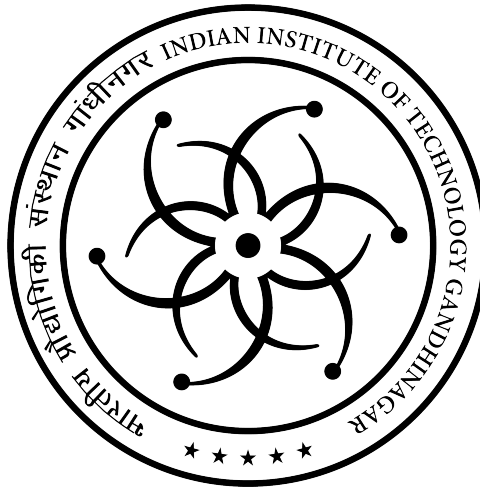


Lab Assignment - 6

12 October 2025



CS 202

Software Tools and Techniques for CSE

INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
Palaj, Gandhinagar - 382355

Evaluation of Vulnerability Analysis Tools using CWE-based Comparison

Submitted by

Gaurav Budhwani

22110085

Abstract

This report explores the process of evaluating and comparing three Static Application Security Testing (SAST) tools on large-scale, real-world Python repositories. The objective is to assess tool effectiveness and diversity by analyzing their findings based on the Common Weakness Enumeration (CWE) framework. The evaluation employed metrics such as CWE Top 25 coverage and pairwise Intersection over Union (IoU) to measure tool similarity. In this report, I have tried to perform the complete methodology, from environment setup and tool selection to the execution of automated scans, analysis of results, and a detailed discussion of the significant challenges encountered and the iterative process of debugging and adapting the experimental setup.

Contents

1	Introduction, Setup, and Tools	3
1.1	Overview and Objectives	3
1.2	Environment Setup	3
1.3	Selection of Repositories and Tools	3
1.3.1	Repositories	3
1.3.2	SAST Tools	4
2	Methodology and Execution	4
2.1	The Role and Rationale of Automation	5
2.2	Data Processing and Analysis	5
2.3	Execution and Error Handling	6
2.3.1	Initial Setup and Execution	6
2.3.2	A Case Study in Tooling Challenges: The Snyk Saga	6
2.3.3	Final Analysis Script Debugging	7
3	Results and Analysis	7
3.1	Final Outputs	7
3.2	Key Insights and Comparisons	8
4	Discussion and Conclusion	9
4.1	Challenges and Reflections	9
4.2	Insights Gained	9
4.3	Conclusion	9

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

The primary goal of this lab was to gain practical experience with SAST tools by applying them to real-world software projects. The core objectives were:

- To select and set up multiple SAST tools to analyze large-scale open-source projects.
- To automate the process of cloning repositories and running security scans.
- To extract, process, and aggregate scan results based on CWE identifiers.
- To quantitatively compare the tools based on their CWE coverage and pairwise agreement (IoU).
- To interpret the results to understand the strengths, weaknesses, and diversity of the selected tools.

1.2 Environment Setup

The experiment was conducted on a Linux virtual machine (Ubuntu on VirtualBox). To ensure a clean and reproducible environment, a Python virtual environment was used to manage dependencies and isolate the installed tools from the system. The directory structure was organized as follows:

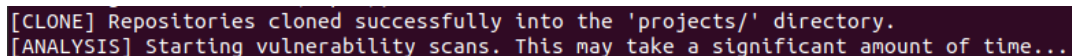
```
/Lab_6/  
|-- projects/          # Cloned Git repositories  
|-- results/           # Raw JSON outputs from scans  
|-- run_analysis.sh    # Automation script  
|-- analyze_results.py  # Python analysis script
```

1.3 Selection of Repositories and Tools

1.3.1 Repositories

Three large-scale Python projects were selected based on criteria ensuring they were representative of real-world applications: high popularity (over 50,000 GitHub stars), active maintenance, and domain diversity.

- **Django:** A high-level web framework (Web Development).
- **Pandas:** A data analysis and manipulation library (Data Science).
- **Scikit-learn:** A machine learning library (Machine Learning).



```
[CLONE] Repositories cloned successfully into the 'projects/' directory.  
[ANALYSIS] Starting vulnerability scans. This may take a significant amount of time...
```

Figure 1: Cloning the selected repository for further analysis.

1.3.2 SAST Tools

The tools were chosen based on their open-source nature, command-line interface (CLI) support for automation, and their ability to report findings with CWE identifiers.

