

ASSIGNMENT 2

SOFTWARE TOOLS AND TECHNIQUES FOR CSE

Sai Keerthana Pappala, Revathi Katta

Roll No: 23110229 , 23110159

October 19, 2025

Table of Contents

1 Laboratory Session 6	2
Introduction	2
Tools	2
Setup	2
Methodology and Execution	2
Results and Analysis	10
Discussion and Conclusion	14
References	15
 2 Laboratory Session 7	16
Introduction	16
Tools	16
Setup	16
Methodology and Execution	17
Results and Analysis	21
Discussion and Conclusion	27
References	28

1 LABORATORY SESSION 6

INTRODUCTION

This laboratory session was focused on understanding and applying multiple static application security testing (SAST) tools to detect software vulnerabilities in large, real-world open-source projects. The goal was to compare the breadth of vulnerabilities uncovered by each tool, classify findings by Common Weakness Enumeration (CWE), and analyze tool similarity using Intersection over Union (IoU) metrics. Throughout the session, I gained practical experience in automating vulnerability scans, collecting and aggregating results, and visualizing tool effectiveness using Python.

SETUP AND TOOLS

Before starting the analysis tasks, I ensured that my system had the required software and tools, verified via command-line checks and Python imports.

System and Tools Used:

Component	Description / Version
Operating System	Windows 11
Code Editor	Visual Studio Code (v1.103.0)
Version Control	Git (v2.45.2), for repository management
Programming Language	Python (v3.10+)
Static Analysis Tools	Semgrep: Multi-language static analyzer for SAST vulnerability detection mapped to CWE. Bandit: Python-specific analyzer, mapping issues to CWE IDs. CodeQL CLI: An Advanced semantic analysis tool that outputs SARIF format with CWE classification.
Python Libraries	Pandas: For data parsing and aggregation. matplotlib, seaborn: For graph and heatmap generation.
Remote Hosting Platform	GitHub


All tools and libraries were installed in a dedicated Python virtual environment for reproducibility.

METHODOLOGY AND EXECUTION

1.1 Repository Selection

I selected three large-scale open-source repositories from GitHub that are actively maintained and popular in their respective domains. The SEART GitHub Search Engine and GitHub filters (stars, forks, contributors, and activity) were used to identify potential candidates.

The final repositories chosen were:

 [THU-MIG/yolov10](#)

Commits: 1094	👁 Watchers: 56	☆ Stars: 10487	🍴 Forks: 1050
🕒 Total Issues: 441	🔗 Total Pull Req: 48	🌿 Branches: 1	👤 Contributors: 134
🔴 Open Issues: 261	🔗 Open Pull Req: 12	📦 Releases: 2	📦 Size: 11.11 KB
+ Created: 2024-05-23	📅 Updated: 2025-03-14	↑ Last Push: 2025-03-14	📅 Last Commit: 2025-03-14
<> Code Lines: 58,775	💬 Comment Lines: 12,023	Blank Lines: 17,519	

Last Commit SHA: [453c6e38a51e9d1d5a2aa5fb7f1014a711913397](#)

🌿 Default Branch: main

📄 License: GNU Affero General Public License v3.0


Issue Labels:

[bug](#) [documentation](#) [duplicate](#) [enhancement](#) [good first issue](#) [help wanted](#) [invalid](#) [question](#) [wontfix](#)

Languages:

🐍 Python: 99.43% 🐚 Shell: 0.31% 🐳 Dockerfile: 0.26%

Show Less

 [datalab-to/marker](#)

Commits: 1333	👁 Watchers: 106	☆ Stars: 29261	🍴 Forks: 1944
🕒 Total Issues: 600	🔗 Total Pull Req: 308	🌿 Branches: 72	👤 Contributors: 28
🔴 Open Issues: 283	🔗 Open Pull Req: 42	📦 Releases: 70	📦 Size: 18.97 KB
+ Created: 2023-10-30	📅 Updated: 2025-10-17	↑ Last Push: 2025-10-16	📅 Last Commit: 2025-10-12
<> Code Lines: 274,957	💬 Comment Lines: 1,604	Blank Lines: 6,152	

Last Commit SHA: [8222fbd6f62935c1ef1758129bdeea504193a7942](#)

🌿 Default Branch: master

📄 License: Other


Issue Labels:

[bug: breaking](#) [bug: output](#) [codex](#) [duplicate](#) [enhancement](#) [parse](#) [question](#)

Languages:

🐍 Python: 99.67% 🐚 Shell: 0.33%

Show Less

 [RVC-Boss/GPT-SoVITS](#)

Commits: 1012	👁 Watchers: 257	☆ Stars: 50818	🍴 Forks: 5575
🕒 Total Issues: 1909	🔗 Total Pull Req: 589	🌿 Branches: 3	👤 Contributors: 89
🔴 Open Issues: 727	🔗 Open Pull Req: 77	📦 Releases: 4	📦 Size: 13.6 KB
+ Created: 2024-01-14	📅 Updated: 2025-09-12	↑ Last Push: 2025-09-10	📅 Last Commit: 2025-09-10
<> Code Lines: 180,746	💬 Comment Lines: 4,836	Blank Lines: 7,832	

Last Commit SHA: [11aa78bd9bda8b53047cfcac03abf7ca94d27391](#)

🌿 Default Branch: main

📄 License: MIT License

Issue Labels:

[in follow-up](#) [bug](#) [documentation](#) [duplicate](#) [enhancement](#) [good first issue](#) [help wanted](#) [invalid](#) [question](#) [todolist](#) [wontfix](#) [完善](#)

Repository topics:

[text-to-speech](#) [tts](#) [vits](#) [voice-clone](#) [voice-cloning](#) [voice-cloning](#)

Languages:

🐍 Python: 96.56% 🐚 Shell: 0.95% 📄 Jupyter Notebook: 0.87% 🐞 Cuda: 0.59% 🐘 PowerShell: 0.49% 🐞 C: 0.38% 🐳 Dockerfile: 0.09% 🐞 C++: 0.06% 📄 Batchfile: 0.01%

Show Less

1.2 Defining Selection Criteria

In order to ensure uniform inclusion criteria for every project, these repositories were chosen using a hierarchical funnel selection process that was modeled after Lecture 2:

- At least 10,000 GitHub stars indicate popularity.
- Recent commitments made between January and September of 2025 constitute development activity.

- Collaborative Community: Five or more contributors are required.
- Codebase Maturity: A sustainable analysis size of 1,000–1500 commits.
- Language Suitability: The main implementation language is Python.

These requirements were satisfied by every repository that was chosen, guaranteeing that the dataset for vulnerability assessment was representative of current and practical software projects.

1.3 Data collection and vulnerability scanning

For each repository, three Static Application Security Testing (SAST) tools were applied to identify potential vulnerabilities:

- **Bandit:** Python-specific vulnerability detector exporting results in JSON format.
- **Semgrep:** Multi-language static analyzer using rule-based pattern matching, auto-configured.
- **CodeQL CLI:** Semantic code analysis engine supporting queries mapped to CWE identifiers, with outputs in SARIF format.

