ASSIGNMENT 2

Software Tools and Techniques for CSE

Tejas Lohia and Zainab Kapadia

23110335 and 23110373

https://github.com/TejasLohia21/Lab6-and-7_23110335

TABLE OF CONTENTS

Contents

Chapter 1

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

Laboratory Session 6

INTRODUCTION

This Laboratory session was held on 8th September 2025 and was aimed towards exploring and evaluating the capabilities of different Static Application Security Testing (SAST) tools in identifying software weaknesses within real-world, large-scale open-source projects. By doing this lab work, I understood the effectiveness and scope of multiple vulnerability analysis tools by using the Common Weakness Enumeration (CWE), a community-developed list of common software and hardware weakness types, to provide a standardized measure for comparison. This was done by analyzing the degree of agreement and disagreement between the tools. To quantify this, we will compute the Intersection over Union (IoU) for each pair of tools based on the CWEs they detect. This metric will show the diversity in their detection capabilities and inform which tool combinations might offer the most useful vulnerability coverage.

TOOLS

We worked with various vulnerability analysis tools which required installation of certain tools and libraries as follows:-

A) Vulnerability Analysis Tools

The following three Static Application Security Testing (SAST) tools were selected for this evaluation:

- **Tool 1:** Cppcheck
- **Tool 2:** Flawfinder
- **Tool 3:** scan-build

B) Open-Source Repositories

The analysis was performed on the following three open-source repositories, chosen based on criteria such as project scale, complexity, and programming language:

- **Repository 1:** libgit2

- **Repository 2:** libsndfile
- **Repository 3:** nghttp2

SETUP

The first thing I did was create a virtual environment in order to isolate the dependencies. I also installed all the required tools i.e., cppcheck, flawfinder and scanbuild using pip install. Moreover, I cloned the repositories locally in the repos_cpp folder in order to expedite the process.

METHODOLOGY AND EXECUTION

(a) Repository Selection:

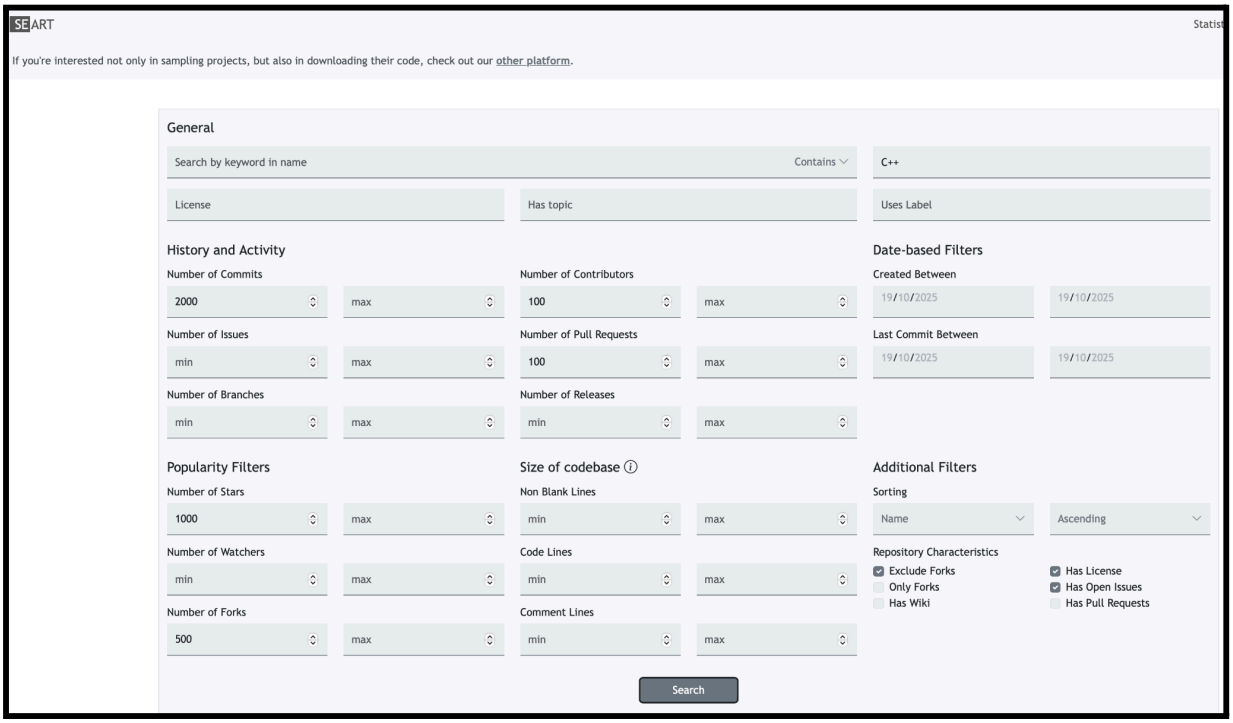
The first task in this lab was to select a large scale open-source repository to carry out our analysis. To make this choice systematic, I used the **SEART GitHub Search Engine**, which allows advanced filtering of repositories based on commits, issues, stars, forks, and primary programming language. This ensured that the repository chosen was not a toy project but a real-world codebase.

Selection Criteria:

For this task, I defined the following **selection criteria**:

- The repository should have **at least 2000 commits**, ensuring sufficient development history to analyze bug-fix patterns meaningfully.
- It should have a minimum of **100 Contributors,** which makes sure we analyze a collaborative codebase.
- The minimum **Pull Requests should be 100**, so that there is an active review and contribution process.
- The **minimum Stars should be 1000 & Forks should be 500**.
- The language should be **C++** as I will be carrying out the analysis using static analysis tools meant for C++.
- **Exclude Forks, Has License, Has Open Issues**: These filters ensure we are working on a real world project and clones.

1



Based on these criteria, I selected the three repositories as libgit2, libsndfile and ngttp2. **libgit2** is a portable, pure C implementation of the Git core methods. It is designed to be a linkable library for other applications that need to interact with Git repositories, without having to shell out to the git command-line tool. It's used by major services like GitHub, GitLab, and Microsoft Visual Studio. Its complexity in handling repository data and network protocols is the reason why we chose it for SAST analysis.

libsndfile is a widely used C library for reading and writing files containing sampled sound (such as WAV, FLAC, and AIFF). It is famous for its simple, elegant API and support for a vast number of audio formats. Due to its role in parsing complex binary file formats, it is susceptible to vulnerabilities related to buffer overflows, integer overflows, and other memory corruption issues.

nghttp2 is a C implementation of the HTTP/2 protocol, the second major version of the

HTTP network protocol. It provides both server and client-side libraries and is used in critical network applications like cURL and the Apache HTTP Server.

(b) Vulnerability Tool Selection:-

The following three open-source Static Application Security Testing (SAST) tools were selected for this evaluation based on their explicit support for C/C++ projects and their ability to report findings using CWE identifiers:

- **Flawfinder**:- Flawfinder is a lightweight and easy-to-use source code scanner that operates by searching for known C/C++ functions that have a history of security vulnerabilities, such as buffer overflows (e.g., strcpy()) and format string bugs. It ranks potential flaws by risk level and directly maps many of them to CWEs.
- **Cppcheck**:- Cppcheck is a more advanced static analysis tool for C/C++ that goes

beyond simple pattern matching. It performs a deeper analysis of the code to detect a wide range of bugs, including memory leaks, null pointer dereferences, and uninitialized variables. It is highly configurable and can produce detailed reports in XML format, which helps in programmatic parsing.

- **Scan-build (Clang Static Analyzer)**:- It is a part of the LLVM compiler infrastructure and is a powerful, high-fidelity tool that performs deep, path-sensitive analysis. It explores the execution paths within the code to find complex bugs that other tools might miss, such as use-after-free errors and dead stores. It integrates directly with the build process (e.g., make), which helps in analyzing large and complex codebases effectively.

(c) Running Vulnerability tools and collecting pairwise findings:

For this part, we first created a python script which automated the pipeline, i.e., running the analysis on each repository, parsing the heterogeneous outputs and extracting standardized CWE data.

The script performs this:

- It defines the target repositories and the output directory for storing results.
- It includes a predefined set of the 2024 "CWE Top 25" weaknesses which has been taken from 2024 CWE Top 25 Most Dangerous Software Weaknesses.

```
projs={
    "libsndfile": "repos_cpp/libsndfile",
    "libgit2": "repos_cpp/libgit2",
    "nghttp2": "repos_cpp/nghttp2"
}

results_dir= "results_cpp"
final= os.path.join(results_dir, "consolidated_cpp.csv")

CWE_TOP_25= {
    "CWE-787", "CWE-79", "CWE-89", "CWE-20", "CWE-125", "CWE-78",
    "CWE-416", "CWE-22", "CWE-352", "CWE-434", "CWE-476", "CWE-502",
    "CWE-269", "CWE-94", "CWE-863", "CWE-77", "CWE-306", "CWE-362",
    "CWE-798", "CWE-918", "CWE-400", "CWE-611", "CWE-502", "CWE-284", "CWE-295"
}
```

4

- It contains functions to execute each SAST tool (Flawfinder, Cppcheck, scan-build) via shell commands using Python's subprocess module.
- Each function is responsible for invoking its corresponding tool with the correct parameters, capturing the output, and parsing it to extract CWE identifiers.
- The main execution loop iterates through each defined project, runs all three analysis tools on it, and stores the structured results.

```
if __name__ == "__main__":
    if os.path.exists(final):
        os.remove(final)

    for proj in projs:
        proj_path = projs[proj]
        run_flawfinder(proj_path, proj, out_format="csv")
        run_cppcheck(proj_path, proj, out_format="xml")
        run_scanbuild(proj_path, proj, out_format="json")

    print(f"All analyses complete! Structured outputs saved in {results_dir}")
```

Tool preparation:-

- **Flawfinder**:
    - The run_flawfinder function assembles and executes the command flawfinder --csv [project_path]. The --csv flag is crucial as it tells Flawfinder to produce a structured output.
    - Flawfinder's CSV output is sent to standard output. The script captures this directly and writes it into a corresponding .csv file in the results directory. It required minimal parsing as its native output format is already structured for easy use.

5

```python
#Flawfinder
def run_flawfinder(proj_path, proj_name, out_format="csv"):
    print(f"🔍 Running Flawfinder on {proj_name}")
    os.makedirs(results_dir, exist_ok=True)
    out_file = os.path.join(results_dir, f"{proj_name}_flawfinder.{out_format}")

    if out_format == "csv":
        cmd = ["flawfinder", "--csv", proj_path]
        stdout, stderr, rc = run_cmd_capture(cmd)
        if rc != 0:
            print(f"[!] Flawfinder failed: {stderr}")
            return []
        with open(out_file, "w") as f:
            f.write(stdout)
        return out_file
    elif out_format == "json":
        cmd = ["flawfinder", "--columns", proj_path]
        stdout, stderr, rc = run_cmd_capture(cmd)
        findings = []
        for line in stdout.splitlines():
            if "CWE-" in line:
                idx = line.find("CWE-")
                cwe_id = line[idx:idx+7]
                findings.append({
                    "Project_name": proj_name,
                    "Tool_name": "Flawfinder",
                    "CWE_ID": cwe_id,
                    "Is_In_CWE_Top_25": "Yes" if cwe_id in CWE_TOP_25 else "No"
                })
        export_json(out_file, findings)
        return out_file
```

- **Cppcheck:**
    - The run_cppcheck function constructs the command cppcheck --xml --enable=all --inconclusive --quiet [project_path]. The --xml flag is used to generate a detailed report, --enable=all ensures maximum rule coverage, and --inconclusive includes findings that are not certain.
    - Cppcheck uniquely writes its XML report to the standard error stream. The script captures this stderr content. The parsing logic, found in the run_cppcheck function, then uses Python's xml.etree.ElementTree.fromstring() to load the XML data. It iterates through all <error> elements in the XML tree and extracts the CWE identifier from the cwe attribute of each element.

6

```python
#Cppcheck
def run_cppcheck(proj_path, proj_name, out_format="xml"):
    print(f"🔎 Running Cppcheck on {proj_name}")
    os.makedirs(results_dir, exist_ok=True)
    out_file = os.path.join(results_dir, f"{proj_name}_cppcheck.{out_format}")

    cmd = ["cppcheck", "--xml", "--enable=all", "--inconclusive", "--quiet", proj_path]
    stdout, stderr, rc = run_cmd_capture(cmd)
    if rc != 0:
        print(f"[!] Cppcheck failed: {stderr}")
    with open(out_file, "w") as f:
        f.write(stderr)

    if out_format == "json":
        findings = []
        try:
            root = ET.fromstring(stderr)
            for error in root.iter("error"):
                cwe = error.attrib.get("cwe")
                if cwe:
                    cwe_id = f"CWE-{cwe}"
                    findings.append({
                        "Project_name": proj_name,
                        "Tool_name": "Cppcheck",
                        "CWE_ID": cwe_id,
                        "Is_In_CWE_Top_25": "Yes" if cwe_id in CWE_TOP_25 else "No"
                    })
            export_json(out_file, findings)
        except ET.ParseError:
            print(f"[!] Failed to parse Cppcheck XML for {proj_name}")

    return out_file
```

- **scan-build**:
    - The run_scanbuild function integrates with the project's build system using the command scan-build -o [report_dir] make -C [project_path] -j4. The -o flag specifies an output directory for the reports, and -C changes the working directory to the project's root before running make.
    - scan-build generates its findings in .plist (Property List XML) files within a timestamped subdirectory. The script's first step is to locate these files using glob.glob(os.path.join(report_dir, "**", "*.plist"), recursive=True).