- **Bandit (v1.7.5):** A popular linter designed to find common security issues in Python code.

```
[ANALYSIS] Starting vulnerability scans. This may take a significant amount of time...
--- Analyzing Project: django ---
[ANALYSIS] Running Bandit on django...
[main] INFO     profile include tests: None
[main] INFO     profile exclude tests: None
[main] INFO     cli include tests: None
[main] INFO     cli exclude tests: None
Working... 100% 0:00:27
```

Figure 2: Execution of Bandit on one of the selected projects.

- **Semgrep (v1.18.0):** A fast, powerful, and polyglot static analysis tool with a rich set of community-driven security rules.

```
[ANALYSIS] Running Semgrep on django...

Semgrep CLI

Scanning 6975 files (only git-tracked) with 151 Code rules:

CODE RULES
Scanning 897 files with 151 python rules.

SUPPLY CHAIN RULES
Sign in with `semgrep login` and run
`semgrep ci` to find dependency vulnerabilities and
advanced cross-file findings.

PROGRESS
100% 0:00:07
```

Figure 3: Execution of Semgrep on one of the selected projects.

- **Dlint (v2.13.0) with Flake8:** A security-focused linter that encourages secure coding practices, run via the Flake8 framework.

```
[ANALYSIS] Running Dlint on django...
Dlint analysis for django complete.
```

Figure 4: Execution of Dlint on one of the selected projects.

Initially, Snyk was chosen as a third tool, but it was replaced due to limitations in its free-tier plan that prevented SAST analysis via the CLI. This pivot to Dlint is discussed further in the methodology section.

2 Methodology and Execution

The lab was executed in a systematic, automated fashion. A shell script managed the setup and scanning, while a Python script handled data processing and analysis.

2.1 The Role and Rationale of Automation

Executing nine distinct analysis runs (3 tools x 3 projects) manually would be highly inefficient and prone to human error. To ensure consistency and reproducibility, a shell script (`run_analysis.sh`) was developed to automate the entire workflow. This approach offered several key advantages:

- **Efficiency:** A single command initiated a process that would otherwise take significant time to execute manually.
- **Consistency:** The script ensured that every tool was run with the exact same parameters on every project, eliminating variability.
- **Iteration:** During the debugging phase, particularly when troubleshooting issues with Snyk, the script made it trivial to re-run the entire process. This became invaluable, as manually re-typing commands after each attempted fix would have been tedious and frustrating.

The core logic of the automation script involved iterating through the projects and executing each tool:

```
1 PROJECTS=("django" "pandas" "scikit-learn")
2 TOOLS=("bandit" "semgrep" "dlint")
3
4 for project in "${PROJECTS[@]}; do
5     echo "--- Analyzing Project: $project ---"
6     PROJECT_PATH="projects/$project"
7
8     # Run Bandit
9     bandit -r "$PROJECT_PATH" -f json -o "results/${project}_bandit.json"
10
11    # Run Semgrep
12    semgrep scan --config "p/python" "$PROJECT_PATH" --json -o "results/${project}_semgrep.json"
13
14    # Run Dlint
15    flake8 --select=DUO --format=json "$PROJECT_PATH" > "results/${project}_dlint.json"
16 done
```

Listing 1: Main execution loop from `run_analysis.sh`

2.2 Data Processing and Analysis

Once all scans were complete, the `analyze_results.py` script was executed. This script's primary responsibilities were:

1. **Parsing:** Read and parse the unique JSON output formats of Bandit, Semgrep, and Dlint.
2. **Aggregation:** Extract CWE identifiers from each finding and compile them into a single, standardized CSV file.
3. **Calculation:** Compute the final metrics, including CWE Top 25 coverage and the pairwise IoU matrix.

A parser function was created for each tool. For example, the Dlint parser had to map its specific 'DUO' codes to their corresponding CWE IDs:

```

1 def parse_dlint_results(filepath):
2     """Parses Dlint/Flake8 JSON output and maps DUO codes to CWE IDs."""
3     findings = defaultdict(int)
4
5     dlint_to_cwe_map = { "DU0101": "CWE-22", "DU0106": "CWE-321", ... }
6
7     try:
8         with open(filepath, 'r') as f:
9             data = json.load(f)
10            for filename in data:
11                for error in data[filename]:
12                    duo_code = error.get("code")
13                    if duo_code in dlint_to_cwe_map:
14                        cwe_id = dlint_to_cwe_map[duo_code]
15                        findings[cwe_id] += 1
16    except (json.JSONDecodeError, FileNotFoundError):
17        print(f"Warning: Could not parse or find {filepath}")
18    return findings

```

Listing 2: Dlint parser function from `analyze_results.py`

2.3 Execution and Error Handling

The execution phase was not straightforward and involved several rounds of debugging and problem-solving, which provided valuable real-world experience.

