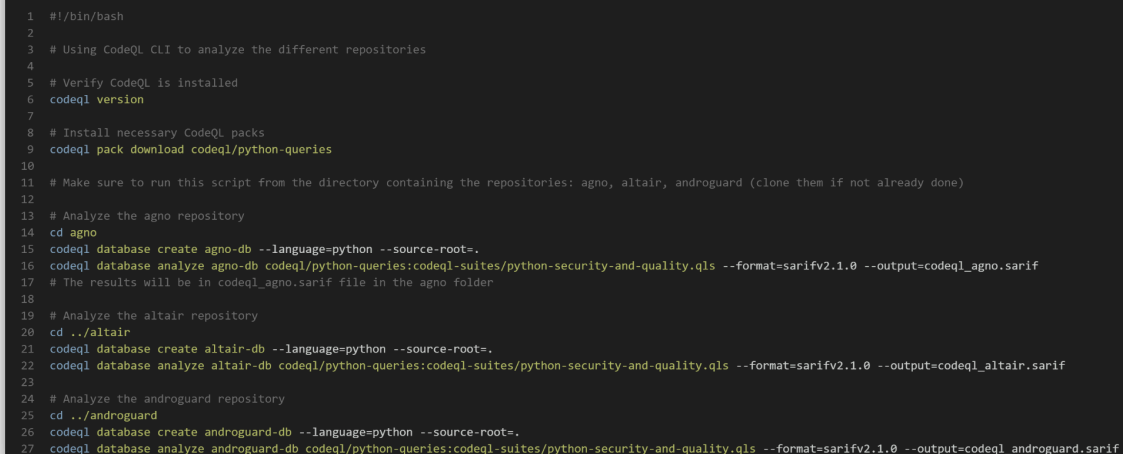
- [CodeQL](#) - Semantic code analysis framework by GitHub. All built-in queries are annotated with “external/cwe/cwe-###”.
- [Semgrep](#) - A Lightweight static analyzer supporting custom and community rules, most of which contain CWE tags.
- [Snyk](#) - Combines SAST and Software Composition Analysis. Provides CWE IDs for both code-level and dependency vulnerabilities.

These tools were chosen as they reported explicit CWE-tagged outputs, they were free and open-source, compatible with Python projects, and were easy to set up on a Windows environment.

## 3) Run Vulnerability Tools on Project and Collect Pairwise Findings

### CodeQL

So now I started to run the tools one by one on all three repositories. First, I started off with CodeQL. I used the following Shell script to run the analysis.



```
1 #!/bin/bash
2
3 # Using CodeQL CLI to analyze the different repositories
4
5 # Verify CodeQL is installed
6 codeql version
7
8 # Install necessary CodeQL packs
9 codeql pack download codeql/python-queries
10
11 # Make sure to run this script from the directory containing the repositories: agno, altair, androguard (clone them if not already done)
12
13 # Analyze the agno repository
14 cd agno
15 codeql database create agno-db --language=python --source-root=.
16 codeql database analyze agno-db codeql/python-queries:codeql-suites/python-security-and-quality.qls --format=sarifv2.1.0 --output=codeql_ago.sarif
17 # The results will be in codeql_ago.sarif file in the agno folder
18
19 # Analyze the altair repository
20 cd ../altair
21 codeql database create altair-db --language=python --source-root=.
22 codeql database analyze altair-db codeql/python-queries:codeql-suites/python-security-and-quality.qls --format=sarifv2.1.0 --output=codeql_altair.sarif
23
24 # Analyze the androguard repository
25 cd ../androguard
26 codeql database create androguard-db --language=python --source-root=.
27 codeql database analyze androguard-db codeql/python-queries:codeql-suites/python-security-and-quality.qls --format=sarifv2.1.0 --output=codeql_androguard.sarif
```

This first checked the CodeQL version to verify installation, then I installed the necessary packages to analyze Python-based repositories. Then, I built a database for each project using:

*codeql database create <repo\_name>-db --language=python --source-root=.*

Finally, I ran the complete security and quality query suite using:

*codeql database analyze agno-db codeql/python-queries:codeql-suites/python-security-and-quality.qls --format=sarifv2.1.0 --output=codeql\_ago.sarif*

The output was a SARIF JSON file containing results from over 120 built-in queries. Each finding included a CWE tag like "external/cwe/cwe-078"

The output was a SARIF JSON file containing results from over 120 built-in queries. Each finding included a CWE tag like "external/cwe/cwe-078". I wrote a small Python script to parse these SARIF files and count the occurrences of each CWE.

## Semgrep

Next, I used Semgrep CLI. After setting it up and cloning the repositories, I ran the following command for each repository:

```
cd <repo_directory>
semgrep scan --config "p/default" --json-output=semgrep_<repo_name>.json
```

This scanned the entire project using a built-in Python security rule set. Each rule already had CWE metadata. The analysis was quick, usually completing in under two minutes per project.

## Snyk

Finally, the last tool I used was Snyk. Again, after setting it up and cloning the repositories, I ran the following commands to analyze each repository:

```
cd <repo_directory>
snyk code test --json > snyk_<repo_name>.json
```

This command generated a JSON file listing each issue detected in the code, along with its CWE ID. I noticed that when I scanned the same project using the Snyk web interface, the results were far more extensive — over 200 vulnerabilities, compared to just a few in the CLI scan. Later, I realized this was because the web scan also includes dependency analysis, while the CLI focuses only on static code issues.

For my lab's CWE analysis, I only used the SAST (code-level) results from the CLI because the assignment specifically focused on static vulnerability detection.

## Data Aggregation and Processing

After obtaining the results from all three tools, I converted their JSON outputs into a single standardized CSV format using a Python script.

```
import json, csv, re
from pathlib import Path
from collections import defaultdict

CWE_RE = re.compile(r"CWE-(\d+)")
ENCODINGS = ("utf-8", "utf-16", "utf-8-sig")

def load_json(p: Path):
    for enc in ENCODINGS:
        try:
            return json.loads(p.read_text(encoding=enc))
        except UnicodeError:
            continue
```

```

        except json.JSONDecodeError:
            break
    return None

def detect_tool(data):
    if isinstance(data, dict):
        if "runs" in data:
            # Check SARIF structure
            for run in data.get("runs", []):
                tool_name = run.get("tool", {}).get("driver", {}).get("name", "").lower()
                if "snyk" in tool_name:
                    return "snyk"
                if "codeql" in tool_name:
                    return "codeql"
        if "results" in data:
            # Check Semgrep structure
            for r in data.get("results", []):
                if "extra" in r and "metadata" in r["extra"] and "cwe" in r["extra"]["metadata"]:
                    return "semgrep"
        return "unknown"

def extract_project_name(file_path: Path):
    stem = file_path.stem.lower()
    for prefix in ("snyk_", "semgrep_", "codeql_"):
        if stem.startswith(prefix):
            return stem[len(prefix):]
    return file_path.parent.name

def extract_snyk(data):
    # SARIF structure
    cwe_counts = defaultdict(int)
    for run in data.get("runs", []):
        rule_map = {}
        for rule in run.get("tool", {}).get("driver", {}).get("rules", []):
            rid = rule.get("id")
            cwes = []
            raw = rule.get("properties", {}).get("cwe", [])
            if isinstance(raw, str):
                raw = [raw]
            for entry in raw:
                for m in CWE_RE.findall(str(entry)):
                    cwes.append(f"CWE-{int(m)}")
            if cwes:
                rule_map[rid] = cwes
        for result in run.get("results", []):
            rid = result.get("ruleId")
            for cwe in rule_map.get(rid, []):
                cwe_counts[cwe] += 1
    return cwe_counts

def extract_semgrep(data):
    cwe_counts = defaultdict(int)
    for r in data.get("results", []):
        raw = r.get("extra", {}).get("metadata", {}).get("cwe")
        if raw is None: continue