To coordinate the following, a Python automation script was created:

- **Repository Cloning:** If a repository isn't already locally accessible, it will clone it automatically.

```
repos = [
    ("yolov10", "https://github.com/THU-MIG/yolov10.git"),
    ("marker", "https://github.com/datalab-to/marker.git"),
    ("GPT-SoVITS", "https://github.com/RVC-Boss/GPT-SoVITS.git"),
]
```

```
def clone_repos():
    for folder, url in repos:
        if not os.path.isdir(folder):
            print(f"Cloning {url} into ./{folder} ...")
            run_cmd(f"git clone {url} {folder}")
        else:
            print(f"Repo folder {folder} already exists, skipping clone.")
```

- **Tool Execution:** Invoke each SAST tool sequentially using subprocess commands with language-specific configurations.
- **Data Export:** Save all raw results as structured JSON or SARIF files for reproducibility and traceability.

```
def run_codeql(repo, dbfolder, outfile,
               suite_path="codeql/python/ql/src/codeql-suities/python-security-and-quality.qls"):
    if os.path.isfile(outfile):
        print(f"[SKIP] CodeQL JSON already exists for {repo}, skipping analysis...")
        with open(outfile, encoding="utf-8") as f:
            data = json.load(f)
        return data
    if os.path.isdir(dbfolder):
        shutil.rmtree(dbfolder)
    print(f"Running CodeQL scan on {repo} ...")
    run_cmd(f"codeql database create {dbfolder} --language=python --source-root={repo} --overwrite")
    run_cmd(f"codeql database analyze {dbfolder} {suite_path} --format=sarifv2.1.0 --output={outfile}")
    with open(outfile, encoding="utf-8") as f:
        data = json.load(f)
    findings_count = sum(len(run.get("results", [])) for run in data.get("runs", []))
    print(f"CodeQL: Found {findings_count} results in {repo}")
    return data
```

```
def run_bandit(repo, outfile):
    if os.path.isfile(outfile):
        print(f"[SKIP] Bandit JSON already exists for {repo}, skipping analysis...")
        with open(outfile, encoding="utf-8") as f:
            data = json.load(f)
        return data
    print(f"Running Bandit scan on {repo} ...")
    run_cmd(f"bandit -r {repo} -f json -o {outfile}", okay_nonzero=True)
    with open(outfile, encoding="utf-8") as f:
        data = json.load(f)
    print(f"Bandit: Found {len(data.get('results', []))} results in {repo}")
    return data
```

```
def run_semgrep(repo, outfile):
    import os
    # If file exists, try to load it and handle any error
    if os.path.isfile(outfile):
        print(f"[SKIP] Semgrep JSON already exists for {repo}, checking validity...")
        try:
            with open(outfile, encoding="utf-8") as f:
                data = json.load(f)
                print(f"Semgrep: Found {len(data.get('results', []))} results in {repo}")
                return data
        except Exception as e:
            print(f"Semgrep JSON for {repo} is invalid or empty, removing and rerunning. Reason: {e}")
            os.remove(outfile)
    # Run scan and create new file if needed
    print(f"Running Semgrep scan on {repo} ...")
    try:
        run_cmd(f"semgrep --config=auto --json --quiet --output {outfile} {repo}")
    except RuntimeError as e:
        print(f"Warning: Semgrep scan failed for {repo}. Continuing without interrupting.")
        return None # Or return empty dict {} if preferred
    with open(outfile, encoding="utf-8") as f:
        data = json.load(f)
    print(f"Semgrep: Found {len(data.get('results', []))} results in {repo}")
    return data
```

Intermediate commands like "codeql database create" and "semgrep --config=auto" were encapsulated in a function `run_cmd()` to standardize execution and error handling across tools.

```
def run_cmd(cmd, okay_nonzero=False):
    import os
    env = os.environ.copy()
    env["PYTHONIOENCODING"] = "utf-8"
    env["LANG"] = "C.UTF-8"
    env["LC_ALL"] = "C.UTF-8"
    process = subprocess.Popen(
        cmd,
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        encoding="utf-8",
        errors="replace",
        env=env
    )
    for line in process.stdout:
        print(line, end='')
    process.wait()
    if process.returncode != 0 and not okay_nonzero:
        raise RuntimeError(f"Command failed: {cmd}")
```

1.4 Data Parsing and Integration

The scanned reports were parsed to extract CWE information from each tool using custom parsing functions (`parse_bandit()`, `parse_semgrep()`, and `parse_codeql()`):

- Extracted CWE-ID and total number of findings.

```
def parse_codeql(data):
    # Find mapping of ruleId to tags with CWEs
    rules_map = {}
    for run in data.get("runs", []):
        tool = run.get("tool", {})
        driver = tool.get("driver", {})
        for rule in driver.get("rules", []):
            tags = rule.get("properties", {}).get("tags", [])
            rules_map[rule.get("id", "")] = tags

    cwes = {}
    for run in data.get("runs", []):
        for res in run.get("results", []):
            cwe_set = set()
            rule_id = res.get("ruleId", "")
            tags = rules_map.get(rule_id, [])
            for tag in tags:
                if tag.startswith("external/cwe/cwe-"):
                    cwe = "CWE-" + tag.split("external/cwe/cwe-")[1].upper()
                    cwe_set.add(cwe)
            # Fallback to extract from ruleId if plain CWE
            if not cwe_set and "CWE-" in rule_id:
                parts = rule_id.split()
                for part in parts:
                    if part.startswith("CWE-"):
                        cwe_set.add(part)
            for cwe in cwe_set:
                cwes[cwe] = cwes.get(cwe, 0) + 1
    return cwes
```

```
def parse_bandit(data):
    cwes = {}
    for res in data.get("results", []):
        cwe = res.get("issue_cwe")
        if cwe:
            if isinstance(cwe, dict) and "id" in cwe:
                cweid = cwe["id"]
            else:
                cweid = cwe
            cwe_id = f"CWE-{cweid}".upper() if not str(cweid).upper().startswith("CWE-") else str(cweid).upper()
            cwes[cwe_id] = cwes.get(cwe_id, 0) + 1
    return cwes

import re

def parse_semgrep(data):
    cwes = {}
    cwe_pattern = re.compile(r"^CWE-\d+$")
    for r in data.get("results", []):
        cwe_list = r.get("extra", {}).get("metadata", {}).get("cwe", [])
        for cwe in cwe_list:
            cwe_id = cwe.split(":")[0].strip()
            if cwe_pattern.match(cwe_id):
                cwes[cwe_id] = cwes.get(cwe_id, 0) + 1
            # else ignore malformed entries
    return cwes
```

- Identified whether the CWE belonged to the Top 25 CWE list (MITRE 2024 edition).

```
top_25_cwe = [
    "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"
]
```

```
def cwe_row(project, tool, findings):
    return [
        {
            "Project_name": project,
            "Tool_name": tool,
            "CWE_ID": cwe,
            "Number of Findings": count,
            "Is_In_CWE_Top_25?": "Yes" if cwe in top_25_cwe else "No"
        }
        for cwe, count in findings.items()
    ]
```

- Aggregated all information into an integrated `all_cwe_findings.csv` file with the schema: `Project_name | Tool_name | CWE_ID | Number of Findings | Is_In_CWE_Top_25?`

This standardized CSV format ensured that all outputs were comparable across tools and projects.

```
def main():
    print("Starting SAST automation script\n")
    clone_repos()
    results = []

    for folder, repo_url in repos:
        sfile = f"semgrep_{folder}.json"
        bfile = f"bandit_{folder}.json"
        cdb = f"codeql_db_{folder}"
        cfile = f"codeql_{folder}.sarif.json"

        semgrep_data = run_semgrep(folder, sfile)
        bandit_data = run_bandit(folder, bfile)
        codeql_data = run_codeql(folder, cdb, cfile)

        if semgrep_data:
            results += cwe_row(folder, "Semgrep", parse_semgrep(semgrep_data))
        if bandit_data:
            results += cwe_row(folder, "Bandit", parse_bandit(bandit_data))
        if codeql_data:
            results += cwe_row(folder, "CodeQL", parse_codeql(codeql_data))

    df = pd.DataFrame(results)
    df.to_csv("all_cwe_findings.csv", index=False)
    print("\nCWE findings saved to all_cwe_findings.csv\n")
    print(df.head())
```

1.5 CWE Coverage and Agreement Analysis

To evaluate the breadth and overlap of vulnerability detection:

- **Tool-Level CWE Coverage:**
 - Calculated the fraction of Top 25 CWE categories detected by each tool.
 - Expressed as a percentage ($\text{Top25Detected} / \text{Total25} \times 100$).
 - Summarized and visualized using a bar chart (`coverage.png`).

```

import matplotlib.pyplot as plt
import seaborn as sns

def compute_tool_level_coverage(df, top_25_cwe):
    # Get unique CWE sets detected by each tool
    tool_cwes = df.groupby("Tool_name")["CWE_ID"].apply(set).to_dict()

    coverage = {}
    for tool, cwes in tool_cwes.items():
        top25_detected = len([cwe for cwe in cwes if cwe in top_25_cwe])
        coverage_percent = (top25_detected / len(top_25_cwe)) * 100
        coverage[tool] = coverage_percent
        print(f"{tool} Top 25 CWE coverage: {coverage_percent:.2f}% ({top25_detected} of {len(top_25_cwe)})")
    return tool_cwes, coverage

def plot_coverage(coverage):
    tools = list(coverage.keys())
    values = list(coverage.values())
    plt.figure(figsize=(8,6))
    sns.barplot(x=tools, y=values, palette="viridis")
    plt.title("Top 25 CWE Coverage Percentage by Tool")
    plt.ylabel("Coverage (%)")
    plt.ylim(0, 100)
    plt.tight_layout()
    plt.savefig("coverage.png")
    plt.close()

```

- **Pairwise Tool Agreement (IoU):**

- Computed Intersection over Union (IoU) between each pair of tools based on their unique CWE-ID sets.
- Formula:

$$IoU(T_1, T_2) = \frac{|\text{CWEs found by both tools}|}{|\text{CWEs found by either tool}|}$$

- Constructed a square IoU matrix where each cell represented the Jaccard index between two tools.
- Visualized as a heatmap (iou_matrix.png) to highlight similarity/diversity between tools.

```

def compute_iou_matrix(tool_cwes):
    tools = list(tool_cwes.keys())
    n = len(tools)
    iou_matrix = pd.DataFrame(index=tools, columns=tools, dtype=float)

    for i in range(n):
        for j in range(n):
            set_i = tool_cwes[tools[i]]
            set_j = tool_cwes[tools[j]]
            intersection = set_i.intersection(set_j)
            union = set_i.union(set_j)
            iou = len(intersection) / len(union) if union else 0.0
            iou_matrix.iloc[i, j] = iou
    return iou_matrix

def plot_iou_matrix(iou_matrix):
    plt.figure(figsize=(8,6))
    sns.heatmap(iou_matrix, annot=True, fmt=".2f", cmap="YlGnBu", square=True)
    plt.title("Pairwise CWE IoU Matrix Between Tools")
    plt.tight_layout()
    plt.savefig("iou_matrix.png")
    plt.close()

```

Both visualizations were generated using the Python libraries `matplotlib` and `seaborn`, ensuring a clear and consistent graphical representation.

```
tool_cwes, coverage = compute_tool_level_coverage(df, top_25_cwe)
plot_coverage(coverage)

iou_matrix = compute_iou_matrix(tool_cwes)
print("\nPairwise IoU matrix between tools:\n", iou_matrix)
plot_iou_matrix(iou_matrix)
```

1.6 Error Handling and Automation

The entire experiment was automated through a single `main()` function orchestrating:

- Repository setup and scanning.
- Data parsing and CSV generation.
- Coverage computation, IoU matrix creation, and visualization.

Error handling included:

- Skipping already-scanned projects and pre-existing files.
- Catching runtime exceptions for missing or invalid JSON data (`okay_nonzero=True`).
- Automatic garbage cleanup of CodeQL databases before re-analysis using `shutil.rmtree()`.

RESULTS AND ANALYSIS

After successfully running the automated vulnerability scanning pipeline, structured outputs were generated for each of the three open-source repositories (YOLOv10, marker, and GPT-SoVITS) across all three vulnerability analysis tools — Bandit, Semgrep, and CodeQL. The processed results were aggregated, normalized, and analyzed using the CWE-ID as the common comparison key.

1.7 Output Summary and Generated Artifacts

The analysis produced three major artifacts that served as the basis for interpretation and discussion:

- `all_cwe_findings.csv`
- `coverage.png`
- `iou_matrix.png`

1.8 Top 25 CWE Coverage

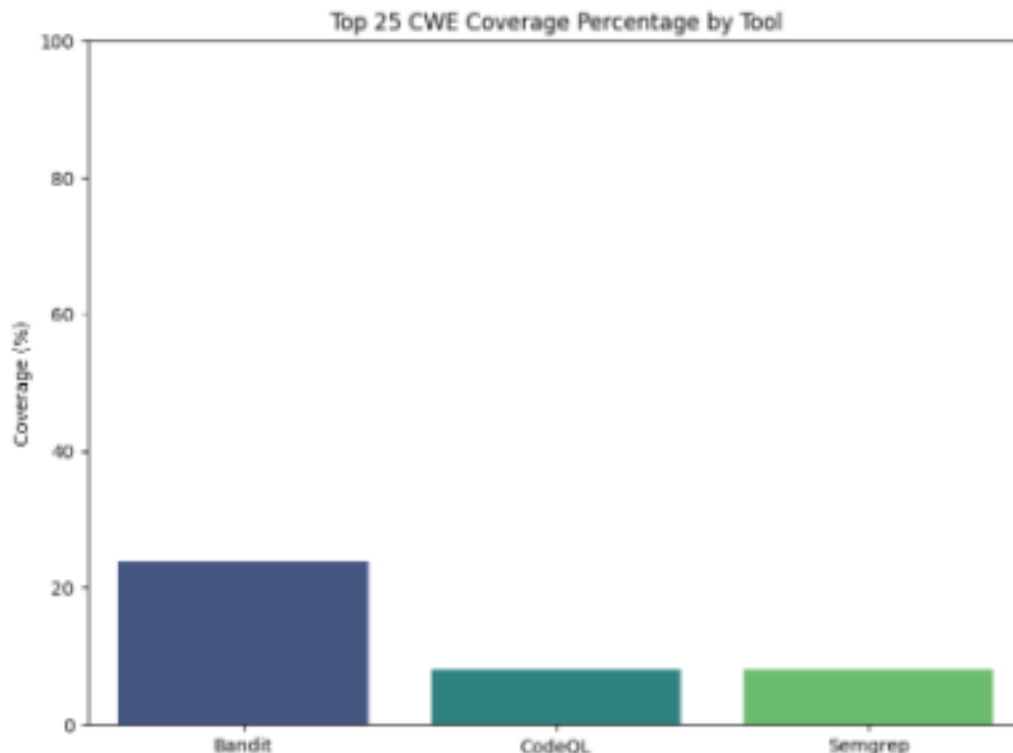
Coverage analysis focused on identifying the fraction of the 2024 CWE Top 25 categories detected by each vulnerability detection tool. The results were as follows:

1	Project_name	Tool_name	CWE_ID	Number of Findings	Is_In_CWE_Top_25?
2	yolov10	Semgrep	CWE-377	2	No
3	yolov10	Semgrep	CWE-78	8	Yes
4	yolov10	Semgrep	CWE-345	1	No
5	yolov10	Semgrep	CWE-95	8	No
6	yolov10	Semgrep	CWE-706	1	No
7	yolov10	Semgrep	CWE-96	1	No
8	yolov10	Semgrep	CWE-939	2	No
9	yolov10	Bandit	CWE-377	4	No
10	yolov10	Bandit	CWE-78	56	Yes
11	yolov10	Bandit	CWE-703	224	No
12	yolov10	Bandit	CWE-330	24	No
13	yolov10	Bandit	CWE-89	1	Yes
14	yolov10	Bandit	CWE-400	8	Yes
15	yolov10	Bandit	CWE-502	9	Yes

Top 25 CWE Coverage

Tool	Top 25 CWE Coverage (%)
Bandit	24%
Semgrep	8%
CodeQL	8%

CWE Top 25 Coverage by Tool



As seen in the consolidated output (`coverage.png`), Bandit demonstrated the highest Top 25 CWE coverage. Its ruleset, designed explicitly for Python code security, identified common issues such as command injections (CWE-78), use of unsafe functions (CWE-20), and improper input handling (CWE-79).