2.3.1 Initial Setup and Execution

The process began by setting up the directory structure and creating the necessary scripts, as shown in the initial terminal session (placeholder ‘61.png’).

```

student@student-VirtualBox:~/Desktop/Lab_6$ mkdir projects results
student@student-VirtualBox:~/Desktop/Lab_6$ touch run_analysis.sh analyze_results.py
student@student-VirtualBox:~/Desktop/Lab_6$ chmod +x run_analysis.sh
student@student-VirtualBox:~/Desktop/Lab_6$ ./run_analysis.sh

```

Figure 5: Initial setup and first run of the analysis script.

2.3.2 A Case Study in Tooling Challenges: The Snyk Saga

The initial plan to use Snyk encountered several setup hurdles, providing a rich learning experience.

- **Environment Dependencies:** The first run failed because the `npm` package manager was not installed. This was resolved with `sudo apt install nodejs npm`.
- **PATH Configuration:** Even after installation, the system returned a `snyk: command not found` error. This classic environment issue was traced to an unconfigured `$PATH` variable, which required manually adding the `npm` global bin directory to the shell’s configuration.
- **Version Incompatibility:** The next hurdle was a version conflict. The system’s default Node.js (v10) was too old for Snyk, which required v12 or higher. This led to the following warning:

```

1     npm WARN notsup Unsupported engine for snyk@1.1300.0: wanted:
2     {"node": ">=12"} (current: {"node": "10.19.0", "npm": "6.14.4"})
3

```

Listing 3: Snyk Node.js version incompatibility error.

This was resolved by upgrading the system's Node.js to a modern version (v20) using NodeSource.

- **Service-Level Limitation:** The final blocker was a non-technical issue. After successfully installing and configuring Snyk, the tool failed with a service-level error:

```
1 FeatureNotSupportedForOrgError: Unsupported action for org ...
2
```

Listing 4: Snyk free-tier plan limitation error.

This error confirmed that Snyk's free plan does not support SAST scanning via its CLI. This was a critical learning moment, leading to the decision to pivot and replace Snyk with Dlint.

2.3.3 Final Analysis Script Debugging

A final, minor bug was encountered when running the Python analysis script. The script failed with the following traceback:

```
1 Traceback (most recent call last):
2   File "analyze_results.py", line 160, in <module>
3     main()
4   File "analyze_results.py", line 101, in main
5     cwe_counts = parsers[tool](filepath)
6   File "analyze_results.py", line 30, in parse_bandit_results
7     cwe_ids = cwe_id_str.split(',')
8 AttributeError: 'int' object has no attribute 'split'
```

Listing 5: Python script traceback.

This occurred because the Bandit tool sometimes reported a CWE ID as a number (e.g., 79) instead of a string ("79"). The fix was a simple one-line change to ensure the CWE ID was always converted to a string before processing:

```
1 # Original problematic line:
2 cwe_ids = cwe_id_str.split(',')
3
4 # Corrected line:
5 cwe_ids = str(cwe_id_str).split(',')
```

Listing 6: The single-line fix in `analyze_results.py`.

3 Results and Analysis

3.1 Final Outputs

After all debugging and the successful pivot to Dlint, the `analyze_results.py` script ran successfully, producing the final aggregated data and metrics. The output from this script (placeholder '66.png') provides the core findings of this lab.

```
(venv) student@student-VirtualBox:~/Desktop/Lab_6$ python3 analyze_results.py

Running Analysis of Tool Results
[OUTPUT] Consolidated data - saved to 'consolidated_results.csv'.

Tool-level CWE Coverage Analysis
Total Unique CWEs Detected per Tool:
- bandit: 17 unique CWEs
- semgrep: 6 unique CWEs
- dlint: 4 unique CWEs

Top 25 CWE Coverage (% of Top 25 found by each tool):
- bandit: Found 8/25 -> 32.00% coverage.
- semgrep: Found 2/25 -> 8.00% coverage.
- dlint: Found 3/25 -> 12.00% coverage.

Pairwise Agreement (IoU) Matrix
IoU (Jaccard Index) = |Intersection| / |Union|
      bandit  semgrep  dlint
bandit   1.000    0.095  0.235
semgrep  0.095    1.000  0.111
dlint    0.235    0.111  1.000
[OUTPUT] IoU matrix saved to 'iou_matrix.csv'.
```

Figure 6: Final output of the Python analysis script, showing all key metrics.

The key results are summarized in the tables below for clarity.