7

Each found .plist file is then parsed with Python's plistlib.load(). Since scan-build does not provide a dedicated CWE field, a regular expression, re.search(r'CWE-\d+'), is applied to the human-readable description field of each diagnostic item to find and extract the associated CWE ID.

```
def run_scanbuild(proj_path, proj_name, out_format="json"):
    print(f"Running scan-build on {proj_name}")
    os.makedirs(results_dir, exist_ok=True)
    report_dir = os.path.join(results_dir, f"{proj_name}_scanbuild_report")
    out_file = os.path.join(results_dir, f"{proj_name}_scanbuild.{out_format}")

    #the -C flag tells make to change to the project directory before running so that the execution context remains clean
    cmd = ["scan-build", "-o", report_dir, "make", "-C", proj_path, "-j4"]
    stdout, stderr, rc = run_cmd_capture(cmd)

    if "scan-build: No bugs found." in stderr or "scan-build: No bugs found." in stdout:
        print(f"  [+] scan-build found no bugs in {proj_name}.")
        export_json(out_file, [])
        return out_file

    if rc != 0 and not glob.glob(os.path.join(report_dir, "**", "*.plist"), recursive=True):
        print(f"[!] scan-build failed for {proj_name}. Return code: {rc}")
        print(f"    STDOUT: {stdout.strip()}")
        print(f"    STDERR: {stderr.strip()}")
        return None

    print(f"  Parsing scan-build results for {proj_name}...")
    findings = []
    # scan-build creates a timestamped subdirectory, so we search recursively for plist files.
    plist_files = glob.glob(os.path.join(report_dir, "**", "*.plist"), recursive=True)

    for plist_file in plist_files:
        try:
            with open(plist_file, 'rb') as f:
                data = plistlib.load(f)
            for diag in data.get('diagnostics', []):
                description = diag.get('description', '')
                # Search for a CWE ID in the description text
                match = re.search(r'CWE-\d+', description)
                if match:
                    cwe_id = match.group(0)
                    findings.append({
                        "Project_name": proj_name,
                        "Tool_name": "scan-build",
                        "CWE_ID": cwe_id,
                        "Is_In_CWE_Top_25": "Yes" if cwe_id in CWE_TOP_25 else "No"
                    })
        except Exception as e:
            print(f"[!] Error parsing plist file {plist_file}: {e}")

    if out_format == "json":
        export_json(out_file, findings)

    return out_file
```

The outputs for the tools are in this format:-

1. Flawfinder: CSV Output

Flawfinder was executed with the --csv flag, instructing it to produce a comma-separated

8

values (CSV) report.

Explanation:

- File, Line, Column: Pinpoints the exact location of the potential weakness in the source code.
- Level: Assigns a risk level (1-5, with 5 being the highest risk).
- Category and Name: Classifies the type of function or issue (e.g., race, chmod).
- Warning: Provides a detailed, human-readable description of the vulnerability. This is often where the CWE is mentioned contextually.
- CWEs: This is the most crucial column for automated analysis. It provides a direct, parseable CWE identifier for the finding. In the first example row, it explicitly lists CWE-362, indicating a potential Race Condition.
- Context: Shows the actual line of code that triggered the warning.

2. Cppcheck: XML Output

The entire report is enclosed in <results> tags, with each finding detailed within an <error> tag.

Explanation:

The XML structure provides clear attributes for each finding:

- <error> tag: Represents a single issue detected by the tool.
- id: A Cppcheck-specific identifier for the type of error (e.g., integerOverflow).
- severity: Classifies the issue's importance (e.g., warning, style, information).
- msg: A concise, human-readable description of the problem.
- cwe: The key attribute for our analysis. It provides the numerical CWE code directly. In the first example, it identifies CWE-190, a classic Integer Overflow vulnerability.
- <location> tag: Specifies the exact file and line number of the issue.

We use an XML parser to iterate through each <error> element and extract the value of

the cwe attribute, which is then standardized (e.g., 190 becomes CWE-190).

3. scan-build (Clang Static Analyzer): Property List (.plist) Output

scan-build generates its primary machine-readable output in the form of .plist files. This is an XML-based format that stores data in key-value pairs, nested within dictionaries and arrays.

Explanation:

The .plist format is highly structured but requires a specific parser (plistlib in Python).

- <dict>: A dictionary or a mapping of key-value pairs.
- <key>: The name of a property (e.g., description, category).
- <string>: The value associated with the preceding key.
- <array>: A list of items. The main findings are stored in an array under the diagnostics key.
- description: It contains a detailed, human-readable explanation of the bug. Crucially, the Clang Static Analyzer embeds the relevant CWE directly into this string.

During the setup and execution of multiple static analysis tools for CWE-based vulnerability detection, several challenges were encountered:

1. Many vulnerability detection tools (e.g., *Polyspace Bug Finder*, *Helix QAC*, *LDRA*, *TrustInSoft Analyzer*) are commercial and require separate licenses. This limited the analysis to open-source tools like **Cppcheck**, **Flawfinder**, and **Scan-Build**. Additionally, configuring these tools to run seamlessly on macOS required adjustments in command-line execution, environment variables, and compiler path settings.
2. Tools such as Scan-Build depend on successful compilation through clang or cmake. Several repositories failed during CMake configuration due to missing source files or dependencies which prevented the tool from performing a full analysis. These failures needed manual inspection and, in some cases, partial builds.
3. Among the tested tools, only few of them explicitly support CWE-based

vulnerability reporting.

4. For many open-source repositories, Scan-Build often reported zero issues even though other tools flagged multiple vulnerabilities. This was primarily due to limited coverage of default checkers or repositories with high-quality code that complied with modern safety standards.

5. Each tool produces results in a different format (text, JSON, XML, CSV). Consolidating outputs into a uniform structure required extensive parsing logic and post-processing scripts.

6. Some tools bundled with macOS (like Clang and LLVM) were slightly outdated, lacking advanced analysis capabilities found in newer releases.

After running each tool on a project, the script generates a final csv file, which is then used for the subsequent results and analysis section. This consolidated file includes the project name, tool name, CWE ID, the number of findings for that CWE, and a flag indicating if the CWE is in the Top 25 list.

(d) Tool-level CWE Coverage Analysis & Pairwise Agreement (IoU) Analysis::

Once I had the consolidated findings, I had to then analyse those findings, which I did using pandas and matplotlib library. I processed the csv file to compute two primary metrics: CWE Top 25 Coverage and Pairwise Intersection over Union (IoU).

For the data preprocessing, I used the normalize_cwe function, which standardizes the format of CWE identifiers (e.g., converting 'cwe-79' to 'CWE-79') to ensure consistency for accurate comparison.

```
def normalize_cwe(cwe: str) -> str:
    # Normalize CWE IDs (e.g., 'cwe-79 ' -> 'CWE-79')
    if pd.isna(cwe):
        return ""
    s = str(cwe).strip().upper()
    if not s.startswith("CWE-") and s.isdigit():
        s = f"CWE-{s}"
    return s
```

The compute_top25_coverage function calculates the breadth of each tool's detection

capabilities. For each tool, it computes:

1. The total number of unique CWEs detected.
2. The number of these unique CWEs that are part of the official 2024 CWE Top 25 list.
3. The coverage percentage, representing what proportion of a tool's unique findings are considered critical, high-impact weaknesses.

```
def compute_top25_coverage(df: pd.DataFrame) -> pd.DataFrame:
    top25_col = "Is_In_CWE_Top_25?"
    top25_bool = df[top25_col]
    if df[top25_col].dtype == object:
        top25_bool = df[top25_col].astype(str).str.strip().str.lower().isin({"1", "true", "yes", "y"})
    df_top = df.copy()
    df_top[top25_col] = top25_bool

    dfp = df_top[df_top["Number of Findings"] > 0].copy()
    uniq_counts = dfp.groupby("Tool_name")["CWE_ID"].nunique().rename("Unique_CWEs")

    top25 = dfp[dfp[top25_col]]
    top25_counts = top25.groupby("Tool_name")["CWE_ID"].nunique().rename("Top25_CWEs")

    out = pd.concat([uniq_counts, top25_counts], axis=1).fillna(0).reset_index()
    out["Top25_CWEs"] = out["Top25_CWEs"].astype(int)
    out["Top25_Coverage(%)"] = np.where(out["Unique_CWEs"] > 0,
                                        100.0 * out["Top25_CWEs"] / out["Unique_CWEs"],
                                        0.0)
    return out.sort_values(by="Top25_Coverage(%)", ascending=False)
```

I then saved this as a new CSV file and generated bar charts to visualize the coverage percentages for each tool.

For the Pairwise Agreement (IoU) Analysis, I calculated the Intersection over Union (IoU) metric by the iou_between_sets function. IoU measures the similarity between two sets of data (in this case, sets of unique CWEs detected by two tools). A value of 1.0 indicates identical findings, while 0.0 means no overlap. I performed this analysis at two levels:

a) **Overall IoU**: It computes a single IoU matrix comparing every pair of tools based on all CWEs found across all three projects. This gives a general sense of how similar the tools are.

b) **Per-Project IoU**: The compute_iou_per_project function generates separate IoU matrices for each of the three repositories. This helps in analysis of whether tool agreement changes depending on the codebase being analyzed. Each IoU matrix is saved as a CSV file and visualized as a heatmap for easy interpretation.

12

```python
def compute_iou_matrix(tool_sets: Dict[str, Set[str]]) -> pd.DataFrame:
    tools = sorted(tool_sets.keys())
    mat = np.zeros((len(tools), len(tools)), dtype=float)
    for i, ti in enumerate(tools):
        for j, tj in enumerate(tools):
            mat[i, j] = iou_between_sets(tool_sets[ti], tool_sets[tj])
    return pd.DataFrame(mat, index=tools, columns=tools)

def compute_iou_per_project(df: pd.DataFrame) -> Dict[str, pd.DataFrame]:
    out = {}
    for proj, sub in df.groupby("Project_name"):
        tool_sets = compute_tool_sets(sub)
        if len(tool_sets) >= 2:
            out[proj] = compute_iou_matrix(tool_sets)
    return out
```

RESULTS AND ANALYSIS

Outputs

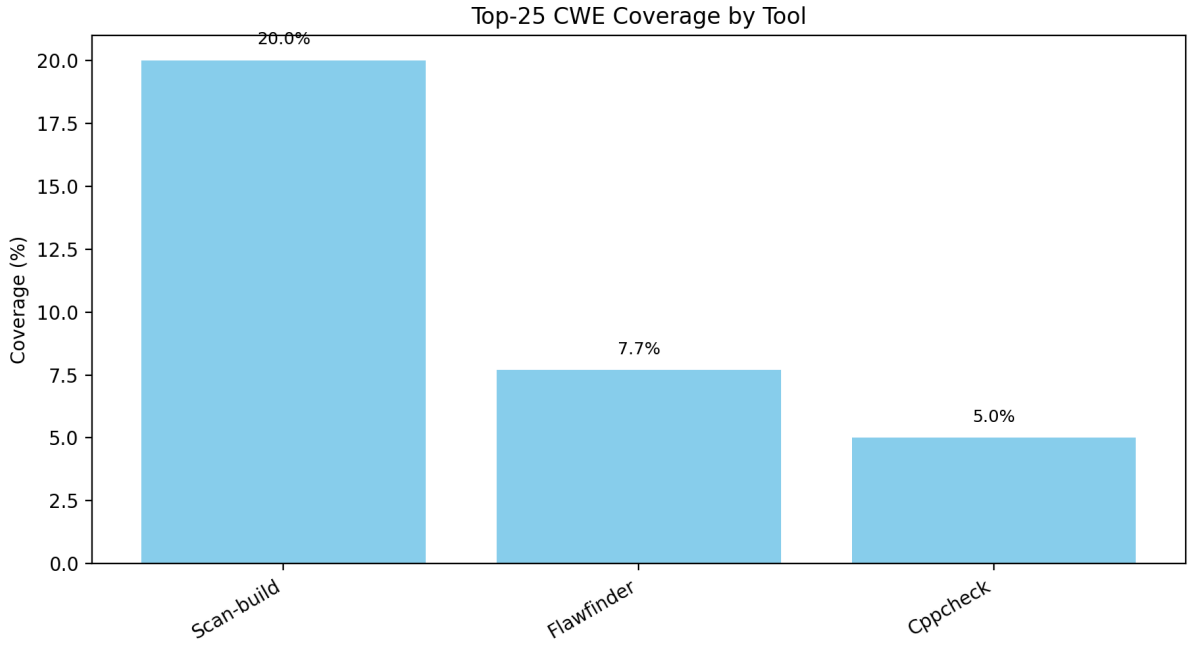
A) Pairwise Findings:-

The first few rows of the csv (The complete csv file link has been added in the reference section) :-

Project_name	Tool_name	CWE_ID	Number of Findings	Is_In_CWE_Top_25?
libgit2	Cppcheck	CWE-195	3	No
libgit2	Cppcheck	CWE-398	546	No
libgit2	Cppcheck	CWE-401	1	No
libgit2	Cppcheck	CWE-457	33	No
libgit2	Cppcheck	CWE-467	1	No
libgit2	Cppcheck	CWE-476	26	Yes
libgit2	Cppcheck	CWE-561	4439	No
libgit2	Cppcheck	CWE-562	4	No
libgit2	Cppcheck	CWE-563	75	No
libgit2	Cppcheck	CWE-570	51	No
libgit2	Cppcheck	CWE-571	40	No
libgit2	Cppcheck	CWE-628	131	No
libgit2	Cppcheck	CWE-664	4	No
libgit2	Cppcheck	CWE-682	1	No
libgit2	Cppcheck	CWE-683	1	No
libgit2	Cppcheck	CWE-686	36	No
libgit2	Cppcheck	CWE-758	4	No
libgit2	Flawfinder	CWE-119	420	No
libgit2	Flawfinder	CWE-120	486	No
libgit2	Flawfinder	CWE-126	932	No
libgit2	Flawfinder	CWE-134	29	No

