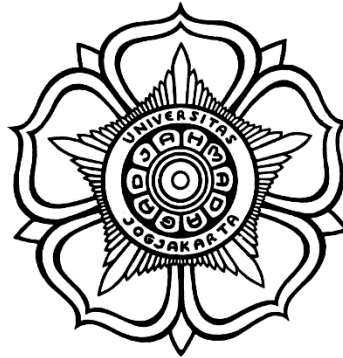


UNDERGRADUATE THESIS

**ANALISA BINER STATIS MELALUI EKSTRAKSI *INSTRUCTION* DAN
OPERAND DAN AGGLOMERATIVE *HIERARCHICAL CLUSTERING***

**BINARY STATIC ANALYSIS THROUGH INSTRUCTION AND
OPERAND EXTRACTION AND AGGLOMERATIVE *HIERARCHICAL*
CLUSTERING**



CHRYST MARK REAL RUMAHORBO

20/457768/PA/19806

**UNDERGRADUATE PROGRAM IN COMPUTER SCIENCE
FACULTY OF MATHEMATICS AND NATURAL SCIENCES**

UNIVERSITAS GADJAH MADA

YOGYAKARTA

2024

HALAMAN PENGESAHAN

SKRIPSI (RISET AKADEMIK)

**BINARY STATIC ANALYSIS THROUGH INSTRUCTION AND OPERAND
EXTRACTION AND AGGLOMERATIVE HIERARCHICAL CLUSTERING
IUP)**

Telah dipersiapkan dan disusun oleh

Chryst Mark Real Rumahorbo

20/457768/PA/19806

Telah dipertahankan di depan Tim Penguji
pada tanggal 13 Juni 2024

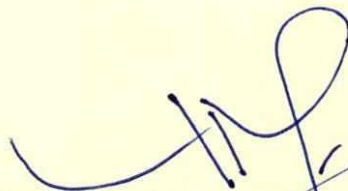
Susunan Tim Penguji



Dr. Yohanes Suyanto, M.I.Kom.
Ketua Penguji



Dr. Azhari, MT
Anggota Penguji



Prof. Dr.-Ing. Mhd. Reza M. I. Pulungan, S.Si., M.Sc
Pembimbing

Mengetahui,
a.n. Dekan FMIPA UGM
Wakil Dekan Bidang Pendidikan, Pengajaran
dan Kemahasiswaan



Prof. Drs. Roto, M.Eng., Ph.D.
NIP. 196711171993031020

PERNYATAAN BEBAS PLAGIASI

Saya yang bertanda tangan di bawah ini:

Nama : Chryst Mark Real Rumahorbo
NIM : 20/457768/PA/19806
Tahun terdaftar : 2020
Program studi : S1 Ilmu Komputer IUP
Fakultas/Sekolah : Fakultas Matematika dan Ilmu Pengetahuan Alam

Menyatakan bahwa dalam dokumen ilmiah Skripsi ini tidak terdapat bagian dari karya ilmiah lain yang telah diajukan untuk memperoleh gelar akademik di suatu Lembaga Pendidikan Tinggi, dan juga tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang/Lembaga lain, kecuali yang secara tertulis disitasi dalam dokumen ini dan disebutkan sumbernya secara lengkap dalam daftar Pustaka.

Dengan demikian saya menyatakan bahwa dokumen ilmiah ini bebas dari unsur-unsur plagiasi dan apabila dokumen ilmiah Skripsi ini di kemudian hari terbukti merupakab plagiasi dari hasil karya penulis lain dan/atau dengan sengaja mengajukan karya atau pendapat yang merupakan hasil karya penulis lain, maka penulis bersedia menerima sanksi akademik dan/atau sanksi hukum yang berlaku.

Yogyakarta, 4 Juni 2024



Chryst Mark Real Rumahorbo
20/457768/PA/19806

FOREWORD

I would like to express my deepest gratitude to those who have supported and guided me throughout the process of completing this thesis. First and foremost, I would like to thank my academic advisor, Prof. Dr.-Ing. Mhd. Reza M. I. Pulungan, S.Si., M.Sc., for his invaluable guidance, insightful feedback, and continuous encouragement. His expertise and dedication were instrumental in the successful completion of this research.

I also extend my sincere thanks to the examination team members Dr. Yohanes Suyanto, M.I.Kom., and Dr. Azhari, MT, for their critical evaluations and constructive suggestions, which greatly improved the quality of this thesis.

Furthermore, I am profoundly grateful to my parents, for their unwavering support, love, and encouragement throughout my academic journey. Their belief in my abilities has been a constant source of motivation.

Finally, I extend my appreciation to all my friends and colleagues who have provided moral support and companionship during the challenging times of my study.

Yogyakarta, 04 June 2024

Author

TABLE OF CONTENT

APPROVAL PAGE	ii
DECLARATION.....	iii
FOREWORD.....	iv
TABLE OF CONTENT.....	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF CODE	xi
ABSTRACT	xii
INTISARI	xiii
CHAPTER I INTRODUCTION	1
1.1 Research Background	1
1.2 Research Problem	2
1.3 Research Scope	3
1.4 Research Objectives	3
1.5 Research Benefits.....	3
CHAPTER II LITERATURE REVIEW	5
CHAPTER III THEORETICAL BASIS	12
3.1 Binary Files	12
3.2 ELF Files.....	12
3.3 Instruction and Operand Extraction	13
3.4 Shannon Entropy	14
3.5 Kullback-Leibler Divergence (KL-Divergence)	15
3.6 Agglomerative Hierarchical Clustering (AHC)	16
3.7 Silhouette Coefficient as an Evaluation Metric	19
3.8 Block Feature Correlation Using Cross-Correlation Function	20
3.9 Correlated Data Flow Graph	21
CHAPTER IV RESEARCH METHODOLOGY	22

4.1 Research Description	22
4.2 Environment Set-Up and Data Acquisition	22
4.2.1 Building Custom Programs	22
4.2.2 Disassembling ELF Files	23
4.3 Feature Extraction	24
4.3.1 Entropy Based Significant Feature Extraction.....	24
4.4 Block Characterization.....	25
4.4.1 Implementing Agglomerative Hierarchical Clustering (AHC).....	25
4.4.2 Evaluating the Clusters Using Silhouette Coefficient.....	26
4.5 Constructing Correlated Data Flow Graphs.....	27
4.5.1 Calculating Block Feature Correlation	27
4.5.2 Generating Correlation Data Flow Graphs	27
CHAPTER V IMPLEMENTATION.....	28
5.1 Development Environment	28
5.2 Dataset Generation.....	28
5.2 Feature Extraction Program	29
5.3 Extracting Significant Features by Entropies Level	32
5.3.1 Noise Removal via Entropy Thresholding.....	32
5.4 Block Characterization Using KL-Divergence and AHC.....	35
5.4.1 Calculating KL-Divergence	35
5.4.2 AHC Implementation	36
5.5 Silhouette Coefficient as Evaluation Metric	38
5.6 Generating Correlated Data Flow Graphs (CDFGs).....	39
5.6.1 Calculating Block Correlation	40
5.6.2 Visualizing Highly Correlated Data Flow Graphs (CDFGs).....	40

CHAPTER VI RESULTS & ANALYSIS.....	42
6.1 Block Characterization Visualized as Dendrograms	42
6.1.1 Functional Clustering of Simple Calculator.....	42
6.1.2 Functional Clustering of Dynamic Array Allocator	44
6.1.3 Functional Clustering of Dynamic CSV Parser	47
6.2 Silhouette Coefficient as an Evaluation Metric	50
6.3 Block Correlation Analysis Using Correlated Data Flow Graphs	52
CHAPTER VII CONCLUSION	56
REFERENCES.....	58
APPENDIX.....	62
Appendix A (Entropy Calculation)	62
Appendix B (Simple Calculator).....	64
Appendix C (Simple Calculator Assembly)	65
Appendix D (Dynamic Array Allocator).....	70
Appendix E (CSV Parser)	71
Appendix F (Dendrogram of CSV Parser Block Clusters)	72
Appendix G (Coefficient Value of Simple Calculator Blocks).....	73
Appendix H (Coefficient Value of Dynamic Array Allocator Blocks).....	74
Appendix I (Coefficient Value of CSV Parser Blocks).....	75
Appendix J (Block Similarity Heatmap of Simple Calculator).....	78
Appendix K (Block Similarity Heatmap of Dynamic Array Allocator)	79

LIST OF TABLES

Table 2.1 Table of Comparison.....	9
Table 5.1 Environment Specifications	28
Table 5.2 Data frame of disassembled code of simple calculator program	30
Table 5.3 First 5 assembly codes distribution probability and entropy	32
Table 5.4 First 5 assembly codes probability and entropy after noise removal....	35
Table 5.5 Top 5 highly correlated blocks (Simple Calculator).....	40
Table 6.1 Cluster Assignment Table (Simple Calculator)	42
Table 6.2 Cluster Assignment Table (Dynamic Array Allocator)	45
Table 6.3 Cluster Assignment Table (CSV Parser)	48

LIST OF FIGURES

Figure 3.1 ELF Format	13
Figure 3.2 Pseudocode for Calculating Entropy using Shannon Entropy.....	15
Figure 3.3 Agglomerative Hierarchical Clustering Algorithm Process	17
Figure 3.4 Pseudocode of Agglomerative Hierarchical Clustering	18
Figure 3.5 Example of Dendrogram Produced by AHC Algorithm	19
Figure 3.6 An example of highly correlated data flow graph	21
Figure 4.1 Example of a disassembled binary file	23
Figure 5.1 Content size of each block (CSV Parser)	31
Figure 5.2 Content size of each block (Dynamic Array Allocator)	31
Figure 5.3 Content size of each block (Simple Calculator)	31
Figure 5.4 Distinct instruction (ϵ) Shannon entropies. (X-axis = Instructions)....	33
Figure 5.5 Distinct left operand (δ) Shannon entropies. (X-axis = Left operand)	33
Figure 5.6 Distinct right operand (γ) Shannon entropies (X-axis = Right operand)	34
Figure 5.7 Block similarity matrix heatmap (Simple Calculator).....	36
Figure 5.8 AHC dendrogram output (Dynamic Array Allocator)	38
Figure 5.9 Silhouette Coefficient values for each block (CSV Parser (Left) and Dynamic Array Allocator (Right)).....	39
Figure 5.10 Maximum correlation values for each block (Dynamic Array Allocator)	41

Figure 5.11 Highly correlated blocks (Dynamic Array Allocator).....	41
Figure 6.1 Dendrogram of Block Clusters (Simple Calculator)	43
Figure 6.2 Block 1 Content (Simple Calculator)	43
Figure 6.3 Average Entropy of Distinct Right Operand per Cluster (Simple Calculator).....	44
Figure 6.4 Dendrogram of Block Clusters (Dynamic Array Allocator)	45
Figure 6.5 Average Entropy of Distinct Right Operand per Cluster (Dynamic Array Allocator)	46
Figure 6.6 Average Entropy of Distinct Instruction per Cluster (CSV Parser)	49
Figure 6.7 Average Entropy of Distinct Right Operand per Cluster (CSV Parser)	49
Figure 6.8 Silhouette Coefficient for Each Block (CSV Parser)	50
Figure 6.9 Silhouette Coefficient for Each Block (Dynamic Array Allocator)	51
Figure 6.10 Silhouette Coefficient for Each Block (Simple Calculator)	51
Figure 6.11 Highly Correlated Blocks (CSV Parser).....	53
Figure 6.12 Highly Correlated Blocks (Dynamic Array Allocator)	53
Figure 6.13 Highly Correlated Blocks (Simple Calculator)	54

LIST OF CODE

Code 5.1 Parsing instructions function to format assembly code	29
Code 5.2 A function to convert the extracted code into CSV format	30
Code 5.3 A loop to write with entropy above the threshold into a new CSV	34
Code 5.4 A function to calculate KL-Divergence between two blocks	35
Code 5.5 A function to convert similarity matrix into distance matrix	37
Code 5.6 Implementation of AHC	37
Code 5.7 Calculating Silhouette Coefficient values for each block	38

ABSTRACT

Binary Static Analysis Through Instruction and Operand Extraction and Agglomerative Hierarchical Clustering

By

Chryst Mark Real Rumahorbo

20/457768/PA/19806

The ever-increasing need to make sure code is reliable, efficient, and secure has fueled the growth of popular static binary code analysis tools. Tools like IDA Pro and objdump command help analysts by disassembling binaries into an opcode/assembly language format in support of manual static code analysis. Despite the existence of such tools, analyzing large binaries manually takes a ton of time and resources. It's easy to miss potential coding flaws or inefficiencies. This paper expands on a lightweight, data-driven methodology that uses highly correlated data flow graphs (CDFGs) to identify coding irregularities such that analysis time and required computing resources are minimized. We combine graph analysis and unsupervised machine learning techniques - this saves analysts time and computing resources. By recognizing the most important flow patterns, analysts can focus on the most statistically significant flow patterns, improving accuracy and efficiency.

Keywords: Agglomerative Hierarchical Clustering, Correlated Data Flow Graphs, Static Binary Analysis, Information Metrics, Unsupervised Learning

INTISARI

Analisa Biner Statis Melalui Ekstraksi Instruction Dan Operand Dan Agglomerative Hierarchical Clustering

Oleh

Chryst Mark Real Rumahorbo

20/457768/PA/19806

Permintaan yang semakin meningkat untuk memastikan kode andal, efisien, dan aman telah mendorong pertumbuhan alat analisis kode biner statis yang populer. Alat seperti *IDA Pro* dan perintah *objdump* membantu analisis dengan membongkar biner menjadi format bahasa *opcode/assembly* untuk mendukung analisis kode statis manual. Meskipun alat-alat tersebut sudah ada, menganalisis biner besar secara manual memerlukan banyak waktu dan sumber daya. Mudah untuk melewatkan potensi cacat atau ketidakefisienan kode. Makalah ini menguraikan metodologi berbasis data yang ringan yang menggunakan *Correlated Data Flow Graphs (CDFGs)* untuk mengidentifikasi ketidakteraturan kode sehingga waktu analisis dan sumber daya komputasi yang diperlukan dapat diminimalkan. Dengan menggabungkan analisis graf dan teknik *unsupervised learning* - ini menghemat waktu dan sumber daya komputasi bagi analisis. Dengan mengenali pola aliran yang paling penting, analisis dapat fokus pada pola aliran yang paling signifikan secara statistik, meningkatkan akurasi dan efisiensi.

