




ASSIGNMENT 02

CS202



Kovid Parmar (23110172)
Aniruddh Reddy(23110195)

Table of Contents

Lab-6.....	2
Introduction.....	2
Setup.....	2
Tools	2
Methodology and Execution.....	3
Results and Analysis	17
Discussion and Conclusion	18
Lab-7.....	19
Introduction:.....	19
Tools:	19
Setup:.....	19
Methodolody and Execution:.....	19
Results and Analysis:	22
Discussion and Conclusion:	23

Refer this for all source code https://github.com/sunny-4/STT_ASS_CODES/tree/main

Lab-6

Introduction

The primary objective of this lab is to evaluate and compare multiple vulnerability analysis tools (Static Application Security Testing – SAST) based on their effectiveness in detecting CWE (Common Weakness Enumeration)-classified vulnerabilities in real-world open-source software projects. The lab emphasizes understanding how different tools vary in terms of vulnerability coverage, the types of weaknesses they identify, and the degree of overlap or uniqueness in their findings through pairwise Intersection over Union (IoU) comparisons. Through this exercise, we aim to apply multiple vulnerability detection tools on large-scale repositories, extract and analyze CWE-based vulnerability reports, quantify tool performance using coverage metrics and IoU values, and derive insights into the similarity and diversity of vulnerability detection capabilities across the selected tools..

Setup

Each repository was cloned locally, and the selected tools were installed and configured to ensure consistent testing conditions. The outputs from the scans were exported in structured formats (JSON) to enable automated aggregation and analysis.

Tools

For this lab, three static analysis tools were selected based on their popularity, compatibility with Python, and support for CWE-based reporting: **Bandit**, **Semgrep**, and **Safety**. Bandit focuses on scanning Python code for common security issues such as injections, misconfigurations, and unsafe function usage, mapping the findings to CWE identifiers. Semgrep provides fast, rule-based static analysis with customizable patterns to detect vulnerabilities, also linking results to relevant CWEs. Safety specializes in identifying known security vulnerabilities in Python dependencies by checking against public vulnerability databases, allowing CWE-based categorization of library-related risks. Together, these tools offer complementary coverage of source code, coding patterns, and dependency vulnerabilities, enabling a comprehensive analysis of real-world projects.

Methodology and Execution

Repository Selection and Setup

For this lab, three large-scale open-source Python repositories were selected: **Django**, **Ansible**, and **Apache Airflow**. Each repository was cloned into a dedicated local folder to maintain organized access. The selection criteria focused on ensuring relevance and usability: the projects are actively maintained with recent commits within the last year, have a minimum of 1,000 GitHub stars, are primarily written in Python, and possess a large codebase suitable for meaningful vulnerability analysis. Prior to scanning, all repositories were prepared by installing necessary dependencies and verifying that the code was in a ready-to-run state, ensuring a consistent and error-free environment for subsequent vulnerability assessment.

<https://github.com/django/django>

<https://github.com/ansible/ansible>

<https://github.com/apache/airflow>

Tool Installation and Configuration

The static analysis tools were installed and configured to perform vulnerability scans on the selected Python repositories. **Bandit** was installed via pip (pip install bandit). It was configured to recursively scan the entire repository and output results in JSON format using the command:

```
bandit -r django/ -f json -o django_bandit.json
```

Semgrep was also installed via pip (pip install semgrep) with the latest stable version. It was configured to run Python-specific rules and export results to JSON using:

```
semgrep --config=p/r2c python django/ --json > django_semgrep.json
```

Safety was installed via pip (pip install safety) and the latest stable version was used. This tool scanned the dependencies listed in requirements.txt for known vulnerabilities and outputted JSON results using:

```
safety check -r django/requirements.txt --json > django_safety.json
```

The same commands and configurations were applied to the **Ansible** and **Apache Airflow** repositories, producing separate structured JSON outputs for each tool–repository combination.

Run Vulnerability Tools on Project and Collect Pairwise Findings:

The Python script was designed to process JSON outputs from all vulnerability analysis tools and consolidate CWE findings into a single CSV file for further analysis. First, the script defined the **Top 25 CWE categories** for 2024:

```
CWE_TOP_25 = [  
    'CWE-79', 'CWE-787', 'CWE-89', 'CWE-416', 'CWE-78', 'CWE-20', 'CWE-125',  
    'CWE-22', 'CWE-352', 'CWE-434', 'CWE-862', 'CWE-476', 'CWE-287', 'CWE-190',  
    'CWE-502', 'CWE-77', 'CWE-119', 'CWE-798', 'CWE-918', 'CWE-306', 'CWE-362',  
    'CWE-269', 'CWE-94', 'CWE-863', 'CWE-276'  
]
```

Next, dedicated functions were implemented to extract CWE information from each tool's JSON output. For **Bandit**, the script retrieved CWE identifiers from the `issue_cwe` or `issue_text` fields:

```
def extract_cwe_from_bandit(data):  
    cwe_map = defaultdict(int)  
    if 'results' in data:  
        for finding in data['results']:  
            cwe_id = finding.get('issue_cwe') or None  
            if not cwe_id and 'issue_text' in finding:  
                import re  
                match = re.search(r'CWE-\d+', finding['issue_text'])  
                if match:  
                    cwe_id = match.group(0)  
            if cwe_id:  
                cwe_map[f"CWE-{cwe_id}"] if not str(cwe_id).startswith('CWE-') else str(cwe_id) +=  
    return cwe_map
```