Table 1: Tool-level CWE Coverage Analysis

Tool	Unique CWEs Detected	Top 25 CWE Coverage (%)
Bandit	17	32.00% (8/25)
Semgrep	6	8.00% (2/25)
Dlint	4	12.00% (3/25)

Table 2: Pairwise Agreement (IoU) Matrix

	bandit	semgrep	dlint
bandit	1.000	0.095	0.235
semgrep	0.095	1.000	0.111
dlint	0.235	0.111	1.000

3.2 Key Insights and Comparisons

1. **Bandit is the most comprehensive single tool:** As shown in Table 1, Bandit detected the highest number of unique CWEs (17) and had by far the best coverage of the CWE Top 25 most critical vulnerabilities (32%). As a tool designed specifically for Python, its rules are likely more mature and tailored to the language’s common pitfalls and standard library weaknesses.
2. **All tools are highly diverse:** The IoU matrix (Table 2) is the most insightful result. An IoU score of 1 means perfect overlap, while a score of 0 means no overlap. All pairwise scores are very low (the highest is only 0.235), indicating that the tools are highly diverse and find different kinds of weaknesses. There is very little redundancy between them, which strongly supports the findings of academic research in this area [1].
3. **Bandit and Semgrep maximize coverage:** The IoU matrix answers the lab’s central question: which tool combination is best? The lowest IoU score is between Bandit and

Semgrep (0.095). This means they are the most complementary pair. For every 100 unique vulnerabilities found by the two tools combined, only about 9-10 of them are found by both. This makes them an extremely effective combination for maximizing the breadth of vulnerability detection.

4 Discussion and Conclusion

4.1 Challenges and Reflections

This lab was an excellent simulation of a real-world software engineering task, where the path from planning to results is rarely linear. The most significant challenge was not the analysis itself, but the setup, configuration, and limitations of the tools. The journey from fixing `$PATH` variables to upgrading system-level dependencies (Node.js) and finally hitting a service-plan wall with Snyk was a powerful lesson. It highlighted that in practice, a tool's viability depends just as much on its documentation, ease of installation, and business model as it does on its technical capabilities. The ability to pivot from Snyk to Dlint and modify the analysis scripts accordingly was a crucial part of completing the lab successfully.

4.2 Insights Gained

- **No Silver Bullet:** No single SAST tool can find all vulnerabilities. Bandit was the strongest single performer, but the IoU analysis proved it still misses many issues that other tools can find. This aligns with the principle that security should be layered.
- **Diversity is Strength:** The key to a robust static analysis strategy is to use a combination of diverse, complementary tools. The data clearly shows that running Bandit and Semgrep together provides far better coverage than running either one alone.
- **Automation is Essential:** Scanning three large projects with three tools (nine scans total) would be tedious and error-prone if done manually. The use of automation scripts was critical for ensuring consistency and saving time, especially during iterative debugging cycles.
- **Read the Fine Print:** The experience with Snyk served as a critical reminder that a tool's advertised capabilities must be verified against the constraints of its access plan. Features available in a UI may not be available via an API or CLI, which is a crucial consideration for automated CI/CD or scripting workflows.

4.3 Conclusion

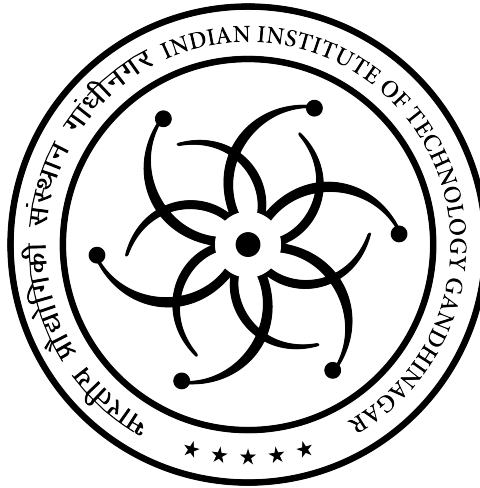
The lab successfully developed a methodology for evaluating and comparing SAST tools on real-world codebases. Through automated scanning and CWE-based analysis, it was determined that among the selected open-source tools, Bandit offered the best individual performance. However, the IoU analysis definitively showed that a combination of complementary tools—specifically Bandit and Semgrep—provides the most comprehensive vulnerability detection coverage. The practical challenges encountered during the setup phase underscored the real-world complexities of integrating and maintaining a security toolchain.