Kata Kunci: *Agglomerative Hierarchical Clustering, Correlated Data Flow Graphs, Analisa Biner Statis, Metrik Informasi, Pembelajaran Tanpa Pengawasan*

CHAPTER I

INTRODUCTION

1.1 Research Background

Despite the numerous critical systems that depend on third-party software, such software's dependability, safety, and privacy-preserving properties still need to be ensured (Obert and Loffredo, 2021). The software's source code is not generally available for users and developers to view, let alone to analyze for quality. However, once the software binary files are disassembled, various characteristics, including function calls, header, text, and data sections, are available for analysis (Ahmad Zabidi, 2012). This lack of transparency can have significant repercussions, mainly when using complex algorithms and data processing methods. Moreover, the need for more transparency in software source code raises concerns about the deployed algorithms' ethics, accuracy, and fairness, primarily as they handle increasingly sensitive and critical tasks. As a result, there is a growing need for efficient, adaptive, and accurate solutions to navigate the complex content of software executable files when the source code is unavailable.

In recent years, the application of machine learning techniques for tackling complex tasks has seen a significant surge, especially in analyzing data or content that is challenging for humans to read. The source code is generally inaccessible in executable files after assembling all files building the programs. Disassembly tools like IDA Pro or objdump for Linux distributions can be employed to deconstruct the architecture of such files. These tools transform the binary structures into assembly language files, which, while retaining many machine-level syntactic elements, are still comprehensible to humans.

In a prior study, Bragen (2015) employed a range of supervised learning algorithms—including Naive Bayes, Support Vector Machines (SVM), K-Nearest Neighbor (KNN), and Self-Organizing Map (SOM)—to categorize files as either “malware” or “benign” based on their instruction sequences. While these methods demonstrated robust performance in accurate classification, they require extensive

preparation and training to achieve the optimal output. An alternative yet underexplored area for addressing this issue involves utilizing unsupervised learning techniques to cluster and classify the feature vectors of assembly files proposed by Obert and Loffredo (2021). Unlike supervised learning, unsupervised approaches do not require an existing dataset or pre-trained model to apply to new data. This inherent flexibility renders unsupervised learning both lightweight and efficient while incurring only a minimal compromise in classification accuracy. Despite the seemingly efficient model, Obert and Loffredo (2021) did not mention the types of software they applied the Agglomerative Hierarchical Clustering (AHC), nor did they mention if they had prior knowledge of the structure of the binary files they tested.

In this proposed research, Obert and Loffredo's (2020) method will be replicated to analyze pre-made software applications with varying complexity. The unsupervised learning algorithm's primary objective is to generate a hierarchical clustering of blocks extracted from disassembled binary files. The approach aims to transform the data into a human-readable format by constructing Highly Correlated Data Flow Graphs (CDFGs). The accuracy and efficiency of the model will be compared against the known structure of the pre-made software applications, which invokes novelty in the research.

1.2 Research Problem

A recent study by Obert and Loffredo (2021) emphasizes the importance of using binary static analysis as an alternative to static code analysis when the source code of a system is not accessible to either users or developers. This issue is particularly relevant for critical systems that depend on third-party software for functionalities such as file sharing and networking, as these systems may not offer guaranteed dependability, safety, and privacy. Despite the significance of this challenge, there is a lack of effective methods for analyzing complex binary code to ensure these critical attributes. Therefore, this research aims to address this gap by creating and evaluating the use of the Agglomerative Hierarchical Clustering (AHC) algorithm in conjunction with generating correlated Control Data Flow

Graphs (CDFGs) as a comprehensive binary static analysis tool. This tool will be assessed for its ability to identify and cluster functional code blocks and enhance data flow analysis, ultimately promoting greater transparency and trust in software systems that rely on third-party components.

1.3 Research Scope

In this research, ELF (Executable and Linkable Format) files targeting x64 architectures will be compiled from pre-made C++ source code. The target programs are within three predefined categories: Arithmetic Operations, Pointer Manipulation, and File I/O Handling. Each ELF will be disassembled using the `objdump` operation to achieve the desired features. Significant features to be extracted from the binary files include instructions (opcodes), left operands, and right operands. Each feature will be assessed using two metrics, Shannon Entropy, and KL-Divergence, to gauge information relevancy and feature distribution similarity, respectively. Subsequently, the block features need to be correlated using a cross-correlation function. Finally, the unsupervised algorithm chosen for this study is Agglomerative Hierarchical Clustering (AHC). It will cluster the feature vectors to make it easy to read and identify irregularities in the code.

1.4 Research Objectives

This research aims to explore the potential of unsupervised learning, particularly Agglomerative Hierarchical Clustering (AHC), in clustering blocks in assembly files both efficiently and accurately using a known structure of pre-made executable files or source code. AHC will produce Highly Correlated Data Flow Graphs (CDFGs) to provide block correlation values and connectivity, allowing us to read the system's structure and detect anomalies likelihood in code through clustering blocks that form functional classes.

1.5 Research Benefits

Previous studies have focused on using supervised learning to detect irregularities in code, such as Bragen (2015) realized. Thus, exploring the use of unsupervised learning, which requires no prior training dataset, offers new potential

for future research to improve both the efficiency and accuracy of code anomaly detection.

CHAPTER II

LITERATURE REVIEW

Source code has always been a trade secret for developers and companies. To prevent unauthorized adversaries from reading and modifying the source code files, developers compile these files into an executable file that contains only a binary structure, which is readable by machines. With the extensive process of disassembling this binary structure into a human-readable format, it is possible to understand the workflow and processes the software undergoes.

The process of disassembling binary files is familiar; currently, numerous software options are available for this task, such as IDA Pro, GHIDRA, and objdump, a built-in Linux disassembling tool. These tools can convert binary files into human-readable structures like assembly files, but the resulting content can be extensive, depending on the number of source code files compiled. It is important to focus on critical features. For instance, Lyda and Hamrock (2007) emphasized the importance of using entropy analysis to measure uncertainty in a series of numbers or bytes. Additionally, Obert and Loffredo (2020) outlined various methods for feature extraction and dimensionality reduction, including Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), and Canonical Correlation Analysis (CCA). While these methods are potential candidates for feature extraction and dimensionality reduction, a thorough analysis of their computational complexity is necessary for optimizing efficiency.

Prior research on randomized Principal Component Analysis (PCA) had been done previously by Warmuth and Kuzmin (2007). The PCA algorithm probabilistically selects a k -dimensional projection matrix (P^t) based on internal parameters in each iteration. Upon receiving a data instance (x^t), it calculates the loss as:

$$\text{Loss}_t = \|x_t - P_t P_t^T x_t\|^2 \quad (1)$$

representing the squared distance between the data and its projection. The internal parameters are then updated based on this loss. Warmuth and Kuzmin (2007) designed the algorithm to minimize the cumulative loss across iterations, aiming to approach the performance of the optimal k -dimensional subspace P determined in hindsight. While PCA can measure low dimensional approximation with provable bounds, it has a computational complexity of $O(d^2n + d^3)$ for n samples and d dimensions.

On the other hand, McLachlan (1992) proposed the Linear Discriminant Analysis (LDA) method used for dimensionality reduction and classification. LDA aims to find a weight vector w that maximizes the function:

$$J(w) = \frac{w^t S_B w}{w^t S_W w} \quad (2)$$

where S_B and S_W are the between-class and within-class scatter matrices, respectively. By maximizing this ratio, LDA aims to achieve maximal class separability. Once w is determined, new samples can be classified by projecting them onto vector $y = w^T x$. LDA can extract significant features from a dataset by finding the linear combination of features that maximizes the separation between different classes, therefore aiding us with a practical classification or dimensionality reduction. However, one downside of LDA is that it assumes that the features are normally distributed and that the classes have identical covariance matrices, which may only be held for some datasets. Furthermore, like PCA, LDA has a computational complexity $O(d^2n + d^3)$ for n samples and d dimensions.

Alternatively, Huang et al. (2009) suggested the potential of Canonical Correlation Analysis (CCA) as a measure of association between two sets of random variables. CCA can find linear combinations of feature vectors within a cross-covariance matrix with maximum correlations where the equation:

$$\text{Correlation}(V_x, V_y) = w_x^T \Sigma_{xy} w_y \quad (3)$$

is maximized with w_x and w_y as weight vectors. Dimensionality reduction is achieved by extracting only features with the highest correlation (Obert and Loffredo, 2020). Unfortunately, CCA has a significant computational complexity of $O(d_1^2n + d_2^2n + d_3^1 + d_3^2)$ where d_1 and d_2 are the dimensions of the two datasets compared.

After conducting further research on each method, it can be concluded that PCA, LDA, and CCA all seek linear combinations of features that accurately describe the data. These methods consider the relationships between features to identify significant ones, and they can be computationally complex. However, according to Borda (2011), measuring the Shannon Entropy of each feature distribution does not account for inter-feature dependencies. In terms of efficiency, Shannon Entropy has a computational complexity of $O(n)$ or $O(1)$ if the distribution is a known priori, making it a valuable metric for selecting significant features when computational resources are limited and when certain feature distributions exhibit low variance (Obert and Loffredo, 2020).

The previously mentioned metrics provide significant vector values that benefit unsupervised learning algorithms. Agglomerative Hierarchical Clustering (AHC) has been widely utilized in various domains. Triayudi & Fitri (2019) introduced a new AHC technique, SLG, which demonstrated high validity in analyzing student activity in online learning (Triayudi & Fitri, 2019; Yang & Li, 2023) compared the performance of an improved AHC algorithm with other standard clustering algorithms, highlighting the significance of AHC in intelligent grid systems (Yang & Li, 2023). Furthermore, Zhao et al. (2005) emphasized the focus on agglomerative methods in hierarchical document clustering, indicating the prevalence and importance of AHC in document datasets (Zhao et al., 2005).

AHC offers several benefits. It effectively analyzes student activity in online learning, as evidenced by the high validity index obtained by the SLG technique (Triayudi & Fitri, 2019). Additionally, AHC is valuable in smart grid systems, as demonstrated by the improved AHC algorithm's performance compared

to other clustering algorithms (Yang & Li, 2023). Moreover, AHC is prevalent in hierarchical document clustering, indicating its suitability for organizing and structuring document datasets (Zhao et al., 2005).

However, AHC also presents limitations. While it is effective in certain domains such as online learning and smart grid systems, its applicability in other areas may be limited. For instance, its performance in clustering document datasets may not be as effective in other types of data. Additionally, the computational complexity of AHC, as highlighted by (Zhao & Karypis, 2002), may pose limitations in handling large datasets (Zhao & Karypis, 2002). Furthermore, AHC has demonstrated its effectiveness in various domains such as online learning, smart grid systems, and hierarchical document clustering. Its benefits include high validity in analyzing student activity, superior performance in smart grid systems, and suitability for organizing document datasets. However, its limitations may arise in handling diverse types of data and dealing with computational complexity, especially in large datasets.

Table 2.1 Table of Comparison

No.	Author	Description	Conclusion
1	Obert and Loffredo (2020)	Application of information metrics and unsupervised learning model, to visualize the content structure of binary files through block correlations and hierarchy.	This research described a simplified approach to data flow analysis that uses unsupervised learning and dimension reduction to identify dominating patterns of data flow, functionally classify blocks, and minimize the complexity of data flows.
2	Lyda and Hamrock (2007)	Provides a reference and guide for entropy metrics usage on different levels of encryption of files for malware.	This research shows, through graphs and quantitative results, the significance of using binary entropy tools to analyze and generate statistics on malware collections that contained packed or encrypted samples.
3	Bragen (2015)	Application of supervised learning classifiers to detect malware through opcode sequence analysis.	Classification of malware files or benign files can be done just by extracting only the opcodes from Portable Executable (PE) files, with consideration of n-grams. While the accuracy of the classifier is better with higher n-grams, the computational complexity increases exponentially.

No.	Author	Description	Conclusion
4	Warmuth and Kuzmin (2007)	Adaptation of an online algorithm design for Principal Component Analysis (PCA).	Warmuth and Kuzmin (2007) developed a new set of techniques for low dimensional approximation with provable bounds and lifted the algorithms and bounds developed for a diagonal case to the matrix case
5	McLachlan (1992)	A framework of discriminant analysis and its use case.	Provides an in-depth discussion of discriminant analysis and its relationship between the group membership and the feature vector.
6	Huang et al. (2009)	Huang et al. (2009) designed non-linear metrics for what was a Linear Canonical Correlation Analysis (LCCA).	The paper signifies the potential and possibility of adapting non-linear methods for Canonical Correlation Analysis (CCA) by introducing a nonlinear and nonparametric kernel method for an independent test for two sets of variables.
7	Triayudi & Fitri (2019)	Triayudi & Fitri (2019) introduced SLG, a new AHC technique, demonstrating high validity in analyzing student activity in online learning.	The new AHC technique, SLG, effectively models student activity in online learning, demonstrating high validity in analyzing student activity with a cophenetic correlation coefficient (CPCC) of 0.9237, 0.9015, 0.9967,

No.	Author	Description	Conclusion
			0.8853, 0.9875 for the five datasets compared to conventional AHC methods.
8	Yang & Li (2023)	Yang & Li (2023) Compared the performance of an improved AHC algorithm with other common clustering algorithms, emphasizing its significance in smart grid systems.	AHC is valuable in identifying vulnerable lines in smart grid systems, as the improved AHC algorithm outperformed other clustering algorithms, highlighting its significance in smart grid systems
9	Zhao et al. (2005)	Emphasizes the focus on agglomerative methods in hierarchical document clustering, indicating the prevalence and importance of AHC in document datasets.	The focus on agglomerative methods in hierarchical document clustering indicates the prevalence and importance of AHC in organizing and structuring document datasets
10	Zhao & Karypis (2002)	Evaluates hierarchical clustering algorithms for document datasets, revisiting the question of the superiority of agglomerative approaches over partitional approaches.	Experimental results showed that AHC consistently leads to better hierarchical solutions than agglomerative or partitional algorithms alone, emphasizing its effectiveness in hierarchical clustering

CHAPTER III

THEORETICAL BASIS

3.1 Binary Files

A binary file is a computer file that contains data in a binary format, which is a sequence of bits. These files can store various types of data, including text, images, audio, and executable programs. Binary files are not human-readable and require specific software to interpret and process the data they contain. They are used to store information in a format that is directly understandable by the computer's hardware, allowing for efficient storage and processing of data. Additionally, binary files are used to represent complex data structures and are essential for tasks such as binary code analysis, binary modification, and binary translation.