B) Tool-level CWE Coverage Analysis:-

14



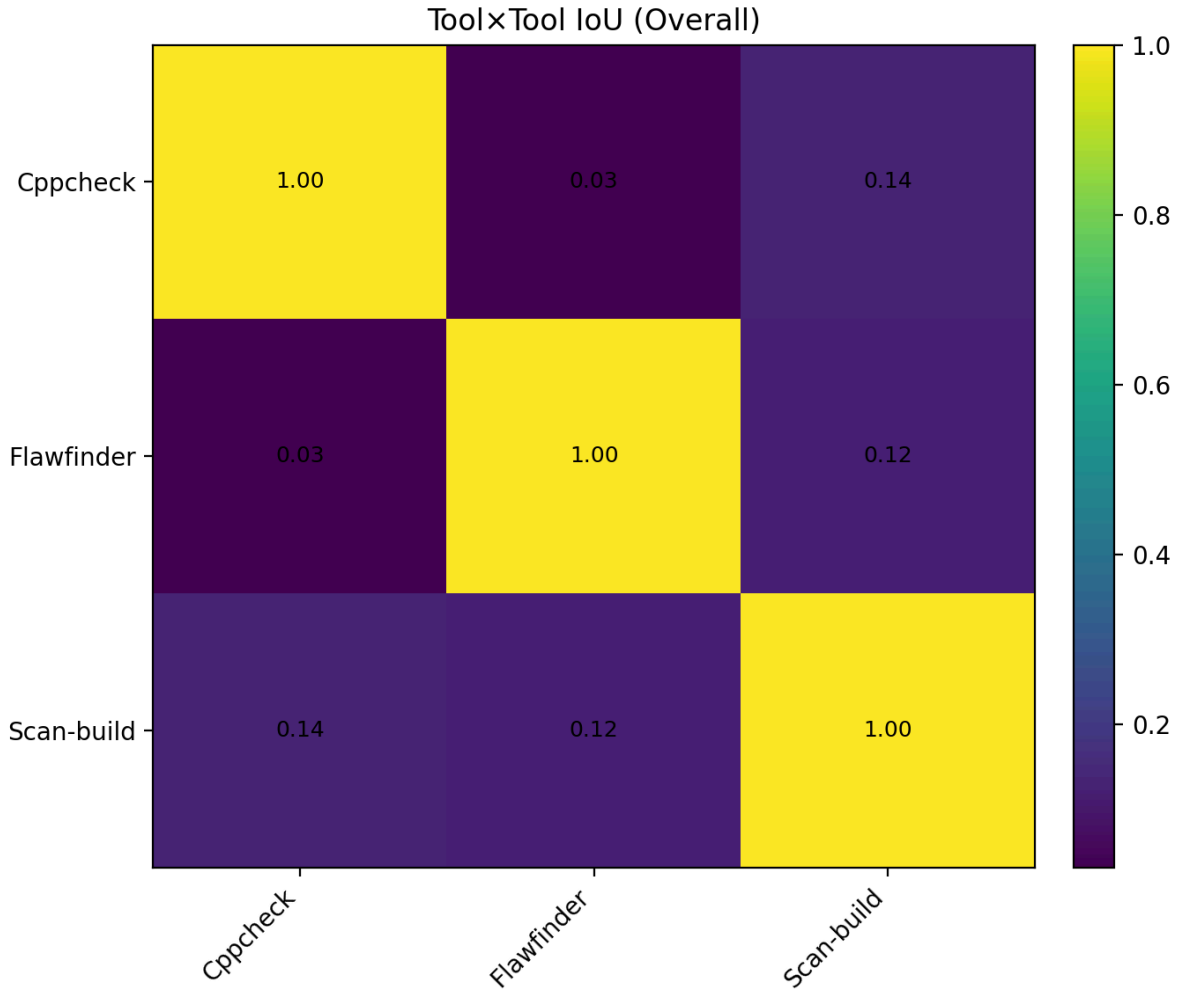
Tool_name	Unique_CWEs	Top25_CWEs	Top25_Coverage(%)
Scan-build	5	1	20.0
Flawfinder	13	**1**	7.7
Cppcheck	20	1	5.0

The table and the bar chart above provide two different perspectives on tool performance. Cppcheck clearly demonstrates the greatest **breadth**, identifying the highest number of unique CWEs (20), which is represented by the total height of its bar. In contrast, **Scan-build** shows the greatest **focus** on critical vulnerabilities. Despite finding the fewest unique CWEs overall (5), 20% of its findings belong to the CWE Top 25.

This analysis highlights an important trade-off. Cppcheck acts as a wide net for a large number of general code quality and security issues, whereas Scan-build is more of a targeted tool for a smaller, potentially higher-impact, set of flaws. Cppcheck is the most

comprehensive tool in terms of the variety of unique CWEs it detects (20 total). However, all three tools are equal in their ability to find at least one type of high-priority vulnerability from the Top 25 list in this specific analysis.

C) Pairwise Agreement (IoU) Analysis (Overall):



The values represent the Jaccard Index calculated from the CWEs found across all projects combined.

Interpreting the Matrix:

The Jaccard Index (IoU) measures the similarity between two sets. The formula is:

IoU(A, B) = |A ∩ B| / |A ∪ B|

(Size of the Intersection of A and B) / (Size of the Union of A and B)

- A value of 1.0**(Brightest Color)** means the sets are identical.
- A value of 0.0**(Darkest Color)** means the sets have no common elements.
- A low value (like those in the matrix) signifies that the tools are highly diverse in their findings.

Key Insights:

	Cppcheck	Flawfinder	Scan-build
Cppcheck	1.0	0.031	0.136
Flawfinder	0.031	**1.0**	0.125
Scan-build	0.136	0.125	1.0

1. **High Diversity**: All off-diagonal values in the matrix are very low (the highest is 0.136). This is the most critical insight: the tools are not redundant. Each tool specializes in finding a different set of vulnerabilities. Running only one of them would cause you to miss the majority of issues detected by the others.
2. **Lowest Overlap**: The pair with the lowest similarity is Cppcheck and Flawfinder (IoU = 0.031). They are almost entirely complementary, meaning they excel at finding completely different types of weaknesses.
3. **Highest Overlap**: The pair with the most similarity is Cppcheck and Scan-build (IoU = 0.136), which is still low. This suggests they have a slightly better, yet still small, common ground in the types of CWEs they can both detect.

Which Tool Combination Maximizes CWE Coverage?

To maximize coverage, you should combine the tools that are **most diverse** (i.e., have the

lowest IoU score) as we need to cover the widest possible range of unique vulnerabilities with minimal redundancy.

Based on the IoU matrix:

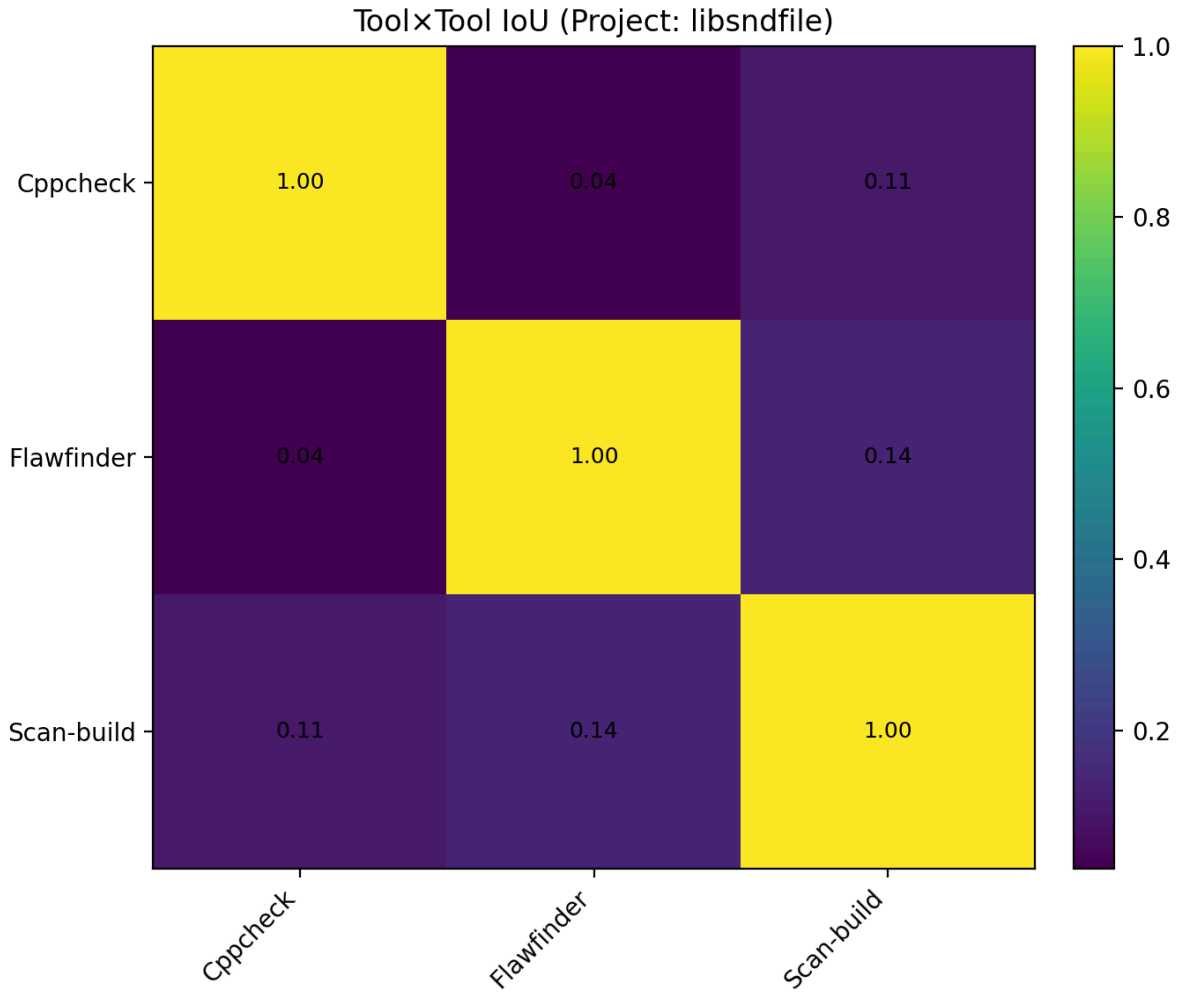
1. The Best Two Tool Combination is Cppcheck and Flawfinder. These two have the lowest IoU (0.031), meaning they are the most different from each other. Using them together provides the broadest coverage because their findings barely overlap.
2. The optimal Strategy is to use all three tools (Cppcheck + Flawfinder + Scan-build). Since all the IoU values are low, each tool contributes significantly to the total pool of unique vulnerabilities found and so combining all the three ensures the most comprehensive analysis possible.

Takeaways of the Analyses

1. There is no single best static analysis tool. Cppcheck finds the most variety of CWEs, but it doesn't find everything. The other tools found vulnerabilities that Cppcheck missed.
2. The primary takeaway is the strength of using a multi-tool approach. The low Jaccard Index scores prove that these SAST tools are complementary, not interchangeable. Relying on a single tool provides a false sense of security.
3. While Cppcheck offered the greatest breadth, all three tools were capable of finding Top 25 CWEs. This suggests that even tools with a smaller overall detection range (like Scan-build in this case, which found only 5 unique CWEs) can still provide critical security value.

D) Repository specific results:-

1) libsndfile:-



The results for libsndfile show a very low degree of overlap among all tools, highlighting their diversity.

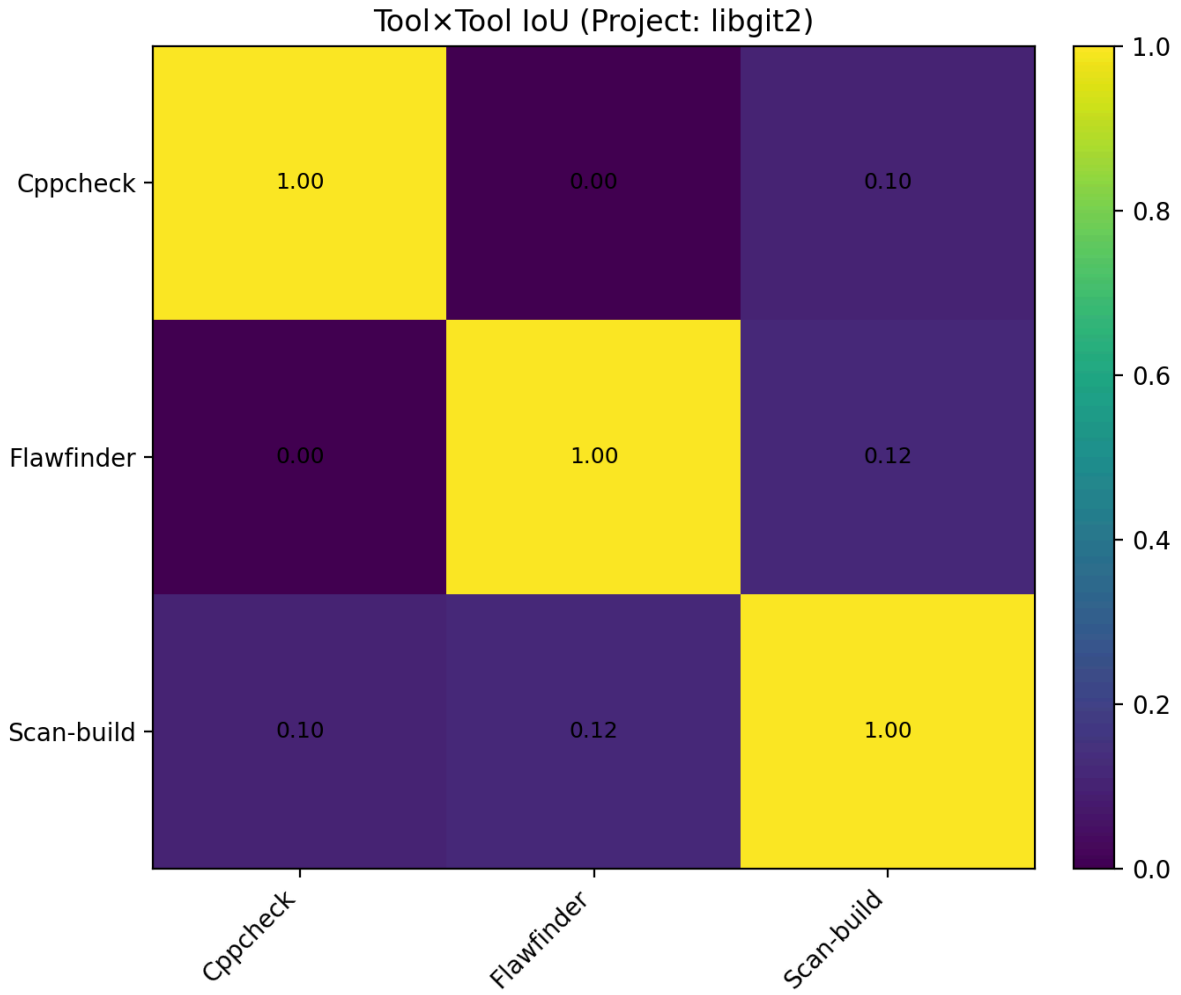
1. Cppcheck vs. Flawfinder (IoU ≈ 0.038): This is the lowest similarity score for this project. It means that the set of unique vulnerabilities found by Cppcheck and the set found by Flawfinder were almost completely different.

2. Cppcheck vs. Scan-build (IoU ≈ 0.105): There is a slightly higher but still very small overlap between these two tools. They share a minor common ground in the types of CWEs they can detect.

3. Flawfinder vs. Scan-build (IoU ≈ 0.143): This pair has the highest similarity in the libsndfile analysis. However, an IoU of 0.143 is still extremely low.