```

```

        if not isinstance(raw, (list, tuple)):
            raw = [raw]
        for item in raw:
            for m in CWE_RE.findall(str(item)):
                cwe_counts[f"CWE-{int(m)}"] += 1
    return cwe_counts

def extract_codeql(data):
    # CodeQL SARIF structure
    cwe_counts = defaultdict(int)

    # Map ruleId -> tags
    rule_tags = {}
    for run in data.get("runs", []):
        for rule in run.get("tool", {}).get("driver", {}).get("rules", []):
            tags = rule.get("properties", {}).get("tags", [])
            rule_id = rule.get("id")
            rule_tags[rule_id] = tags

    # Extract CWE IDs from results
    for run in data.get("runs", []):
        for result in run.get("results", []):
            rule_id = result.get("ruleId")
            tags = rule_tags.get(rule_id, [])
            for tag in tags:
                tag_lower = tag.lower()
                if "cwe-" in tag_lower:
                    cwe_id = tag_lower.split("/")[-1].upper()
                    if CWE_RE.match(cwe_id):
                        cwe_counts[cwe_id] += 1

    return cwe_counts

EXTRACTORS = {
    "snyk": extract_snyk,
    "semgrep": extract_semgrep,
    "codeql": extract_codeql
}

def gather_logs(logs_path: Path):
    files = []
    for subfolder in ["codeql", "semgrep", "snyk"]:
        folder_path = logs_path / subfolder
        if folder_path.is_dir():
            files.extend(folder_path.rglob("*.json"))
            files.extend(folder_path.rglob("*.sarif"))
    return files

def main():
    logs_path = Path("logs") # Automatically traverse the logs folder
    out_path = Path("consolidated_cwe.csv")

    top25 = [
        "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"
    ]

    rows = []
    for f in gather_logs(logs_path):
        data = load_json(f)
        if data is None:
            continue
        tool = detect_tool(data)
        if tool not in EXTRACTORS:
            continue
        project = extract_project_name(f)
        counts = EXTRACTORS[tool](data)
        for cwe, n in counts.items():
            rows.append((project, tool, cwe, n, "Yes" if cwe in top25 else "No"))

    rows.sort()
    with out_path.open("w", newline="", encoding="utf-8") as fh:
        w = csv.writer(fh)
        w.writerow(["Project_name", "Tool_name", "CWE_ID", "Number of Findings",
"Is_In_CWE_Top_25"])
        w.writerows(rows)
    print(f"Wrote {out_path} ({len(rows)} rows)")

if __name__ == "__main__":
    main()

```

This Python script automates the process of **collecting, parsing, and consolidating CWE vulnerability data** from our three tools. It recursively scans a “logs” directory for all “.json” and “.sarif” result files, automatically detects which tool generated each file, and then extracts CWE IDs and their occurrence counts using tool-specific parsing functions. The script also checks whether each detected CWE belongs to the MITRE Top 25 CWE list, marks it accordingly, and writes all the aggregated results into a single CSV file (consolidated\_csv.csv). Each row in the final output contains the project name, tool name, CWE ID, number of findings, and whether it is part of the CWE Top 25.

#### 4) CWE Coverage and IOU Analysis

For this, I created a new Python notebook called “analysis.py” and then I imported the necessary libraries and then imported the consolidated\_csv.csv file and defined the Top 25 CWEs

```

1  # Load the Dataset
2
3  df = pd.read_csv("consolidated_csv.csv")
4  print(df.info())
5
6  top25 = [
7      "CWE-79", "CWE-787", "CWE-89", "CWE-352", "CWE-22", "CWE-125", "CWE-78", "CWE-416",
8      "CWE-862", "CWE-434", "CWE-94", "CWE-20", "CWE-77", "CWE-287", "CWE-269", "CWE-502",
9      "CWE-200", "CWE-863", "CWE-918", "CWE-119", "CWE-476", "CWE-798", "CWE-190", "CWE-400", "CWE-306"
10 ]

```

Then to extract the set of CWE IDs (unique) detected by each tool, I ran the following line of code:

```

1 # Get the set of unique CWE IDs detected by each tool
2
3 tool_cwe = df['CWE_ID'].groupby(df['Tool_name']).apply(set).to_dict()
4 print(tool_cwe)
5

```

Then to calculate the top 25 CWE coverage percentage at tool level, I ran the following:

```

1 # Compute Top 25 CWE coverage (%) at the tool level.
2
3 top25_set = set(top25)
4 coverage = {}
5
6 for tool, cwes in tool_cwe.items():
7     covered = len(top25_set & cwes)
8     total = len(cwes)
9     coverage_pct = (covered / total * 100) if total > 0 else 0
10    coverage[tool] = coverage_pct
11    print(f"{tool}:")
12    print(f"    Unique CWE IDs found: {total}")
13    print(f"    Top 25 CWE IDs found: {covered}")
14    print(f"    Top 25 CWE Coverage: {coverage_pct:.2f}%\n")
15
16 print("Summary Table:")
17 print("Tool\tUnique_CWE\tTop25_CWE\tCoverage(%)")
18 for tool in coverage:
19     total = len(tool_cwe[tool])
20     covered = len(top25_set & tool_cwe[tool])
21     print(f"{tool}\t{total}\t{covered}\t{coverage[tool]:.2f}")

```

And then to visualize the same, I ran the following code:

```

1 # Visualization of Top 25 CWE coverage by tool
2
3 fig, axs = plt.subplots(1, 2, figsize=(12, 5))
4
5 # Bar chart
6 axs[0].bar(coverage.keys(), coverage.values())
7 axs[0].set_ylabel("Top 25 CWE Coverage (%)")
8 axs[0].set_title("Top 25 CWE Coverage by Tool")
9
10 # Pie chart
11 axs[1].pie(coverage.values(), labels=coverage.keys(), autopct='%1.1f%%')
12 axs[1].set_title("Top 25 CWE Coverage Distribution")
13
14 plt.tight_layout()
15 plt.show()

```

And to understand this better, I plotted tool-wise bar charts of the number of findings for each CWE detected whilst highlighting the top 25 CWEs. The code for this is below:

```
1 # Toolwise bar charts: Number of Findings for each CWE detected by each tool, highlighting Top 25 CWEs
2
3 for tool in df['Tool_name'].unique():
4     df_tool = df[df['Tool_name'] == tool]
5     cwe_counts_tool = df_tool.groupby('CWE_ID')['Number of Findings'].sum().sort_values(ascending=False)
6     is_top25_tool = cwe_counts_tool.index.isin(top25_set)
7     plt.figure(figsize=(10, 6))
8     bars = plt.bar(cwe_counts_tool.index, cwe_counts_tool.values, color=['orange' if t else 'skyblue' for t in is_top25_tool])
9     for idx, bar in enumerate(bars):
10         if is_top25_tool[idx]:
11             bar.set_edgecolor('red')
12             bar.set_linewidth(2)
13     plt.title(f"{tool} CWE Findings")
14     plt.xlabel("CWE ID")
15     plt.ylabel("Number of Findings")
16     plt.xticks(rotation=90)
17     plt.tight_layout()
18     plt.show()
```