3.2 ELF Files

An ELF (Executable and Linkable Format) file is a common standard file format for executable files, object code, shared libraries, and core dumps. It is used on Unix and Unix-like systems, including Linux, and is designed to be a portable and executable file format. ELF files contain headers that define the file structure, program and section headers, and various sections such as the `.text` section, which holds the executable code, and the `.data` section, which holds initialized data.

The characteristics of an ELF file include its ability to support multiple architectures, its extensibility to add new sections, and its use of program headers to define the program's memory layout when it is executed. ELF files play a crucial role in executing programs on Unix-based systems and are essential for the functioning of the operating system and applications. The `.text` section in an ELF file contains the program's executable code in machine code format. This section is essential for the program's execution as it holds the CPU's instructions.

The `.text` section interacts with other sections, such as the `.data` section, which holds initialized data, and the `.bss` section, which holds uninitialized data. During the program's execution, the `.text` section interacts with these other sections

by accessing and modifying the data they contain, allowing the program to perform its intended functions. The .text section is critical for the program's functioning and is a fundamental component of the ELF file format, enabling the execution of the program's instructions on the underlying hardware.

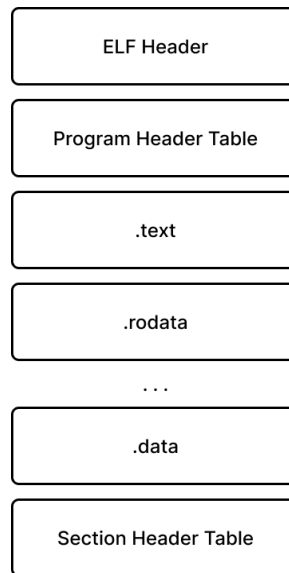


Figure 3.1 ELF Format

Figure 3.1 illustrates the structure of an ELF file, which shows the organization of different sections within the file, including the ELF Header, Program Header Table, various sections like .text (code), .rodata (read-only data), and .data (initialized data), followed by the Section Header Table. The text section of the ELF file, containing both the blocks and their corresponding instructions and operands, will be the central area of interest for data flow analysis. The instructions or opcodes, left operands, and right operands are the specific features of the disassembled file to be extracted. Visualizing the interaction between these features will allow us to analyze and understand the underlying pattern or behavior of the program without the need for the source code.

3.3 Instruction and Operand Extraction

To disassemble a binary file in Linux, one can use the objdump tool, a widely used disassembler for extracting assembly instructions from binary files

(Subedi et al., 2018). Objdump is a powerful tool that can be leveraged to analyze the memory usage of assembly code (Verbeek et al., 2020). It is also used for performing call-graph analysis and studying the usage of the Linux API in disassembled binaries. However, it is important to note that complete disassembly coverage is only possible if the binary contains relocation and symbolic debugging information (Pappas et al., 2012). The correct base address is also crucial for accurate cross-references in instances where the address reference uses absolute addresses rather than offsets in the binary file (Zhu et al., 2016). The objdump output will consist of block addresses, instructions, left operand, and right operand. An example of the output of the objdump tool can be seen in Figure 4.1.

Once the assembly instructions are extracted using objdump, regular expressions can extract specific elements such as instructions, left operands, and right operands from the disassembled code. Regular expressions have been the dominant approach for text extraction and have been widely used for information extraction in various domains (Li et al., 2008). They have also been utilized for filtering and extracting specific patterns from disassembled code (Ladas et al., 2023). However, it is essential to be mindful of the practical problems faced by developers when using, fixing, and testing regular expressions, as highlighted in recent studies (Wang et al., 2021; Wang, 2021).

3.4 Shannon Entropy

Shannon Entropy, a concept named after Claude Shannon, serves as a barometer for the unpredictability or information density in a dataset. At its core, it utilizes the formula:

$$H(x) = \sum_{i=1}^n p(x_i) \log_b p(x_i) \quad (4)$$

where $p(x_i)$ symbolizes the likelihood of a specific value or symbol appearing in the data. This metric shines in its ability to encapsulate the essence of information or uncertainty within a dataset. Essentially, Shannon Entropy outputs a singular value, reflecting the average informational output of a random variable. A higher value denotes more significant uncertainty or rich information content, while a

lower value suggests the opposite. Its feature selection and extraction efficiency are invaluable, especially in pinpointing the most informative elements within a dataset, as Cover & Thomas (2001) noted.

```
Procedure CalculateEntropy(data)
{
1.   entropy := 0;
2.   total_count := sum(data.values());
3.   FOR EACH value IN data DO
4..   probability := data[value] / total_count;
5.   entropy -= probability * Log2(probability);
6.   RETURN entropy;
7.   END Procedure
}
```

Figure 3.2 Pseudocode for Calculating Entropy using Shannon Entropy

Figure 3.2 shows a procedure to quantify the uncertainty or diversity within a dataset. It initializes the entropy sum, iterates over each data point to compute its probability, and then aggregates these probabilities, weighted by the logarithm of the probability, to arrive at the entropy value. This value reflects the informational richness of the dataset, and it's useful in fields like machine learning and data analysis to evaluate the significance of different features.

In deconstructing a binary file to extract elements like instructions, and the left and right operands, Shannon Entropy is a powerful tool to help in sifting through the features based on their informational weight. By evaluating the entropy of various components in a disassembled file, features with higher entropy values symbolize more uncertainty and information, which should be prioritized. Conversely, features with lower entropy are less informative and can be sidelined. This methodology streamlines the feature extraction process and enhances the analysis of the binary code to find the best features as representatives of a block function.

3.5 Kullback-Leibler Divergence (KL-Divergence)

KL-Divergence, or Kullback-Leibler Divergence, quantifies how one probability distribution diverges from another. Using the formula:

$$D_{KL}(P\|Q) = \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{p(x)}{q(x)} \right) \quad (5)$$

Equation (5) compares two distributions, P and Q , based on the probabilities $p(x)$ and $q(x)$ of an event x in these distributions. This measure is particularly useful in analyzing disassembled files, where it assesses similarity between blocks based on features like instructions, left operand, and right operand, quantifying differences between two probability distributions.

In comparing blocks of assembly, KL-Divergence calculates the divergence between the probability distributions of features like instructions and operands in different blocks, revealing the degree of similarity or dissimilarity among them. Blocks with lower KL-Divergence values show higher similarity, possibly indicating related functionalities. Conversely, higher values suggest significant differences, pointing to distinct functionalities. This method is vital for identifying blocks with analogous feature distributions, aiding in deciphering the binary file's structure and functionality.

3.6 Agglomerative Hierarchical Clustering (AHC)

Agglomerative hierarchical clustering or AHC is widely used in unsupervised learning, particularly in constructing a hierarchical tree based on the similarity between data points (Yang & Li, 2023). This method follows a bottom-up approach, where each data point starts as its cluster and then iteratively merges with the closest cluster based on a similarity measure, ultimately forming a hierarchical cluster tree (Yang & Li, 2023). The agglomerative hierarchical clustering method is known for its versatility and has been applied in various domains, such as intrusion detection systems, smart grid systems, and online learning analysis (Song et al., 2014; Triayudi & Fitri, 2019). For instance, in intrusion detection systems, Song et al. (2014) utilized agglomerative hierarchical clustering combined with mutual information theory to group features for effective intrusion detection.

Furthermore, agglomerative hierarchical clustering has been applied in diverse fields, such as document clustering, multi-document summarization, and visual representations (Naveen & Nedungadi, 2014; Noroozi & Favaro, 2016; Tamura et al., 2012). This method has also been extended to address specific challenges, as seen in the work of Tamura et al. Rabiner (1989) who proposed a two-stage clustering approach using agglomerative hierarchical algorithms and one-pass k-means to overcome the computational complexity inherent in agglomerative hierarchical clustering. Additionally, the theoretical basis of agglomerative hierarchical clustering has been discussed in the literature, emphasizing its mathematical structure and its potential for a wide range of applications (Liu et al., 2019). Therefore, agglomerative hierarchical clustering serves as a fundamental technique in unsupervised learning, offering a flexible and robust approach for clustering and analyzing complex datasets across various domains.

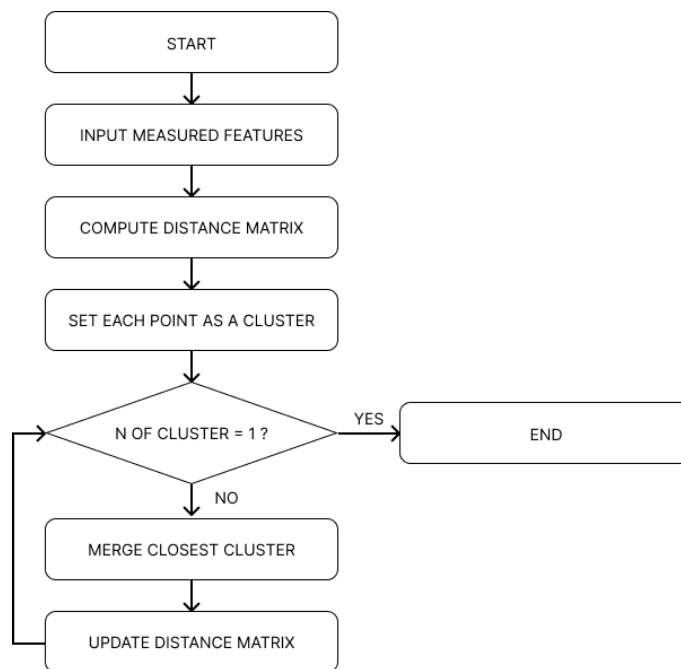


Figure 3.3 Agglomerative Hierarchical Clustering Algorithm Process

Figure 3.3 illustrates the stages of the Agglomerative Hierarchical Clustering algorithm. It begins with inputting measured features for data points,

followed by the computation of a distance matrix to quantify the similarity between points. Each data point is initially considered as an individual cluster. The algorithm then iteratively merges the closest clusters and updates the distance matrix accordingly. This process continues until all points are grouped into a single cluster, marking the end of the algorithm's execution.

```

Procedure AgglomerativeHierarchicalClustering( $V_1, V_2, V_3, \dots, V_n$ )
{
1.  measured_features :=  $V_1, V_2, V_3, \dots, V_n$ ;
2.  distance_matrix := Compute_Distance_Matrix(measured_features);
3.  clusters := Set_Each_Point_As_A_Cluster(measured_features);
4.  WHILE Length(clusters) > 1 DO
5.      {closest_pair} := Find_Closest_Cluster_Pair(distance_matrix, clusters);
6.      clusters := Merge_Clusters(clusters, closest_pair);
7.      distance_matrix := Update_Distance_Matrix(distance_matrix, clusters, closest_pair);
8.  END WHILE
9.  RETURN clusters;
10. END Procedure
}

```

Figure 3.4 Pseudocode of Agglomerative Hierarchical Clustering

Figure 3.4 outlines the Agglomerative Hierarchical Clustering algorithm based on the general process shown in Figure 3.3. It begins by treating each data point V_i as a separate cluster and then calculate a distance matrix to represent the dissimilarity between every pair of clusters. The algorithm's core is a loop that continuously identifies and merges the two closest clusters until all points are unified into a single cluster. This loop works by finding the pair of clusters closest to each other, merging them into one, and then updating the distance matrix to reflect the new distances between the newly formed cluster and all other clusters. The process repeats, reducing the number of clusters by one each time until only one remains. This final cluster is the algorithm's output, representing the hierarchical clustering of the input data points. The product of the AHC is a dendrogram that displays the dissimilarity between blocks based on the feature distribution. The example output of the algorithm would appear as Figure 3.5 shows.

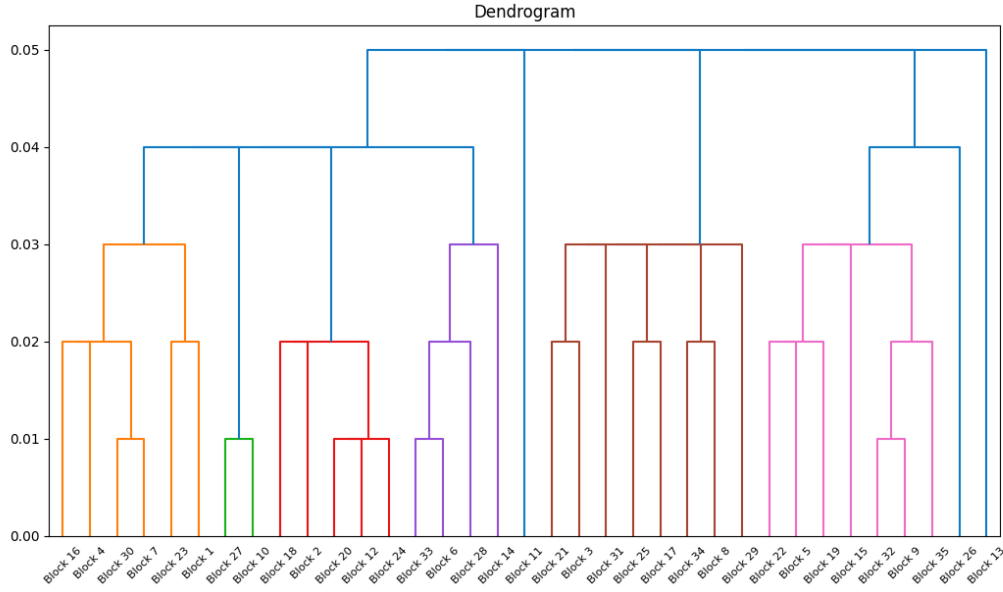


Figure 3.5 Example of Dendrogram Produced by AHC Algorithm

3.7 Silhouette Coefficient as an Evaluation Metric

The Silhouette Coefficient is a widely used metric in unsupervised learning to evaluate the quality of clustering (Wang & Srinivasan, 2015). It measures how similar an object is to its cluster compared to other clusters, providing a way to assess the appropriateness of the clustering process (Ferwana, 2019). The coefficient is calculated using the mean intra-cluster distance and the mean nearest-cluster distance for each sample, with values ranging from -1 to 1. A high value indicates that the object is well-matched to its cluster and poorly matched to neighboring clusters, signifying a good clustering result (Ferwana, 2019). This metric is particularly beneficial in agglomerative unsupervised learning, as it provides a quantitative measure of the compactness and separation of clusters, aiding in assessing the accuracy of the clustering process (Webb et al., 2022).