	Cppcheck	Flawfinder	Scan-build
Cppcheck	1.000	0.038	0.105
Flawfinder	0.038	**1.000**	0.143
Scan-build	0.105	0.143	1.000

2) libgit2:-

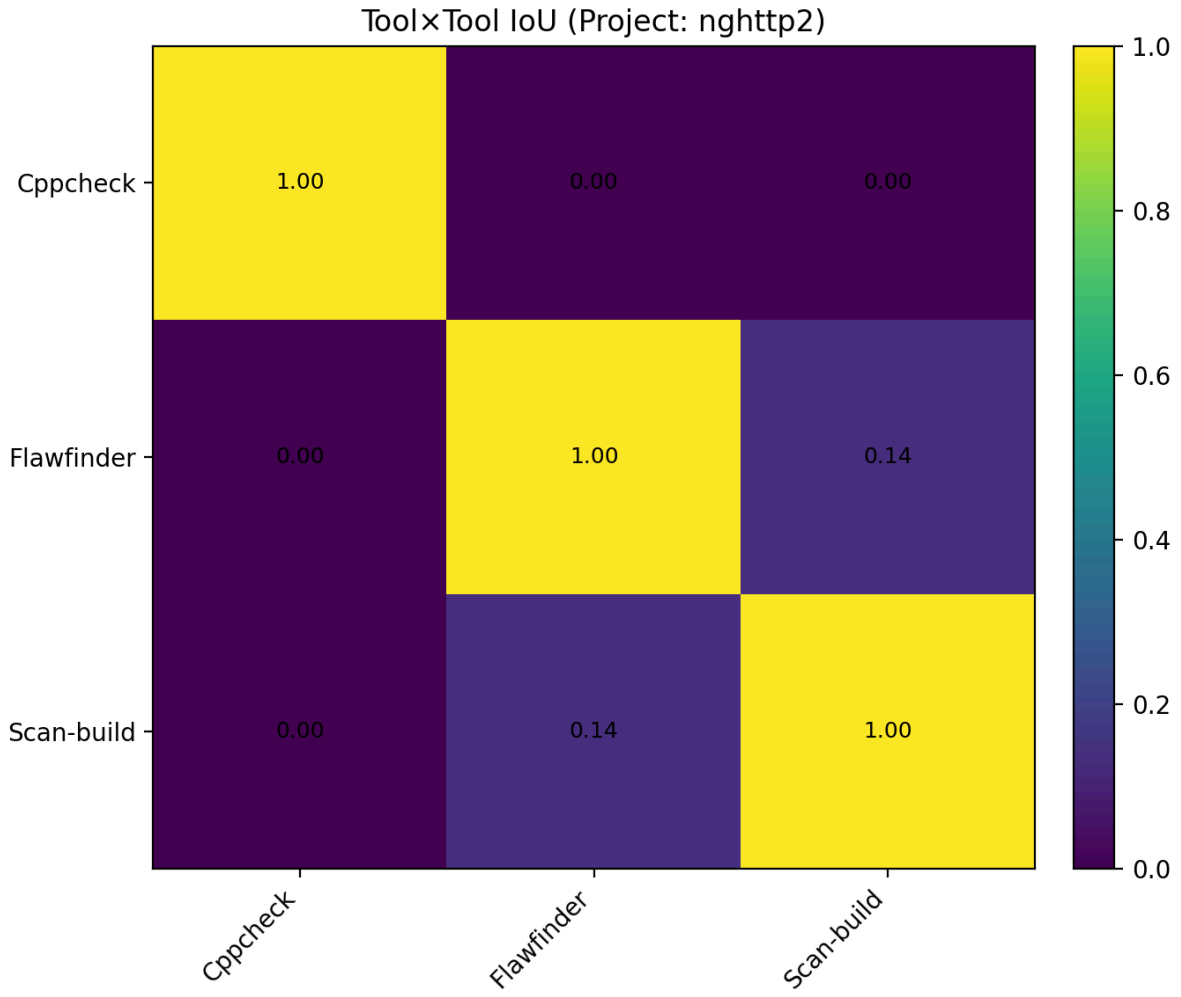


20

1. Cppcheck vs. Flawfinder (IoU = 0.000): This is the most significant finding as a score of zero means there was no overlap between the vulnerabilities found by Cppcheck and those found by Flawfinder.
2. Cppcheck vs. Scan-build (IoU = 0.100):- A small overlap exists, indicating they share a common capability to find at least one type of CWE present in this project.
3. Flawfinder vs. Scan-build (IoU = 0.125): Similar to the other pair, this demonstrates low similarity. These tools also found mostly different vulnerabilities.

	Cppcheck	Flawfinder	Scan-build
Cppcheck	1.000	0.000	0.100
Flawfinder	0.000	**1.000**	0.125
Scan-build	0.100	0.125	1.000

3) ngtttp2:-



The nghttp2 results are perhaps the most striking, showing that one tool's findings were completely unique.

1. Cppcheck vs. Flawfinder (IoU = 0.000) and Cppcheck vs. Scan-build (IoU = 0.000):- The vulnerabilities detected by Cppcheck had zero overlap with the findings of either Flawfinder or Scan-build. Hence, for nghttp2, Cppcheck is an important tool as it identifies a set of issues that were completely invisible to the other two analyzers.

2. Flawfinder vs. Scan-build (IoU ≈ 0.143):- This is the only pair of tools with any overlap at all for this project. They share a small common ground, but are still largely diverse.

	Cppcheck	Flawfinder	Scan-build
Cppcheck	1.000	0.000	0.000
Flawfinder	0.000	**1.000**	0.143
Scan-build	0.000	0.143	1.000

CONCLUSION

Challenges

The process of analyzing C/C++ repositories for CWE-based vulnerabilities highlighted both the potential and limitations of static analysis tools. While tools like Cppcheck and Flawfinder provided reliable CWE mappings, other widely-used tools such as Scan-Build and Clang-Tidy often reported few or no issues which highlights the dependency of results on tool capabilities and configuration.

A key challenge was navigating tool compatibility, output inconsistencies, and repository-specific build requirements. Some repositories failed to build under certain tools, while others required manual adjustments for the analysis to proceed. Parsing and consolidating results into structured formats demanded careful scripting and attention to detail.

Reflections

Reflecting on the experience, it became evident that selecting the right tools for vulnerability analysis is crucial. Open-source tools offer accessibility but may have limitations in coverage or CWE mapping, whereas commercial tools promise comprehensive detection but they require license. Additionally, working with multiple tools highlighted the importance of cross-verification, as relying on a single tool may overlook critical vulnerabilities.

Lessons Learned

The key lessons learned from this project are:

1. Understanding the strengths, limitations, and output formats of tools before integrating them into an analysis pipeline is important.
2. Scripts must handle different output formats and error conditions to ensure smooth data consolidation.
3. Comparing outputs from multiple tools helps in capturing a broader set of vulnerabilities and increases confidence in the results.
4. Licensing, environment setup, and repository dependencies can significantly affect the feasibility and coverage of vulnerability detection.

Hence, in this lab, we got to know the importance of systematic tool evaluation and careful planning in conducting reliable vulnerability assessments.

REFERENCES

1) https://drive.google.com/file/d/195OJgPdnY7ixEyMarA5ZdkL7tVR8deG_/view
2) https://cwe.mitre.org/
3) https://en.wikipedia.org/wiki/Jaccard_index
4) Github repo:- https://github.com/zainabkapadia52/Evaluation-of-Vulnerability-Analysis-Tools-using-CWE-based-Comparison
5) CSV file:- https://drive.google.com/file/d/1sVGUY1Qa5yizhmCrr7buHC3ZeCc1N-hW/view?usp=sharing

Chapter 2

Lab 7 - Reaching Definitions Analyzer for C Programs

2.1 Introduction

The aim of this laboratory was to provide hands-on experience with the program analysis techniques. This laboratory focuses on implementation of automated Control Flow Graph (CFG) generation from C programs, and performing Reaching Definitions (RD) analysis on the generated CFGs. The RD analysis helps in understanding how variable definitions propagate through the program, which is crucial for various applications such as optimization, debugging, and code comprehension.

2.2 Tools

- **Programming Language:** Python 3.12.9 — Used for code and using pylint library.
- **Editor/IDE:** Visual Studio Code — Used for coding, debugging and execution.
- **Version Control:** Git and GitHub — Used to track and the changes in the code, and to improve maintainability of the codebases.
- **Gcc compiler:** Gcc version 14.2.0 — Used to compile and run C programs.
- **Virtual Environment:** venv — To prevent library version conflicts by isolating working environments.

2.3 Setup

To configure GitHub for this project, I had to setup virtual environment on my machine and Visual Studio Code (VS Code) as the code editor.

2.4 Program corpus Selection

The three programs chosen are written in C language.

They satisfied the following criteria:

- The programs are small to medium-sized, with a manageable number of lines of code (between 200 to 300 lines).
- The programs contains a single main function as mentioned in the question and also the file is the only standalone source file for the ease of analysis.
- Chosen program contain conditionals like if-else, loops like for, while.
- The programs include variable assignments and definitions that can be tracked through the RD analysis.

2.5 Control Flow Graphs

A Control Flow Graph (CFG) is a graphical representation of all paths that might be traversed through a program during its execution. In a CFG, nodes represent basic blocks (a sequence of consecutive statements with a single entry point and a single exit point), and directed edges represent the flow of control between these blocks.

Rules to find leaders:

- The first statement is a leader.
- Any statement that is the target of a conditional or unconditional jump is a leader.
- Any statement that immediately follows a conditional or unconditional jump is a leader.

METHODOLOGY AND EXECUTION

Gcc compiled and executed three C codes. Following pipeline was used to generate CFG and generate other information.

- **Detect leaders:**
  - Parsing the entire code and extracting the lines to categorize them as leaders.
  - Using the detected leaders and the ending points through the codes, basics blocks were build each corresponding to the node of the CFG.
  - Using the basic blocks generated, edges were added to connect these blocks based on the control flow. Fall through edges wered added when the blocks did not end with return, break or continue. Two way edges wered added for if, while and for headers.
  - Based on the blocks and edges, Cyclomatic complexity was generated.

- For reaching definitions, enumerated through the assignments and computed gen(B) and kill(B). Using this iterated until convergence for the forward flow.

- **Defining dictionary to map various keywords**

```
p = {
    "label": re.compile(r'^\s*([A-Za-z_]\w*)\s*:\s*(?!:)'),
    "case": re.compile(r'^\s*case\b[^:]*:\s*$'),
    "default": re.compile(r'^\s*default\s*:\s*$'),
    "iff": re.compile(r'^\s*if\s*\('),
    "else": re.compile(r'^\s*else\b'),
    "for": re.compile(r'^\s*for\s*\('),
    "while": re.compile(r'^\s*while\s*\('),
    "do": re.compile(r'^\s*do\s*(?:\{)?\s*$'),
    "goto": re.compile(r'\bgoto\s+([A-Za-z_]\w*)\s*;'),
    "break": re.compile(r'^\s*break\s*;'),
    "cont": re.compile(r'^\s*continue\s*;'),
    "ret": re.compile(r'^\s*return\b'),
    "func": re.compile(r'^\s*(?:[A-Za-z_]\w*\s+)*[A-Za-z_]\w*\s*\([^;]*\)\s*\{?\s*$'),
}
```

Figure 2.1: Defining keywords for leader identification

  - **Explanation of the dictionary:**
    - This snippet defines a dictionary for compiled regex patterns.
    - These regex patterns are used to recognize the keywords and the flow in the code to identify the leaders and the endings of the block which are required to form the nodes and hence the Control Flow Graphs.

- **Support functions**
  - **Explanation of support functions**
    - readfile in the basic function which takes the code as the input reading it as the text file in UTF8 text and returning it as string.
    - structural_only is the function which is used to find the parts in the code which are actually executable by checking the content inside the brackets.
    - next_stmt is the function which is used to find the next realexecutable statement the in the codes after current line using the structural_only code.
    - first_stmt_in_bock is the function which finds the first executable statement after the start line.

- **This is the main function to identify the leader lines from the code**
- This function is to find all the indices that represent the entry point of logical blocks and statements.
  - **Explanation of Code for finding leaders:**

```
def readfile(fn):
    with open(fn, "r", encoding="utf-8", errors="ignore") as f:
        return f.read()

def structural_only(s):
    t = s.strip()
    return t in ("{","}","};")

def next_stmt(lines, j):
    k = j + 1
    while k < len(lines):
        t = lines[k]
        if t.strip() and not t.lstrip().startswith("#") and not structural_only(t):
            return k
        k += 1
    return None

def first_stmt_in_block(lines, start_line):
    k = start_line
    if "{" not in lines[k]:
        while k < len(lines) and "{" not in lines[k]:
            k += 1
    k += 1
    while k < len(lines):
        t = lines[k]
        if t.strip() and not t.lstrip().startswith("#") and not structural_only(t):
            return k
        k += 1
    return None
```

Figure 2.2: Support functions

* The code is passed as the input to the function leader, and the code is split into lines stored in the variable lines. ls is initialized set to store unique leaders line indices. labels is a dictionary to map label names to their line indices.
* In the first step we enumerate thru all the lines, and check if any of the lines has a match with withe function header using regex and then first_stmt_in_block function is called to find the first executable statement in the function block and add it to the set of leaders.
* Then we find teh first executable statement in the code and add it to the set of leaders.
* Next we check if it matches a label, and if it does, we record its position marking it as the leader.
* Again the code iterates throught some predefined header_keys and checks if the lines contains any construct like iff, else, for, while or do, and if it does, its marked as the leaders. After that, we iterate further using the function to find the next executable statement and add it to the ls set.
* After keywords like break, continue or return, the next statement can be a flow entry point, so again added to the set ls.
* After closing brackets, the next statemet could be a leader, so added to the set ls. This is followed by goto statement, where the target label is marked as the leader.

```
def leaders_from_src(src):
    lines = src.splitlines()
    ls = set()
    labels = {}

    for i, s in enumerate(lines):
        if p["func"].match(s):
            ls.add(i)
            k = first_stmt_in_block(lines, i)
            if k is not None:
                ls.add(k)

    for i, s in enumerate(lines):
        if s.strip() and not s.lstrip().startswith("#"):
            ls.add(i)
            break

    for i, s in enumerate(lines):
        m = p["label"].match(s)
        if m:
            labels[m.group(1)] = i
            ls.add(i)
        if p["case"].match(s) or p["default"].match(s):
            ls.add(i)

    header_keys = ("iff","else","for","while","do")
    for i, s in enumerate(lines):
        if any(p[k].search(s) for k in header_keys):
            ls.add(i)
            k = next_stmt(lines, i)
            if k is not None:
                ls.add(k)

        if p["break"].search(s) or p["cont"].search(s) or p["ret"].search(s):
            k = next_stmt(lines, i)
            if k is not None:
                ls.add(k)

        if s.strip() == "}":
            k = next_stmt(lines, i)
            if k is not None:
                ls.add(k)

        m = p["goto"].search(s)
        if m and m.group(1) in labels:
            ls.add(labels[m.group(1)])

    idxs = sorted(ls)
    return idxs, lines
```

Figure 2.3: Distribution of number of fix types.