Next, I had to compute IoU for each tool pair using the formula for Jaccard Index (IoU):

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

And then I created a (tool x tool) IoU matrix, using the following code:

```
1 # Pairwise Agreement (IoU) Analysis
2
3 from itertools import product
4
5 tools = list(tool_cwe.keys())
6 iou_matrix = pd.DataFrame(index=tools, columns=tools, dtype=float)
7
8 for t1, t2 in product(tools, repeat=2):
9     set1, set2 = tool_cwe[t1], tool_cwe[t2]
10     intersection = len(set1 & set2)
11     union = len(set1 | set2)
12     iou = intersection / union if union else 0.0
13     iou_matrix.loc[t1, t2] = iou
14
15 print("Pairwise IoU Matrix:")
16 print(iou_matrix)
```

I also plotted a heatmap of the same for better visualization with the following code:

```
1 # Plotting the heatmap of IoU matrix
2
3 sns.heatmap(iou_matrix, annot=True, cmap="Blues")
4 plt.title("Pairwise Tool IoU (Jaccard Index)")
5 plt.show()
```

Here is the link to the [GitHub repository](#) containing all codes for this lab.

## Results & Analysis

The set of unique CWE IDs detected by each tool is as follows:

Tool	Detected CWE IDs
<b>CodeQL</b>	CWE-570, CWE-079, CWE-020, CWE-036, CWE-563, CWE-095, CWE-772, CWE-396, CWE-315, CWE-390, CWE-497, CWE-665, CWE-088, CWE-312, CWE-359, CWE-099, CWE-584, CWE-073, CWE-094, CWE-532, CWE-571, CWE-209, CWE-022, CWE-023, CWE-117, CWE-116, CWE-295, CWE-561, CWE-078, CWE-685, CWE-581, CWE-628
<b>Semgrep</b>	CWE-116, CWE-078, CWE-295, CWE-095, CWE-079, CWE-327, CWE-939, CWE-089, CWE-706, CWE-502, CWE-353, CWE-1333, CWE-319, CWE-798, CWE-522
<b>Snyk</b>	CWE-916, CWE-023, CWE-295, CWE-079, CWE-346, CWE-611, CWE-942, CWE-089, CWE-547, CWE-798

Here are the quantified results, including the tool-wise top 25 CWE coverage percentage:

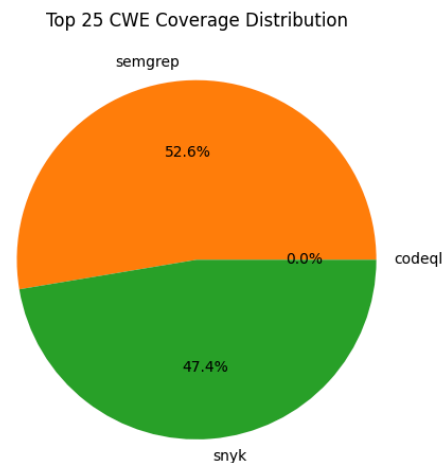
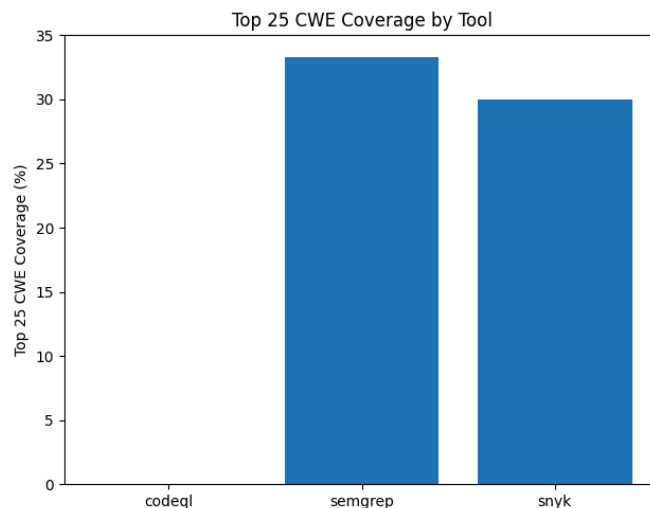
```
codeql:
Unique CWE IDs found: 32
Top 25 CWE IDs found: 0
Top 25 CWE Coverage: 0.00%

semgrep:
Unique CWE IDs found: 15
Top 25 CWE IDs found: 5
Top 25 CWE Coverage: 33.33%

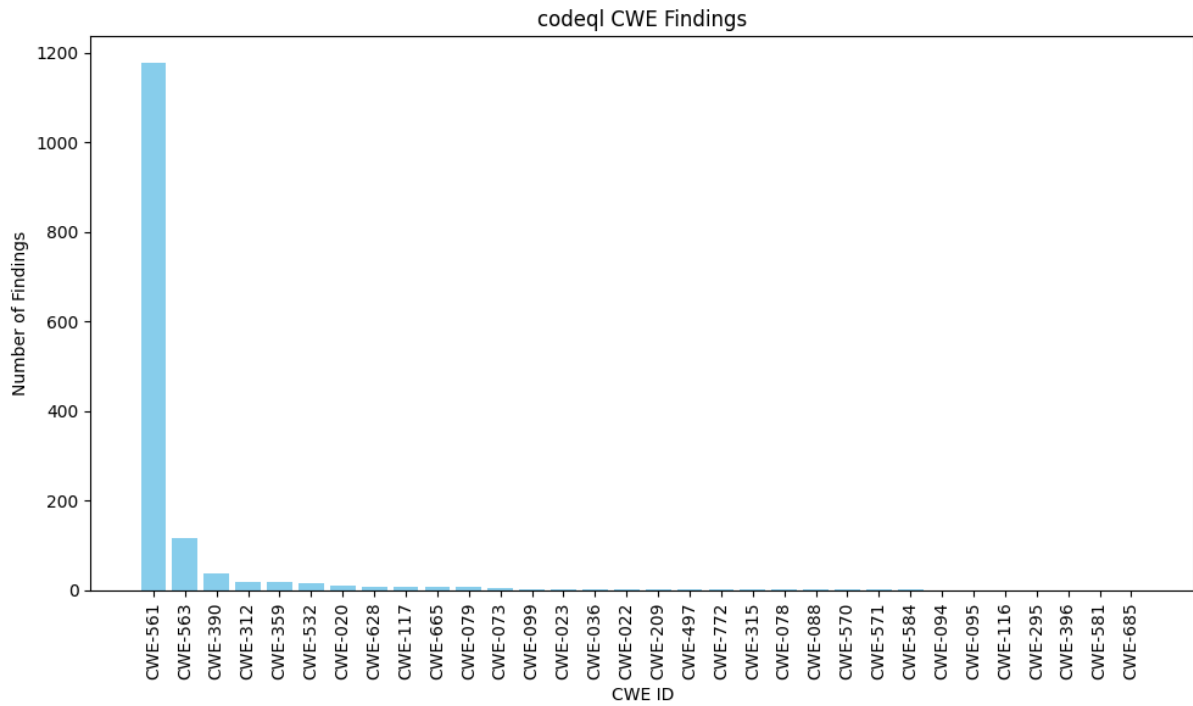
snyk:
Unique CWE IDs found: 10
Top 25 CWE IDs found: 3
Top 25 CWE Coverage: 30.00%

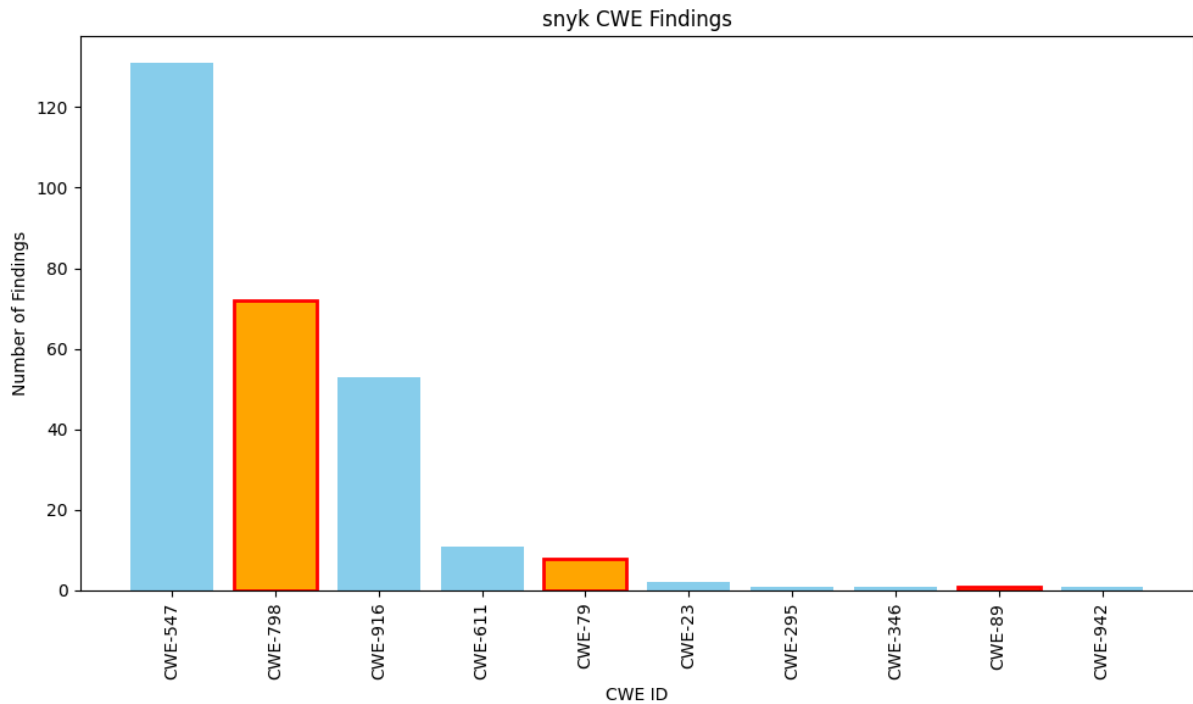
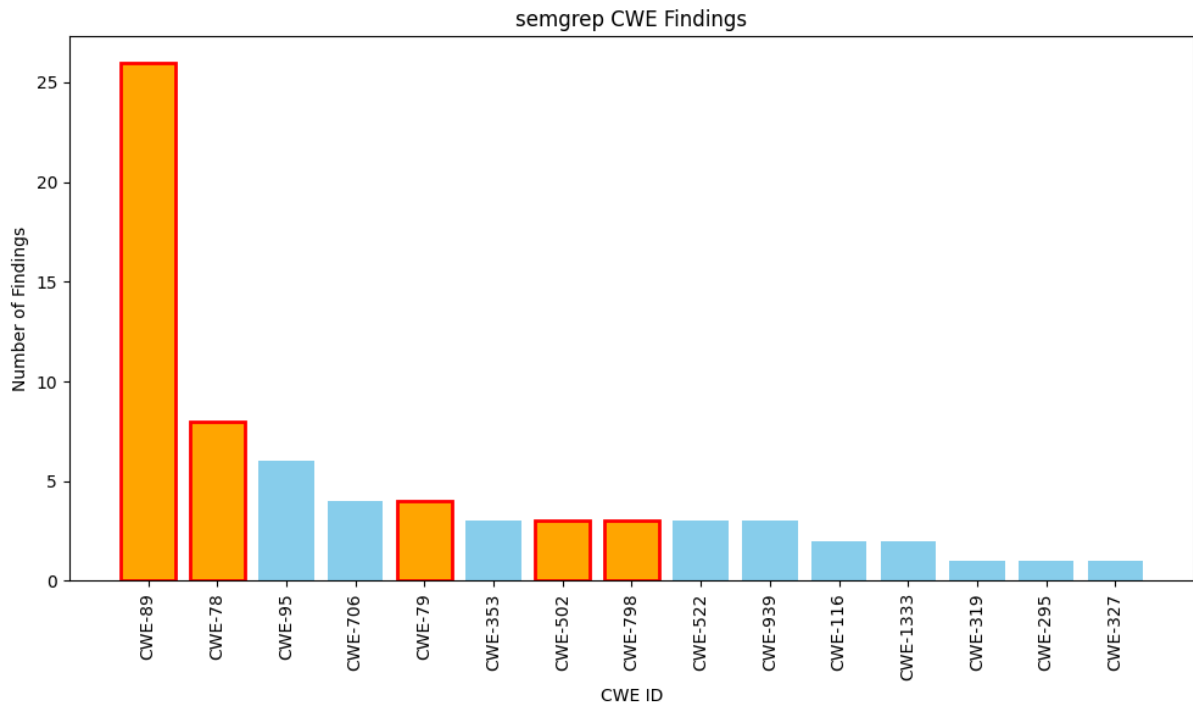
Summary Table:
Tool    Unique_CWE    Top25_CWE    Coverage(%)
codeql  32             0             0.00
semgrep 15             5             33.33
snyk    10             3             30.00
```

And here are a couple of graphs to visualize it better:



Next, we have tool-wise bar charts showing number of findings for each CWE detected by each tool, highlighting Top 25 CWEs:



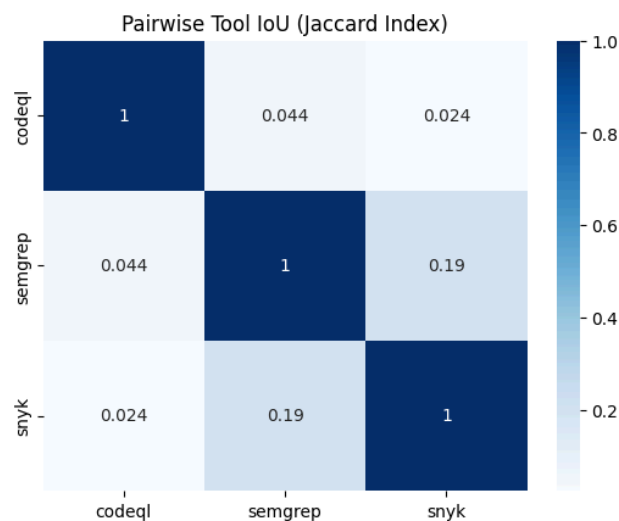


From the results, CodeQL detected the highest number of unique CWE IDs (32 in total), showing that it can identify a wide range of different types of weaknesses. However, none of these belonged to the Top 25 CWE list, which means CodeQL mainly focuses on general coding and structural issues instead of the most critical or commonly exploited ones. Its analysis is deeper and more semantic, aimed at understanding how data moves through the code rather than just spotting surface-level flaws.