The Silhouette Coefficient offers several advantages for evaluating clustering accuracy in agglomerative unsupervised learning. It provides a clear and intuitive measure of how well-defined the clusters are, enabling the identification of the optimal number of clusters for a given dataset (Maulana & Al-Khowarizmi, 2022). Additionally, the Silhouette Coefficient allows for comparing different

clustering algorithms and parameter settings, facilitating the selection of the most suitable approach for a specific dataset (Webb et al., 2022). Furthermore, it aids in identifying potential issues such as overlapping clusters or poorly separated data points, thereby guiding the refinement of the clustering process to improve accuracy (Wang & Srinivasan, 2015). Overall, the Silhouette Coefficient is a valuable tool in agglomerative unsupervised learning, enabling data scientists to assess and optimize the accuracy of clustering results quantitatively.

$$\text{Silhouette Coefficient} = \frac{b-a}{\max(a,b)} \quad (6)$$

Equation (6) represents the average distance from the object to the other objects in its cluster, thus quantifying how well it fits within its cluster. Conversely, b is the smallest average distance from the object to objects in a different cluster, thus reflecting the object's separation from the nearest cluster that it is not a part of. The Silhouette Coefficient ranges from -1 to +1, where a value close to +1 denotes that the object is well placed within its cluster and far from neighboring clusters. A value near 0 indicates that the object is on or very close to the decision boundary between two neighboring clusters. Lastly, a value towards -1 suggests that the object is poorly placed and might have been assigned to the wrong cluster.

3.8 Block Feature Correlation Using Cross-Correlation Function

The cross-correlation function measures the similarity between two data series, like feature vectors of block F_1 and block F_2 , and is key in data flow analysis of disassembled binary files. It helps establish relationships among instructions and operands by correlating feature vectors, enabling analysts to classify code blocks in the binary file. This process offers insights into the data flow and interdependencies, which are crucial for understanding the code's behavior and functionality. The maximum cross-correlation vector $K_q = \max(C_d)$ for all instructions, left operands, and right operands can be found using the equation:

$$C_d = \frac{\sum_{a=1}^N (F1_{a-\text{mean}(R)}) (F2_{a-d-\text{mean}(F2)})}{\sqrt{\sum_{a=1}^N (F1_{a-\text{mean}(F1)})^2} \sqrt{\sum_{a=1}^N (F2_{a-d-\text{mean}(F2)})^2}} \quad (7)$$

3.9 Correlated Data Flow Graph

The Correlated Data Flow Graph (CDFG) is a representation of a program's control and data flow, where the vertices represent the program's features, and the edge weights represent the correlation between these features. In the context of analyzing disassembled binary files, CDFGs are used to establish relationships among instructions and operands by correlating feature vectors, enabling analysts to classify code blocks in the binary file.

The maximum cross-correlation vector K_q obtained using the cross-correlation function is used as the edge weight values for the edges connecting the vertices in the adjacency matrix, forming the CDFG. When analyzing CDFGs, the vertices that are strongly connected with edge weights K_q greater than a predetermined correlation value are of most interest in data flow analysis. This approach offers insights into the data flow and interdependencies, which are crucial for understanding the behavior and functionality of the code in the binary file. A premade example of a CDFG with dummy data showing different blocks and their correlations can be found in Figure 3.5.

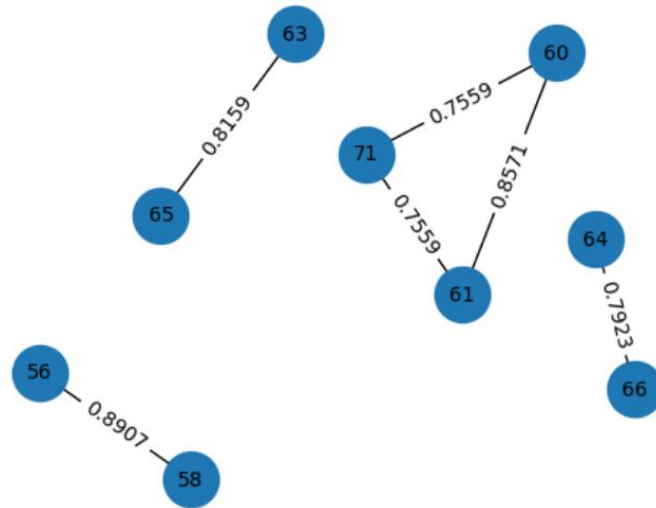


Figure 3 6 An example of highly correlated data flow graph

CHAPTER IV

RESEARCH METHODOLOGY

4.1 Research Description

The research methodology and application of previously mentioned metrics and methods in the theoretical basis section are proposed in the following sections. The methodology proposed in this research is a reproduction of the paper by Obert and Loffredo (2020). A critical goal of the research is to ensure the reproducibility of the research, ensuring that the same methods should result similarly. The experiment's output will be Correlated Data Flow Graphs (CDFGs) which can be used to analyze the block structure and functions of a program without the need of the source code.

4.2 Environment Set-Up and Data Acquisition

The research will be conducted using ELF (Executable and Linkable Format) files, which are specific to Unix-like systems. Therefore, a Linux-based environment is essential for accurate testing and analysis. Throughout the experiments, we will primarily utilize Debian-based Linux distributions, with a particular emphasis on Ubuntu. These distributions provide a robust foundation for our research. Additionally, for compatibility with Windows systems, Linux virtual machines may be employed to suit the x86/x86-x64 architecture. These virtual environments will replicate the necessary conditions, ensuring consistency across platforms.

4.2.1 Building Custom Programs

To ensure the validity of our research results, we will develop custom programs from the ground up. This strategy allows us full transparency into the program's functionality and background operations. Given large-scale programs' extensive size and complexity with numerous features, the custom-built programs will incorporate only basic functionalities. The research will focus on testing three types of programs that cover arithmetic operations (simple calculator), pointer manipulation (dynamic array allocator), and I/O file handling (CSV parser). All

programs will be coded in C++ and compiled with GCC. By the end of this process, we aim to have three functioning programs, each demonstrating a different capability.

4.2.2 Disassembling ELF Files

To get the disassembled content of the ELF files, we need to run the `objdump` command on the target ELF file. This will expose the assembly structure of the program. With the appropriate flags, we can select the desired range of content to avoid noise features.

```
objdump -d -j .text program.exe > output.txt
```

The above command provides a template for utilizing `objdump` to disassemble an ELF file of interest. A critical segment for observation is the text section, which contains the executable code. To isolate this section, the `-j .text` flag is employed in conjunction with the disassembly option `-d`. The basic command outputs the disassembled contents directly to the terminal. To preserve the output in a format-like Figure 4.1, it is necessary to append the redirection operator `> output.txt` to the command. This operation diverts the disassembled text section into a text file for further examination.

```
00000000000021f0 <__cxx_global_var_init>:
21f0: 55                push    %rbp
21f1: 48 89 e5          mov     %rsp,%rbp
21f4: 48 8d 3d 0e 5f 00 00 lea     0x5f0e(%rip),%rdi
21fb: e8 70 ff ff ff    call    2170 <_ZNSt8ios_base4InitC1Ev@plt>
2200: 48 8b 3d f1 5d 00 00 mov     0x5df1(%rip),%rdi
2207: 48 8d 35 fb 5e 00 00 lea     0x5efb(%rip),%rsi
220e: 48 8d 15 e3 5e 00 00 lea     0x5ee3(%rip),%rdx
2215: e8 c6 fe ff ff    call    20e0 <__cxa_atexit@plt>
221a: 5d                pop     %rbp
221b: c3                ret
221c: 0f 1f 40 00       nopl    0x0(%rax)
```

Figure 4.1 Example of a disassembled binary file

4.3 Feature Extraction

The type of data we want to extract from disassembling the file is a significant feature of the program. In this case, the features are the instructions, left operands, and right operands. To extract the features from the disassembled file, we can use a Python script that runs over the disassembled file such as in Figure 4.1 by identifying the blocks first, then the instructions, left operands, and right operands contained within the block.

4.3.1 Entropy Based Significant Feature Extraction

The content of disassembled binary files often includes outlier data, or 'noise', which does not contribute to defining the functionality of the blocks. To isolate only pertinent features, Shannon Entropy can be applied to calculate the distribution entropy of each distinct feature within a block. This process involves utilizing a Python program to analyze the disassembled text file, which encompasses the names of the blocks along with their respective instructions and operands.

Initially, the program is tasked with navigating through the file to pinpoint blocks, identifiable by their ID and name as illustrated in Figure 4.1. For instance, consider the block identified as `00401000 <_start>:`. Regular expressions are employed to sift through the text file and detect this specific block naming pattern, characterized by the sequence `<[^>]+>:`, or, more broadly, any text enclosed between `<` and `>:`. Upon successfully identifying the blocks, the program then proceeds to evaluate each block, calculating the frequency of each unique feature. This frequency data is then used in conjunction with the Shannon Entropy equation, as outlined in Equation (4), to determine the entropy of each feature. A pattern may emerge, revealing numerous features with low entropy values, suggesting their ubiquity within the block, and implying a lack of significance in representing the block's functionality.

To address this issue, a threshold can be established to filter out features whose entropy values fall below a certain level. This effectively isolates features

whose entropies are sufficiently high to be deemed indicative of the block's functionality. This methodology is systematically applied to each block across all disassembled programs.

4.4 Block Characterization

In the characterization of blocks based on their functionality, this study employs two primary metrics: the Shannon Entropy of feature distribution and the Kullback-Leibler (KL) Divergence for assessing similarity. Shannon Entropy quantifies the extent to which a feature accurately represents a block's functionality, as previously explained. Conversely, KL-Divergence measures the similarity in block functionality by examining the distribution of features across pairs of blocks.

Furthermore, the associated feature vectors facilitate the use of Agglomerative Hierarchical Clustering (AHC) for classifying the blocks. This clustering approach enables the delineation of functional association patterns among the blocks. Consequently, blocks clustered together are deemed to belong to the same functional category and are labeled accordingly. Significantly, AHC's role as an unsupervised learning algorithm facilitates classification without the need for a pre-trained model. This approach offers greater efficiency compared to supervised learning algorithms, which require pre-labeled, pre-trained models.

4.4.1 Implementing Agglomerative Hierarchical Clustering (AHC)

Agglomerative Hierarchical Clustering (AHC) begins by treating each data point as a distinct cluster. This initial approach creates as many clusters as there are data points, ensuring that all potential groupings and inherent structures within the data are considered from the onset. The algorithm proceeds iteratively, at each step merging the pair of clusters with the least dissimilarity, determined by a preselected linkage criterion. This criterion is central to the clustering process, with common variants including single linkage, complete linkage, average linkage, and Ward's method. The chosen criterion significantly influences the cluster characteristics, shaping them into compact spheres or elongated chains.

Throughout the iterative process, AHC continually updates the proximity matrix to reflect the dissimilarities between newly merged clusters and all remaining clusters. This constant recalibration of the matrix is crucial for accurately identifying the next pair of clusters to merge. The evolving cluster hierarchy is represented in a dendrogram, a tree-like diagram that visually illustrates the sequence of mergers and the corresponding distances. This dendrogram acts as a navigational tool for determining the ideal number of clusters. Setting a dissimilarity threshold allows for the dendrogram's truncation, where the intersecting lines indicate the cluster count. This threshold is strategically selected to maximize intra-cluster homogeneity and inter-cluster heterogeneity, ensuring that the resulting divisions are meaningful and representative of the data's inherent structure.

4.4.2 Evaluating the Clusters Using Silhouette Coefficient

In unsupervised learning, the lack of labels complicates the evaluation of clusters. The Silhouette Coefficient is a useful tool for assessing whether features are correctly grouped, as it measures the similarity between a given feature and all cluster contents. This coefficient facilitates the determination of feature-cluster congruence. The evaluation process involves iterating this comparison for each feature across all clusters using Equation (6), enhancing the accuracy of cluster assignments.

The expected outcome of the evaluation metric is a set of values, each ranging from -1 to 1, for every feature. These values quantify the degree of alignment between each feature and its respective cluster. By graphing the features in relation to their Silhouette Coefficient values and clusters, we can evaluate the quality of the clustering. This, in turn, provides insight into the roles of the features and blocks within their functional clusters. This assessment is crucial as it directly influences our understanding of the functionality of these elements.

4.5 Constructing Correlated Data Flow Graphs

As illustrated in Figure 3.2, a Correlated Data Flow Graph (CDFG) facilitates the visualization of inter-block connectivity, which is based on the similarity of feature vectors. To attain this visualization, it is necessary to systematically compare each feature line within two blocks, considering a specified index gap. This comparison is conducted using Equation (7).

4.5.1 Calculating Block Feature Correlation

Upon determining the entropy and similarity of features across blocks, the resultant vector values are obtained. Each feature line within a block corresponds to a distinct vector value. To quantify the correlation of features between two blocks, the method iteratively computes the similarity of vectors from each pair of blocks, and computation execution uses Equation (7). The objective of employing Equation (7) is to identify the maximum cross-correlation vector across all possible combinations of instructions and left and right operands.

4.5.2 Generating Correlation Data Flow Graphs

In the Correlated Data Flow Graphs (CDFG) analysis, an adjacency matrix is constructed, wherein the features serve as vertices and edge weight values for the connections between these vertices (blocks). Within this framework, vertices that exhibit strong connections, characterized by edge weights K_q exceeding a predefined correlation threshold, are particularly significant for data flow analysis. Employing Equation (7), the analysis identifies blocks with maximal correlation by examining the relationships between instructions and operands associated with each block.

CHAPTER V

IMPLEMENTATION

5.1 Development Environment

The research will leverage two distinct operating systems at different stages of the workflow. The initial data generation phase will utilize Linux Ubuntu within a virtualized environment provided by VirtualBox. This selection is driven by the necessity to compile C++ programs using the Linux Clang compiler, which produces Executable and Linkable Format (ELF) files. Subsequently, the data will undergo processing and manipulation within a Windows 11 environment. This shift is motivated by the potential performance benefits offered by the native operating system on the underlying hardware.

Table 5.1 Environment Specifications

Technology	Specification
Operating System	Linux Ubuntu 22.04.2 64-bit Windows 11 23H2 64-bit
C++ Version	C++17
Clang Version	Clang 17
Work Station	Visual Studio Code 1.89.1

5.2 Dataset Generation

The dataset to be used in this research are three disassembled programs: a simple calculator using switch cases, a dynamic array allocator (DAA), and a CSV parser program. All programs are written in C++ and compiled using Clang compiler which can be found in this research repository as “simple_calculator.cpp”, “dynamic_array_allocator.cpp”, and “csv_parser.cpp” and their respective compiled ELF files and disassembled ELF files. Additionally, source code and output graphs and tables can also be found in this research repository¹.