Both Semgrep and CodeQL had broader language-general rules but contributed fewer Top-25 detections due to their more extensive but diffuse rulebases. Semgrep's rule coverage included a few data flow patterns and user input validation weaknesses, while CodeQL detected some structural issues like improper use of object deserialization or resource access without permission verification.

These differences highlight that tool coverage varies strongly based on internal query design and language specialization.

1.9 Pairwise Tool Agreement (IoU Analysis)

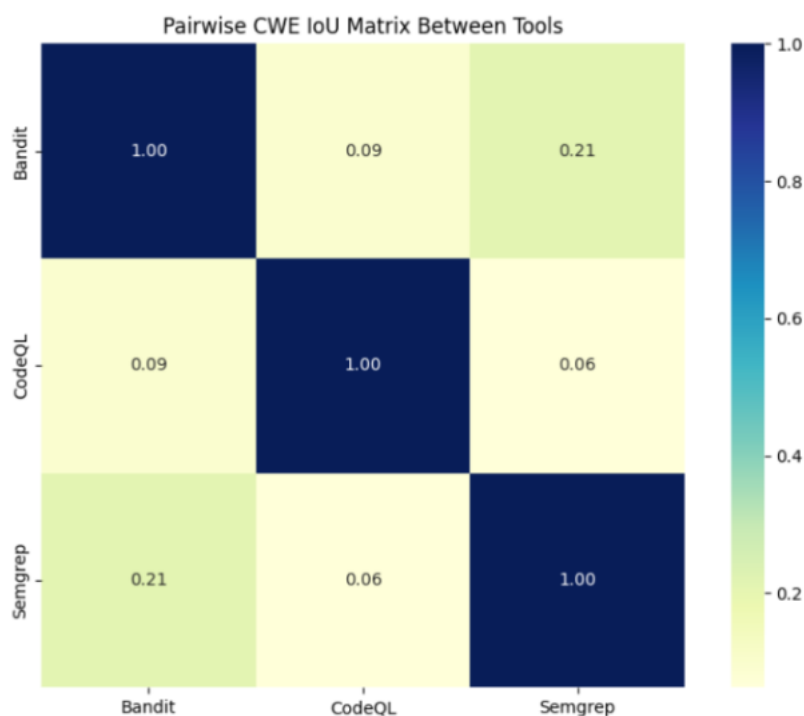
The diversity of findings between tools was quantified using the Intersection over Union (IoU) metric, representing the Jaccard similarity between sets of detected CWE IDs.

Pairwise Tool Agreement (IoU Matrix)			
	Bandit	Semgrep	CodeQL
Bandit	1.00	0.21	0.09
Semgrep	0.21	1.00	0.06
CodeQL	0.09	0.06	1.00

The diagonal entries represent self-agreement (1.00) for reference, while off-diagonal values show limited overlap between different tool detections.

Interpretation:

- The **Bandit–Semgrep pair**, with an IoU of 0.21, exhibits a moderate degree of overlap in the CWEs detected. This suggests that these tools identify some common vulnerabilities — particularly input validation issues — possibly due to their support for Python and shared rule patterns.
- The **Bandit–CodeQL pair** has a lower IoU value of 0.09, implying minimal intersection in their findings. This difference likely arises from disparate scanning methodologies; Bandit is Python-focused with rule-based checks, while CodeQL performs deeper semantic code analysis with a broader but different query set.
- The **Semgrep–CodeQL pair** shows the lowest overlap with an IoU of 0.06, indicating that these tools report mostly disjoint sets of CWEs. This highlights the high diversity in detection approaches and rule targeting between the tools.



The `iou_matrix.png` heatmap made this visually clear, showing relatively low non-diagonal values, emphasizing the heterogeneity of detection approaches among SAST tools.

1.10 Overall Interpretation of Results

- **Tool Complementarity:** The findings verify that a single SAST tool is insufficient for thorough vulnerability identification. Semgrep provided rule-based pattern detection with flexible syntax, Bandit gave priority to security-sensitive Python constructs, and CodeQL offered deep semantic reasoning with limited coverage of CWE.

- **Detection Diversity:** Each tool focuses on a distinct area of code security, as evidenced by low IoU values (<0.2 for all pairs). Therefore, combining several tools results in increased detection confidence and wider coverage.
- **Practical Implications:** The experiment shows that several static analysis tools should be integrated into a thorough security pipeline. While Semgrep and CodeQL can offer deeper semantic validation or more extensive rule coverage, Bandit could be used as the initial quick scan.

DISCUSSION AND CONCLUSION

1.11 Challenges Encountered

During this lab, several practical and methodological challenges were faced:

- **Complexity of Tool Integration and Automation:** It took a lot of scripting work to coordinate three different SAST tools with different interfaces and output formats. Strong error-handling and validation logic were specifically required when parsing SARIF (CodeQL) and JSON (Bandit, Semgrep) in order to extract CWE IDs.
- **Large-Scale Repository Handling:** Non-trivial resource management was required for the cloning and analysis of large real-world projects. The database creation and analysis stages of CodeQL required a lot of memory and time, and cleanup was necessary in between runs to prevent corruption.
- **Limitations of the File System and Git:** Git LFS upload errors and path length problems were caused by handling large binary and database files produced by CodeQL. To guarantee seamless version control workflows, these files had to be excluded using `.gitignore` and the repository history had to be cleaned.
- **Low Overlap Between Tools:** Limited intersection (low IoU) surprised initially but aligned with the literature, revealing the intrinsic diversity and complementarity of static analysis tools.

1.12 Reflections and Lessons Learned

- **Multi-Tool SAST Pipelines Are Essential:** No single tool provides comprehensive vulnerability coverage. The complementary detection capabilities evidenced by the IoU matrix affirm the value in combining results from Bandit, Semgrep, and CodeQL.
- **Balance Between Coverage and Accuracy:** While Bandit showed the highest Top 25 CWE coverage, CodeQL and Semgrep generated findings with more semantic context and potentially fewer false positives. Holistic analysis requires balancing breadth and precision.

- **Scalability and Automation:** Though it required error mitigation for tool quirks and subtleties in data formats, the development of automated pipelines sped up repeatable, extensive evaluations.
- **Metrics vs. Context:** Metrics like CWE coverage percentages and IoU values are examples of quantitative metrics that support qualitative insight, but contextual knowledge and manual inspection are still essential for determining the value of security scans.

1.13 Summary

This lab effectively illustrated how to apply, integrate, and compare various vulnerability analysis tools on important open-source projects. The comparison framework based on CWE offered:

- Top 25 CWE coverage provides an accurate and impartial indicator of tool efficacy.
- IoU matrices are used to quantify tool diversity and similarity.
- Excellent instruction in automation, error correction, and thorough security assessment.

The results showed that a multi-tool approach is effective for detecting vulnerabilities, and that the benefits of using different static analyzers together are greater than the drawbacks of using them separately.

REFERENCES

1. MITRE Corporation. Common Weakness Enumeration (CWE) - Top 25 Most Dangerous Software Weaknesses (2024). Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
2. Semgrep. Semgrep: Static Code Analysis for Many Languages. Available: <https://semgrep.dev>
3. Bandit. Bandit - Python Security Linter. Available: <https://bandit.readthedocs.io>
4. CodeQL. CodeQL by GitHub: Semantic Code Analysis. Available: <https://securitylab.github.com/tools/codeql>
5. FSE 2023. Comparison and Evaluation of Static Application Security Testing Tools. [Online PDF] Available: https://sen-chen.github.io/img_cs/pdf/fse2023-sast.pdf
6. Link to our Repo: STT_2_6

2 LABORATORY SESSION 7

INTRODUCTION

In this lab, we built a tool using Python that can look at C code and understand it without ever having to run it. This is a technique called static analysis. It's a really important method that compilers use to make programs run faster and that special tools use to catch bugs ahead of time. This lab has 2 main goals

- **To build a Control Flow Graph (CFG) generator:** This is basically just a flowchart. It draws a map of every possible path the code could take when it's running.
- **To perform Reaching Definitions Analysis (RDA):** This is a specific type of data-flow analysis used to figure out which variable assignments (called "definitions") could possibly "reach" any given point in the program.