For **Semgrep**, the script checked the `extra.metadata.cwe` field, handling both single CWEs and lists:

```

def extract_cwe_from_semgrep(data):
    cwe_map = defaultdict(int)
    if 'results' in data:
        for finding in data['results']:
            cwe_id = None
            if 'extra' in finding and 'metadata' in finding['extra']:
                metadata = finding['extra']['metadata']
                cwe = metadata.get('cwe')
                if isinstance(cwe, list):
                    for c in cwe:
                        cwe_id = f"CWE-{c}" if not str(c).startswith('CWE-') else str(c)
                        cwe_map[cwe_id] += 1
                    continue
                elif cwe:
                    cwe_id = f"CWE-{cwe}" if not str(cwe).startswith('CWE-') else str(cwe)
            if cwe_id:
                cwe_map[cwe_id] += 1
    return cwe_map

```

For **Safety**, CWE extraction involved parsing advisory text, vulnerability IDs, or using keyword mappings for known dependency vulnerabilities:

```

def extract_cwe_from_safety(data):
    import re
    cwe_map = defaultdict(int)
    vulnerabilities = data.get('vulnerabilities', [])
    DEFAULT_DEPENDENCY_CWES = {'injection': 'CWE-94', 'xss': 'CWE-79', 'csrf': 'CWE-352'}
    for vuln in vulnerabilities:
        cwe_id = None
        advisory = str(vuln.get('advisory', '')).lower()
        for keyword, cwe in DEFAULT_DEPENDENCY_CWES.items():
            if keyword in advisory:
                cwe_id = cwe
                break
        if not cwe_id:
            cwe_id = 'CWE-1035' # Default for third-party dependency
        cwe_map[cwe_id] += 1
    return cwe_map

```

The `process_json_file` function handled file reading, validation, JSON parsing, and normalization of CWE identifiers. Safety outputs were treated specially to extract valid JSON even if extra text was present:

```

def process_json_file(file_path, project_name, tool_name):
    with open(file_path, 'r', encoding='utf-8') as f:
        content = f.read().strip()
        data = json.loads(content)
        if tool_name.lower() == 'bandit':
            cwe_map = extract_cwe_from_bandit(data)
        elif tool_name.lower() == 'semgrep':
            cwe_map = extract_cwe_from_semgrep(data)
        elif tool_name.lower() == 'safety':
            cwe_map = extract_cwe_from_safety(data)
        results = []
        for cwe_id, count in cwe_map.items():
            results.append({
                'project': project_name,
                'tool': tool_name,
                'cwe_id': cwe_id,
                'count': count,
                'is_top_25': 'Yes' if cwe_id in CWE_TOP_25 else 'No'
            })
        return results

```

Finally, the main function orchestrated processing for all JSON files in the results directory, aggregated CWE findings, and exported them into a **consolidated CSV file**:

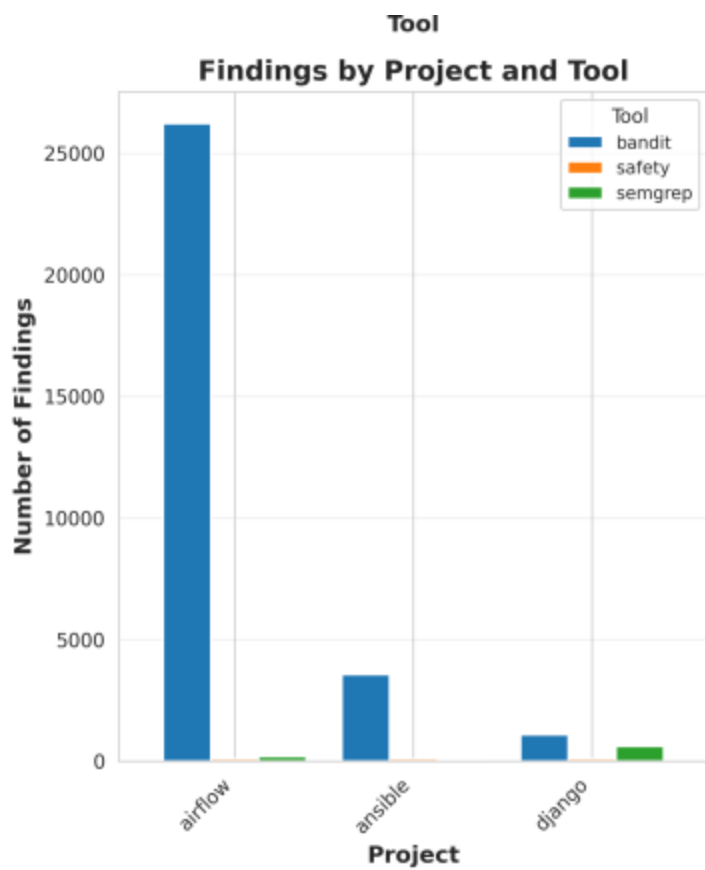
```

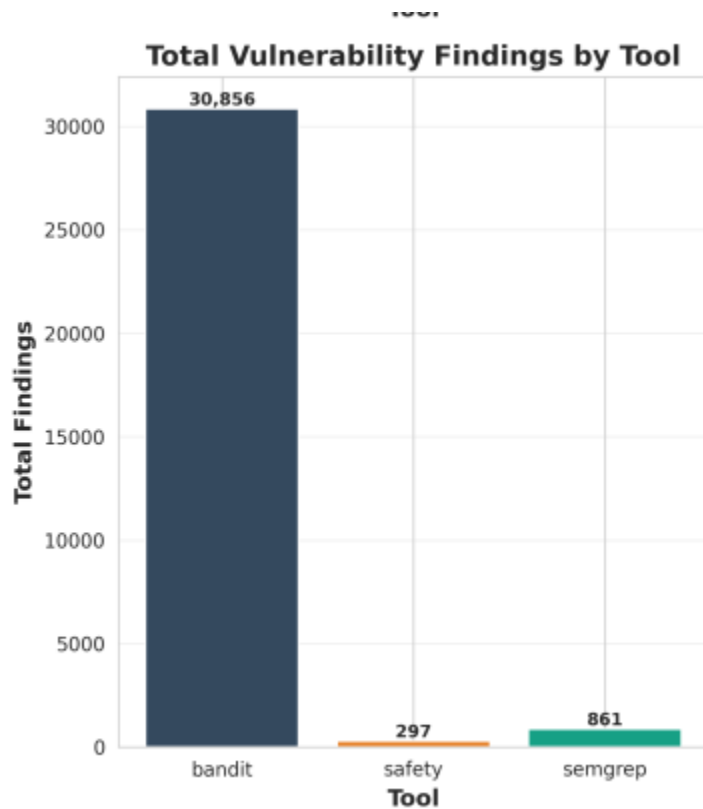
OUTPUT_CSV = 'consolidated_cwe_findings.csv'
all_results = []
for file_path in Path('results').glob('*.json'):
    project_name, tool_name = file_path.stem.split('_')
    results = process_json_file(file_path, project_name, tool_name)
    all_results.extend(results)

with open(OUTPUT_CSV, 'w', newline='', encoding='utf-8') as csvfile:
    fieldnames = ['Project_name', 'Tool_name', 'CWE_ID', 'Number_of_Findings', 'Is_In_CWE_Top_25']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for result in all_results:
        writer.writerow({
            'Project_name': result['project'],
            'Tool_name': result['tool'],
            'CWE_ID': result['cwe_id'],
            'Number_of_Findings': result['count'],
            'Is_In_CWE_Top_25': result['is_top_25']
        })