¹ For access to the data set, please refer to the GitHub repository: <https://github.com/Markkreel/BINARY-STATIC-ANALYSIS-THROUGH-INSTRUCTION-AND-OPERAND-EXTRACTION-AND-AHC-ALGORITHM>

5.2 Feature Extraction Program

Assembly language instructions, while fundamental to computer architecture, are challenging to parse and analyze due to their low-level representation. As illustrated in Figure 4.1, the text segment of an assembly file contains memory addresses alongside the corresponding instructions and operands. To facilitate the classification of instruction blocks based on functional similarity, a structured and labeled format, such as a CSV file, is desirable. This section details the development of a Python script designed to extract four key elements from each instruction block: address, instruction mnemonic, left operand, and right operand. The script further assigns a unique block identifier to each sequence of instructions terminating with a control flow instruction. We call the program `extractor.py`.

```
def parse_instruction(line):
    line = line.strip().split(";")[0]

    match = re.search(r"^([0-9a-f]+):", line)
    address = int(match.group(1), 16) if match else None

    parts = re.split(r"(\s+|\s, (?<!\s)|\s)", line, maxsplit=3)
    instruction = parts[0]
    left_operand = parts[1].strip() if len(parts) > 1 else None
    right_operand = parts[2].strip() if len(parts) > 2 else None

    return {
        "address": address,
        "instruction": instruction,
        "left_operand": left_operand,
        "right_operand": right_operand,
    }
```

Code 5.1 Parsing instructions function to format assembly code

In code 5.1, the `parse_instruction()` function is the heart of the parsing process. It meticulously cleans each line by removing leading whitespace and comments. Regular expressions then extract the numerical address (without `0x`) and split the line into instruction and operands. The function handles cases with no operands or internal commas (e.g., register names). The source code can be found in this research repository as “`extractor.py`”.

```

def convert_to_csv(assembly_content):
    data = []
    block_id = 0
    for line in assembly_content.splitlines():
        if line.strip():
            instruction_data = parse_instruction(line)
            if instruction_data["address"]:
                data.append(
                    [
                        block_id,
                        instruction_data["address"],
                        instruction_data["instruction"],
                        instruction_data["left_operand"],
                        instruction_data["right_operand"],
                    ]
                )
            else:
                block_id += 1

```

Code 5.2 A function to convert the extracted code into CSV format

The `convert_to_csv()` function serves as the bridge between the assembly code and the desired CSV format (as depicted in Code 5.2). It iterates through each line of the assembly code, leveraging the `parse_instruction()` function to extract the relevant information. Block identification relies on the presence or absence of whitespace between addresses. If a parsed line holds a valid address, the function appends it to a data list along with its corresponding block identifier, address, instruction, and any left and right operands as distinct elements. Finally, upon processing all lines, the function constructs the CSV string with a header row and formatted data rows. The output of the `extractor.py` program produces a CSV file with the following data frame sample in Table 5.2:

Table 5.2 Data frame of disassembled code of simple calculator program

	Block_ID	Address	Instruction	Left Operand	Right Operand
1					
2	1	00000000000010b0	push	%rbp	
3	1	00000000000010b1	mov	%rsp	%rbp
4	1	00000000000010b4	lea	0x2fa6(%rip)	%rdi
5	1	00000000000010bb	call	1080<_ZNSt8ios_base4InitC1Ev@pl...	
6	1	00000000000010c0	mov	0x2f31(%rip)	%rdi

Content visualization can help understand the assembly code block size distribution for three programs of varying complexities. The content of these blocks is visualized in bar graphs (Figures 5.1–5.3) where instructions are depicted in purple, left operands in red, and right operands in green.

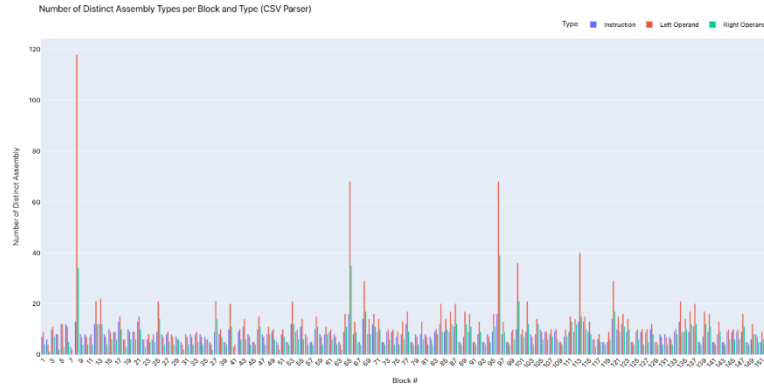


Figure 5.1 Content size of each block (CSV Parser)

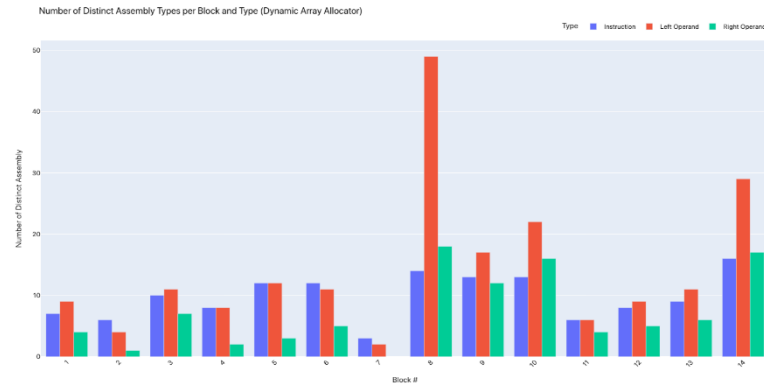


Figure 5.2 Content size of each block (Dynamic Array Allocator)

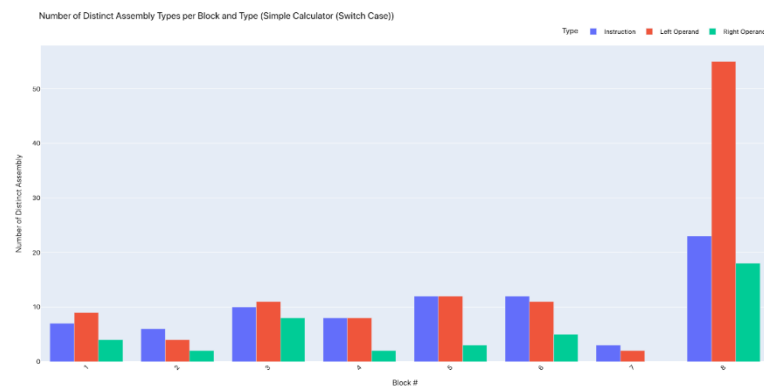


Figure 5.3 Content size of each block (Simple Calculator)

A cursory inspection of the block size distribution reveals a heterogeneity in code density. Some blocks exhibit a noticeably larger size compared to others,

suggesting potential differences in operational complexity. However, it is crucial to acknowledge that definitive conclusions regarding operational intensity cannot be solely drawn from block size. Further analysis is necessary to substantiate this initial observation.

5.3 Extracting Significant Features by Entropies Level

Shannon entropy provides a quantitative measure of the information content within a dataset. Data points with low entropy exhibit a high degree of predictability, often signifying redundancy, or noise. By calculating the Shannon entropy for each data point, we can identify those with values significantly lower than the average. Low-entropy points likely represent repetitive patterns or artifacts with minimal informational value. Subsequently, these points can be flagged for removal or further investigation to determine if their elimination impacts the integrity of the remaining data.

Using the program `shannon_entropy.py` (Appendix A), we calculate the probability of a specific assembly code appearing within their ζ and their type. The probability, substituted into equation 4, will give us the entropy of each assembly code relative to its block (ζ). The output is printed into a CSV file, see Table 5.3, which we can filter out in the future if necessary. The full source code can be found in this research repository as “entropy.py”.

Table 5.3 First 5 assembly codes distribution probability and entropy

	Block_...	Type	Assembly	Probability	Entropy
1					
2	1	Instruction	push	0.09090909090909091	0.3144937835124816
3	1	Left Operand	%rbp	0.18181818181818182	0.44716938520678134
4	1	Instruction	mov	0.18181818181818182	0.44716938520678134
5	1	Left Operand	%rsp	0.09090909090909091	0.3144937835124816
6	1	Right Operand	%rbp	0.09090909090909091	0.3144937835124816

5.3.1 Noise Removal via Entropy Thresholding

Individual data points within each block (ζ) often comprise hundreds of operational lines. However, these blocks are constructed with a limited set of instructions (ϵ) and operands (δ, γ). This redundancy leads to frequent repetition of

specific instructions and operands, consequently resulting in **low entropy**. Low-entropy data points offer minimal discriminatory power; they are more likely to appear frequently within a block and thus fail to effectively represent the broader dataset. To address this challenge, we employ a filtering mechanism that distinguishes between informative, high-entropy data and these repetitive, low-entropy elements. This process leverages the Z-score metric, which calculates the deviation of each data point's entropy (instructional, operand-based) relative to the average entropy of its corresponding block.

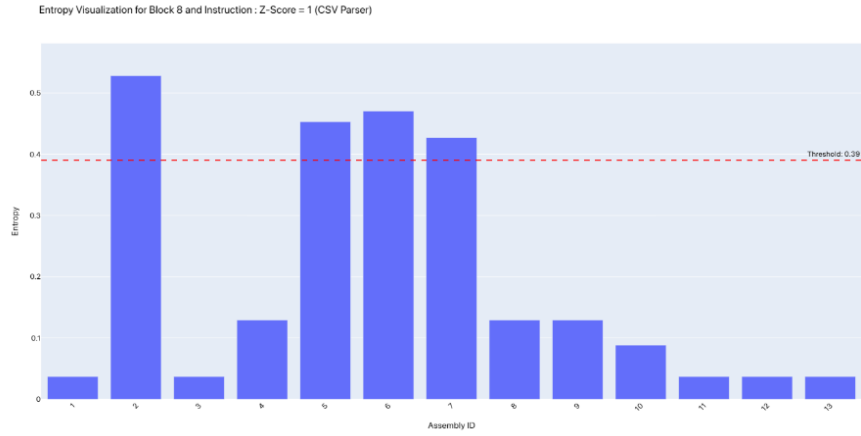


Figure 5.4 Distinct instruction (ϵ) Shannon entropies. (X-axis = Instructions)

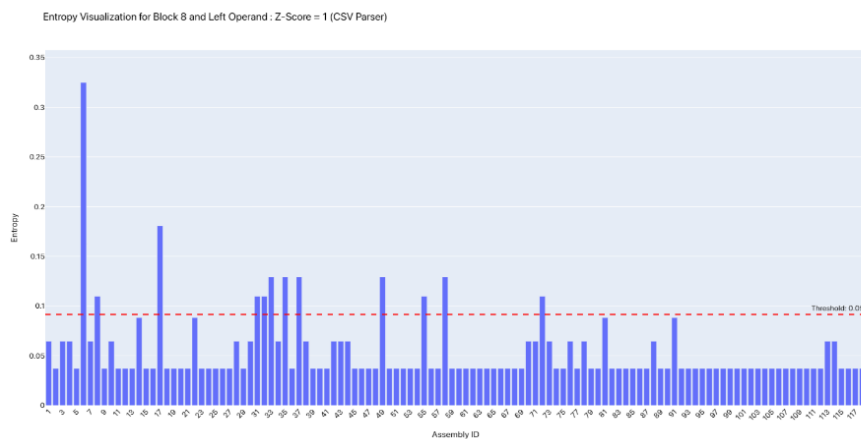


Figure 5.5 Distinct left operand (δ) Shannon entropies. (X-axis = Left operand)

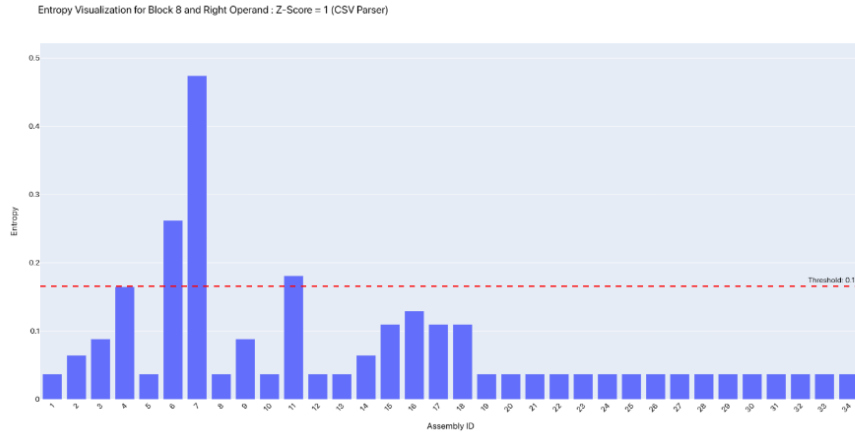


Figure 5.6 Distinct right operand (γ) Shannon entropies (X-axis = Right operand)

Figure 5.4–5.6 depict the Shannon entropy distribution for distinct instruction (ϵ), left operand (δ), and right operand (γ) elements within the 8th block of the CSV Parser program. A threshold (red line) is established to differentiate between low- and high-entropy data points. Assembly code values falling below this threshold are deemed noise and subsequently excluded from the dataset as they are unlikely to reflect the inherent characteristics of their corresponding block (ζ). The full source code can be found in this research repository as “threshold.py”.

```
for row in reader:
    block_id = row["Block_ID"]
    variable_type = row["Type"]
    entropy = float(row["Entropy"])
    threshold = thresholds[block_id].get(
        variable_type, 0
    ) # Default to 0 if threshold is not found
    if entropy >= threshold:
        writer.writerow(row)
```

Code 5.3 A loop to write with entropy above the threshold into a new CSV

The noise removal process is illustrated in Code 5.3. The program iterates through the original CSV file, identifying data points with entropy exceeding the predefined threshold. These elements are then selectively extracted and written into a new CSV file. This procedure yields a refined dataset comprised of instructions

and operands that provide a more accurate representation of their respective blocks, as visualized in Table 5.4.