∗ Finally, the set ls is returned as the output of the function after being sorted.

- **Generation of basic blocks from the leaders identified**

```
def nonstruct(s):
    t = s.strip()
    return t and t not in ("{","}","};") and not t.startswith("#")

def build_blocks_from_leaders(lines, leaders):
    leaders = sorted(set(leaders))
    leaders.append(len(lines))
    blocks = []
    for idx in range(len(leaders)-1):
        start = leaders[idx]
        end = leaders[idx+1]
        s = start
        while s < end and not nonstruct(lines[s]):
            s += 1
        e = end
        while e > s and not nonstruct(lines[e-1]):
            e -= 1
        if s < e:
            blocks.append((s, e))
    return blocks
```

Figure 2.4: Generating basic blocks.

  - **Explanation of Code for building basic blocks:**
    * nonstruct functions check if the code line is a real statement and not just structure or a comment
    * build_blocks_from_leaders function takes the code lines and the identified leaders as input. These blocks are basic ones with Continuous lines without jump.
    * The leaders are then sorted and duplicates are removed.
    * For every pair of consecutive leader indices, start variable holds the index of this starting of the block and end variable for the end of the block.
    * Then by iteration from the beginning and from the end of the block, we find the first and last executable statements in the block using the nonstruct function.
    * This is added to the blocks and returned.
- **Generation of control flow graphs**
- This part of the code uses regular expressions and code block analysis to detect the flow from one basic block to another, thereby constructing a control flow graph (CFG) of the program.
  - **Explanation of Code for building control flow graphs:**
    * We define regex patters to identify various control flow constructs like if, else, for, while, do, break, continue, return and goto.

```
is_branch_hdr = re.compile(r'^\s*(if|while|for)\b')
is_return     = re.compile(r'^\s*return\b')
is_break      = re.compile(r'^\s*break\s*;')
is_continue   = re.compile(r'^\s*continue\s*;')

def nonstruct(s):
    t = s.strip()
    return t and t not in ("{","}","};") and not t.startswith("#")

def last_stmt_line(lines, s, e):
    for k in range(e - 1, s - 1, -1):
        if nonstruct(lines[k]):
            return k
    return None

def build_cfg_edges(lines, blocks):
    edges = []
    names = [f"B{i}" for i in range(len(blocks))]
    for i, (s, e) in enumerate(blocks):
        name = names[i]
        k = last_stmt_line(lines, s, e)
        fall = True
        branch2 = False
        if k is not None:
            tail = lines[k]
            if is_return.match(tail) or is_break.match(tail) or is_continue.match(tail):
                fall = False
            elif is_branch_hdr.match(tail):
                branch2 = True
        if fall and i + 1 < len(blocks):
            edges.append((name, names[i + 1], "fall"))
        if branch2:
            if i + 1 < len(blocks):
                edges.append((name, names[i + 1], "true"))
            if i + 2 < len(blocks):
                edges.append((name, names[i + 2], "false"))
    return names, edges

def cfg_metrics(num_blocks, edges):
    N = num_blocks
    E = len(edges)
    CC = E - N + 2
    return N, E, CC
```

Figure 2.5: Generating CFG.

2.5.1 Regex patterns

* The construction of edges in the Control Flow Graph (CFG) is based on the analysis of control statements and block boundaries. For each basic block, the outgoing edges are determined by the following rules:
  - **is_branch_hdr:** Matches any line starting with if, while, or for. Used to find branching statements.
  - **is_return:** Matches any line with a return statement, typically an exit point.
  - **is_break:** Matches any line with a break; statement, found in loopsswitches.
  - **is_continue:** Matches any line with a continue; statement, found in loops to skip to next.
* nonstruct function check if the code line is a real statement and not just structure or a comment

    * last_stmt_line finds the last real, non-structural code statement within a block.
    * build_cfg_edges is the main function used to generate the edges of the CFG based on blocks and lines of code.
    * Each block is given a name from B0, B1 and so on.
    * For every block, we find the last executable statement using last_stmt_line function and its index is assigned to the variable k.
    * If this last line is not a return, break or continue statement, we add a fall-through edge to the next block.
    * If the last line is a branching header (if, while, for), if true then, jump to next block and if false, jump to the block after that.
    * cfg_metrics is used to calculate Cyclomatic complexity using the formula E - N + 2P, where E is number of edges, N is number of nodes and P is number of connected components.

- **Reaching Definitions Analysis**
  - **Support Functions for Reaching definitions analysis:**
    * assign_pat is a regular expression that matches assignments to variables, and is used to find where a variable gets a new value.
    * nonstruct is a code which is to check if a code is a real statement, not just a structural line.
    * block_names is a function that generates symbolix names for each code block.
    * collect_definitions finds all variable definitions within each block of code. For every block, it splits the compound using statement ; and it checks for assignment using the function assign_pat. If a match is found, it records the variable name and the line number where it was defined. Reconfirms by checking not structural or comment line and then it records tuple.
    * Returns a list of all variables definitions found in the blocks.
    * geb_kill_per_block checks for each block, which assignments were killed in that block. It builds a set of killed definitions assignment to the same variable in other blocks.
    * preds is the final function, which finally generates the edges in the graph.
  - **Reaching Definitions:**
    * Each block is assigned a symbolic name like B0, B1 and so on.

```
assign_pat = re.compile(r'\b([A-Za-z_]\w*(?:\[[^\]]+\])?)\s*=')

def nonstruct(s):
    t = s.strip()
    return t and t not in ("{","}","};") and not t.startswith("#")

def block_names(blocks):
    return [f"B{i}" for i in range(len(blocks))]

def collect_definitions(lines, blocks):
    defs = []
    names = block_names(blocks)
    for bi,(s,e) in enumerate(blocks):
        bname = names[bi]
        for li in range(s, e):
            sline = lines[li]
            if not nonstruct(sline):
                continue
            parts = [p for p in sline.split(';') if p.strip()]
            for stmt in parts:
                if '=' in stmt and '==' not in stmt:
                    m = assign_pat.search(stmt)
                    if m:
                        var = m.group(1).strip()
                        defs.append((None, var, li, stmt.strip(), bname))
    out = []
    for k,(nid,var,li,txt,bname) in enumerate(defs, start=1):
        out.append((f"D{k}", var, li, txt, bname))
    return out

def gen_kill_per_block(defs, blocks):
    names = block_names(blocks)
    all_by_var = defaultdict(set)
    for Dk,var,li,txt,b in defs:
        all_by_var[var].add(Dk)
    gen = {b:set() for b in names}
    for Dk,var,li,txt,b in defs:
        gen[b].add(Dk)
    kill = {b:set() for b in names}
    for b in names:
        vars_in_b = set(var for (Dk,var,li,txt,bb) in defs if bb == b)
        kset = set()
        for v in vars_in_b:
            kset |= (all_by_var[v] - gen[b])
        kill[b] = kset
    return gen, kill

def preds(edges):
    P = defaultdict(set)
    for u,v,lbl in edges:
        P[v].add(u)
    return P
```

Figure 2.6: Support functions.

```
def reaching_definitions(blocks, edges, gen, kill):
    names = block_names(blocks)
    IN = {b:set() for b in names}
    OUT = {b:set() for b in names}
    P = preds(edges)
    changed = True
    it = 0
    logs = []
    while changed and it < 100:
        changed = False
        it += 1
        rows = []
        for b in names:
            in_new = set()
            for p in P[b]:
                in_new |= OUT[p]
            out_new = gen[b] | (in_new - kill[b])
            rows.append((b, sorted(gen[b]), sorted(kill[b]), sorted(in_new), sorted(out_new)))
            if in_new != IN[b] or out_new != OUT[b]:
                changed = True
            IN[b] = in_new
            OUT[b] = out_new
        logs.append(rows)
    return IN, OUT, logs
```

Figure 2.7: Reaching Definitions.

* IN and OUT for each block contains the mapping, that will eventually hold which definitions reach that block beforeafter.
* preds function finds which blocks are predecessor to each block.
* Runs an interation until convergence, that is it runs until there is a change capped to 100 iterations.
* Within each iteration, for every block we start with an empty set.
* For every predecessor block p, union together all definitions that can exit the block thus giving candidate definitions that could reach the entrance of this block.
* candidates that are reassighned during the iteration, are removed because of being overwritten.
* We also add the locally generated variables.
* Finally in_new and out_new are updated and in case there is no change in these for any of the blocks, we stop the iteration.

- **Analysis and result generating codes**

```
def stmt_lines(lines, s, e):
    out = []
    for k in range(s, e):
        parts = [p.strip() for p in lines[k].rstrip().split(';') if p.strip()]
        for p in parts:
            out.append(p + ";")
    return out

def html_block_label(lines, s, e, name):
    esc = lambda x: html.escape(x, quote=False)
    rows = "".join(f"<tr><td align='left'>{esc(p)}</td></tr>" for p in stmt_lines(lines, s, e))
    header = f"<tr><td bgcolor='#eeeeee'><b>{esc(name)} [{s},{e})</b></td></tr>"
    if not rows:
        rows = "<tr><td align='left'>(empty);</td></tr>"
    return f"<<table border='0' cellborder='1' cellspacing='0'>{header}{rows}</table>>"

def export_cfg_dot_html(lines, blocks, edges, program_name="code"):
    names = [f"B{i}" for i in range(len(blocks))]
    with open("cfg.dot", "w", encoding="utf-8") as f:
        f.write("digraph CFG {\n")
        f.write('  graph [fontname="Courier", nodesep=0.35, ranksep=0.4];\n')
        f.write('  node [shape=plaintext, fontname="Courier"];\n')
        f.write('  edge [fontname="Courier"];\n')
        for i,(s,e) in enumerate(blocks):
            label = html_block_label(lines, s, e, names[i])
            f.write(f"  {names[i]} [label={label}];\n")
        for u,v,lbl in edges:
            f.write(f'  {u} -> {v} [label="{lbl}"];\n')
        f.write(f'  labelloc="t";\n  label="CFG for {program_name}";\n')
        f.write("}\n")
```

Figure 2.8: Dot file generation

```python
def export_rd_csv(defs, gen, kill, IN, OUT, prefix="rd"):
    with open(f"{prefix}_defs.csv", "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(["Dk","var","line","block","stmt"])
        for Dk,var,li,txt,b in defs:
            w.writerow([Dk,var,li,b,txt])
    order = sorted(gen.keys(), key=lambda x:int(x[1:]))
    with open(f"{prefix}_sets.csv", "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(["Block","gen","kill","IN","OUT"])
        for b in order:
            w.writerow([
                b,
                " ".join(sorted(gen[b])),
                " ".join(sorted(kill[b])),
                " ".join(sorted(IN[b])),
                " ".join(sorted(OUT[b]))
            ])
```

Figure 2.9: Generating CSV.

```python
def print_Dk_map(defs):
    print("Definition IDs:")
    for Dk,var,li,txt,b in defs:
        print(f"{Dk}: var={var}, line={li}, block={b}, stmt=[{txt}]")

def print_gen_kill(gen, kill):
    order = sorted(gen.keys(), key=lambda x:int(x[1:]))
    print("\nGEN/KILL:")
    for b in order:
        print(f"{b}: gen={sorted(gen[b])} kill={sorted(kill[b])}")

def print_final_in_out(IN, OUT):
    order = sorted(IN.keys(), key=lambda x:int(x[1:]))
    print("\nFinal IN/OUT:")
    for b in order:
        print(f"{b}: IN={sorted(IN[b])} OUT={sorted(OUT[b])}")
```

Figure 2.10: Printing other results.