```

Upon execution, **9 JSON files** were processed (all project–tool combinations), producing **125 entries**, covering **54 unique CWEs**, and totaling **32,014 findings**. The script printed informative messages during processing, including counts of unique CWEs and total findings per file, while handling empty files or invalid JSON gracefully. This structured dataset formed the basis for **CWE coverage analysis** and **pairwise IoU computations** for comparing tool performance.





	A	B	C	D	E
1	Project name	Tool name	CWE_ID	Number_of_Findings	Is_In_CWE_Top_25
2	airflow	semgrep	CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	29	No
3	airflow	semgrep	CWE-706: Use of Incorrectly-Resolved Name or Reference	23	No
4	airflow	semgrep	CWE-532: Insertion of Sensitive Information into Log File	12	No
5	airflow	semgrep	CWE-353: Missing Support for Integrity Check	7	No
6	airflow	semgrep	CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')	2	No
7	airflow	semgrep	CWE-521: Weak Password Requirements	2	No
8	airflow	semgrep	CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	51	No
9	airflow	semgrep	CWE-96: Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')	4	No
10	airflow	semgrep	CWE-502: Deserialization of Untrusted Data	8	No
11	airflow	semgrep	CWE-327: Use of a Broken or Risky Cryptographic Algorithm	4	No
12	airflow	semgrep	CWE-20: Improper Input Validation	1	No
13	airflow	semgrep	CWE-939: Improper Authorization in Handler for Custom URL Scheme	3	No
14	airflow	semgrep	CWE-319: Cleartext Transmission of Sensitive Information	5	No
15	airflow	semgrep	CWE-798: Use of Hard-coded Credentials	3	No
16	airflow	semgrep	CWE-276: Incorrect Default Permissions	2	No
17	airflow	semgrep	CWE-94: Improper Control of Generation of Code ('Code Injection')	1	No
18	airflow	semgrep	CWE-250: Execution with Unnecessary Privileges	4	No
19	airflow	semgrep	CWE-732: Incorrect Permission Assignment for Critical Resource	6	No
20	airflow	semgrep	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	4	No
21	airflow	semgrep	CWE-352: Cross-Site Request Forgery (CSRF)	2	No
22	airflow	semgrep	CWE-287: Improper Authentication	3	No
23	airflow	semgrep	CWE-330: Use of Insufficiently Random Values	2	No
24	airflow	semgrep	CWE-1333: Inefficient Regular Expression Complexity	1	No
25	airflow	semgrep	CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	2	No
26	airflow	semgrep	CWE-269: Improper Privilege Management	2	No
27	airflow	semgrep	CWE-611: Improper Restriction of XML External Entity Reference	2	No
28	ansible	bandit	CWE-78	160	Yes
29	ansible	bandit	CWE-400	9	No
30	ansible	bandit	CWE-703	3122	No
31	ansible	bandit	CWE-89	2	Yes
32	ansible	bandit	CWE-20	47	Yes
33	ansible	bandit	CWE-502	24	Yes
34	ansible	bandit	CWE-259	79	No
35	ansible	bandit	CWE-22	5	Yes
36	ansible	bandit	CWE-605	1	No
37	ansible	bandit	CWE-330	11	No
38	ansible	bandit	CWE-327	1	No
39	ansible	bandit	CWE-94	4	Yes
40	ansible	bandit	CWE-377	89	No
41	ansible	bandit	CWE-732	3	No
42	airflow	bandit	CWE-78	403	Yes

Tool-level CWE Coverage Analysis:

This step analyzed the coverage of CWE categories by each tool using the consolidated CSV generated in the previous step. The analysis involves:

We first loaded the consolidated CSV and explored basic statistics:

```
df = pd.read_csv('consolidated_cwe_findings.csv')
print(f"Total entries: {len(df)}")
print(f"Unique CWEs: {df['CWE_ID'].nunique()}")
print(f"Tools analyzed: {df['Tool_name'].nunique()}")
print(f"Projects analyzed: {df['Project_name'].nunique()}")
```