Table 5.4 First 5 assembly codes probability and entropy after noise removal

1	Block_ID	Type	Assembly	Probability	Entropy
2	1	Left Operand	%rbp	0.18181818181818182	0.44716938520678134
3	1	Instruction	lea	0.2727272727272727	0.5112188503407658
4	1	Right Operand	%rdi	0.18181818181818182	0.44716938520678134
5	2	Instruction	push	0.16666666666666666	0.430827083453526
6	2	Left Operand	%rbp	0.3333333333333333	0.5283208335737187

5.4 Block Characterization Using KL-Divergence and AHC

Building upon the concept introduced in Chapter 3.6, we can leverage KL-Divergence to cluster functionally similar blocks. This technique calculates the similarity between two blocks by comparing the probability distributions of their constituent instructions (ϵ), left operands (δ), and right operands (γ) within their respective ζ vectors. In essence, KL-Divergence quantifies the difference between these distributions, providing a measure of how closely aligned two blocks are in terms of their operational characteristics.

5.4.1 Calculating KL-Divergence

As a reminder, the KL-Divergence metric ranges from 0 to positive infinity, where 0 signifies identical distributions (perfectly similar blocks) and increasingly larger values indicate progressively lower similarity. Code 5.4 illustrates the calculation of block similarity using the KL-Divergence formula presented in Equation (5).

```
def calculate_kl_divergence(p, q):
    epsilon = 1e-10 # small constant to avoid log(0)
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log2(p / q))
```

Code 5.4 A function to calculate KL-Divergence between two blocks

To effectively visualize these pairwise similarities, we can construct a block similarity matrix. This matrix, typically rendered as a heatmap as shown in Figure 5.7, offers a color-coded representation of the similarity scores between all block pairs. Hotter colors (typically red or orange) indicate higher similarity, whereas cooler colors (typically blue or green) signify lower similarity. Heatmaps provide a valuable tool for efficiently comprehending the overall similarity landscape within a dataset. Similarity matrix heatmap for simple calculator program and DAA program can be seen in Appendix J and Appendix K. Unfortunately, the CSV parser program has too many blocks to display in the form of a heatmap. The similarity matrix values can be found in this research repository as “csv_parser_block_similarity_normalized.csv” in GitHub.

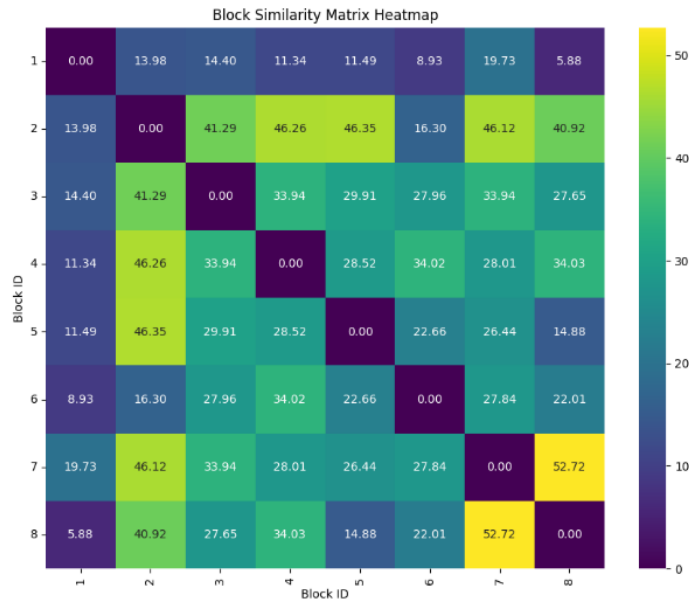


Figure 5.7 Block similarity matrix heatmap (Simple Calculator)

5.4.2 AHC Implementation

The values we would like to feed into the AHC algorithm are the similarity values we obtained from calculating the similarity between two blocks using KL-Divergence equation. First, the program will read the similarity matrix which takes the input of the similarity matrix CSV and the number of blocks in the assembly program.

```
def similarity_to_distance(similarity_matrix):
    max_similarity = np.max(similarity_matrix)
    distance_matrix = max_similarity - similarity_matrix
    return distance_matrix
```

Code 5.5 A function to convert similarity matrix into distance matrix

Code 5.5 shows a code snippet implementing a function called `similarity_to_distance()` that transforms a similarity matrix obtained from block comparisons into a corresponding distance matrix. The function first identifies the maximum similarity value within the entire matrix `max_similarity`. It then subtracts each element in the similarity matrix from this maximum value, thereby generating a new matrix `distance_matrix`. This conversion reflects the intuition that higher similarity scores in the original matrix translate to smaller distances in the resulting distance matrix. In essence, this approach establishes a distance metric where highly similar blocks (close to 1.0 similarity) exhibit minimal distances, while increasingly dissimilar blocks (lower similarity values) manifest as progressively larger distances within the distance matrix.

```
def perform_ahc(distance_matrix):
    Z = linkage(distance_matrix, method="ward")
    return Z
```

Code 5.6 Implementation of AHC

Building upon the distance matrix (obtained from the previous step), the `perform_ahc()` function leverages Agglomerative Hierarchical Clustering (AHC) to establish a hierarchical clustering structure (`Z`) for the data. The `linkage` function within SciPy is employed, utilizing Ward's minimum variance method for merging clusters during the hierarchical process. This approach iteratively merges the most similar clusters (those with the smallest distances in the distance matrix) until a single, all-encompassing cluster is formed. The resulting hierarchical clustering structure (`Z`) serves as a crucial output for further analysis and visualization of the underlying relationships within the data. An example of the dendrogram graph of the AHC implementation can be seen in Figure 5.8 which visualizes the growth of

the clusters. The full source code can be found in this research repository as “agglomerative_hierarchical_clustering.py”.

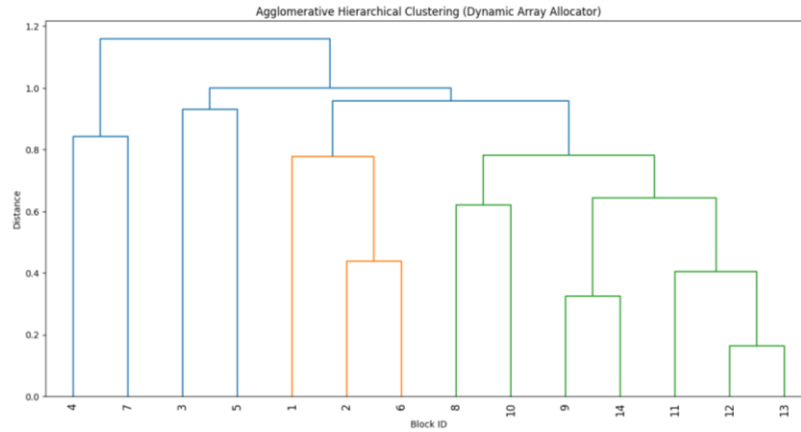


Figure 5.8 AHC dendrogram output (Dynamic Array Allocator)

5.5 Silhouette Coefficient as Evaluation Metric

As proposed to differentiate this research to prior research, Silhouette Coefficient is applied to the AHC algorithm to evaluate the accuracy of the clustering.

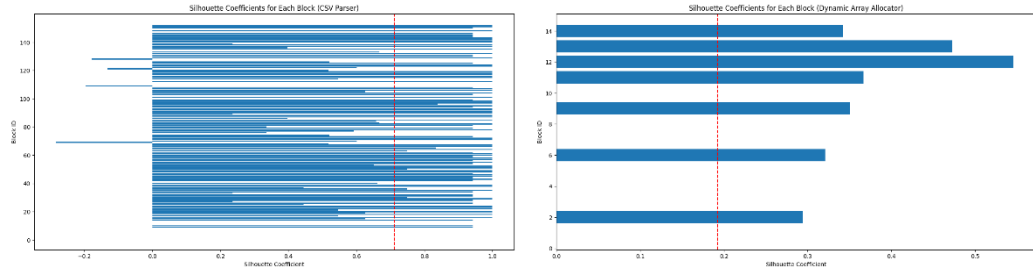
```
silhouette_values = silhouette_samples(distance_matrix,
                                     clusters,
                                     metric="precomputed")
average_silhouette_score = silhouette_score(
    distance_matrix, clusters, metric="precomputed"
)
```

Code 5.7 Calculating Silhouette Coefficient values for each block

Code 5.7 leverages the silhouette coefficient metric to assess the quality of the clustering generated by the Agglomerative Hierarchical Clustering (AHC) process. The `silhouette_samples()` function calculates the silhouette coefficient for each individual block within the dataset, considering the provided distance matrix (`distance_matrix`) and the resulting cluster assignments (`clusters`). The silhouette coefficient itself reflects how well a block is placed within its designated cluster. Values closer to 1 indicate a good fit, suggesting the block resides within a cluster with high similarity to other blocks and significant

dissimilarity to blocks in other clusters. Conversely, values closer to -1 imply a potentially poor fit, where the block might be positioned closer to members of a different cluster.

Furthermore, the code calculates the average silhouette score across all blocks using the `silhouette_score()` function. This score provides a single numerical value summarizing the overall cluster quality. Scores closer to 1 represent a well-defined clustering structure, while lower scores indicate a potentially suboptimal clustering outcome. These silhouette-based metrics serve as valuable tools for evaluating the effectiveness of the AHC clustering and potentially refining the clustering parameters for improved results. An example of the Silhouette Coefficient values for each block for the clusters shown in Figure 5.8 can be seen in Figure 5.9 below. The implementation of Silhouette Coefficient evaluation metric can be found in the AHC algorithm implementation, which can be found in this research repository as “`agglomerative_hierarchical_clustering.py`”.



**Figure 5.9 Silhouette Coefficient values for each block
(CSV Parser (Left) and Dynamic Array Allocator (Right))**

5.6 Generating Correlated Data Flow Graphs (CDFGs)

Clustering blocks based on similarity offer a visual representation of groups of blocks with analogous functionality. However, it does not indicate whether these similar blocks interact with each other. This research extends beyond examining the functional similarities between blocks by also considering the maximum correlation between blocks, which is based on the correlation between the instructions and operands associated with each ζ .

5.6.1 Calculating Block Correlation

The maximum correlation between two blocks can be determined using Equation (7), where each block, denoted as ζ , encompasses three variables at a given time frame. Each time frame, or block series, consists of three variables due to the nature of assembly addresses, which contain three elements: instruction (ϵ), left operand (δ), and right operand (γ). We treat all assembly instructions and operands within a block as a time series, sorted by their address order, and compare the similarities between two block series at frame $\phi = 3$. The series are iterated with a delay ∂ until the end of the series is reached. The highest value obtained from the cross-correlation function is taken as the correlation value between the two blocks, indicating the degree of interconnectivity between them.

Table 5.5 Top 5 highly correlated blocks (Simple Calculator)

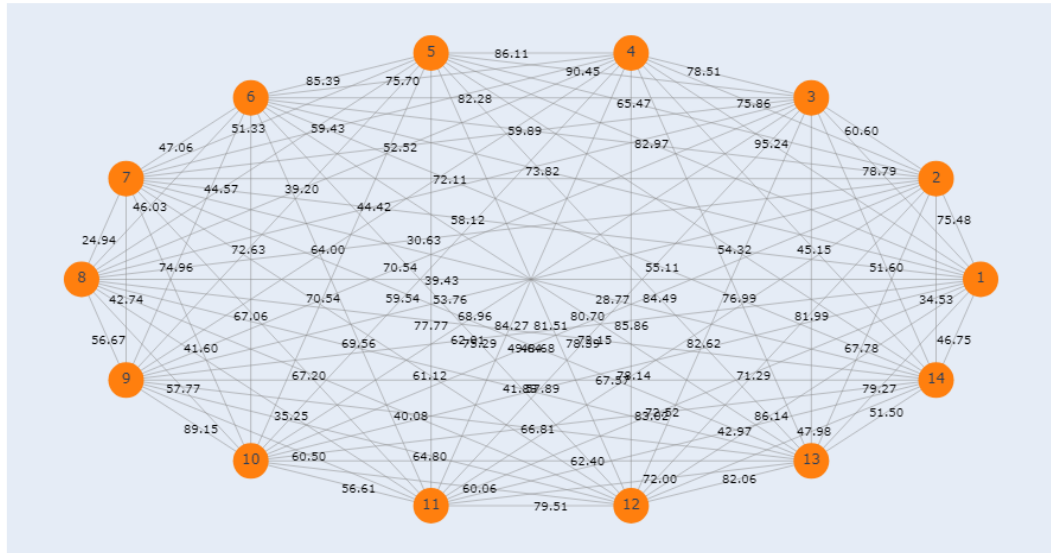
Block1	Block2	Correlation
1	4	93.804192
3	5	89.077958
5	6	87.429088
4	5	85.826985
3	6	84.496137

Table 5.5 presents the top five highly correlated blocks within the simple calculator assembly code. The correlation values range from 0 to 100, where a value of 100 signifies a high degree of correlation, and a value of 0 indicates no correlation.

5.6.2 Visualizing Highly Correlated Data Flow Graphs (CDFGs)

Visualizing the blocks as nodes and the correlation values as edge weights produces a correlated data flow graph (CDFG). The *Networkx* and *Plotly* libraries are powerful tools that streamline this visualization process. For each program, two types of CDFGs are generated: one that visualizes all block correlations and another

that visualizes the top 10 highly correlated blocks. Figures 5.10 and 5.11 display the example maximum correlation values between blocks for the dynamic array allocator assembly code. The full source code can be found in this research repository as “cdfg.ipynb”.



**Figure 5.10 Maximum correlation values for each block
(Dynamic Array Allocator)**



Figure 5.11 Highly correlated blocks (Dynamic Array Allocator)

CHAPTER VI

RESULTS & ANALYSIS

6.1 Block Characterization Visualized as Dendrograms

The Agglomerative Hierarchical Clustering (AHC) algorithm offers an efficient and informative analysis of binary file content by clustering assembly blocks based on content distribution similarities, calculated using the Kullback-Leibler (KL) Divergence metric. AHC also provides a unique cluster visualization through dendrograms, which explicitly depict the hierarchical relationships between clusters. This tree-like structure elucidates the clustering process and illustrates how data points are grouped at various levels. The height of the branches in a dendrogram represents the distance between clusters, allowing for a visual assessment of the similarity or dissimilarity between clusters at different levels. The output of the AHC algorithm clearly demonstrates its suitability for lightweight and rapid binary analysis, as evidenced by the results presented in this chapter for each program.