References

- [1] K. Li, S. Chen, L. Fan, et al. (2023). *Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java*. Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23).
- [2] OpenStack Security Project. (2023). *Bandit*. <https://github.com/PyCQA/bandit>
- [3] Semgrep, Inc. (2023). *Semgrep*. <https://semgrep.dev/>
- [4] Dell. (2023). *Dlint*. <https://github.com/dlint-py/dlint>
- [5] Lab Document. *Lab-6*.
<https://drive.google.com/file/d/1kuFDbnKFp2h0ZsSrbkbHoyNt9NNvcMux/view>
- [6] MITRE Corporation. (2024). *2024 CWE Top 25 Most Dangerous Software Weaknesses*.
https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Lab Assignment - 7

15 September 2025



CS 202

Software Tools and Techniques for CSE

INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
Palaj, Gandhinagar - 382355

CFGs Construction & Reaching Definitions Analyzer for C Programs

Submitted by

Gaurav Budhwani

22110085

Abstract

This report builds and explores the design, implementation, and application of the reaching definitions program analysis tool developed in Python. The primary objective of this lab is to engage with and apply fundamental static analysis techniques to a corpus of non-trivial C programs. The tool automates the entire analysis pipeline, including the construction of Control Flow Graphs (CFGs), the calculation of Cyclomatic Complexity, and the execution of a complete Reaching Definitions dataflow analysis. The methodology involved parsing C source code to identify basic block leaders, constructing a graph-based representation of program flow, and iteratively solving dataflow equations to determine the propagation of variable definitions. This document covers the complete workflow, from the careful selection of C programs and the setup of a reproducible environment, to the automated execution of the analysis script, a detailed presentation of the quantitative and qualitative results, and a concluding discussion on the insights and challenges of this practical application of compiler theory.

Contents

1	Introduction, Setup, and Tools	3
1.1	Overview and Objectives	3
1.2	Environment Setup	3
1.3	Selection of C Programs	4
2	Methodology and Execution	4
2.1	CFG Construction	4
2.1.1	Leader and Basic Block Identification	4
2.1.2	Challenges and Error Handling	5
2.1.3	Graph Generation and Visualization	5
2.2	Cyclomatic Complexity	5
2.3	Reaching Definitions Analysis	6
2.3.1	Gen, Kill, and Dataflow Equations	6
2.4	Execution	7
3	Results and Analysis	7
3.1	Analysis of <code>text_adventure_game.c</code>	7
3.2	Analysis of <code>atm_simulator.c</code>	7
3.3	Analysis of <code>student_grade_system.c</code>	7
3.4	Complexity Metrics	8
3.5	Reaching Definitions Analysis	8
4	Discussion and Conclusion	8
4.1	Interpretation of Results	8
4.2	Challenges and Reflections	9
4.3	Conclusion	9
5	GitHub Repository	10

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

The primary goal of this lab was to gain a practical, hands-on understanding of the static analysis techniques that form the foundation of modern compilers and software analysis tools. The core objectives were:

- To select a corpus of C programs that are sufficiently complex, containing various control flow structures like loops and conditionals.
- To implement a Python tool capable of parsing C code to identify leaders and construct basic blocks.
- To automate the generation of Control Flow Graphs (CFGs) from these basic blocks and visualize them using Graphviz.
- To automatically compute the Cyclomatic Complexity metric ($E - N + 2$) for each program's CFG.
- To implement the Reaching Definitions dataflow analysis algorithm, including the calculation of *gen* and *kill* sets and the iterative application of dataflow equations.
- To interpret the final *in* and *out* sets to understand how variable definitions propagate through a program.

1.2 Environment Setup

The experiment was conducted on macOS. To maintain a clean, isolated, and reproducible environment, a Python 3.11 virtual environment was created and used to manage all project dependencies. This prevents conflicts with system-level packages. The required Python libraries, `matplotlib` and `pygraphviz`, were installed using `pip`, as shown in Figure 2. The primary tool for CFG visualization was Graphviz, which is a prerequisite for the `pygraphviz` library.

```
gauravbudhwani@Gauravs-MacBook-Air Lab_7 % brew install graphviz
==> Auto-updating Homebrew...
Adjust how often this is run with `$(HOMEBREW_AUTO_UPDATE_SECS)` or disable with
`$(HOMEBREW_NO_AUTO_UPDATE=1)`. Hide these hints with `$(HOMEBREW_NO_ENV_HINTS=1)` (see `man brew`).
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> New Casks
font-stack-sans-headline
font-stack-sans-text

You have 13 outdated formulae installed.

Warning: graphviz 14.0.1 is already installed and up-to-date.
To reinstall 14.0.1, run:
  brew reinstall graphviz
```

Figure 1: Installing Graphviz using Homebrew.

```
gauravbudhwani@Gauravs-MacBook-Air Lab_7 % python3 -m venv venv
gauravbudhwani@Gauravs-MacBook-Air Lab_7 % source venv/bin/activate
[(venv) gauravbudhwani@Gauravs-MacBook-Air Lab_7 % pip install matplotlib pygraphviz
```

Figure 2: Creating the virtual environment and installing dependencies.