Semgrep found 15 unique CWEs, and 5 of them were part of the CWE Top 25, giving it a 33.33% coverage. This shows that Semgrep's rule-based scanning is good at catching high-severity and frequently exploited issues, such as injection or cross-site scripting (XSS), even though it finds fewer total vulnerabilities compared to CodeQL.

Snyk detected 10 unique CWEs, with 3 of them belonging to the Top 25, giving a 30% coverage. This suggests that Snyk focuses more on known and high-impact vulnerabilities, using predefined patterns and vulnerability databases rather than performing in-depth code analysis like CodeQL.

Next, the pairwise IoU showed low overlap amongst the tools, below is a heatmap visualization of it:



The diagonal values (1.0) show perfect overlap for each tool with itself, while the other numbers compare how similar the CWE results are between different tools.

From the matrix, I noticed that the overlap between tools was very low. **CodeQL** shared very few CWEs with **Semgrep** or **Snyk** (only about 2–4%), which means it detects a mostly different set of vulnerabilities. This is expected because CodeQL uses **semantic analysis**, which looks deeper into the logic and data flow of the code rather than just matching surface patterns.

**Semgrep** and **Snyk** had the highest overlap (around 19%), showing that both tools tend to find similar, common issues — usually those found in the **CWE Top 25**, such as injection or configuration problems. However, each tool still has its own unique findings because they use different detection methods and rule sets.

## Discussions & Conclusion

Each tool detected vulnerabilities in its own way. **CodeQL** found the highest number of CWE types (32), showing its strength in analyzing complex code structures and logic. However, none of its results matched the **CWE Top 25**, which means it focused more on general coding and safety issues rather than the most severe security risks. This likely happened because CodeQL's built-in Python queries prioritize accuracy and context, even if they miss some high-impact weaknesses.

**Semgrep**, though it found fewer CWEs, achieved the best **Top 25 coverage (33.33%)**, making it effective at catching the most common and dangerous vulnerabilities such as injections and XSS. It was also much faster and easier to use, which made testing more efficient. **Snyk** detected the fewest CWEs but still covered **30% of the Top 25**, showing that it performs well for known and dependency-related issues. The **low IoU scores** between tools showed that each one finds mostly different kinds of weaknesses, so using them together gives a more complete and balanced view of a project's security.

While working on this lab, I faced a few issues — especially with CodeQL’s setup, which required creating databases and managing query paths, and with Snyk’s CLI, which showed fewer results than its web dashboard. Despite these challenges, this experiment helped me understand how static analysis tools work within the **CWE framework** and how their results can be compared and interpreted. I also learned to extract and analyze CWE-based findings, compute **IoU values**, and visualize results to compare tool performance. Overall, I realized that combining multiple tools provides the most complete picture of software security.

## References

1. [Lab 6 GitHub Repository](#)
  2. [Lab Assignment 6](#)
  3. [List of Static Analysis Tools](#)
  4. [2024 CWE Top 25 Most Dangerous Software Weaknesses](#)
  5. [CodeQL CLI Documentation](#)
  6. [Semgrep CLI Documentation](#)
  7. [Snyk CLI API Documentation](#)
  8. [Agno GitHub Repository](#)
  9. [Altair GitHub Repository](#)
  10. [Androguard GitHub Repository](#)
-



# Lab 7 – Reaching Definitions Analyser for C Programs

---

## Introduction

### Objective

The objective of this lab is to implement and analyse *Reaching Definitions* on real C programs through program analysis techniques. The task involves constructing **Control Flow Graphs (CFGs)**, identifying **basic blocks**, and applying **data-flow equations** iteratively until convergence. By doing so, the analysis determines which variable assignments can reach a specific program point.

This lab aims to go beyond theoretical understanding and apply these concepts to non-trivial C programs (200–300 LOC each), demonstrating how compiler-level analyses work under the hood of modern software tools.

### Tools

- **Programming Language:** Python 3.10+
- **Operating System:** Windows 11 (any OS compatible)
- **Libraries Used:**
  - Graphviz (for CFG visualisation)
  - Matplotlib (for graphical outputs, if needed)
- **Text Editor:** VS Code / PyCharm / Jupyter Notebook
- **Reference Materials:** Lecture 7 slides and official Graphviz documentation

### Setup

I worked on this lab using Python 3.10 with Graphviz installed via “pip install graphviz”. To visualise CFGs, I used the .dot file export format and rendered it using the dot -Tpng file.dot -o file.png command.

Three standalone C programs were used as the corpus for this analysis:

1. **Program 1:** “prog1\_gradebook.c” – A Student Gradebook Analyser featuring conditionals, nested loops, and multiple assignments.
2. **Program 2:** “prog2\_string\_analyzer.c” – A String and Word Analyser with nested ifs, loops, and explicit assignments.

3. **Program 3:** “prog3\_grid\_bfs.c” – A Grid BFS Pathfinding algorithm using multiple definitions of variables, while loops, and conditional logic.

Each program was analysed individually, beginning with leader identification, followed by basic block formation, CFG generation, and finally, Reaching Definitions computation.

## Methodology & Execution

### 1) Program Corpus Selection

The chosen programs fulfil the required structural variety:

Program	Description	Key Features
Gradebook Analyzer	Computes grades and averages for multiple students	If/else chains, nested loops, multiple reassignments
String Analyzer	Tokenises and classifies input text	While + for loops, nested conditionals, and assignments
Grid BFS	BFS traversal on a 2D grid	Queue operations, nested loops, and multiple reaching definitions

The rationale behind selecting these programs was to ensure representation across numeric computation, string manipulation, and algorithmic logic, thereby ensuring a broad spectrum of control flow complexity.

### 2) CFG Construction

The **leaders** for each program were identified based on the rules taught in class and mentioned in the assignment:

1. The first instruction of the program is a leader.
2. Any target of a jump or branch instruction is a leader.
3. Any instruction following a branch is also a leader.