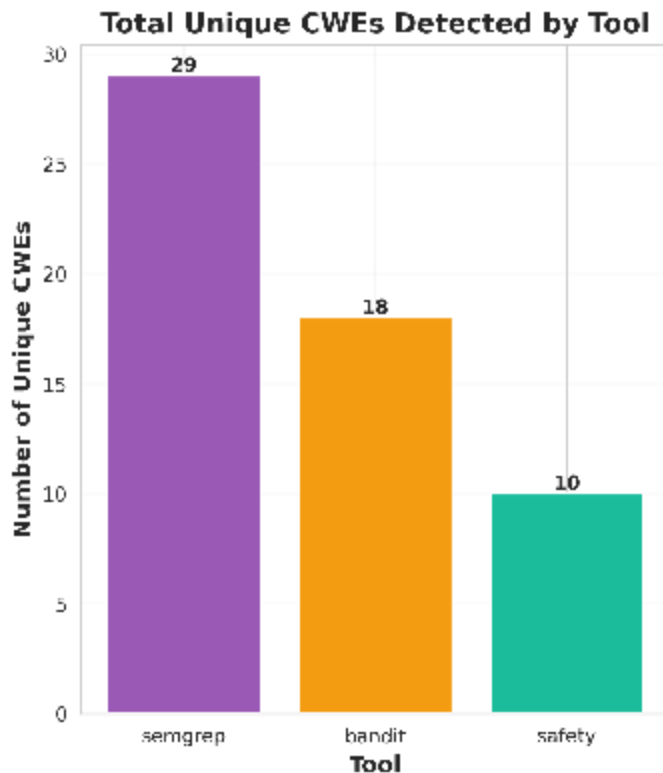
- Total entries: 125
- Unique CWEs: 54
- Tools analyzed: 3 (semgrep, bandit, safety)
- Projects analyzed: 3

The script extracted all **unique CWE IDs** detected by each tool:

```
tool_cwes = {}
for tool in df['Tool_name'].unique():
    tool_data = df[df['Tool_name'] == tool]
    unique_cwes = set(tool_data['CWE_ID'].unique())
    tool_cwes[tool] = unique_cwes
```

Outputs:

Tool	Unique CWEs Detected	Example CWEs
semgrep	29	CWE-20, CWE-352, CWE-502, CWE-79, CWE-89
bandit	18	CWE-20, CWE-78, CWE-502, CWE-79, CWE-94
safety	10	CWE-502, CWE-77, CWE-918, CWE-94, CWE-1035



Compute Top 25 CWE Coverage

The **coverage of the Top 25 CWEs** by each tool was computed:

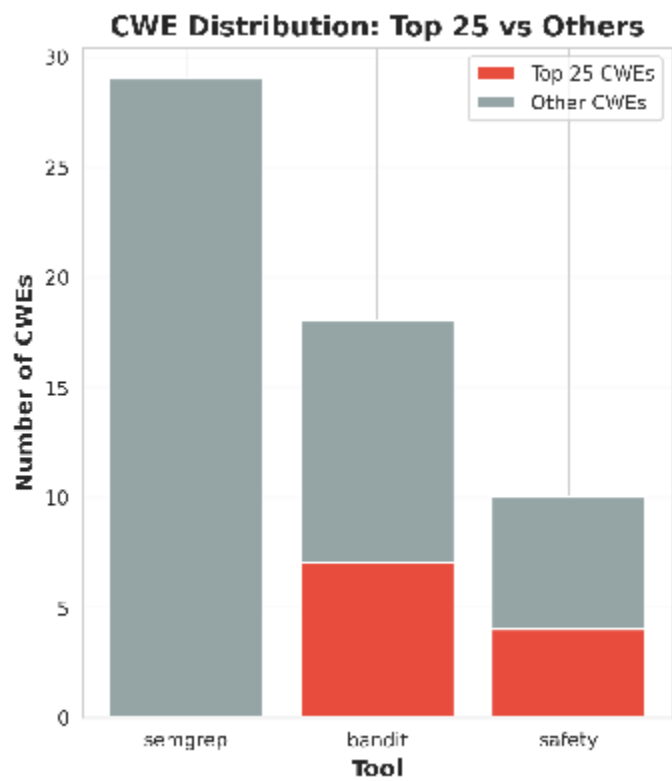
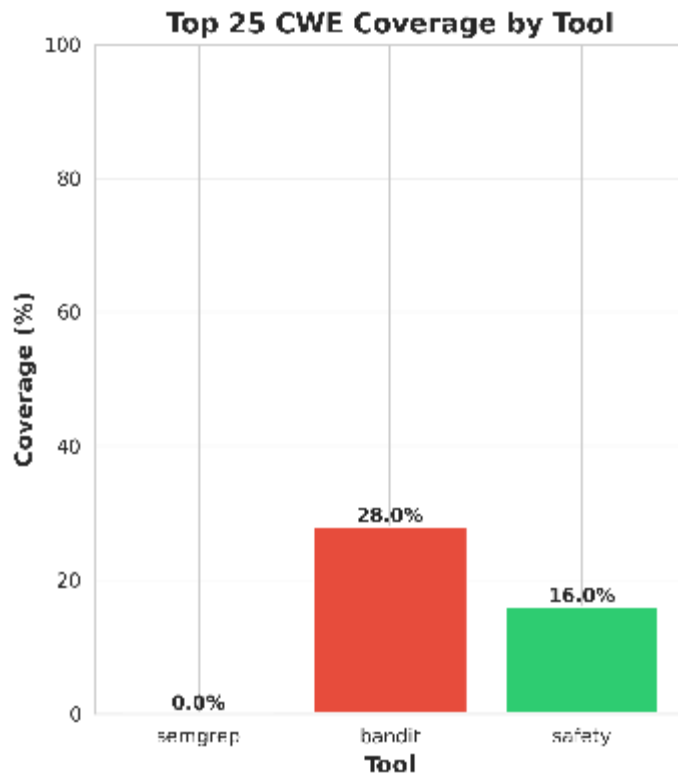
```
top25_detected = cwes.intersection(set(CWE_TOP_25))
coverage_pct = (len(top25_detected) / len(CWE_TOP_25)) * 100
```