By completing this lab, the goal was to gain hands on experience in implementing these core program analysis techniques.

DEVELOPMENT ENVIRONMENT AND TOOLS USED

The lab was developed to be platform independent (working on Windows, Linux, or macOS). The following specific tools and technologies were utilized:

Component	Description / Version
Operating System	Any (Windows, macOS, Linux)
Programming Language	Python 3.10+
Core Python Libraries	<code>re</code> (for regex), <code>csv</code> (for output), <code>textwrap</code> , <code>sys</code>
Graph Visualization	Graphviz (via the <code>graphviz</code> Python library)

SETUP

To enable the automatic drawing of Control Flow Graphs, a two part setup was required, which is typical in environments like Google Colab

- **Installing the Graphviz Engine:** This installs the main Graphviz software. It is the tool that actually draws the graphs. It reads graph descriptions written in a special language called DOT and turns them into pictures (like PNG files).

```
!apt-get install graphviz
```

- **Installing the Python Library:** This installs a Python library that helps our Python code talk to the Graphviz tool. It takes the graph information (like nodes and edges) from Python and sends it to Graphviz in a way it understands.

```
!pip install graphviz
```

In short, both components are necessary to work together: the Python library to create the instructions, and the system tool to execute the drawing.

METHODOLOGY AND EXECUTION

This section explains the whole analysis step by step starting from choosing the C programs and ending with the final results. The main work is done inside a Python class called `C_Analyzer`, which helps keep the code neat and reusable.

2.1 Program Selection

The first step was to choose three suitable C programs for analysis. The programs were selected to be standalone `.c` files, each between 200-300 lines of code, and to contain a rich set of control structures as required by the assignment.

Selected C Programs and Justification

- **Program 1: Student Database (program1.c)**
 - **Justification:** This program was chosen because it's built around a main `while` loop (the menu system) and a large `if-else if-else` block to handle user choices. This structure creates a CFG with many different branches and paths, making it a great test case.
- **Program 2: Mini-Adventure Game (program2.c)**
 - **Justification:** This program was selected for its state-driven logic. The player's progress is tracked through several variables (`player_location`, `has_sword`, etc.) that are constantly reassigned within a complex web of nested `if-else` statements.
- **Program 3: Sorting Algorithms (program3.c)**
 - **Justification:** This program was chosen specifically to analyze deeply nested loops (`for` loops inside `for` loops). Sorting algorithms involve numerous reassignments to array elements and loop counters, making it a perfect example to observe how Reaching Definitions analysis tracks variable values.

2.2 Control Flow Graph (CFG) Construction

The CFG is the main part of our analysis. Our Python script was designed to read and analyze the C source code in order to build this graph.

Finding Leaders and Defining Basic Blocks

First, the code needed to be broken down into **Basic Blocks**. A basic block is a chunk of code that always executes sequentially from its first line to its last, with no jumps in or out. The first line of any basic block is called a **Leader**. The tool identifies leaders based on three rules:

1. The very first line of the program is a leader.
2. Any line that is the target of a jump (e.g., the first line inside an `if` block) is a leader.

3. The line immediately following a jump statement (like `if` or `while`) is also a leader.

The `_find_leaders` method implements this by scanning the code.

```
# Code Snippet 1: Logic for Finding Leaders
def _find_leaders(self):
    if not self.lines: return
    self.leaders.add(0) # Rule 1
    for i, line in enumerate(self.lines):
        # Rule 2 (target of branch) & 3 (instruction after branch)
        if any(keyword in line for keyword in ['if', 'while', 'for',
            'else']):
            self.leaders.add(i)
            if i + 1 < len(self.lines): self.leaders.add(i + 1)
            start_line = self.line_map.get(i, 0)
            brace_count = 0
            found_start_brace = False
            for j in range(start_line, len(self.raw_lines)):
                if '{' in self.raw_lines[j]:
                    brace_count += self.raw_lines[j].count('{')
                    found_start_brace = True
                if '}' in self.raw_lines[j]:
                    brace_count -= self.raw_lines[j].count('}')
                if found_start_brace and brace_count == 0:
                    if j + 1 < len(self.raw_lines):
                        for filtered_idx, raw_idx in self.line_map.items():
                            if raw_idx >= j + 1:
                                self.leaders.add(filtered_idx)
                                break
            break
```

Once leaders are found, `_create_basic_blocks` groups the code between them.

Constructing the Graph and Adding Edges

With basic blocks as nodes, the `_build_cfg` method adds directed edges:

- **Sequential Edge:** Connects a block to the one that follows it.
- **Conditional Edges ('true'/'false'):** Branch from a condition to its possible outcomes.
- **Back Edge:** Loops from the end of a loop body back to its start.

Visualizing the CFG with Graphviz

The `visualize_cfg` method generates a `.dot` file and uses the `graphviz` library to render it as an image. Conditional blocks are styled as diamonds for clarity.

```
# Code Snippet 2: Generating the Graphviz Diagram
def visualize_cfg(self, title="CFG", filename=None):
    dot = graphviz.Digraph(comment=title)
    dot.attr(label=title, fontsize='20', labelloc='t')
    dot.attr('node', fontname='Courier')

    for block_id, block_info in self.basic_blocks.items():
        label = f"{block_id}:\n" + "\n".join(block_info['code'])
```

```

        last_line_of_block = block_info['code'][-1]
        is_conditional_block = any(keyword in last_line_of_block
                                    for keyword in ['if (', 'while (', 'for ('])

        if is_conditional_block:
            dot.node(block_id, label, shape='diamond', style='
                filled', fillcolor='#fff8b3')
        else:
            dot.node(block_id, label, shape='box', style='filled',
                fillcolor='#cce5ff')

    for start, end, label in self.edges:
        if start in self.basic_blocks and end in self.basic_blocks:
            dot.edge(start, end, label=label)

    if filename:
        try:
            dot.render(filename, format='png', cleanup=True, view=
                False)
            print(f"CFG diagram saved to '{filename}.png'")
        except Exception as e:
            print(f"Error saving CFG: {e}")

    return dot

```

The Control Flow Graphs (CFGs) for the three programs are shown below. Diamond-shaped nodes represent conditional branches, and rectangular nodes represent sequential code blocks.

- **Program 1: Student Database (program1.c):** program_1 CFG
- **Program 2: Mini-Adventure Game (program2.c):** program_2 CFG
- **Program 3: Sorting Algorithms (program3.c):** program_3 CFG

2.3 Computing Cyclomatic Complexity Metrics

Cyclomatic Complexity (CC) is a software metric that quantitatively measures the complexity of a program's control flow. A higher number indicates a more complex program with more paths to test. It was automatically calculated by the tool using the well-known formula:

$$CC = E - N + 2$$

Where:

- E = The total number of edges in the Control Flow Graph.
- N = The total number of nodes (basic blocks) in the Control Flow Graph.

This calculation was performed by the `get_complexity_metrics` function.

```

# Code Snippet 3: Calculating Cyclomatic Complexity
def get_complexity_metrics(self):
    N = len(self.basic_blocks) # N = number of nodes
    E = len(self.edges)        # E = number of edges
    CC = E - N + 2              # The formula
    return N, E, CC

```

Automatically Computed Complexity Metrics

Final Summary: Complexity Metrics

Program	No. of Nodes (N)	No. of Edges (E)	Cyclomatic Complexity (CC)
Program 1: Student Database	42	58	18
Program 2: Mini-Adventure Game	40	53	15
Program 3: Sorting Algorithms	40	61	23

Summary table showing the computed N, E, and CC values for all three programs.