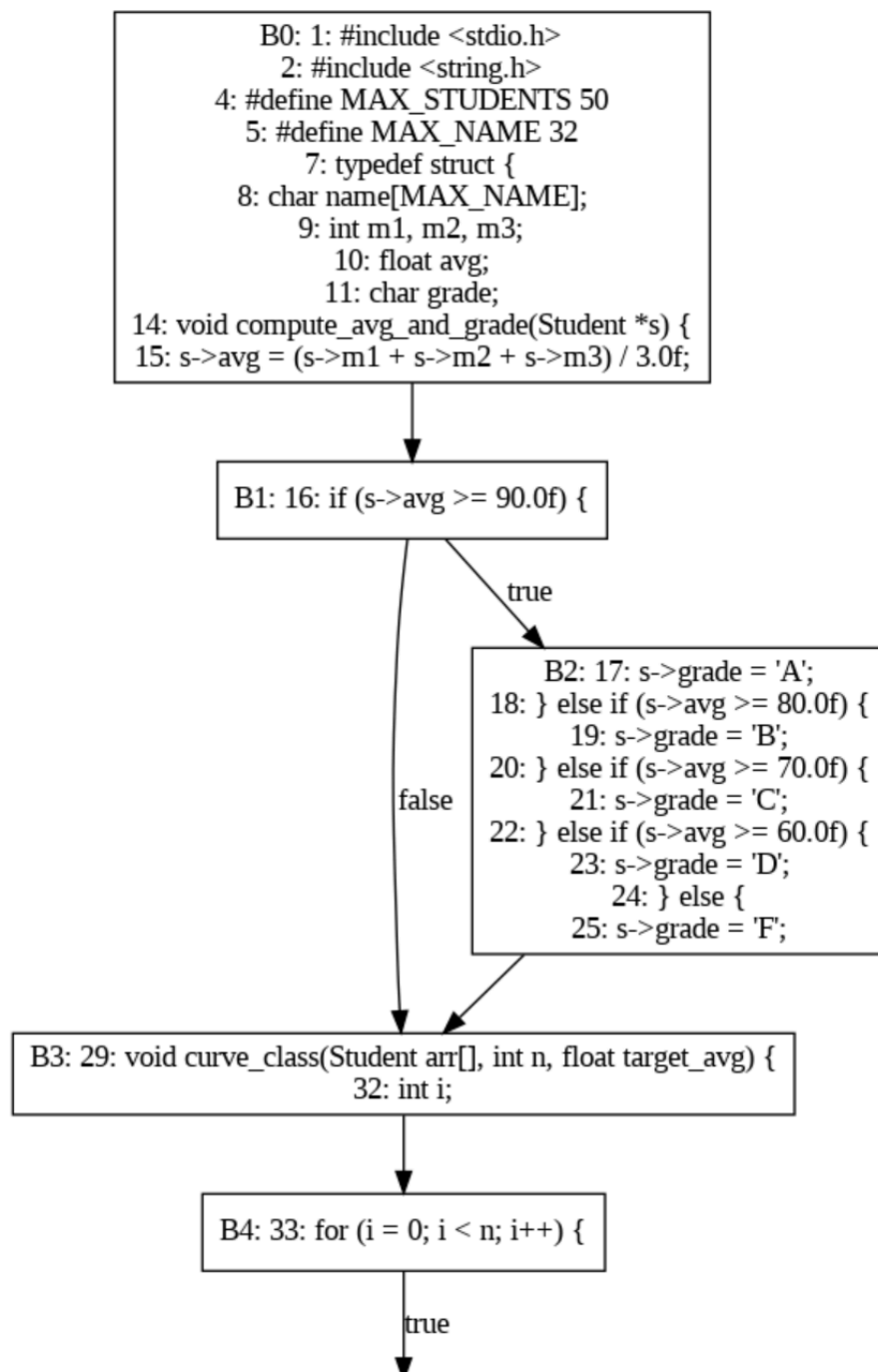
Each leader defined the start of a **basic block**, and the code between leaders was grouped into that block, keeping in mind that each basic block has exactly one entry and one exit point.

Example for *Program 1 (Gradebook)*:

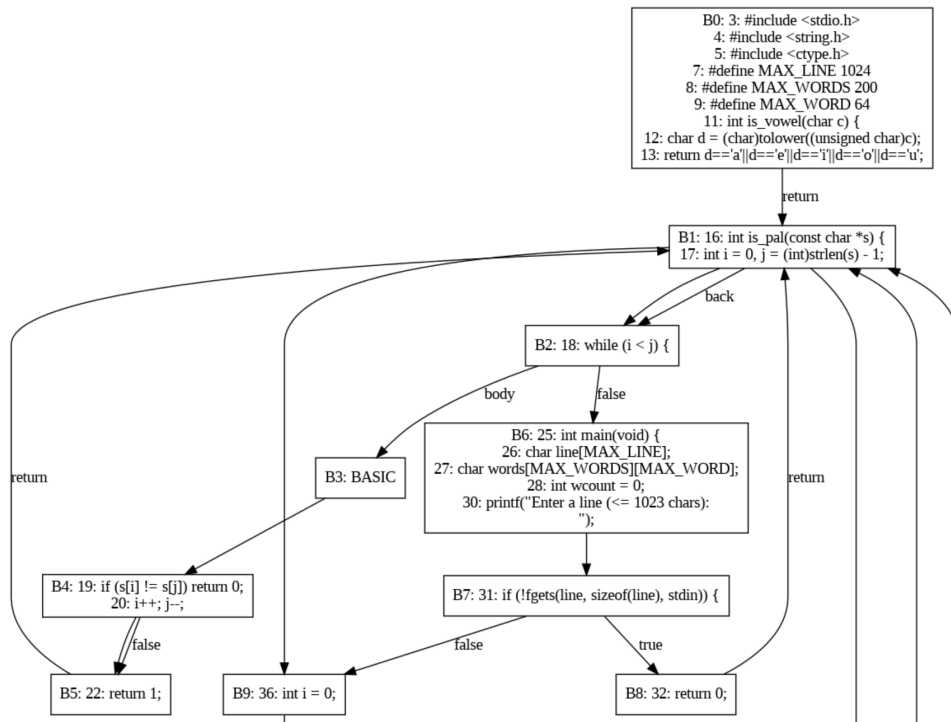
Block	Code Summary
B0	Variable declarations, input for number of students
B1	If condition validating input

B2	For loop reading student marks
B3	Conditional grade assignments
B4	Average computation and print statements
B5	Query loop for student search