2.6 Results and Observations for Code1.c

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.322s)
python code.py
0 int main() {
1     int n = 200;
2     for (i = 0; i < n; i++) {
3         data[i] = (i * alpha + i * i * beta + gamma) % mod;
4         if (data[i] > 50) {
5             data[i] = data[i] - 30;
6         else if (data[i] < -20) {
7             data[i] = data[i] + 10;
8         else {
9             data[i] = data[i] + 5;
10         if (data[i] % 2 == 0) {
11             pos++;
12         else if (data[i] % 3 == 0) {
13             neg++;
14         else {
15             zero++;
16         total += data[i];
17     for (i = 0; i < n; i++) {
18         int t = data[i];
19         if (t > 70) {
20             t = t - (i % 7);
21         else if (t < 20) {
22             t = t + (i % 11);
23         else {
24             t = t * 2 - (i % 9);
25         temp[i] = t;
26     for (iteration = 0; iteration < 50; iteration++) {
27         for (i = 0; i < n; i++) {
28             int m = (data[i] + iteration * 3 + i * 2) % 123;
29             if (m < 0) {
30                 m = -m;
31             if (m % 2 == 0) {
32                 m = m / 2;
33             else {
34                 m = m * 3 + 1;
```

Figure 2.11: Identified leaders for code 1.

In this step, we mark leader lines in code1.c that start basic blocks: the function entry, each loop and conditional header, each case/default label, and the first statement after conditional or loop bodies. These leaders partition the program into maximal straight-line regions to be used for CFG construction in the next step.
For code1.c, number of identified leaders = **131**.

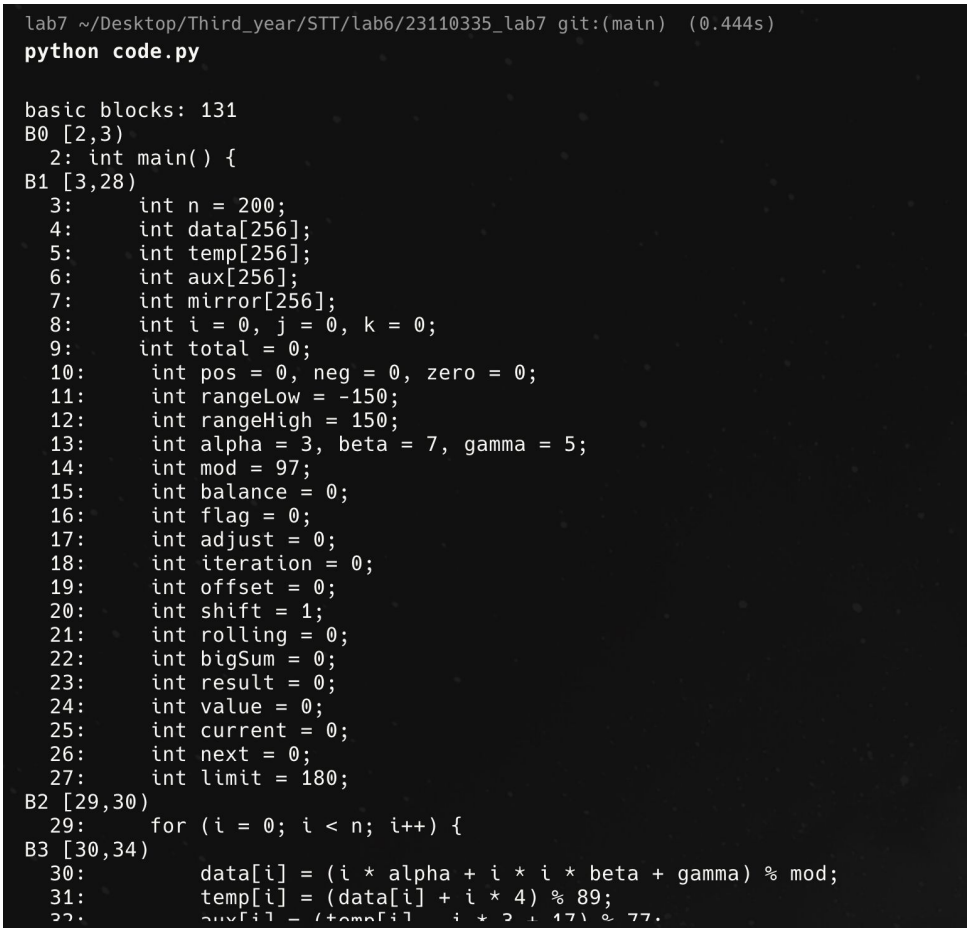


Figure 2.12: Basic blocks for code1.

This figure lists all the basic blocks using the identified leaders from the previous step. Each block contains a sequence of statements with a single entry and exit point. These blocks will serve as nodes in the Control Flow Graph (CFG) constructed in the next step. For every block Bk, the range [s,e) shows its line indices and the indented lines show the statements it contains, excluding brace-only and preprocessor lines.
For code1.c, number of basic blocks = **131**.

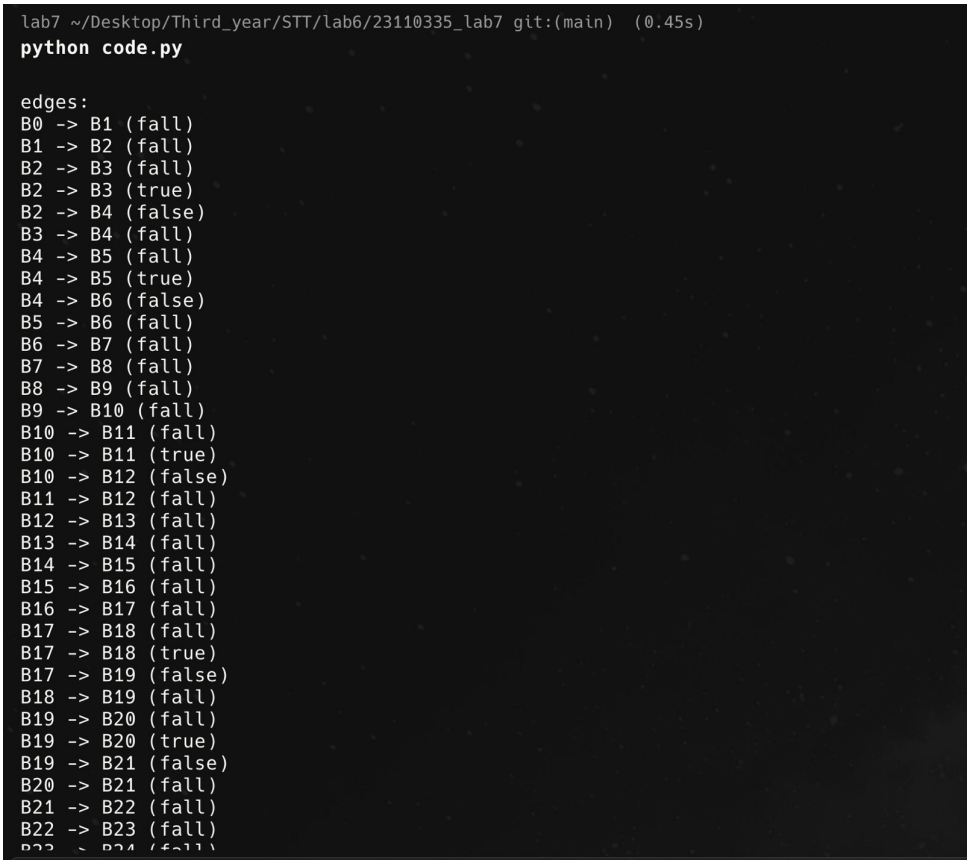


Figure 2.13: Edges for the identified blocks.

For code1.c, number of edges = **203**.

This list enumerates the CFG edges between basic blocks. A fall edge denotes sequential flow to the next block, while true/false edges arise from conditional headers (if/while/for) and split control accordingly. Using the overall structure, we find out the number of edges, nodes in the CFG, and then we calculate the Cyclomatic Complexity (CC) using the formula CC = E - N + 2P, where E is edges, N is nodes, and P is connected components.

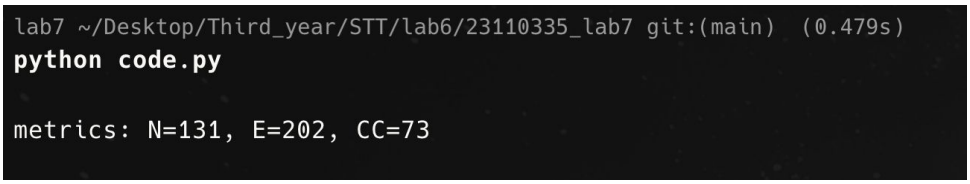


Figure 2.14: CC value for code1.

CFG Cyclomatic complexity (CC) quantifies the number of linearly independent paths through the program. A higher CC indicates more complex control flow, which impacts maintainability. For code1.c, CC = **73**, indicating normal complexity with multiple decision points and loops.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.442s)
python code.py

Definitions (90):
D1 @ line 3 in B1: n = ... ; [int n = 200]
D2 @ line 8 in B1: i = ... ; [int i = 0, j = 0, k = 0]
D3 @ line 9 in B1: total = ... ; [int total = 0]
D4 @ line 10 in B1: pos = ... ; [int pos = 0, neg = 0, zero = 0]
D5 @ line 11 in B1: rangeLow = ... ; [int rangeLow = -150]
D6 @ line 12 in B1: rangeHigh = ... ; [int rangeHigh = 150]
D7 @ line 13 in B1: alpha = ... ; [int alpha = 3, beta = 7, gamma = 5]
D8 @ line 14 in B1: mod = ... ; [int mod = 97]
D9 @ line 15 in B1: balance = ... ; [int balance = 0]
D10 @ line 16 in B1: flag = ... ; [int flag = 0]
D11 @ line 17 in B1: adjust = ... ; [int adjust = 0]
D12 @ line 18 in B1: iteration = ... ; [int iteration = 0]
D13 @ line 19 in B1: offset = ... ; [int offset = 0]
D14 @ line 20 in B1: shift = ... ; [int shift = 1]
D15 @ line 21 in B1: rolling = ... ; [int rolling = 0]
D16 @ line 22 in B1: bigSum = ... ; [int bigSum = 0]
D17 @ line 23 in B1: result = ... ; [int result = 0]
D18 @ line 24 in B1: value = ... ; [int value = 0]
D19 @ line 25 in B1: current = ... ; [int current = 0]
D20 @ line 26 in B1: next = ... ; [int next = 0]
...
```

Figure 2.15: First 20 Defined blocks.

This output lists the first 20 variable definitions found in code1.c. Each definition is represented as a tuple (variable name, line number) indicating where a variable is assigned a value. These definitions form the basis for the Reaching Definitions analysis performed later.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.451s)
python code.py

GEN/KILL per block:
B0 gen: [] kill: []
B1 gen: ['D1', 'D10', 'D11', 'D12', 'D13', 'D14', 'D15', 'D16', 'D17', 'D18', 'D19', 'D2', 'D20', 'D21', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D
9'] kill: ['D22', 'D30', 'D37', 'D38', 'D45', 'D46', 'D51', 'D58', 'D63', 'D70', 'D71', 'D72', 'D73', 'D75', 'D79', 'D81', 'D82', 'D83', 'D84',
 'D85', 'D86', 'D88', 'D89', 'D90']
B2 gen: ['D22'] kill: ['D2', 'D30', 'D38', 'D46', 'D51', 'D58', 'D63', 'D70', 'D75', 'D82', 'D86']
B3 gen: ['D23', 'D24', 'D25', 'D26'] kill: ['D27', 'D28', 'D29', 'D36', 'D43', 'D44', 'D47', 'D48', 'D49', 'D50', 'D55', 'D56', 'D57', 'D59', '
D60', 'D61', 'D69', 'D77']
B4 gen: [] kill: []
B5 gen: ['D27'] kill: ['D23', 'D28', 'D29', 'D43', 'D44', 'D55', 'D56', 'D57', 'D59', 'D60', 'D61', 'D69', 'D77']
B6 gen: [] kill: []
B7 gen: ['D28'] kill: ['D23', 'D27', 'D29', 'D43', 'D44', 'D55', 'D56', 'D57', 'D59', 'D60', 'D61', 'D69', 'D77']
B8 gen: [] kill: []
B9 gen: ['D29'] kill: ['D23', 'D27', 'D28', 'D43', 'D44', 'D55', 'D56', 'D57', 'D59', 'D60', 'D61', 'D69', 'D77']
B10 gen: [] kill: []
B11 gen: [] kill: []
B12 gen: [] kill: []
B13 gen: [] kill: []
B14 gen: [] kill: []
B15 gen: [] kill: []
B16 gen: [] kill: []
B17 gen: ['D30'] kill: ['D2', 'D22', 'D38', 'D46', 'D51', 'D58', 'D63', 'D70', 'D75', 'D82', 'D86']
B18 gen: ['D31', 'D32'] kill: ['D33', 'D34', 'D35', 'D64', 'D65', 'D66', 'D67', 'D68']
B19 gen: [] kill: []
B20 gen: ['D33'] kill: ['D31', 'D32', 'D34', 'D35', 'D64', 'D65', 'D66', 'D67', 'D68']
B21 gen: [] kill: []
B22 gen: ['D34'] kill: ['D31', 'D32', 'D33', 'D35', 'D64', 'D65', 'D66', 'D67', 'D68']
B23 gen: [] kill: []
B24 gen: ['D35'] kill: ['D31', 'D32', 'D33', 'D34', 'D64', 'D65', 'D66', 'D67', 'D68']
B25 gen: ['D36'] kill: ['D24']
B26 gen: ['D37'] kill: ['D12']
B27 gen: ['D38'] kill: ['D2', 'D22', 'D30', 'D46', 'D51', 'D58', 'D63', 'D70', 'D75', 'D82', 'D86']
B28 gen: ['D39'] kill: ['D40', 'D41', 'D42']
B29 gen: [] kill: []
```

Figure 2.16: Early reachable blocks.

This figure shows the gen/kill sets per basic block for Reaching Definitions. For each block Bk, gen lists the definitions Dk created inside the block, while kill lists other definitions of the same variables that are invalidated by writing new values in this block

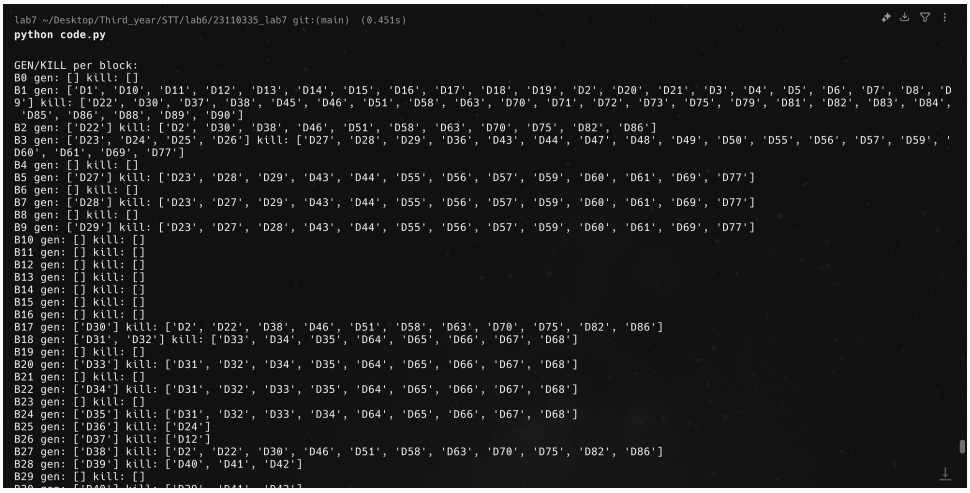


Figure 2.17: Gen and kill of blocks.

Iterations until convergence. Figure 2.18 shows the forward data-flow iterations for reaching definitions. In each iteration, every basic block B updates its set using $\text{in}[B] = \bigcup_{P \in \text{pred}(B)} \text{out}[P]$ and $\text{out}[B] = \text{gen}[B] \cup \big(\text{in}[B] \setminus \text{kill}[B]\big)$. The rows illustrate how definition IDs go downward along the CFG edges across progressive rounds until no set is changed, indicating convergence to the fixed point.

Figure 2.18: Iterations.

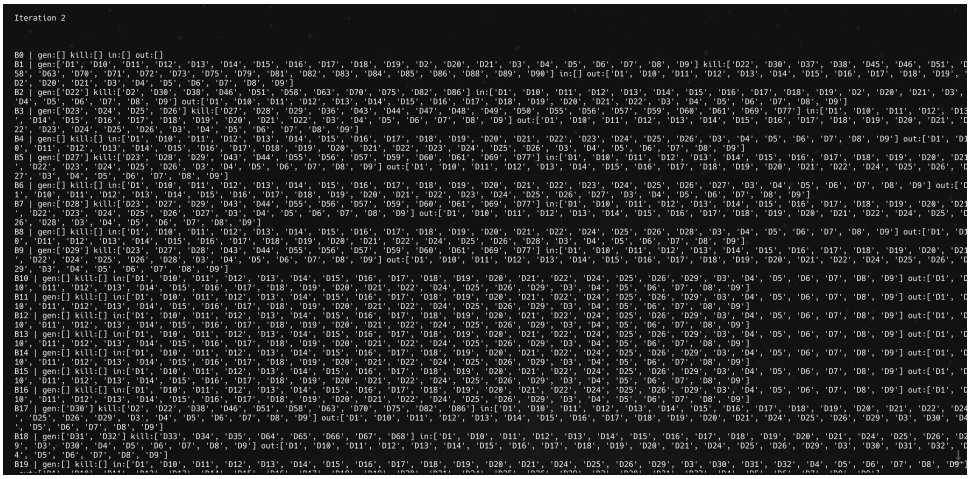


Figure 2.19: Iteration - 2.

Figure 2.20: Final IN and OUT.

The RD analysis converged in two iterations: after the second round of updates, no block's in[·] or out[·] set changed, so we say that the flows have converged to the final result.

2.7 Results for Code2.c

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.3s)
python code.py
3 int main() {
4     int i = 0;
10     for (i = 0; i < n; i++) {
11         arr[i] = (i * i + 3 * i + 7) % 97;
13     int sum = 0;
14     for (i = 0; i < n; i++) {
15         sum = sum + arr[i];
16         if (arr[i] % 2 == 0) {
17             arr[i] = arr[i] / 2;
19         else {
20             arr[i] = arr[i] * 3 + 1;
23     int even = 0;
25     for (i = 0; i < n; i++) {
26         if (arr[i] % 2 == 0) {
27             even = even + 1;
29         else {
30             odd = odd + 1;
33     int threshold = (sum / n) % 50;
35     for (i = 0; i < n; i++) {
36         if (arr[i] > threshold) {
37             greater = greater + 1;
39         else {
40             arr[i] = arr[i] + threshold / 2;
43     int iteration = 0;
44     while (iteration < 100) {
45         for (i = 0; i < n; i++) {
46             if (arr[i] % 5 == 0) {
47                 arr[i] = arr[i] + iteration;
49             else {
50                 arr[i] = arr[i] - iteration;
52             if (arr[i] < 0) {
53                 arr[i] = -arr[i];
56         iteration = iteration + 1;
58     int max = arr[0];
60     for (i = 1; i < n; i++) {
61         if (arr[i] > max) {
62             max = arr[i];
64         if (arr[i] < min) {
```