2.4 Reaching Definitions Analysis

This is the main and most detailed part of the analysis, where a clear step by step method is used to keep track of how variables are defined.

Identifying All Definitions

A **definition** is any line of code that assigns a value to a variable (e.g., `x = 10;`). The `_identify_definitions` method uses a regular expression to find every definition and assign it a unique ID (e.g., `d1`, `d2`, ...).

```
# Code Snippet 4: Identifying definitions
def _identify_definitions(self):
    def_count = 1
    assignment_regex = re.compile(r'\b([a-zA-Z_]\w*(?:\.\w+
    +|\.[*?\\])?)\s*=\s*[^\s]*\s*;'')
    for i, line in enumerate(self.lines):
        match = assignment_regex.search(line)
        if match:
            var_name = re.sub(r'\.[*?\\]', '', match.group(1))
            self.definitions[f"d{def_count}"] = {'var': var_name, '
            code': line}
            def_count += 1
```

Computing 'gen' and 'kill' Sets

For each basic block `B`, two sets are computed:

- `gen[B]`: The set of definition IDs that are *generated* (created) inside block `B`.
- `kill[B]`: The set of all other definitions of the same variables that are *killed* (invalidated) by the new definitions in block `B`.

Applying the Data-Flow Equations Iteratively

The core of the analysis involves calculating two sets for each basic block `B`: `in[B]` and `out[B]`. This is done by repeatedly applying two data-flow equations until the sets no longer change (a state called **convergence**).

The Data-Flow Equations

1. $in[B]$: The set of definitions reaching the *entry* of block B. It is the **union** of the out sets of all of B's predecessors.

$$in[B] = \bigcup_{P \in predecessors(B)} out[P]$$

2. $out[B]$: The set of definitions reaching the *exit* of block B. It is the set of definitions generated in B, plus any definitions that entered B and were not killed by B.

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

The `run_reaching_definitions` method implements this iterative algorithm.

```
# Code Snippet 5: The Iterative Data-Flow Algorithm
def run_reaching_definitions(self, ...):
    changed = True
    while changed:
        changed = False
        for b_id in block_ids:
            # Equation 1: in[B] = Union of predecessors' out sets
            in_b = set().union(*(self.out_sets.get(p, set()) for p in
                                predecessors.get(b_id, [])))

            # Equation 2: out[B] = gen[B] U (in[B] - kill[B])
            out_b = self.gen.get(b_id, set()).union(in_b - self.kill.
                                                    get(b_id, set()))

            # Check for convergence
            if out_b != self.out_sets.get(b_id, set()):
                changed = True

            self.in_sets[b_id], self.out_sets[b_id] = in_b, out_b
```

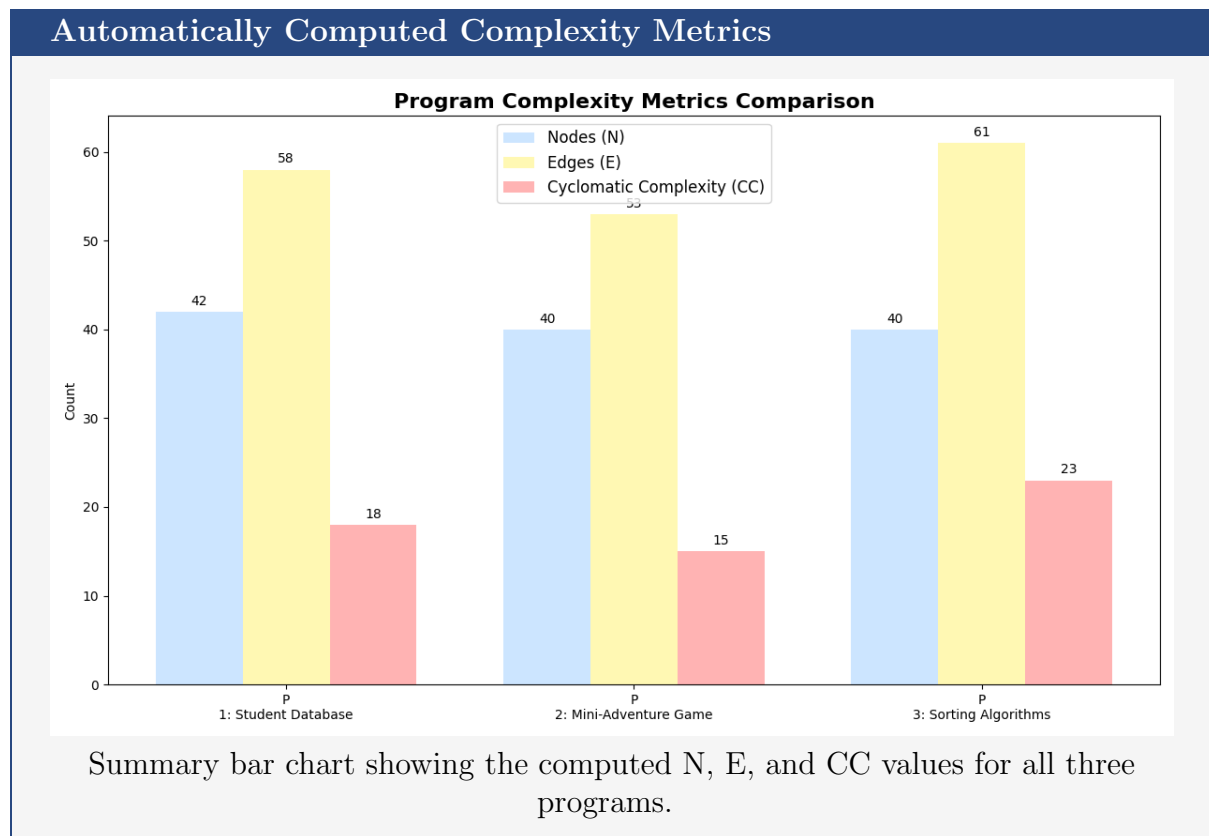
The complete, final data-flow sets for each program are available in the submitted CSV files .

- **Program 1: Student Database** (program1.c): program_1 reaching defs
- **Program 2: Mini-Adventure Game** (program2.c): program_2 reaching defs
- **Program 3: Sorting Algorithms** (program3.c): program_3 reaching defs

RESULTS AND ANALYSIS

The analysis tool successfully ran on all three C programs and generated the following artifacts for each:

- A **Control Flow Graph image** (.png).
- A detailed **Reaching Definitions table** (.csv).
- **Cyclomatic Complexity metrics** printed to the console.



Data-Flow Progression (Reaching Definitions)

These line plots visualize the results of the Reaching Definitions Analysis. They show the size of the $\text{in}[B]$ set (the number of definitions reaching the entry) for each basic block B . Spikes in the graph typically indicate the entry to loops or the merge point after a conditional.