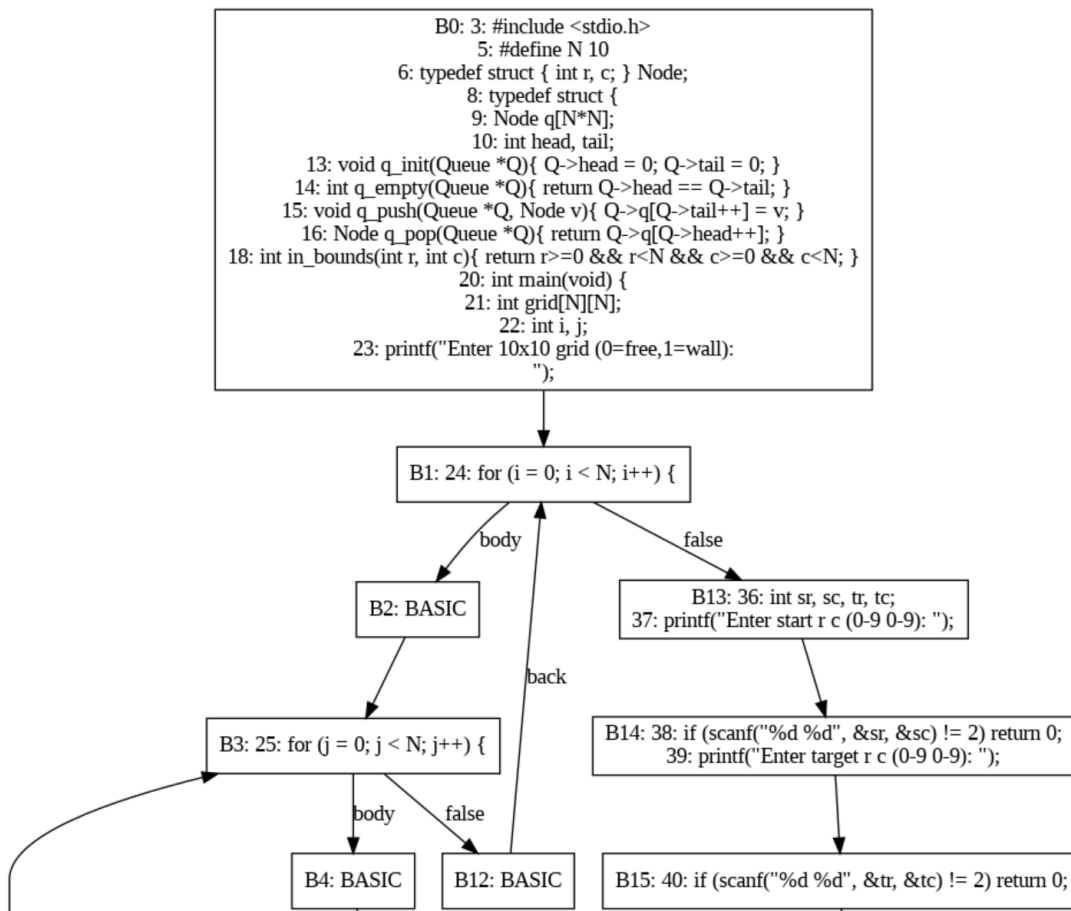
Using this, a **Control Flow Graph (CFG)** was constructed where each basic block represented a node, and directed edges represented control flow (sequential, conditional, and loop edges). Each of the following images also contains links to the GitHub repo where they are hosted and can be viewed fully.



*CFG for Program 1 – Gradebook*



**CFG for Program 2 – String Analyser**



**CFG for Program 3 – Grid BFS**

### 3) Cyclomatic Complexity Computation

For each CFG, the following metrics were computed automatically:

Program	Nodes (N)	Edges (E)	Cyclomatic Complexity (CC = E - N + 2)
Gradebook	48	69	23
String Analyzer	46	68	24
Grid BFS	41	62	23

Higher CC values indicate more complex control flow, as seen in the string analyser program due to nested loops and multiple branching conditions.

Program No.	File Name	No. Of Nodes (N)	No. Of Edges (E)	Cyclomatic Complexity (CC)
1	prog1_gradebook.c	48	69	23
2	prog2_string_analyzer.c	46	68	24
3	prog3_grid_bfs.c	41	62	23

*Table showing calculated Cyclomatic Complexities*

### 4) Reaching Definitions Analysis

For each program, assignment statements (definitions) were identified and given unique IDs (D1, D2, ...) and so on.

Example for *Program 2 – String Analyser*:

Definition ID	Statement
D1	letters = 0;
D2	digits = 0;
D3	vowels = 0;
D4	consonants = 0;
D5	freq[i] = freq[i] + 1;

Each basic block *B* was then annotated with:

- **gen[B]** – Definitions generated within the block
- **kill[B]** – Definitions of the same variable in other blocks

Using the standard data-flow equations:

$$\begin{aligned} \text{in}[B] &= \bigcup \text{out}[P] \quad (\text{for all predecessors } P) \\ \text{out}[B] &= \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \end{aligned}$$

We initialise all  $\text{in}[B]$  and  $\text{out}[B]$  to be empty.

These equations were applied iteratively until the sets stabilised (convergence) and the sets stopped changing.

Iteration	Basic-Block	gen[B]	kill[B]	in[B]	out[B]
0	ENTRY	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	B1	{D1}	$\emptyset$	$\emptyset$	$\emptyset$
0	B2	{D2}	{D15, D20, D24, D3}	$\emptyset$	$\emptyset$
0	B3	{D10, D11, D12, D13, D14, D15, D3, D4, D5, D6, D7, D8, D9}	{D10, D11, D12, D13, D14, D15, D2, D20, D24, D3, D4, D5, D6, D7, D8, D9}	$\emptyset$	$\emptyset$
0	B4	{D16}	$\emptyset$	$\emptyset$	$\emptyset$
0	B5	{D17, D18, D19, D20}	{D15, D17, D18, D19, D2, D20, D24, D3}	$\emptyset$	$\emptyset$
0	B6	{D21, D22, D23, D24}	{D15, D17, D18, D19, D2, D20, D21, D22, D23, D24, D3}	$\emptyset$	$\emptyset$
0	B7	{D25}	$\emptyset$	$\emptyset$	$\emptyset$
0	EXIT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Iteration	Basic-Block	gen[B]	kill[B]	in[B]	out[B]
0	ENTRY	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	B1	{D1}	$\emptyset$	$\emptyset$	$\emptyset$
0	B2	{D2}	{D15, D20, D24, D3}	$\emptyset$	$\emptyset$
0	B3	{D10, D11, D12, D13, D14, D15, D3, D4, D5, D6, D7, D8, D9}	{D10, D11, D12, D13, D14, D15, D2, D20, D24, D3, D4, D5, D6, D7, D8, D9}	$\emptyset$	$\emptyset$
0	B4	{D16}	$\emptyset$	$\emptyset$	$\emptyset$
0	B5	{D17, D18, D19, D20}	{D15, D17, D18, D19, D2, D20, D24, D3}	$\emptyset$	$\emptyset$
0	B6	{D21, D22, D23, D24}	{D15, D17, D18, D19, D2, D20, D21, D22, D23, D24, D3}	$\emptyset$	$\emptyset$
0	B7	{D25}	$\emptyset$	$\emptyset$	$\emptyset$
0	EXIT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

*Reaching Definitions table – Gradebook Initial*

3	ENTRY	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	B1	{D1}	$\emptyset$	$\emptyset$	{D1}
3	B2	{D2}	{D15, D20, D24, D3}	{D1}	{D1, D2}
3	B3	{D10, D11, D12, D13, D14, D15, D3, D4, D5, D6, D7, D8, D9}	{D10, D11, D12, D13, D14, D15, D2, D20, D24, D3, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D15, D2, D3, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D15, D3, D4, D5, D6, D7, D8, D9}
3	B4	{D16}	$\emptyset$	{D1, D10, D11, D12, D13, D14, D15, D3, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D15, D16, D3, D4, D5, D6, D7, D8, D9}
3	B5	{D17, D18, D19, D20}	{D15, D17, D18, D19, D2, D20, D24, D3}	{D1, D10, D11, D12, D13, D14, D15, D16, D3, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D16, D17, D18, D19, D20, D4, D5, D6, D7, D8, D9}
3	B6	{D21, D22, D23, D24}	{D15, D17, D18, D19, D2, D20, D21, D22, D23, D24, D3}	{D1, D10, D11, D12, D13, D14, D16, D17, D18, D19, D20, D21, D22, D23, D24, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D16, D21, D22, D23, D24, D4, D5, D6, D7, D8, D9}
3	B7	{D25}	$\emptyset$	{D1, D10, D11, D12, D13, D14, D16, D21, D22, D23, D24, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D16, D21, D22, D23, D24, D25, D4, D5, D6, D7, D8, D9}
3	EXIT	$\emptyset$	$\emptyset$	{D1, D10, D11, D12, D13, D14, D16, D21, D22, D23, D24, D25, D4, D5, D6, D7, D8, D9}	{D1, D10, D11, D12, D13, D14, D16, D21, D22, D23, D24, D25, D4, D5, D6, D7, D8, D9}

*Reaching Definitions table – Gradebook Converged*

The process was repeated similarly for the *String Analyser* and *Grid BFS* programs. The detailed Reaching Definitions table can be found in the GitHub Repository using this [link](#)

## Interpretation of Results

### Program 1 — gradebook.c

#### How I Checked

I examined the  $\text{in}[B]$  sets for each basic block and grouped together definition IDs that refer to the same variable.