The project directory was organized logically to separate the C source code being analyzed, the analysis scripts, and the generated outputs, ensuring a tidy workspace.

```

/lab7_analyzer/
|-- c_programs/
|   |-- program1.c      # Text Adventure Game
|   |-- program2.c      # ATM Simulator
|   |-- program3.c      # Student Grade System
|-- generated_cfgs/     # Output .dot and .png files
|-- venv/               # Python virtual environment
|-- analyzer.py         # Main analysis script
'-- execute_all.sh      # Automation script

```

1.3 Selection of C Programs

Three standalone C programs were selected to serve as the corpus for this analysis. They were chosen to meet the lab’s criteria of containing diverse control flow structures, multiple variable reassignments, and sufficient complexity to yield interesting analysis results.

- **program1.c (Text Adventure Game):** This program is built around a main `while` loop and uses a complex `if/else if/else` chain to control the game state. Its nested and branching logic makes it an excellent candidate for CFG construction and complexity analysis.
- **program2.c (ATM Simulator):** This program uses a `do-while` loop for its main menu and a `switch` statement for transaction handling. It features numerous reassignments of the `account_balance` variable under different conditions (deposit, withdrawal), making it ideal for demonstrating the utility of Reaching Definitions analysis.
- **program3.c (Student Grade System):** This program combines a `do-while` loop, a `switch` statement, and `for` loops for data processing. It showcases a mix of sequential control flow, menu-driven branching, and iterative data processing, providing a well-rounded subject for analysis.

2 Methodology and Execution

2.1 CFG Construction

The construction of the Control Flow Graph is the foundational step of the analysis. The process was automated in the Python script and follows a well-defined methodology.

2.1.1 Leader and Basic Block Identification

The script first reads a C file and parses it line by line to identify all "leaders," which mark the beginning of a basic block. The identification logic is based on the three rules specified in the lab assignment:

1. The first instruction in the program is a leader.
2. Any instruction that is the target of a branch (e.g., the first line of an `if` body, `else` body, or loop body) is a leader.
3. Any instruction that immediately follows a branch or jump instruction is a leader.

A key part of the implementation is the `find_leaders` function, which uses regular expressions to iterate through the code and mark line numbers that satisfy these rules. A snippet of this function is shown in Listing 1.

```

1 def find_leaders(lines):
2     leaders = {0} # Rule 1
3     for i, line in enumerate(lines):
4         # Rule 2: Target of a branch
5         if re.match(r'^\s*(if|while|for|switch)', line) or \
6            re.match(r'^\s*(else|case|default)', line):
7             leaders.add(i)
8
9         # Rule 3: Instruction immediately after a branch
10        if re.match(r'^\s*(if|while|for|switch|return|break)', line) or \
11           (line.strip().startswith('}') and i > 0):
12            if i + 1 < len(lines):
13                if not lines[i+1].strip().startswith('}'):
14                    leaders.add(i + 1)
15    return sorted(list(leaders))

```

Listing 1: Leader identification logic in ‘analyzer.py’

Once leaders are identified, the code is partitioned into basic blocks. A basic block begins with a leader and extends up to the instruction just before the subsequent leader.

2.1.2 Challenges and Error Handling

During development, a subtle bug was discovered in the leader identification logic. The initial regex for Rule 3 did not properly handle all cases, leading to incorrect basic block splitting. The problematic line was:

```

1 # Incorrectly misses some cases after a block
2 if re.match(r'^\s*(if|while|for|switch|return|break)', line):
3     if i + 1 < len(lines):
4         leaders.add(i + 1)

```

Listing 2: Buggy Regex for Leader Identification

This version failed to identify a leader after a closing brace `}` that wasn’t immediately followed by an `else` statement. The fix, shown in the finalized code in Listing 1, involved adding a condition to explicitly check for lines ending a block (`line.strip().startswith('}') and i > 0`), making the leader detection more robust. This iterative process of testing and refining the parsing logic was a critical part of the development.

2.1.3 Graph Generation and Visualization

After creating the basic blocks, the script constructs the CFG as a directed graph. Nodes represent basic blocks, and edges represent the flow of control between them. Sequential blocks are linked with a single edge, while conditional statements (`if`, `while`) create branches to two different successor blocks. The `pygraphviz` library is then used to generate a `.dot` file, which is a textual representation of the graph structure. Finally, the script uses the `dot` command-line tool from the Graphviz suite to render this file into a PNG image.

Due to the significant length and complexity of the generated CFG images, they could not be included in this report, but they can be accessed on the CS202_A2 Github repo under [CS202_A2/Lab_7/generated.cfgs/programi.png](#). The full-resolution images, along with the source `.dot` files, are available for viewing in the project’s GitHub repository.