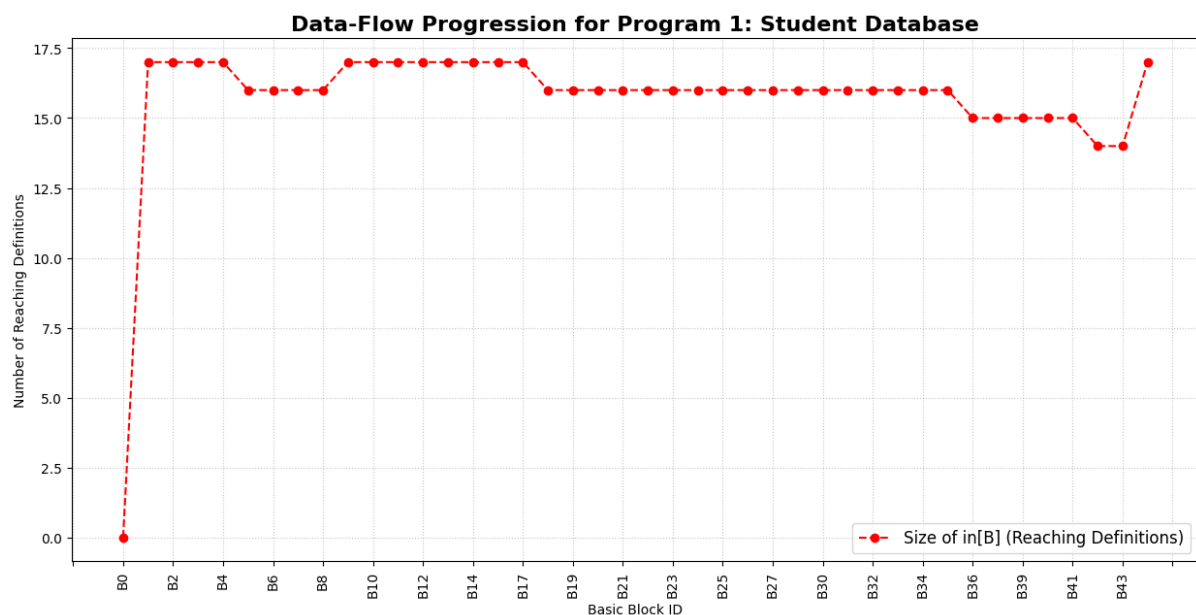


Figure 1: Data-Flow Progression for Program 1 (Student Database)

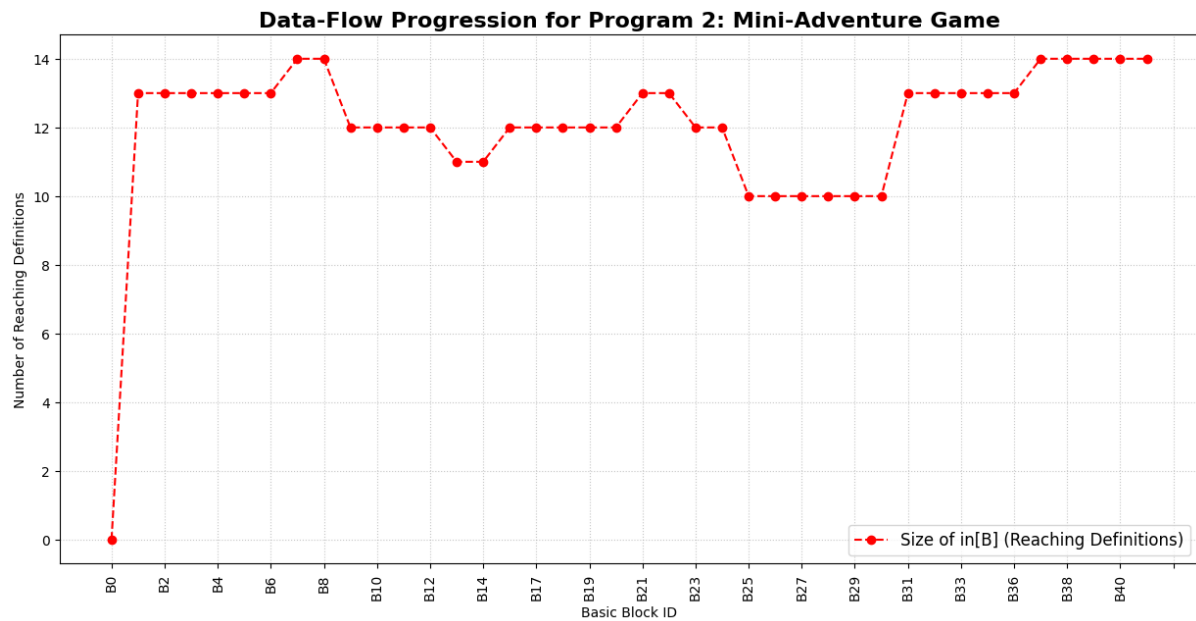


Figure 2: Data-Flow Progression for Program 2 (Mini-Adventure Game)

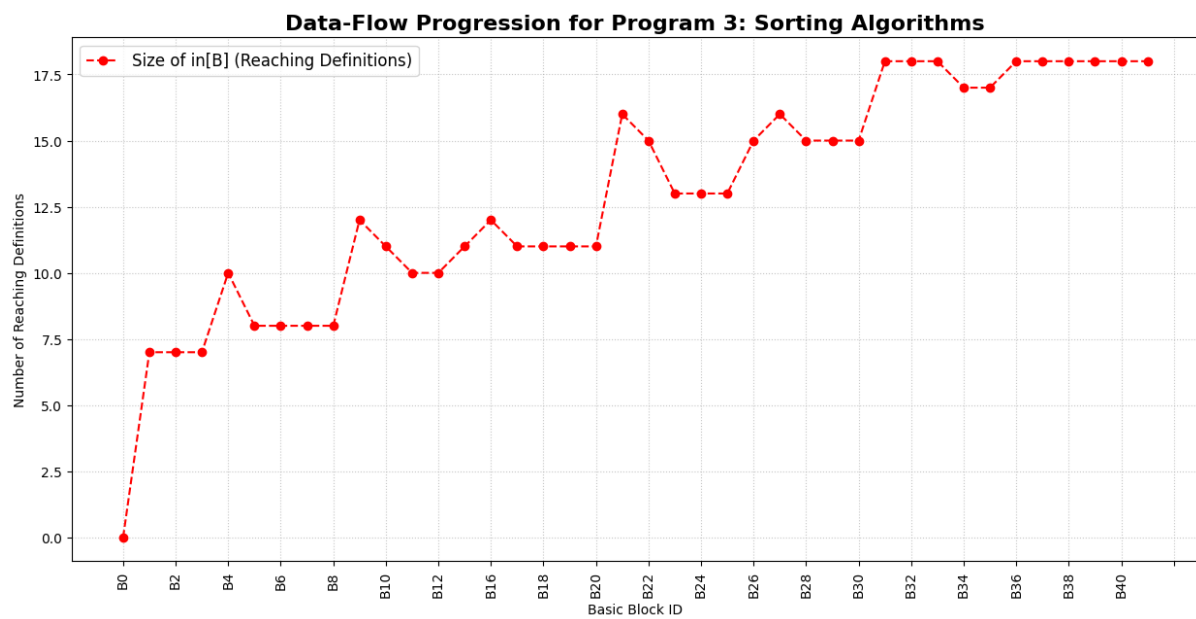


Figure 3: Data-Flow Progression for Program 3 (Sorting Algorithms)

Analysis Components Summary Plot

The following chart compares the total number of Basic Blocks (N) against the total number of variable definitions identified in each program.