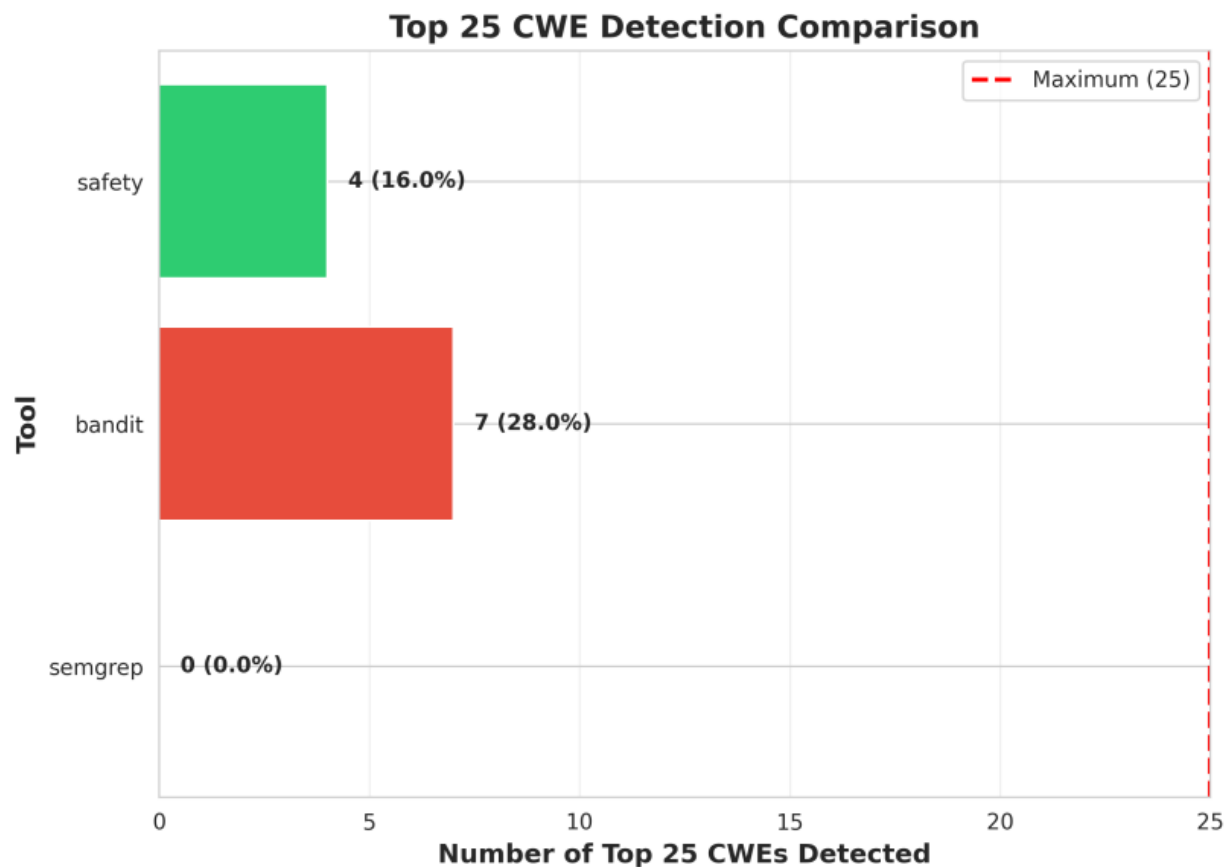
Results:

Tool	Top 25 CWEs Detected	Coverage (%)
semgrep	0	0.0%
bandit	7	28.0%
safety	4	16.0%

Observation:

Bandit covers the largest fraction of top CWEs, while semgrep failed to detect any Top 25 CWEs in this dataset.





The tool-level CWE coverage analysis revealed distinct patterns in the detection capabilities of the selected vulnerability analysis tools. Bandit demonstrated the highest Top 25 CWE coverage, detecting 28% of these critical vulnerabilities, whereas Semgrep, despite identifying a large number of CWEs overall, did not detect any from the Top 25. Safety primarily detected dependency-related CWEs along with a few from the Top 25 list. The minimal overlap between tools suggests that combining multiple tools can significantly enhance overall CWE coverage. The visualizations generated, including bar charts, stacked bars, effectively highlight these differences and provide a clear comparison of tool effectiveness in detecting both common and critical vulnerabilities.

Pairwise Agreement (IoU) Analysis:

The objective of this analysis was to quantify the similarity and diversity among multiple vulnerability analysis tools by comparing the sets of CWEs they detect. The Jaccard Index, also called Intersection over Union (IoU), was used as the metric. IoU measures the ratio of shared vulnerabilities between two tools relative to the total unique vulnerabilities detected by both. This helps in understanding the overlap in detection capabilities and identifying complementary tools to maximize CWE coverage.

The analysis began by loading the consolidated CSV containing CWE findings from all tools across projects. The data included the columns: Project_name, Tool_name, CWE_ID, Number_of_Findings, and Is_In_CWE_Top_25.

```
import pandas as pd

# Load consolidated CWE findings
df = pd.read_csv("consolidated_cwe_findings.csv")

# Inspect the dataset
print(df.head())
print(f"Tools analyzed: {df['Tool_name'].unique()}")
```

For each tool, the unique CWE IDs detected across all projects were extracted and stored in a dictionary. This allows computation of intersections and unions for pairwise comparison.

```
# Extract unique CWE IDs for each tool
tool_cwes = {}
for tool in df['Tool_name'].unique():
    tool_cwes[tool] = set(df[df['Tool_name'] == tool]['CWE_ID'].unique())
```