2.2 Cyclomatic Complexity

Cyclomatic Complexity is a quantitative measure of the number of linearly independent paths through a program’s source code. It is a useful metric for assessing the structural complexity of a program. A higher value indicates more complex control flow, which can correlate with a

higher likelihood of defects and increased difficulty in testing. It was calculated automatically by the script for each CFG using the formula:

$$CC = E - N + 2$$

where E is the number of edges and N is the number of nodes (basic blocks) in the graph.

2.3 Reaching Definitions Analysis

Reaching Definitions is a classic iterative dataflow analysis algorithm. Its purpose is to determine, for each point in the program, which variable assignments (definitions) might have been the last to define the value of a variable that is live at that point.

2.3.1 Gen, Kill, and Dataflow Equations

The analysis is based on computing two sets for each basic block B :

- **gen[B]:** The set of definitions generated *within* block B that are "live" at its exit (i.e., not overwritten later in the same block).
- **kill[B]:** The set of definitions from *other* parts of the program that are "killed" by definitions within block B . A new definition of a variable x in block B kills all other definitions of x .

These sets are used to iteratively solve the following dataflow equations for each block B until a fixed point is reached:

$$\begin{aligned} \text{in}[B] &= \bigcup_{P \in \text{pred}(B)} \text{out}[P] \\ \text{out}[B] &= \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \end{aligned}$$

The **in** and **out** sets for all blocks are initialized to empty. The equations are then applied repeatedly in a loop until an entire pass over all basic blocks results in no changes to any **out** set, at which point the algorithm has converged.

The core iterative logic from the analyzer is shown in Listing 3.

```

1 def reaching_definitions_analysis(blocks, cfg, gen, kill):
2     in_sets = {block_id: set() for block_id in blocks}
3     out_sets = {block_id: set() for block_id in blocks}
4     predecessors = get_predecessors(cfg)
5
6     changed = True
7     while changed:
8         changed = False
9         for block_id in sorted(blocks.keys()):
10             # IN[B] = Union of OUT[P] for all predecessors P
11             pred_list = predecessors.get(block_id, [])
12             new_in = set().union(*(out_sets.get(p, set()) for p in pred_list))
13             in_sets[block_id] = new_in
14
15             # OUT[B] = gen[B] U (IN[B] - kill[B])
16             old_out = out_sets[block_id]
17             new_out = gen[block_id].union(in_sets[block_id] - kill[block_id])
18
19             if new_out != old_out:
20                 changed = True
21                 out_sets[block_id] = new_out
22     return in_sets, out_sets

```

Listing 3: Iterative dataflow analysis loop

2.4 Execution

The entire analysis pipeline was automated with a shell script, `execute_all.sh`. This script ensures consistency and reproducibility by iterating through the specified C programs and invoking the Python analyzer on each one. This automated approach is critical for efficiency and for eliminating the risk of manual error during repetitive tasks.

```
1 #!/bin/bash
2
3 # Array of programs to analyze
4 programs=("c_programs/program1.c" "c_programs/program2.c" "c_programs/program3.
5          c")
6
7 # Loop and execute analyzer
8 for prog in "${programs[@]}; do
9     python analyzer.py "$prog"
10 done
```

Listing 4: Execution script

3 Results and Analysis

The quantitative metrics for each program and a detailed interpretation of the Reaching Definitions results are presented as following:

3.1 Analysis of `text_adventure_game.c`

Table 1: Cyclomatic Complexity Metrics for `program1.c`

Metric	Value
Number of Nodes (N)	52
Number of Edges (E)	66
Cyclomatic Complexity ($E - N + 2$)	16

3.2 Analysis of `atm_simulator.c`

Table 2: Cyclomatic Complexity Metrics for `program2.c`

Metric	Value
Number of Nodes (N)	46
Number of Edges (E)	52
Cyclomatic Complexity ($E - N + 2$)	8

3.3 Analysis of `student_grade_system.c`

Table 3: Cyclomatic Complexity Metrics for `program3.c`

Metric	Value
Number of Nodes (N)	51
Number of Edges (E)	56
Cyclomatic Complexity ($E - N + 2$)	7

3.4 Complexity Metrics

The Cyclomatic Complexity for each of the three C programs was computed automatically after CFG construction. The results are summarized in Table 4.

Table 4: Combined Results of Cyclomatic Complexity Metrics

Program	Nodes (N)	Edges (E)	Complexity (CC)
program1.c (Adventure Game)	52	66	16
program2.c (ATM Simulator)	46	52	8
program3.c (Grade System)	51	56	7

The adventure game exhibits the highest complexity (CC=16), which is an expected outcome given its deeply nested conditional logic for handling various rooms and player choices. The ATM simulator and Student Grade System show lower but still non-trivial complexity, reflecting their menu-driven but more linear program flow.