Figure 2.21: Generating leaders from the source code.

For code2.c, number of identified leaders = **122**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.43s)
python code.py

basic blocks: 122
B0 [2,3)
  2: int main() {
B1 [3,9)
  3:     int i = 0;
  4:     int j = 0;
  5:     int k = 0;
  6:     int n = 60;
  7:     int arr[200];
  8:     int aux[200];
B2 [9,10)
  9:     for (i = 0; i < n; i++) {
B3 [10,11)
  10:         arr[i] = (i * i + 3 * i + 7) % 97;
B4 [12,13)
  12:     int sum = 0;
B5 [13,14)
  13:     for (i = 0; i < n; i++) {
B6 [14,15)
  14:         sum = sum + arr[i];
B7 [15,16)
  15:         if (arr[i] % 2 == 0) {
B8 [16,17)
  16:             arr[i] = arr[i] / 2;
B9 [18,19)
  18:         else {
B10 [19,20)
  19:             arr[i] = arr[i] * 3 + 1;
B11 [22,24)
  22:     int even = 0;
```

Figure 2.22: Printing blocks from the identified leaders.

For code2.c, number of basic blocks = **122**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.464s)
python code.py

edges:
B0 -> B1 (fall)
B1 -> B2 (fall)
B2 -> B3 (fall)
B2 -> B3 (true)
B2 -> B4 (false)
B3 -> B4 (fall)
B4 -> B5 (fall)
B5 -> B6 (fall)
B5 -> B6 (true)
B5 -> B7 (false)
B6 -> B7 (fall)
B7 -> B8 (fall)
B7 -> B8 (true)
B7 -> B9 (false)
B8 -> B9 (fall)
B9 -> B10 (fall)
B10 -> B11 (fall)
B11 -> B12 (fall)
B12 -> B13 (fall)
B12 -> B13 (true)
B12 -> B14 (false)
B13 -> B14 (fall)
B13 -> B14 (true)
B13 -> B15 (false)
B14 -> B15 (fall)
B15 -> B16 (fall)
B16 -> B17 (fall)
B17 -> B18 (fall)
B18 -> B19 (fall)
B18 -> B19 (true)
B18 -> B20 (false)
B19 -> B20 (fall)
```

Figure 2.23: Edges between the blocks.

For code2.c, number of edges = **203**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.332s)
python code.py

metrics: N=122, E=203, CC=83
```

Figure 2.24: Printing the Cyclomatic complexity.

For code2.c, Cyclomatic Complexity (CC) = **83**.

Figure 2.25: Reaching Definitions.

This output lists the first 20 variable definitions found in code2.c. Each definition is represented as a tuple (variable name, line number) indicating where a variable is assigned a value. These definitions form the basis for the Reaching Definitions analysis performed later.

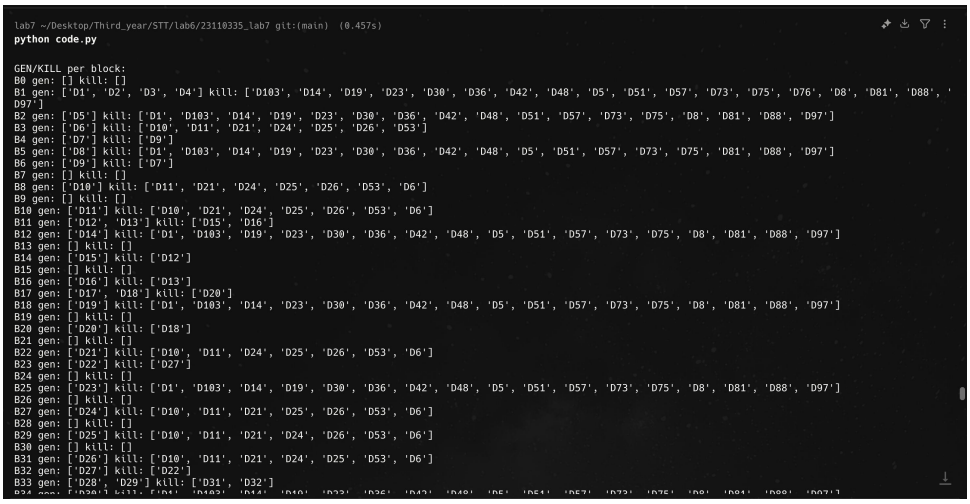


Figure 2.26: Generating and kills per block.

This figure shows the gen/kill sets per basic block for Reaching Definitions. For each block Bk, gen lists the definitions Dk created inside the block, while kill lists other definitions of the same variables that are invalidated by writing new values in this block

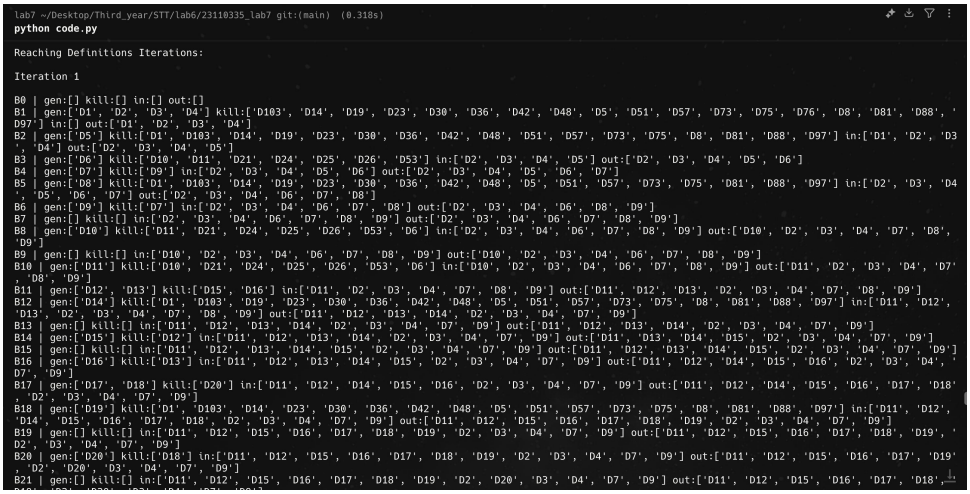


Figure 2.27: Iteration 1.

Figure 2.28: Iteration 2.

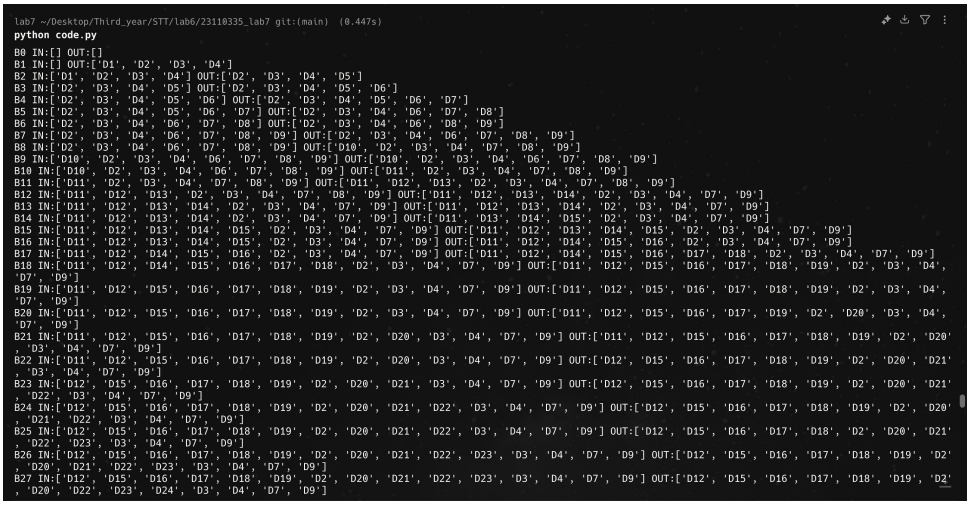


Figure 2.29: Final IN and OUT after iteration convergence.

Reaching–definitions converged quickly: after Iteration 1 and Iteration 2, no further

changes occurred in any block's in[·] or out[·] sets. The third panel reports these final fixed-point sets per block.

2.8 Results for Code3.c

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.443s)
python code.py
3 int main() {
4     int i = 0;
16     for (i = 0; i < n; i++) {
17         arr[i] = (i * 13 + 5 * (i % 7) + 11) % 97;
19     for (i = 0; i < n; i++) {
20         total = total + arr[i];
23     int average = total / n;
25     for (i = 0; i < n; i++) {
26         int value = arr[i];
27         if (value > pivot) {
28             value = value - (value / 5);
35             for (j = 0; j < 3; j++) {
36                 value = (value + j + i) % 91;
38                 if (balance % 7 == 0) {
39                     value = value + 2;
41                 else {
42                     value = value - 3;
45             arr[i] = value;
47         else {
48             value = value + (pivot - value) / 2;
53             for (j = 0; j < 2 * (i % 3 + 1); j++) {
54                 value = (value + 5 + j) % 97;
55                 if (value % 6 == 0) {
56                     x = x + j;
58                 else {
59                     y = y + i;
61                 if (x + y + z > 400) {
62                     x = x / 2;
67             arr[i] = value;
71     int round = 0;
72     while (round < 50) {
73         int sumPart = 0;
74         for (i = 0; i < n; i++) {
75             sumPart = sumPart + arr[i];
77         if (sumPart % 2 == 0) {
78             for (i = 0; i < n; i++) {
79                 arr[i] = (arr[i] + i + round) % 100;
82             balance = balance + sumPart / 2;
```

Figure 2.30: Generating leaders from the source code.

For code3.c, number of identified leaders = **104**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.43s)
python code.py

basic blocks: 104
B0 [2,3)
  2: int main() {
B1 [3,15)
  3:     int i = 0;
  4:     int j = 0;
  5:     int n = 60;
  6:     int arr[120];
  7:     int aux[120];
  8:     int total = 0;
  9:     int x = 3;
  10:     int y = 7;
  11:     int z = 9;
  12:     int balance = 0;
  13:     int iteration = 0;
  14:     int mode = 1;
B2 [15,16)
  15:     for (i = 0; i < n; i++) {
B3 [16,17)
  16:         arr[i] = (i * 13 + 5 * (i % 7) + 11) % 97;
B4 [18,19)
  18:     for (i = 0; i < n; i++) {
B5 [19,21)
  19:         total = total + arr[i];
  20:         aux[i] = arr[i] % 10;
B6 [22,24)
  22:     int average = total / n;
  23:     int pivot = (average % 20) + 10;
B7 [24,25)
  24:     for (i = 0; i < n; i++) {
B8 [25,26)
  25:         int value = arr[i];
B9 [26,27)
  26:         if (value > pivot) {
B10 [27,34)
  27:             value = value - (value / 5);
  28:             arr[i] = value + (i % 7);
```

Figure 2.31: Printing blocks from the identified leaders.

For code3.c, number of basic blocks = **104**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.594s)
python code.py

edges:
B0 -> B1 (fall)
B1 -> B2 (fall)
B2 -> B3 (fall)
B2 -> B3 (true)
B2 -> B4 (false)
B3 -> B4 (fall)
B4 -> B5 (fall)
B4 -> B5 (true)
B4 -> B6 (false)
B5 -> B6 (fall)
B6 -> B7 (fall)
B7 -> B8 (fall)
B7 -> B8 (true)
B7 -> B9 (false)
B8 -> B9 (fall)
B9 -> B10 (fall)
B9 -> B10 (true)
B9 -> B11 (false)
B10 -> B11 (fall)
B11 -> B12 (fall)
B11 -> B12 (true)
B11 -> B13 (false)
B12 -> B13 (fall)
B13 -> B14 (fall)
B13 -> B14 (true)
B13 -> B15 (false)
B14 -> B15 (fall)
B15 -> B16 (fall)
B16 -> B17 (fall)
B17 -> B18 (fall)
B18 -> B19 (fall)
B19 -> B20 (fall)
B20 -> B21 (fall)
```

Figure 2.32: Edges between the blocks.