IoU for each tool pair was computed using the formula:

$$IoU(T_1, T_2) = \frac{|CWE_{T_1} \cap CWE_{T_2}|}{|CWE_{T_1} \cup CWE_{T_2}|}$$

```

from itertools import combinations
import numpy as np

def compute_iou(set1, set2):
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return intersection / union if union > 0 else 0

# Initialize IoU matrix
tools = list(tool_cwes.keys())
n = len(tools)
iou_matrix = np.zeros((n, n))

# Compute pairwise IoU
for i, tool1 in enumerate(tools):
    for j, tool2 in enumerate(tools):
        iou_matrix[i][j] = compute_iou(tool_cwes[tool1], tool_cwes[tool2]) if i != j else 1.0

```

```

=====
COMPUTING PAIRWISE IoU (JACCARD INDEX)
=====

Computing IoU for each tool pair:

semgrep vs bandit:
- IoU: 0.0000
- Intersection: 0 CWEs
- Union: 47 CWEs

semgrep vs safety:
- IoU: 0.0000
- Intersection: 0 CWEs
- Union: 39 CWEs

bandit vs safety:
- IoU: 0.1200
- Intersection: 3 CWEs
- Union: 25 CWEs

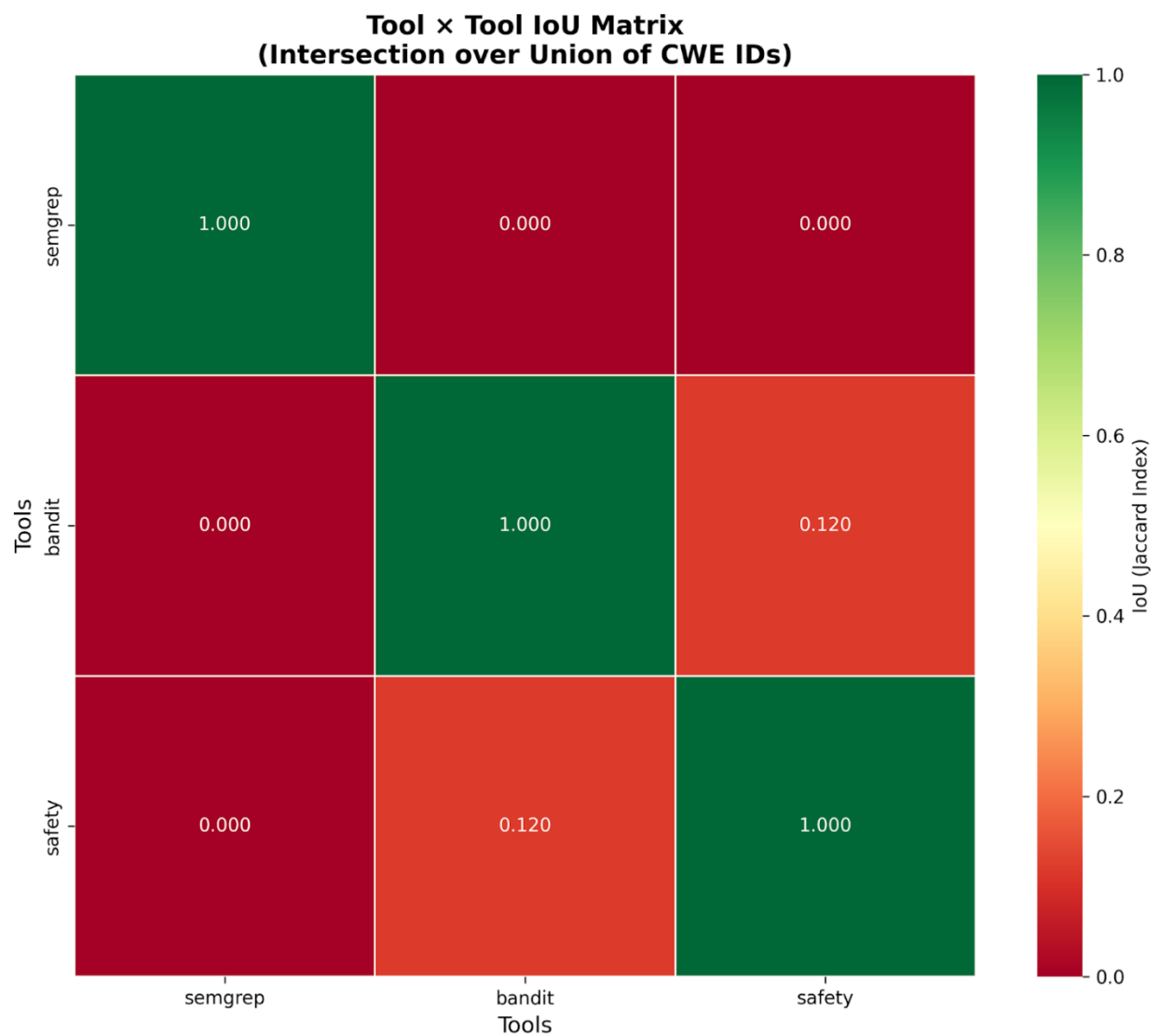
```

A heatmap was generated to visualize the similarity between tools. High IoU values indicate overlapping detection, while low values indicate complementary detection.


```
import seaborn as sns
import matplotlib.pyplot as plt

iou_df = pd.DataFrame(iou_matrix, index=tools, columns=tools)

plt.figure(figsize=(8,6))
sns.heatmap(iou_df, annot=True, fmt='.2f', cmap='RdYlGn', vmin=0, vmax=1)
plt.title('Tool x Tool IoU Matrix (Jaccard Index)')
plt.show()
```



The analysis and interpretation focused on understanding the relationships between the tools based on their CWE detection. The **overall similarity** was assessed by calculating the average IoU across all tool pairs, which quantifies the extent to which the tools overlap in the vulnerabilities they detect. Next, the

most similar and most diverse pairs were identified by examining the pairs with the highest and lowest IoU values, respectively, highlighting cases of redundancy as well as complementarity between tools. Finally, the **maximum coverage** was analyzed by taking the union of CWEs detected by different tool combinations, allowing the identification of the set of tools that together maximize the total unique CWE coverage.