6.1.1 Functional Clustering of Simple Calculator

The simplest program among the three, the simple calculator (see Appendix B), employs switch cases to accept an operator symbol and two integer inputs, resulting in only 8 assembly blocks in its binary file. As depicted in Figure 5.3, the program's operations are predominantly concentrated in the 8th block. This is because the 8th block, or the `<main>` block, contains the main driver code, corresponding to the `main()` function in C++.

Table 6.1 Cluster Assignment Table (Simple Calculator)

Block	1	2	3	4	5	6	7	8
Cluster	1	1	2	3	3	1	4	1

Table 6.1 and Figure 6.1 visualize the cluster assignment for each block based on feature distribution similarity. Most blocks, including the main driver

block 8, fall within cluster 1. This is expected given the content of these blocks. Block 1, labeled <__cxx_global_var_init> shown in Figure 6.2, handles the initialization of global variables, which are accessible throughout the program. This block primarily involves data movement instructions such as `lea`, `mov`, `push`, and `pop`, indicating its role in managing variables as a significant part of its functionality. On the other hand, block 8, labeled <main> (see Appendix C) also operates with variables initialized by block 1 (in this case, using the input variables from user). Other blocks that fall into the same cluster appear to have similar instructions and operands.

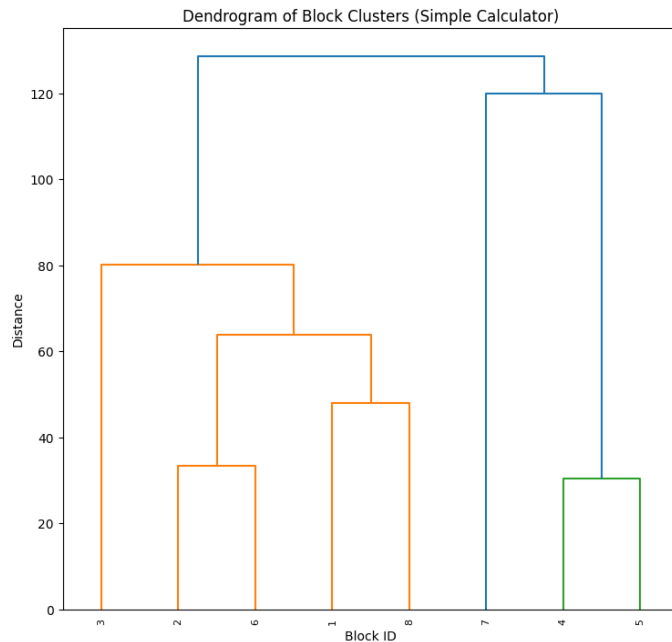


Figure 6.1 Dendrogram of Block Clusters (Simple Calculator)

```

00000000000010b0 <__cxx_global_var_init>:
10b0: 55                push    %rbp
10b1: 48 89 e5          mov     %rsp,%rbp
10b4: 48 8d 3d a6 2f 00 00 lea     0x2fa6(%rip),%rdi
10bb: e8 c0 ff ff ff    call    1080 <_ZNSt8ios_base4InitC1Ev@plt>
10c0: 48 8b 3d 31 2f 00 00 mov     0x2f31(%rip),%rdi
10c7: 48 8d 35 93 2f 00 00 lea     0x2f93(%rip),%rsi
10ce: 48 8d 15 83 2f 00 00 lea     0x2f83(%rip),%rdx
10d5: e8 66 ff ff ff    call    1040 <__cxa_atexit@plt>
10da: 5d                pop     %rbp
10db: c3                ret
10dc: 0f 1f 40 00       nopl    0x0(%rax)

```

Figure 6.2 Block 1 Content (Simple Calculator)

In terms of functionality, blocks 1, 2, 6, and 8 exhibit similar operations, primarily involving variable manipulations, which explains their grouping within the same cluster. Figure 6.3 shows that Cluster 1 has the highest number of distinct right operands with average to high entropy values, indicating that data movement and manipulation tasks are commonly found in blocks belonging to Cluster 1. Blocks such as 3 and 5, however, are less frequent and depend on specific program functionalities. For instance, block 3, labeled `<start>`, marks the point where execution begins when the program is loaded into memory, an event that typically occurs only once. Conversely, block 5, labeled `<register_tm_clones>`, is likely involved in thread creation and managing thread clones, like block 4, labeled `<deregister_tm_clones>`, which handles deregistering threads and thread clones. Consequently, blocks 4 and 5 are grouped within the same cluster.

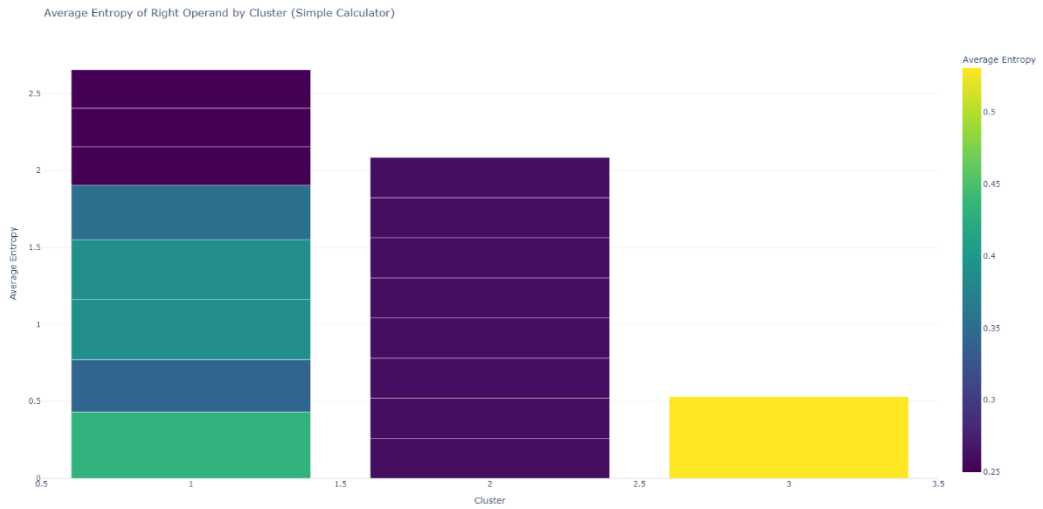


Figure 6.3 Average Entropy of Distinct Right Operand per Cluster (Simple Calculator)

6.1.2 Functional Clustering of Dynamic Array Allocator

The dynamic array allocator (DAA) program is a software component responsible for the allocation and deallocation of memory for dynamic arrays. It operates by dynamically allocating contiguous blocks of memory at runtime based on the specified size of the array, allowing for the creation of arrays whose size is

not known beforehand. In this research, the program utilizes functions like `new` and `delete` in C++ to manage memory allocation and deallocation, ensuring efficient memory usage and preventing memory leaks. The DAA program is the second largest program, introducing the use of functions outside the main driver, private and public variables, and classes (see Appendix D or refer to the GitHub repository).

As previously mentioned, the Agglomerative Hierarchical Clustering (AHC) algorithm generates a dendrogram that illustrates the clusters each block belongs to based on feature distribution similarity. Figure 6.4 provides an alternative visualization of the cluster assignments for each block.

Table 6.2 Cluster Assignment Table (Dynamic Array Allocator)

Block	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Cluster	3	2	3	4	4	3	5	3	2	2	1	1	1	2

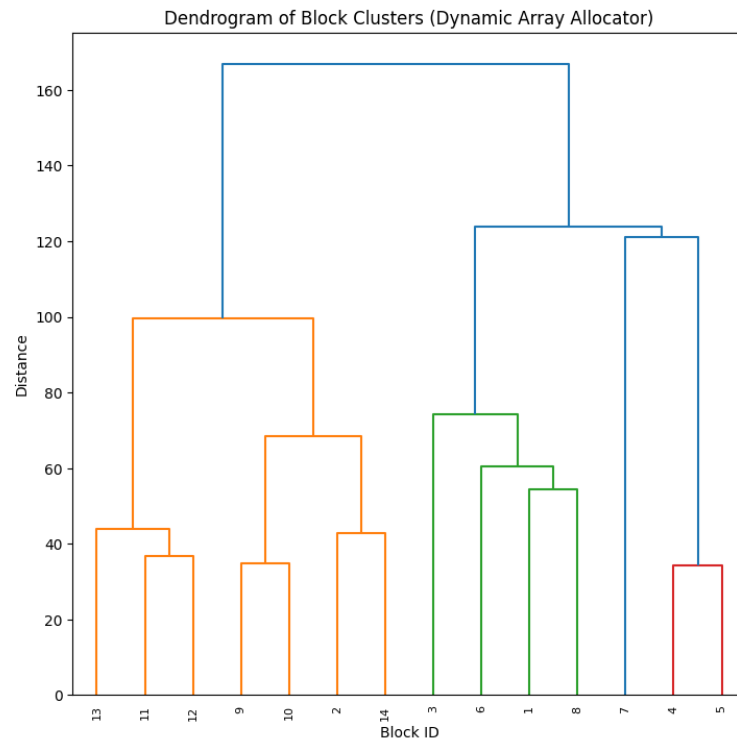


Figure 6.4 Dendrogram of Block Clusters (Dynamic Array Allocator)

Compared to the simple calculator program, the Dynamic Array Allocator (DAA) program has nearly double the number of blocks in its assembly code, with most operations concentrated around the 8th block, as shown in Figure 5.3. Similar to the block clustering in the simple calculator, blocks 1 and 8 are clustered together because they are heavily involved in data movement and manipulation. In contrast, a series of dynamic array constructor blocks are clustered together. For instance, block 9, labeled `<_ZN12DynamicArrayIiEC2Ei>`, and block 10, labeled `<_ZN12DynamicArrayIiE9push_backEi>`, are in Cluster 2, while block 11, labeled `<_ZNK12DynamicArrayIiE7getSizeEv>`, block 12, labeled `<_ZN12DynamicArrayIiEixEi>`, and block 13, labeled `<_ZN12DynamicArrayIiED2Ev>`, are in Cluster 1, see “dynamic_array_allocator_disassembled.asm”. These blocks contain specific instructions for initializing and manipulating dynamic arrays, distinct from variable initialization.

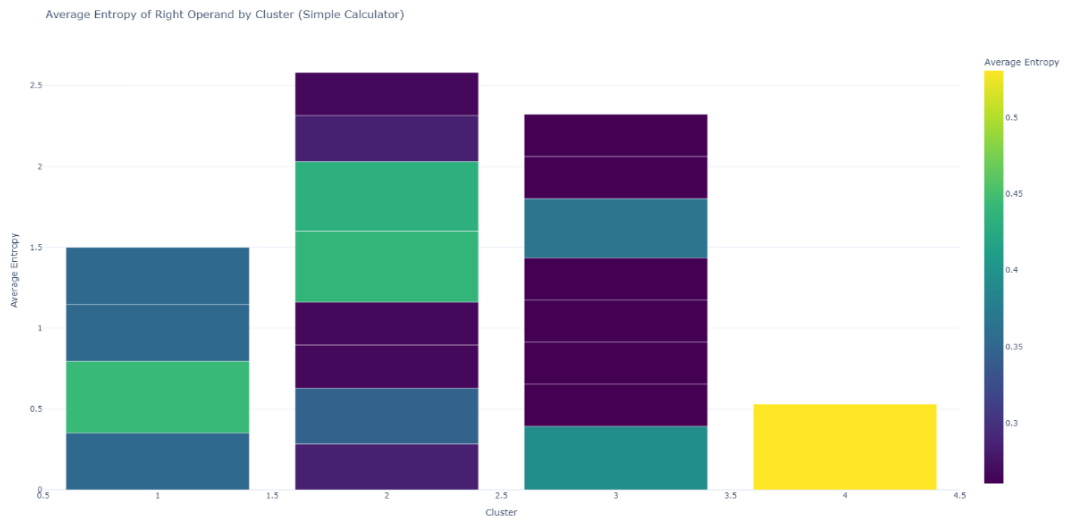


Figure 6.5 Average Entropy of Distinct Right Operand per Cluster (Dynamic Array Allocator)

Functionality can also be assessed by examining the operands manipulated by the assembly, particularly the right operands, as they carry the data. Figure 6.5 shows that Cluster 2, which mainly consists of blocks for initializing and manipulating dynamic arrays, has the highest number of distinct right operands,

with some high entropy values. The average entropy value of distinct right operands in Cluster 2 is higher than that in Cluster 3, which predominantly consists of blocks initializing global variables. This indicates very dynamic data movement within the blocks in Cluster 2, as there are many data transactions involving the right operands. Utilizing AHC and entropy visualization, we can discern which blocks are similar in terms of functionality and identify the tasks for which specific blocks are responsible.

6.1.3 Functional Clustering of Dynamic CSV Parser

The CSV parser program (see Appendix E or refer to the GitHub repository) is the most complex program out of all, consisting of 152 blocks and thousands of instructions and operands despite having the least number of code lines in the source code. The large size of disassembled content happens due to the use of libraries within the code, in which the program needs to communicate and extract functions from the specific library. The CSV parser program reads data from a specified CSV file and stores it in a vector of structures. It begins by including necessary libraries for file and string handling and defines a `CSVRow` structure to hold a single row of CSV data with a vector of strings named `fields`. In the `main()` function, it attempts to open the CSV file using an input file stream and checks if the file is successfully opened, printing an error message, and exiting if it is not. The program then reads the file line by line, tokenizing each line into individual fields separated by commas, and stores these fields in the `fields` vector of a `CSVRow` object. Each `CSVRow` object is then added to a `rows` vector. After reading all lines, the program iterates over the `rows` vector to print each field, separated by spaces, for each row. This process effectively parses the CSV file and prints its contents, demonstrating basic file I/O and string manipulation in C++.

Figure 5.1 shows a couple of blocks with large content size that need further analysis such as block 8, 65, and 97. Like the other two programs previously analyzed, block 8 is the `<main>` block for the main driver codes, but it is quite difficult to tell what block 65 and 97 do. This is where the functional clustering is starting to become more beneficial as a tool for larger scale programs because the

algorithm can cluster the blocks with unknown functionality with other blocks with similar features distribution values.

As shown in Appendix F and table 6.3, despite the large number of assembly blocks in the program, many of them have similar feature distribution, resulting in only a few clusters, in this case 4 clusters. Going back to the previous concern of blocks with large content size, based on the feature distribution similarity, block 65 and block 97 belong to cluster 1. The blocks in cluster 1 mainly consist of mov instructions, which indicates that the blocks in cluster 1 have a common functionality which is to move data into the registers. For example, the mov instruction has a distribution probability of 62% and 53% in block 65 and 97 respectively, and the same case appears to apply to the majority of blocks in cluster 1, which indicates a large data movement within the blocks in cluster 1.