For code3.c, number of edges = **167**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (0.491s)
python code.py

metrics: N=104, E=167, CC=65
```

Figure 2.33: Printing the Cyclomatic complexity.

For code3.c, Cyclomatic Complexity (CC) = **65**.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (3.466s)
python code.py

Definitions (115):
D1 @ line 3 in B1: i = ... ; [int i = 0]
D2 @ line 4 in B1: j = ... ; [int j = 0]
D3 @ line 5 in B1: n = ... ; [int n = 60]
D4 @ line 8 in B1: total = ... ; [int total = 0]
D5 @ line 9 in B1: x = ... ; [int x = 3]
D6 @ line 10 in B1: y = ... ; [int y = 7]
D7 @ line 11 in B1: z = ... ; [int z = 9]
D8 @ line 12 in B1: balance = ... ; [int balance = 0]
D9 @ line 13 in B1: iteration = ... ; [int iteration = 0]
D10 @ line 14 in B1: mode = ... ; [int mode = 1]
D11 @ line 15 in B2: i = ... ; [for (i = 0]
D12 @ line 16 in B3: arr[i] = ... ; [arr[i] = (i * 13 + 5 * (i % 7) + 11) % 97]
D13 @ line 18 in B4: i = ... ; [for (i = 0]
D14 @ line 19 in B5: total = ... ; [total = total + arr[i]]
D15 @ line 20 in B5: aux[i] = ... ; [aux[i] = arr[i] % 10]
D16 @ line 22 in B6: average = ... ; [int average = total / n]
D17 @ line 23 in B6: pivot = ... ; [int pivot = (average % 20) + 10]
D18 @ line 24 in B7: i = ... ; [for (i = 0]
D19 @ line 25 in B8: value = ... ; [int value = arr[i]]
D20 @ line 27 in B10: value = ... ; [value = value - (value / 5)]
...
```

Figure 2.34: Reaching Definitions.

```
lab7 ~/Desktop/Third_year/STT/lab6/23110335_lab7 git:(main)  (3.25s)
python code.py

GEN/KILL per block:
B0 gen: [] kill: []
B1 gen: ['D1', 'D10', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9'] kill: ['D100', 'D102', 'D103', 'D104', 'D105', 'D106', 'D107', 'D108', 'D109', 'D11', 'D1
11', 'D112', 'D113', 'D114', 'D13', 'D14', 'D18', 'D23', 'D24', 'D25', 'D26', 'D27', 'D29', 'D34', 'D35', 'D36', 'D37', 'D38', 'D40', 'D41', 'D42', 'D43', 'D44
', 'D49', 'D51', 'D54', 'D55', 'D56', 'D57', 'D58', 'D61', 'D62', 'D63', 'D64', 'D65', 'D67', 'D68', 'D69', 'D70', 'D71', 'D72', 'D73', 'D74', 'D75', 'D76', 'D
77', 'D78', 'D79', 'D80', 'D82', 'D90', 'D94']
B2 gen: ['D11'] kill: ['D1', 'D100', 'D13', 'D18', 'D49', 'D51', 'D58', 'D82', 'D90', 'D94']
B3 gen: ['D12'] kill: ['D21', 'D32', 'D45', 'D52', 'D59', 'D84', 'D87']
B4 gen: ['D13'] kill: ['D1', 'D100', 'D11', 'D18', 'D49', 'D51', 'D58', 'D82', 'D90', 'D94']
B5 gen: ['D14', 'D15'] kill: ['D22', 'D4', 'D46', 'D53', 'D60', 'D85', 'D86', 'D88']
B6 gen: ['D16', 'D17'] kill: []
B7 gen: ['D18'] kill: ['D1', 'D100', 'D11', 'D13', 'D49', 'D51', 'D58', 'D82', 'D90', 'D94']
B8 gen: ['D19'] kill: ['D20', 'D28', 'D30', 'D31', 'D33', 'D39']
B9 gen: [] kill: []
B10 gen: ['D20', 'D21', 'D22', 'D23', 'D24', 'D25', 'D26'] kill: ['D102', 'D103', 'D104', 'D105', 'D106', 'D107', 'D108', 'D109', 'D111', 'D112', 'D113', 'D114
', 'D12', 'D15', 'D19', 'D28', 'D29', 'D30', 'D31', 'D32', 'D33', 'D34', 'D35', 'D36', 'D37', 'D39', 'D40', 'D41', 'D42', 'D43', 'D44', 'D45', 'D46', 'D5', 'D5
2', 'D53', 'D54', 'D55', 'D56', 'D57', 'D59', 'D6', 'D60', 'D61', 'D62', 'D63', 'D64', 'D65', 'D69', 'D7', 'D70', 'D71', 'D72', 'D73', 'D74', 'D75', 'D76', 'D7
7', 'D78', 'D79', 'D8', 'D80', 'D84', 'D85', 'D86', 'D87', 'D88']
B11 gen: ['D27'] kill: ['D2', 'D38']
B12 gen: ['D28', 'D29'] kill: ['D105', 'D114', 'D19', 'D20', 'D26', 'D30', 'D31', 'D33', 'D37', 'D39', 'D54', 'D61', 'D62', 'D8']
B13 gen: [] kill: []
B14 gen: ['D30'] kill: ['D19', 'D20', 'D28', 'D31', 'D33', 'D39']
B15 gen: [] kill: []
B16 gen: ['D31'] kill: ['D19', 'D20', 'D28', 'D30', 'D33', 'D39']
B17 gen: ['D32'] kill: ['D12', 'D21', 'D45', 'D52', 'D59', 'D84', 'D87']
B18 gen: [] kill: []
B19 gen: ['D33', 'D34', 'D35', 'D36', 'D37'] kill: ['D102', 'D103', 'D104', 'D105', 'D106', 'D107', 'D108', 'D109', 'D111', 'D112', 'D113', 'D114', 'D19', 'D20
', 'D23', 'D24', 'D25', 'D26', 'D28', 'D29', 'D30', 'D31', 'D39', 'D40', 'D41', 'D42', 'D43', 'D44', 'D5', 'D54', 'D55', 'D56', 'D57', 'D6', 'D61', 'D62', 'D63
', 'D64', 'D65', 'D69', 'D7', 'D70', 'D71', 'D72', 'D73', 'D74', 'D75', 'D76', 'D77', 'D78', 'D79', 'D8', 'D80']
B20 gen: ['D38'] kill: ['D2', 'D27']
B21 gen: ['D39'] kill: ['D19', 'D20', 'D28', 'D30', 'D31', 'D33']
B22 gen: [] kill: []
B23 gen: ['D40'] kill: ['D102', 'D106', 'D108', 'D111', 'D23', 'D34', 'D42', 'D5', 'D55', 'D63', 'D69', 'D72', 'D75', 'D78']
B24 gen: [] kill: []
B25 gen: ['D41'] kill: ['D103', 'D107', 'D112', 'D24', 'D35', 'D43', 'D56', 'D6', 'D64', 'D70', 'D73', 'D76', 'D79']
B26 gen: [] kill: []
B27 gen: ['D42', 'D43', 'D44'] kill: ['D102', 'D103', 'D104', 'D106', 'D107', 'D108', 'D109', 'D111', 'D112', 'D113', 'D23', 'D24', 'D25', 'D34', 'D35', 'D36',
```

Figure 2.35: Generating and kills per block.

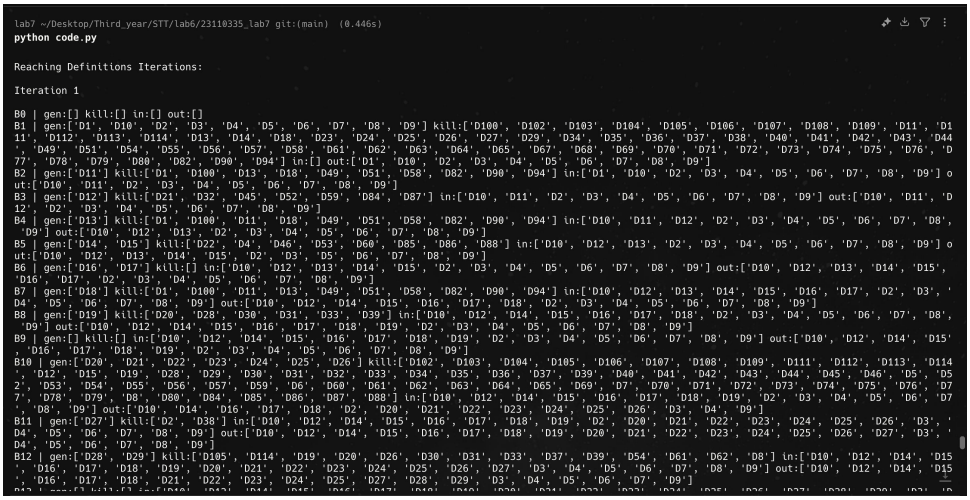


Figure 2.36: Iteration 1.

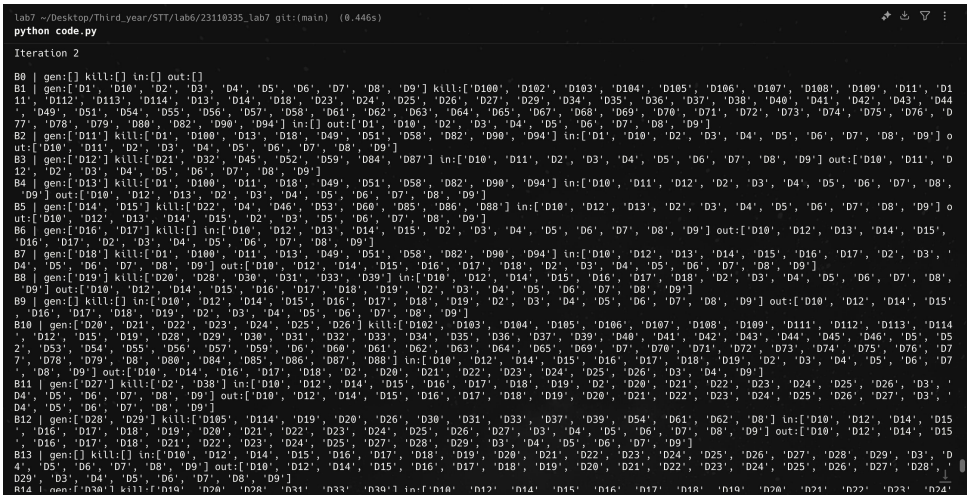


Figure 2.37: Iteration 2.

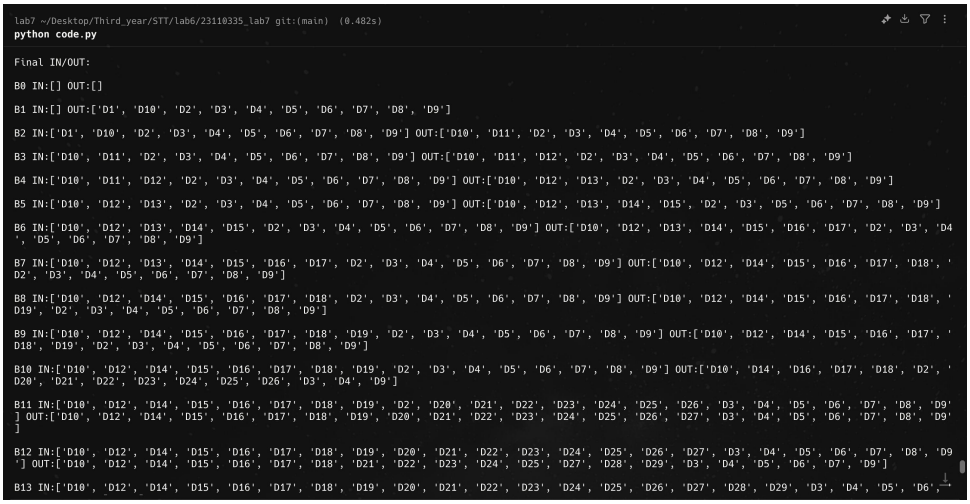


Figure 2.38: Final IN and OUT after iteration convergence.

The reaching–definitions process for this program stabilized quickly: after two iterations (Figures 2.36 and 2.37), no entries changed, and the final panel reports the fixed-point IN and OUT sets for every block (Figure 2.38). These sets summarize which definition IDs are visible on entry to and exit from each basic block at convergence.

2.9 Result and Discussion

Program No.	No. of Nodes (N)	No. of Edges (E)	Cyclomatic Complexity (CC)
Code 1	131	202	73
Code 2	122	203	83
Code 3	104	167	65

Table 2.1: CFG metrics summary for each program.

This CSV lists every assignment as a unique definition ID (Dk) with its variable, source line, basic block, and original statement text, providing a canonical reference for later data–flow tables.

This CSV reports, for each basic block, the sets gen[B], kill[B], in[B], and out[B] as definition IDs, summarizing which definitions are created, overwritten, and reach the block at entry and exit after convergence.

2.10 Multiple possible reaching definitions

Variables can have multiple reaching definitions at a program point whenever different control–flow paths assign to the same variable before those paths join. In the final in[B] sets, this appears as several definition IDs (e.g., D3, D17, D21) for the same variable reaching the entry of block B; correspondingly, out[B] may still contain multiple IDs if the block does not redefine that variable. Typical causes include:

1. if/else chains that assign to the same variable in each branch;
2. loop bodies that reassign a variable and feed a back edge to the header;
3. switch/case arms with assignments that join later.

Thus, any variable whose ID set size $|\text{in}[B] \cap \text{Defs}(x)| > 1$ at some block B has multiple possible reaching definitions at that point, implying its value depends on the path taken.

Conclusion

We constructed control–flow graphs (CFGs) for each program by finding leaders and forming basic blocks, followed by adding fall–through and branch edges to obtain nodes N and edges E, with cyclomatic complexity computed as $CC = E - N + 2$. The HTML–styled node labels ensured one–statement–per–line readability in the figures. For data–flow, we

performed Reaching Definitions: assigned unique IDs (Dk) to all assignments, computed per–block gen$[B]$ and kill$[B]$, and iterated the forward equations $\text{in}[B] = \bigcup_{P \in \text{pred}(B)} \text{out}[P]$ and $\text{out}[B] = \text{gen}[B] \cup \big(\text{in}[B] \setminus \text{kill}[B]\big)$ to a fixed point (converging in two iterations). The exported CSVs (definitions map and per–block sets) provide data that match the tables in the report. Overall, the final IN/OUT table finds which definitions reach each block at entry and exit, enabling downstream optimizations and test–planning guided by CFG complexity.

References

- Reaching definition (Wikipedia): https://en.wikipedia.org/wiki/Reaching_definition
- Cornell CS notes — Reaching Definitions and SSA: https://www.cs.cornell.edu/courses/cs4120/2023sp/notes.html?id=reachdef
- Kildall (1973), A Unified Approach to Global Program Optimization (ACM DL): https://dl.acm.org/doi/10.1145/512927.512945