To ensure correctness, I used *kill sets* from single-definition blocks to safely determine which definitions corresponded to the same variable.

Whenever  $\text{in}[B]$  contained two or more of those IDs at the same time, it indicated that the variable had **multiple reaching definitions** at that program point.

## Findings

Only **one variable** shows up with multiple reaching definitions:

- **Variable Group:** {D2, D15, D20, D24}  
*(This is the loop/index variable that's initialised and then incremented in different loops.)*
- **Blocks Affected:** B3 and B6  
*(The in[B] of these blocks contains two or more of D2/D15/D20/D24.)*

## Reasoning / Intuition

This variable is redefined along different paths — for example, initialised at the top of a loop, incremented inside, and possibly re-initialised in another loop.

When control flow merges, multiple definitions of the same variable can reach the join point, leading to multiple reaching definitions in the corresponding in[B].

## Program 2 — string\_analyzer

### Overview

In this program, **multiple variables** have more than one reaching definition at the same program point.

### Groups Identified

#### **Group {D1, D9} — Counter/Position Variable**

- **Blocks:** B3, B4, B5, B6, B7, B8, B9, B10, B11, EXIT
- **Interpretation:**  
Initialised once (e.g.,  $i = 0$ ) and incremented along several paths (e.g.,  $i++$  inside different loops or conditions).  
At control-flow joins, both “previous” and “incremented” definitions reach the same block.

#### **Group {D2, D3, D7, D15, D21} — Word/Character Tracker**

- **Blocks:** B3, B5, B8
- **Interpretation:**  
Defined across different stages (initialization → scanning → analysis).  
It may be reset and advanced in separate phases, causing multiple reaching definitions at the entry of certain blocks.

#### **Group {D4, D6} — Secondary Counter / Temporary Variable**

- **Blocks:** B5
- **Interpretation:**  
Defined differently along alternative paths that later merge, resulting in multiple reaching definitions at the join.

## General Intuition

The analyzer alternates between scanning, tokenizing, and summarizing stages.

Several counters are repeatedly initialized and incremented in different loops or branches.

These alternative execution paths cause multiple reaching definitions to appear in the in[B] sets for the corresponding variables.

### Program 3 — *grid\_bfs*

#### Overview

This program features complex control-flow joins typical of BFS traversal, including loops for neighbour exploration and backtracking.

As a result, **several variables** have multiple reaching definitions at numerous program points.

#### Groups Identified

##### **Group {D1, D6, D11, D16} — Main Loop/Index Variable**

- **Blocks:** B2–B22, B3–B21, and EXIT
- **Interpretation:**  
Initialized once, then updated in the BFS main loop and during neighbor processing.  
Different paths may update it before the next join, leading to multiple reaching definitions almost throughout the traversal.

##### **Group {D2, D5, D12, D15} — Secondary Index/Counter**

- **Blocks:** B3, B8, B9
- **Interpretation:**  
Serves as an inner-loop or coordinate variable, updated under multiple control paths.

##### **Group {D20, D25, D30, D37} — Path-Length / Step Counter**

- **Blocks:** B12–B22 and EXIT
- **Interpretation:**  
Updated along different branches (e.g., *found* vs. *not found*, *break* vs. *continue*).  
By the time the loop head or termination check executes, multiple definitions can reach it.

##### **Group {D29, D36} — Flag / State Variable**

- **Blocks:** B18, B20, B21
- **Interpretation:**  
Represents flags such as “path found” or “path check OK.”  
Set differently along alternate branches, leading to multiple reaching definitions at merge points.



## Takeaway

In BFS-style programs, variables that are **initialised at loop entries** and **updated within conditionals inside the loop** (e.g., *if in bounds and not visited*) tend to have multiple reaching definitions.

This behaviour is precisely what the in[B] sets reflect in the analysis.

## Results & Analysis

The Reaching Definitions analysis revealed multiple interesting insights:

- **Program 1 (Gradebook):**  
Variables such as “m1”, “m2”, and “m3” had multiple reaching definitions across the looping construct, confirming that previous assignments could propagate depending on the control flow path.
- **Program 2 (String Analyser):**  
The definitions for frequency counters (freq[i]) and totals (letters, vowels, etc.) propagated across several blocks. Loops and nested ifs caused overlapping reaching definitions for vowels and consonants, indicating strong variable coupling within loops.
- **Program 3 (Grid BFS):**  
The analysis produced the most complex data flow due to the BFS loop. Variables like “dist” and “vis” were redefined multiple times within nested loops, resulting in numerous overlapping reaching definitions. This validates BFS as an excellent candidate for demonstrating iterative data-flow analysis.

Overall, the dataflow propagation matched theoretical expectations, with definitions stabilising correctly and accurately reflecting live variable states at each point.

## Discussions & Conclusion

This lab provided hands-on experience with **control flow construction**, **dataflow analysis**, and **automated complexity metrics**. Through implementing Reaching Definitions Analysis, I learned how definitions propagate within loops, conditionals, and nested constructs.

The results confirmed that:

- CFGs can be automatically generated and visualised using Graphviz.
- Cyclomatic Complexity effectively quantifies control structure density.
- Reaching Definitions Analysis helps identify which variable definitions may affect any point in the program — a vital concept for compiler optimisation and static analysis tools.

### Challenges faced:

- Handling nested loops with multiple entry/exit points while maintaining CFG consistency.

- Managing the iterative data-flow updates efficiently until convergence.
- Rendering large CFGs cleanly using Graphviz for medium-sized programs.

Overall, this experiment solidified my understanding of **compiler-level dataflow analysis** and its practical use in software testing and optimisation frameworks.

The GitHub repository for this lab can be found [here](#)

## References

- Lecture 7 slides
  - Lab Assignment 7
  - Graphviz Documentation – <https://graphviz.org/>
  - PyGraphviz Documentation – <https://pygraphviz.github.io/>
  - *Data-Flow Analysis (Wikipedia)* – [https://en.wikipedia.org/wiki/Data-flow\\_analysis](https://en.wikipedia.org/wiki/Data-flow_analysis)
  - Research Paper: “Exploring Data Flow Analysis in Modern Software Systems” (IEEE 2023)
  - Provided C programs: prog1\_gradebook.c, prog2\_string\_analyzer.c, prog3\_grid\_bfs.c
-