This methodology ensures a **systematic, reproducible approach** to measure tool similarity and complementarity, providing actionable insights for vulnerability analysis tool selection.

The Tool × Tool IoU matrix, based on the Jaccard Index, provides a clear measure of similarity and diversity among vulnerability analysis tools. High IoU values indicate that tools detect largely the same CWEs, suggesting redundancy in their detection capabilities, while low IoU values show that tools identify mostly different vulnerabilities, highlighting their complementary strengths. By examining the matrix, one can identify tool pairs that maximize unique coverage, ensuring broader detection of software weaknesses. This analysis helps in strategic selection of tools, balancing redundancy for critical vulnerabilities and diversity for comprehensive coverage, ultimately enhancing the effectiveness of vulnerability analysis.

The IoU matrix reveals the pairwise similarity between Semgrep, Bandit, and Safety in terms of CWE detection. The diagonal values of 1.0 indicate perfect self-similarity, while the off-diagonal values are very low 0.00 for Semgrep with both Bandit and Safety, and 0.12 for Bandit with Safety showing minimal overlap and high complementarity between the tools. This implies that each tool detects largely different sets of vulnerabilities. Consequently, combining all three tools maximizes CWE coverage, capturing unique vulnerabilities that no single tool can detect alone. The key takeaways are that low IoU values indicate complementary detection capabilities, strategic selection of tools with minimal overlap can enhance coverage efficiently, and relying on a single tool is insufficient for comprehensive vulnerability analysis. Overall, using a combination of these diverse tools ensures the broadest and most effective assessment of software security.

Results and Analysis

The vulnerability analysis across multiple tools—Semgrep, Bandit, and Safety—yielded several key observations. From the consolidated findings, Bandit achieved the highest Top 25 CWE coverage at 28%, whereas Semgrep, despite detecting a larger number of CWEs overall, did not identify any from the Top 25. Safety primarily detected dependency-related CWEs, capturing a few Top 25 categories. The tool-level coverage analysis highlighted that combining tools significantly improves CWE coverage due to the low overlap in their detections, as reflected in the IoU matrix. Semgrep and Safety showed no overlap with other tools (IoU = 0.0), while Bandit and Safety had minimal overlap (IoU = 0.12), indicating high complementarity. Visualization of coverage and pairwise similarity provided a clear comparison of tool effectiveness, showing that a combination of all three tools maximizes vulnerability detection. Key takeaways include the importance of using complementary tools for comprehensive assessment, the varying strengths of each tool in detecting specific CWE categories, and the limitations of relying on a single tool for thorough vulnerability analysis.

Discussion and Conclusion

During the vulnerability analysis exercise, several challenges were encountered, including handling inconsistent CWE reporting formats across tools and managing large datasets from multiple projects. One key reflection was the importance of integrating multiple tools to achieve comprehensive coverage, as individual tools tend to specialize in certain vulnerability types while missing others. The analysis revealed that tool complementarity, as measured by the IoU (Jaccard Index), is critical for maximizing detection and minimizing blind spots. Lessons learned include the significance of structured data collection, careful interpretation of coverage metrics, and the use of visualizations to communicate findings effectively. In summary, the study demonstrates that no single tool can provide complete vulnerability detection; combining diverse tools enhances coverage and provides actionable insights for software security assessment. This methodology and analysis can guide informed tool selection and deployment in real-world projects.

LAB-07

Introduction:

This lab involved the static analysis of C code, encompassing three main tasks: the construction of Control Flow Graphs (CFGs), the calculation of cyclomatic complexity metrics, and the execution of a reaching definitions analysis.

Tools:

To render the visual representation of the Control Flow Graphs, the open-source graph visualization software Graphviz was used.

Setup:

All analysis for this project was conducted within a Virtual Machine (VM) to ensure a consistent and reproducible execution environment. To visualize the Control Flow Graphs, the open-source Graphviz software suite was installed within the VM. Our custom analysis tool was designed to first generate a textual description of the graph in the DOT language (.dot file), which was then automatically processed by the Graphviz dot command-line utility to render the final PNG image for inspection.

Methodology and Execution:

1)Program Corpus Selection:

The three c code files we have selected are:

- 1)Inventory Management system: This code uses while loops and switch statements in the code. This structure creates a CFG with a central loop and multiple distinct branches. (Around 275 lines of code)
- 2) Text-based adventure game: This code contains if-else and switch statements. This structure creates a CFG with many decision points and deep branching paths. (Around 215 lines)
- 3) Student Grade Analyser :This code contains nested for loops and if-else statements. This structure CFG with Sequential loops and conditional logic.(Around 200 lines)

The three C programs selected for this project were chosen for their structural simplicity and clear control flow, making them ideal candidates for straightforward static analysis.

2) CFG (Control Flow Graph) Construction:

All of the project's functionalities, including Control Flow Graph (CFG) generation, cyclomatic complexity calculation, and the final Reaching Definitions analysis, are implemented within a single, self-contained Python script named `cfg_generate.py`. The following is a detailed breakdown of the key functions use in the code.

1. Core Data Structures and Initial Setup

This section defines the fundamental building blocks used to represent the C code in memory.