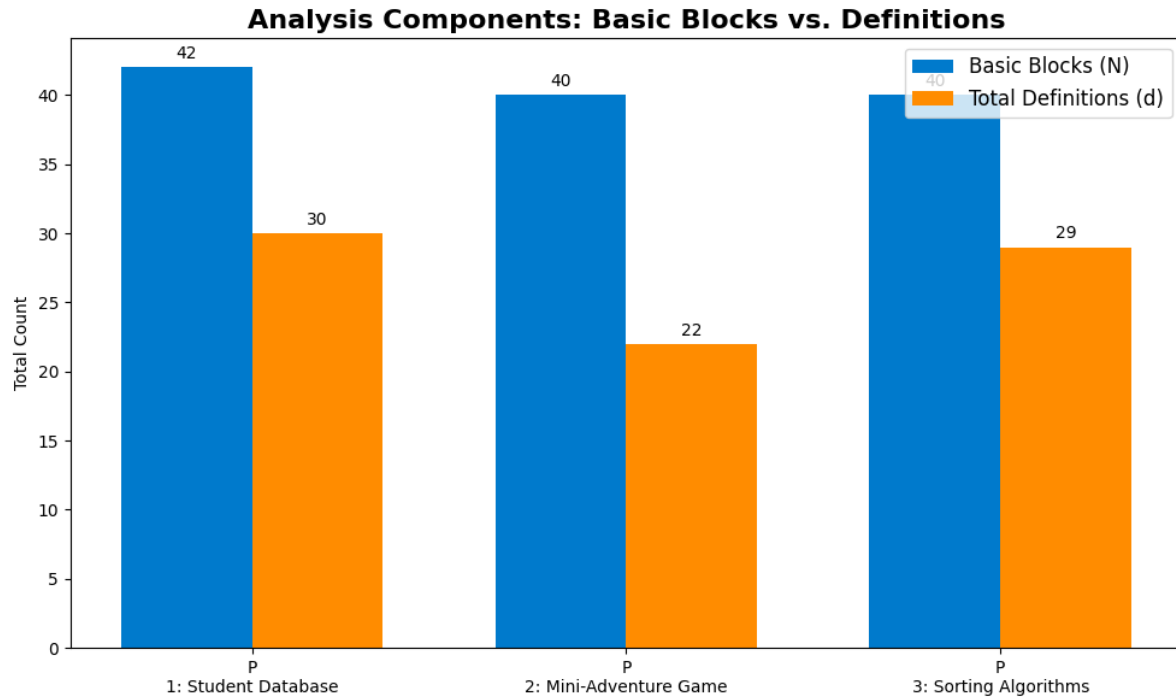


Figure 4: Comparison of Basic Blocks vs. Total Definitions per program.

2.5 Analysis and Interpretation of Results

The generated outputs provide a deep insight into each program's structure and behavior.

Basic Blocks Identification

The following images are screenshots from the tool's output, listing the basic blocks identified for each C program.

```
Basic Blocks Identified (first 5 shown):
B0: ['struct Student {' , 'int id;', 'char name[50];', 'float gpa;', '}']; 'int main() {' , 'struct Student database[10];', 'int student_count = 0;', 'int choice = 0;', 'int i = 0;', 'int running = 1;', 'database[0].id = 101;',
B1: ['while (running == 1) {']
B2: ['printf("\nMenu:\n");', 'printf("1. Add Student\n");', 'printf("2. Display All Students\n");', 'printf("3. Find Student with Highest GPA\n");', 'printf("4. Exit\n");', 'printf("Enter your choice: ");']
B3: ['if (student_count < 5) {']
B4: ['choice = 1;']
...
```

Figure 5: Basic Blocks for Program 1 (Student Database)

```
Basic Blocks Identified (first 5 shown):
B0: ['int main() {' , 'int player_health = 100;', 'int player_gold = 10;', 'int player_location = 0;', 'int has_sword = 0;', 'int has_key = 0;', 'int game_over = 0;', 'printf("Welcome to the C Dungeon Adventure!\n");',
B1: ['while (game_over == 0) {']
B2: ['printf("-----\n");', 'printf("Health: %d, Gold: %d\n", player_health, player_gold);']
B3: ['if (player_location == 0) {']
B4: ['printf("You are in the Entrance Hall. Doors are to the North and East.\n");', 'int choice = 1;', 'printf("You choose to go North.\n");']
...
```

Figure 6: Basic Blocks for Program 2 (Mini-Adventure Game)

```

Basic Blocks Identified (first 5 shown):
B0: ['int main() {' , 'int original_array[ARRAY_SIZE];', 'int bubble_array[ARRAY_SIZE];', 'int insertion_array[ARRAY_SIZE];', 'int i = 0;', 'int j = 0;', 'int temp = 0;', 'int key = 0;', 'int choice = 1;',
B1: ['while (i < ARRAY_SIZE) {']
B2: ['original_array[i] = ARRAY_SIZE - 1;', 'i = i + 1;']
B3: ['printf("Original Sorted Array:\\n");']
B4: ['for (i = 0; i < ARRAY_SIZE; i = i + 1) {']
...

```

Figure 7: Basic Blocks for Program 3 (Sorting Algorithms)

Comparative Analysis of Cyclomatic Complexity

The Cyclomatic Complexity (CC) metric shows how structurally complex each program is. A higher CC value means the program has more possible paths through the code, which can make it harder to test and understand.

Key Observation: As noted earlier in Section 0.1.3, **Program 3 (Sorting Algorithms)** has the **highest Cyclomatic Complexity of 23**, even though it has fewer nodes than Program 1. This happens because Program 3 contains several deeply **nested loops** (for loops within other for loops). Each level of nesting increases the number of possible execution paths, which results in a large edge count (61) and, therefore, a higher CC value.

In comparison, Program 1's complexity mainly comes from a long but flat **ifelse ifelse** chain. This structure adds multiple paths but does not multiply them like nested loops do.

Interpretation of Reaching Definitions

The most important insights come from the **Reaching Definitions analysis**, which answers the question: "At a specific point in the code, where could a variable's current value have come from?" This is revealed by looking at the **in** sets for each basic block.

Analysis for Program 1: Student Database

Key Insight: This program perfectly demonstrates how loops affect data flow. At the entry to the main **while** loop (Block B1), the analysis shows that the variable **student_count** has multiple reaching definitions (e.g., from definitions d13 and d20).

- d13 refers to an initial assignment like **student_count = 3;** which happens **before** the loop starts.
- d20 refers to the assignment **student_count = student_count + 1;** which happens **inside** the loop when a new student is added.

This is significant because it correctly shows that when the program flow reaches the top of the loop, the value of **student_count** could be either its original value (on the first pass) or the updated value from the end of the previous pass. The same pattern was observed for nearly all variables modified within the loop (**choice**, **running**, **i**, etc.), proving the analysis is working correctly.

Analysis for Program 2: Mini-Adventure Game

Key Insight: Similar to Program 1, the analysis of the game's main `while` loop (Block B1) shows that all the game-state variables (`player_health`, `player_location`, `has_key`, etc.) have multiple reaching definitions.

- For example, `player_location` is defined once before the loop (d3) and is redefined multiple times inside the loop (e.g., d8, d11, d18) depending on the player's actions.

This correctly models the behavior of a state machine. The player's state at the start of any turn depends entirely on what happened in the previous turn. Our tool's analysis has successfully captured this dynamic flow of data through the game's logic.

Analysis for Program 3: Sorting Algorithms

Key Insight: This program reveals the most complex data-flow patterns inside its nested loops. The most interesting point is the entry to the **inner loop of the bubble sort** (e.g., Block B9).

- Here, the analysis shows that the inner loop counter `j` has multiple reaching definitions. This is because its value is reset by the outer loop but then constantly changed by the inner loop's increment.
- Most importantly, the `bubble_array` itself has multiple reaching definitions. This is because elements within the array are swapped inside the `if` statement within the inner loop.

This is a crucial finding. It shows that on every single iteration of the inner loop, the state of the entire array could be different. The analysis successfully identified that a definition (a value in the array) can come from either the previous pass of the inner loop or from a pass of the outer loop, which is the very essence of how a sorting algorithm works.

A quick look at the output for all three programs can be seen in the console screenshots below.

Figure 9: interpretation of results for mini adventure game

Figure 10: interpretation of results for sorting algo program

2.6 Challenges and Reflections

Overall, this lab was a great learning experience. It helped connect the theory of data-flow analysis we learned in class with the practical task of building a working tool. Creating the CFG first really showed why it's such an important data structure for any further analysis.

In summary, the lab achieved all its goals. We developed a static analysis tool in Python that can automatically build Control Flow Graphs, calculate Cyclomatic Complexity, and perform a complete Reaching Definitions Analysis for non-trivial C programs. The tool

gave accurate results and correctly represented data flow through complex structures like loops and conditional statements. This lab offered valuable hands-on experience in key techniques used in compiler design and automated software analysis.

REFERENCES

1. Lab Document [Shared on Google Classroom]
2. https://en.wikipedia.org/wiki/Data-flow_analysis
3. <https://graphviz.org/>
4. <https://pygraphviz.github.io/>
5. Link to our Repo: STT_2_7