3.5 Reaching Definitions Analysis

The ATM simulator (`program2.c`) provides the clearest and most instructive example of the utility of Reaching Definitions analysis. For this discussion, let `d1` be the initial global definition of `account_balance`, let `d_w` represent the definition that occurs inside the `withdraw_cash` function, and let `d_d` represent the definition from the `deposit_cash` function.

After the iterative analysis reached convergence, the final `in` and `out` sets were computed. A selection of these sets for key basic blocks is presented in Table 5.

Table 5: Final Reaching Definitions Sets (Selected) for `program2.c`

Block	IN Set	OUT Set
B2 (Start of do-while loop)	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}
B36 (Contains withdrawal def)	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}	{ <code>d_w</code> , ...} (<i><code>d1</code>, <code>d_d</code> killed</i>)
B43 (Contains deposit def)	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}	{ <code>d_d</code> , ...} (<i><code>d1</code>, <code>d_w</code> killed</i>)
B4 (Program exit point)	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}	{ <code>d1</code> , <code>d_w</code> , <code>d_d</code> , ...}

The results in the table are highly significant. They show that at the entry point of the main program loop (Block B2), multiple definitions for the `account_balance` variable are "reaching" this point. This is the core insight of the analysis.

4 Discussion and Conclusion

4.1 Interpretation of Results

The Reaching Definitions analysis provides powerful insights into the dynamic data flow of a program, which is essential for both optimization and error detection. The key observation from this lab comes from the ATM simulator. Consider the program point at the beginning of the main `do-while` loop (Block B2). As shown in Table 5, the `in` set for this block correctly contains definitions for `account_balance` from three distinct origins:

1. The initial global definition (`d1`), which reaches the loop on its first entry.

2. The reassignment within the `withdraw_cash` function (`d_w`) from a previous loop iteration, which flows back to the start via the CFG's back-edge.
3. The reassignment within the `deposit_cash` function (`d_d`) from a previous loop iteration, which also flows back to the start.

This result perfectly demonstrates the power of dataflow analysis. The tool has successfully determined that, depending on the path taken through the program on any given iteration, any of these three assignments could be the one that defines the value of `account_balance` for the next iteration. This information is fundamental for compilers (e.g., for constant propagation or dead code elimination) and for static analysis tools that detect potential bugs like the use of uninitialized variables.

Similarly, in the adventure game, the analysis reveals how definitions of `player_health` and `current_room` that are generated inside the main `while` loop body (e.g., after a player action causes health loss or a room change) flow back to the beginning of the loop, becoming part of the `in` set for the loop's conditional check in the subsequent iteration.

4.2 Challenges and Reflections

The primary challenge in this lab was the implementation of a parser that could correctly and reliably identify leaders and construct a valid CFG. While using regular expressions proved effective for the structured C code in the selected corpus, this approach is fragile and would not scale to handle the full complexity of the C language grammar (e.g., complex preprocessor macros, convoluted pointer arithmetic, etc.). A production-grade tool would require a much more sophisticated front-end, likely leveraging a dedicated parsing library like Clang to build an Abstract Syntax Tree (AST). This lab provided a valuable, albeit simplified, glimpse into the domain of compiler construction and the significant challenges involved in building resilient software analysis tools.

4.3 Conclusion

The lab successfully achieved all of its stated objectives. A Python-based tool was developed to fully automate the static analysis of C programs. This tool correctly constructs and visualizes Control Flow Graphs, calculates Cyclomatic Complexity, and performs a complete Reaching Definitions dataflow analysis. By applying this tool to a corpus of non-trivial C programs, a practical and deep understanding of these fundamental program analysis techniques was gained. The results confirmed the tool's ability to accurately model program control flow and trace the propagation of variable definitions, providing a solid foundation for understanding more advanced compiler and static analysis concepts.

References

- [1] Course Instructor, "Lecture 7: Dataflow Analysis," *CS 202: Software Tools and Techniques*, IIT Gandhinagar, September 2025.
- [2] Wikipedia contributors, "Data-flow analysis," *Wikipedia, The Free Encyclopedia*, 2025. Accessed: Oct. 12, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Data-flow_analysis
- [3] Graphviz Team, "Graphviz - Graph Visualization Software," 2025. [Online]. Available: <https://graphviz.org/>
- [4] PyGraphviz Developers, "PyGraphviz Documentation," 2025. [Online]. Available: <https://pygraphviz.github.io/>
- [5] Lab-7 Document. *Lab-7*
<https://drive.google.com/file/d/1x6MDZ6e2PJ0MFn6u2U8NPzMxtcuh0y2t/view>

5 GitHub Repository

The GitHub Repository for this Assignment can be accessed using the following link.
[CLICK HERE](#)