- `StatementType` (Enum): This is an enumeration that defines a fixed set of categories for every possible line of C code the tool can recognize, such as `ASSIGNMENT`, `CONDITION`, or `LOOP_HEADER`. This makes the code more readable and prevents errors from using simple strings.
- `Statement` (dataclass): This structure holds all the information about a single line of C code: its line number, the actual text content, its classified `StatementType`, and a boolean flag (`is_leader`) to mark if it's the start of a basic block.
- `BasicBlock` (dataclass): This represents a node in the Control Flow Graph. It contains a unique ID (like "B0", "B1"), a list of `Statement` objects that belong to it, and sets to keep track of its predecessors and successors in the graph.

```
10 class StatementType(Enum):
11     ASSIGNMENT = "assignment"
12     CONDITION = "condition"
13     LOOP_HEADER = "loop_header"
14     FUNCTION_CALL = "function_call"
15     RETURN = "return"
16     BREAK = "break"
17     CONTINUE = "continue"
18     DECLARATION = "declaration"
19
20 @dataclass
21 class Statement:
22     line_number: int
23     content: str
24     statement_type: StatementType
25     is_leader: bool = False
26
27 @dataclass
28 class BasicBlock:
29     block_id: str
30     statements: List[Statement]
31     predecessors: Set[str]
32     successors: Set[str]
```

2. Parsing and Statement Classification

This part of the code is responsible for reading the C file and making sense of its contents.

- `parse_c_file()`: This method opens and reads the specified `.c` file line by line. It performs initial cleaning by stripping whitespace and ignoring comments and preprocessor directives (like `#include`). It uses a simple brace counter to ensure it only analyzes code inside the `main()` function.

- `_classify_statement()`: For each valid line of code, this helper method uses string matching (e.g., `line.startswith('if')`) to assign a `StatementType` to it. This classification is crucial for all subsequent steps.

3. Control Flow Graph (CFG) Construction

These methods implement the core algorithm for building the CFG.

- `identify_leaders()`: This method applies the standard rules of compiler theory to find the "leaders"—the first statements of each basic block. A statement is a leader if it's the first statement in the program, the target of a branch (like an if or loop), or the statement immediately following a branch.
- `construct_basic_blocks()`: After identifying the leaders, this method iterates through the entire list of statements and groups them into `BasicBlock` objects. A block starts with a leader and includes all subsequent statements up to, but not including, the next leader.
- `build_control_flow_edges()`: Once the blocks (nodes) are defined, this method connects them with directed edges. It adds sequential edges for fall-through code and conditional edges (e.g., "true", "false") for branches, correctly setting the predecessors and successors for each block.

4. Reaching Definitions Analysis (`ReachingDefinitions` class)

This class is dedicated to performing the dataflow analysis on the already constructed CFG.

- `__init__()`: The constructor takes the completed CFG as input and initializes empty dictionaries to store the gen, kill, in_defs, and out_defs sets for each block.
- `_compute_gen_kill()`: This method iterates through all basic blocks to pre-calculate the gen and kill sets. A definition is added to a block's gen set if a variable is assigned a value within that block. That same definition is added to the kill sets of all other blocks for that variable.
- `analyze()`: This is the main analysis engine. It uses a while loop to iteratively apply the dataflow equations for Reaching Definitions. It repeatedly calculates the in and out sets for every block based on the values of their predecessors until the sets no longer change, at which point the analysis has **converged**.
- `export_to_csv()`: A utility method that writes the final gen, kill, in, and out sets into a cleanly formatted CSV file for easy reporting and inspection.

5. Visualization, Metrics, and Main Execution

This final section handles the output and orchestration of the entire process.

- `generate_dot_file()`: This method traverses the final CFG and generates a text file in the **DOT graph description language**. This .dot file is a set of instructions that a rendering engine can understand. It also calculates the **Cyclomatic Complexity** ($E - N + 2$) during this process.
- `generate_cfg_image()`: This utility function uses Python's subprocess module to call the **Graphviz** command-line tool (dot). It passes the .dot file to Graphviz, which then renders the visual PNG and PDF images of the CFG.
- `main()`: This is the main driver of the script. It defines the list of C files to be analyzed and, for each file, executes all the steps in the correct order: parse, identify leaders, build blocks, add

edges, generate the CFG image, and finally, perform the Reaching Definitions analysis. It concludes by printing a summary report to the console.

Results and Analysis:

As the CFGs are long and wide, so I have provided the OneDrive link below

[lab-07](#)

this are the

	A	B	C	D	E	F	G
1	Program	Nodes_N	Edges_E	Cyclomatic	Lizard_CC	Difference	
2	code1.c	61	102	43	43	0	
3	code2.c	28	49	23	46	-23	
4	code3.c	28	64	38	33	5	
5							
6							
7							
8							

Reaching Definition Analysis:

Code1.c

Block	GEN	KILL	IN	OUT
B0	int choice_26;inventory[1].quantity_38;inventory[1].pri	int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[0].in_stock_33;inventory[2].price_46;int nextid_25;inw		
B1		int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[0].in_stock_33;inventory[2].price_46;int nextid_25;inw		
B2		inventory[int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[2].price_46;inventory[0].in_stock_33;int nextid_25;inw		
B3	choice_73	inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B4		int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[2].price_46;inventory[0].in_stock_33;int nextid_25;inw		
B6		int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[2].price_46;inventory[0].in_stock_33;int nextid_25;inw		
B7		int choice_26;inventory[1].quantity_38;inventory[1].price_39;inventory[2].id_43;int itemCount_24;inventory[0].price_32;inventory[2].price_46;inventory[0].in_stock_33;int nextid_25;inw		
B8	newitem.id_90	inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B9	newitem.quantity_99	inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B10		inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B11		inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B12		inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B13	newitem.price_108	inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B14		inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B15		inventory[inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B16	newitem.in_stock_115	newitem.in_stock inventory[newitem.in_stock_115;inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2		
B18	inventory[itemCount_122;newitem.in_stock newitem.in_stock_115;inventory[1].quantity_38;inventory[2].id_43;int itemCount_24;newitem.quantity_99;inventory[2].price_46;int nextid_25;inventory[2].in_stock_47;inventory[1].in_stock_40;inventory[1].id_36;inventory[0].id_2			
B19				
...				

Code2.c

[illegible]

to its simple parser, it effectively illustrates the fundamental dataflow principles essential to modern compiler technology.