Table 6.3 Cluster Assignment Table (CSV Parser)

Block	1	2	3	4	5	6	7	...	147	148	149	150	151	152
Cluster	2	1	2	2	2	2	2	...	1	4	4	1	4	4

Block 97 and block 65 are likely responsible for the construction of CSV row objects using the `new_allocator` from the GNU C++ library. This block focuses on object instantiation, where a high entropy of mov instructions, which can be seen in figure 6.6, indicates the transfer of data between registers and memory locations. The mov instructions are crucial for setting up the initial state of the CSVRow object, initializing its members, and ensuring the correct data is loaded into the appropriate registers. The left operand of each mov typically represents the destination register or memory location, while the right operand (figure 6.7) signifies the source of the data being moved. This block's operations are fundamental to creating robust CSVRow objects, which will be utilized in subsequent parsing processes.

Conversely, block 65 pertains to dynamic memory management within a vector of strings, specifically handling reallocation and insertion of elements. The

prevalence of `mov` instructions in this block highlights its role in managing memory addresses and pointers during reallocation, ensuring that new memory is correctly assigned, and existing data is preserved and transferred. The right operand often holds the source data or address, while the left operand indicates the target location, crucial for maintaining data integrity during vector resizing operations. Attention must be paid to the intricate sequence of these instructions, as they are critical for the efficient handling of memory, preventing data corruption and ensuring the vector's elements are accurately managed. The interplay of `mov` instructions across both blocks underscores their pivotal role in object construction and memory management, central to the efficient parsing and storage of CSV data.



Figure 6.6 Average Entropy of Distinct Instruction per Cluster (CSV Parser)

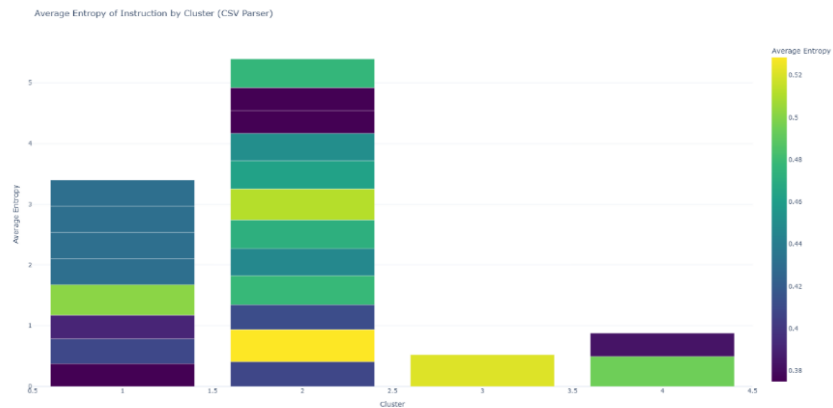


Figure 6.7 Average Entropy of Distinct Right Operand per Cluster (CSV Parser)

6.2 Silhouette Coefficient as an Evaluation Metric

As outlined at the outset of this research, an evaluation metric is applied to assess the quality of the Agglomerative Hierarchical Clustering (AHC) algorithm as a tool for light and efficient binary static analysis. In previous discussions in Chapter 6.8, the potential of the silhouette coefficient was explored, as it offers a clear indication of the similarity of data to the average data within the cluster. Alongside the AHC algorithm, Equation 6 is applied to each clustered block to compare the similarity values to the cluster they belong to. This process generates a horizontal bar graph, as illustrated in Figure 6.8, where the y-axis represents ζ and the x-axis represents the correctness range from -1 to 1, where 1 denotes correct placement and -1 indicates incorrect placement.

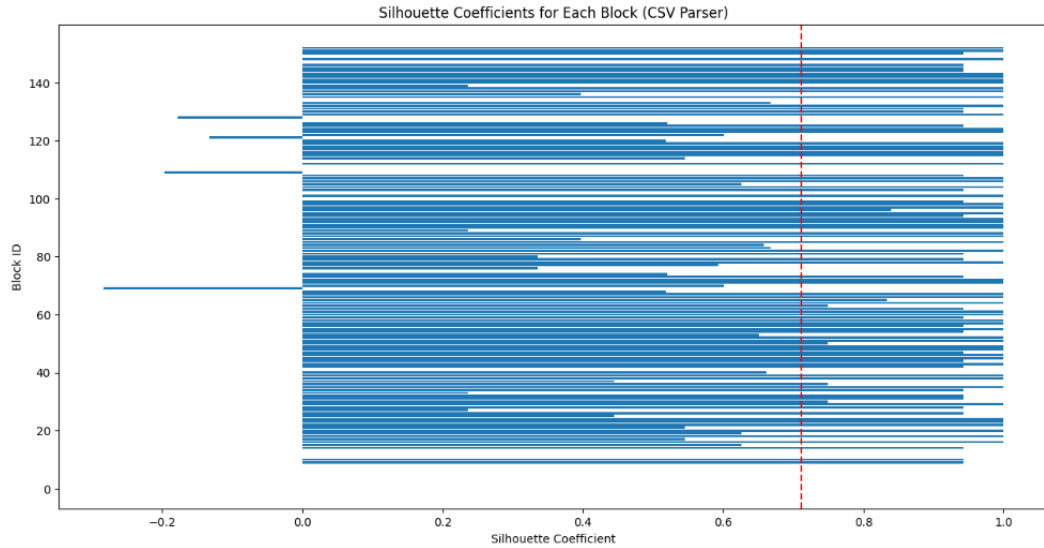


Figure 6.8 Silhouette Coefficient for Each Block (CSV Parser)

Figure 6.8 visualizes the silhouette coefficient values for each block and assesses how accurately the blocks are placed relative to their cluster. The red line represents the average silhouette coefficient for all blocks. In the case of the CSV parser program, the average coefficient value of the clustering is approximately 0.71. A coefficient average above 0.5 indicates strong clustering, where all blocks

are suitably clustered. The silhouette coefficient values for the DAA and simple calculator programs are displayed in Figures 6.9 and 6.10, respectively.

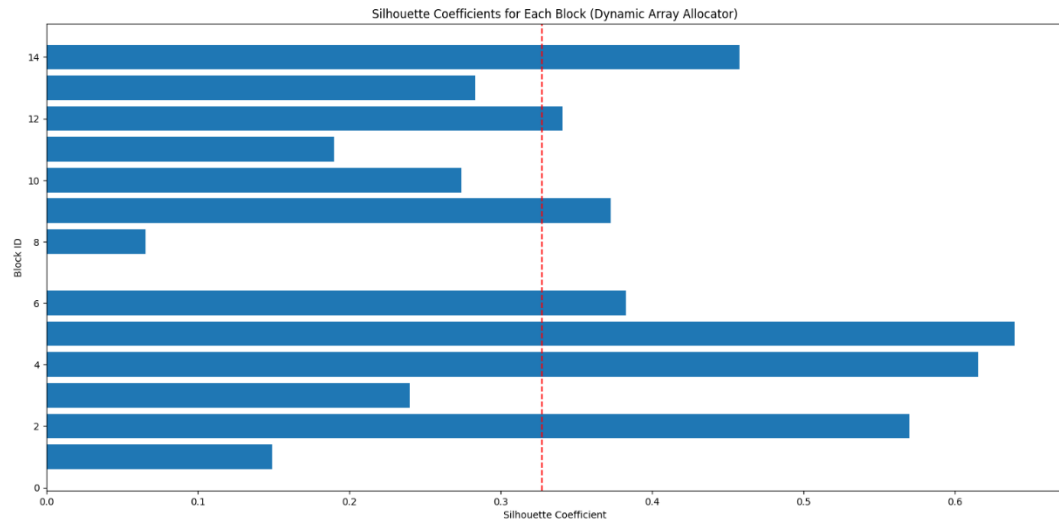


Figure 6.9 Silhouette Coefficient for Each Block (Dynamic Array Allocator)

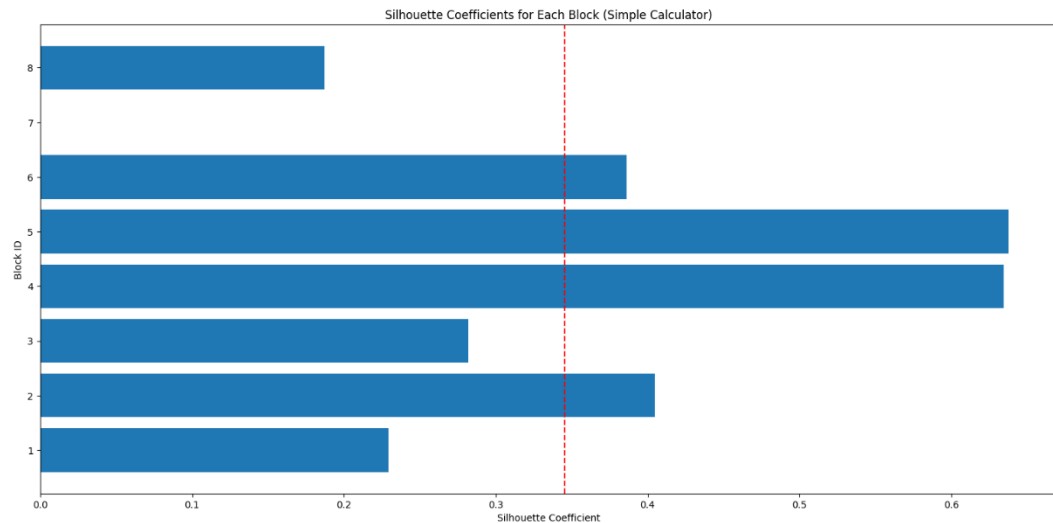


Figure 6.10 Silhouette Coefficient for Each Block (Simple Calculator)

The quality and accuracy of the AHC algorithm significantly depend on the data size, as evident from the coefficient values of the DAA program and simple calculator in Figures 6.9 and 6.10, respectively. An average coefficient value between 0.3 and 0.5 indicates average performance in clustering the blocks. In smaller-scale datasets, this may occur due to a lack of data variety, causing some blocks to be suitable for multiple clusters. The AHC algorithm implemented in this

research was tailored specifically for the data type being analyzed, leaving minimal room for improvement, as the parameters, methods, and thresholds have been optimized to achieve the highest average coefficient. A larger data set with a greater variety of data, such as a program with multiple functionalities, could enhance the algorithm's accuracy by better distinguishing differences.

A significant challenge when analyzing smaller programs using the AHC algorithm is the limited variety in features, such as instructions and operands. In programs like the simple calculator, which mainly involves variable manipulation, there is a scarcity of instruction and right operand variety. Consequently, as shown in Figure 6.10, the average coefficient is maximized around 0.35. Similarly, the DAA program predominantly focuses on constructing dynamic arrays and allocating variables to addresses, which primarily consist of data movement instructions and right operands, resulting in an average coefficient value maximized at 0.33. In contrast, implementing the AHC algorithm on the CSV parser program assembly yields a desired average coefficient maximized around 0.71, with minimal overlapping and misplaced blocks. A complete table of coefficient values for each block can be seen in Appendix G, Appendix H, and Appendix I for simple calculator program, DAA program, and CSV parser program respectively.

6.3 Block Correlation Analysis Using Correlated Data Flow Graphs

As proposed prior to this research, an implementation of correlation data flow graphs (CDFGs) offers a comprehensive approach to understanding the interconnection between blocks, elucidating the movement of instructions and operands between two blocks at a given frame ϕ and delay ∂ . Visualizing the maximum correlation value between blocks yields a weighted graph, where the nodes represent the block ζ ID and the weights denote the correlation value. This approach provides a nuanced understanding of the relationships between blocks within a program, shedding light on the flow of data and the dependencies between various components.

The correlation data flow graphs serve as powerful tools for analyzing the intricacies of program execution, highlighting the patterns of interaction between different segments of code. By examining the correlations between blocks, researchers can discern underlying structures and identify critical pathways within the program. Moreover, these graphs facilitate the identification of performance bottlenecks, potential vulnerabilities, and areas for optimization. Figures 6.11, 6.12, and 6.13 below illustrate the filtered highly correlated graphs for the programs.

Figure 6.11 Highly Correlated Blocks (CSV Parser)

Figure 6.12 Highly Correlated Blocks (Dynamic Array Allocator)

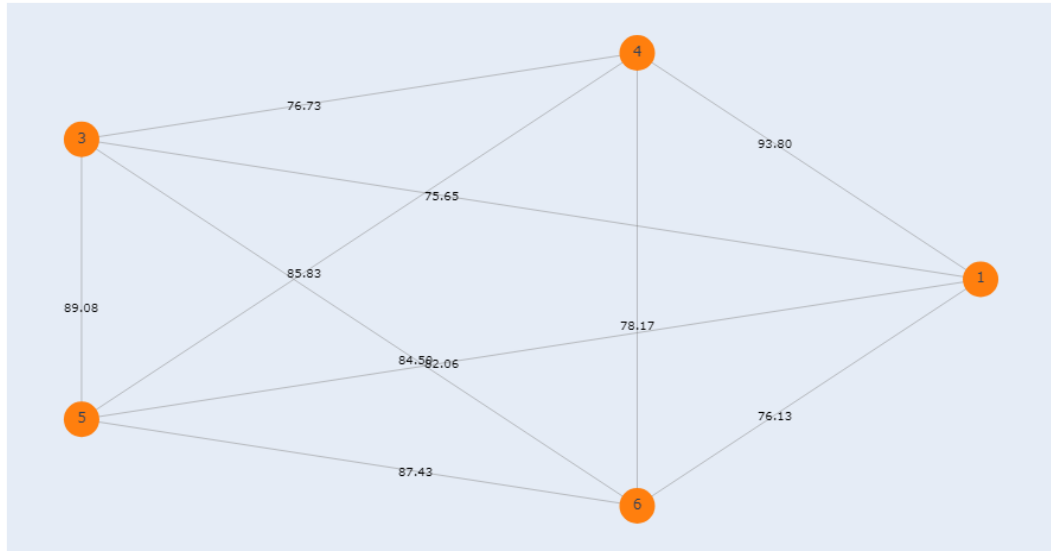


Figure 6.13 Highly Correlated Blocks (Simple Calculator)

In most cases, assembly blocks interact with each other through a variety of mechanisms, primarily centered around data transfer, memory access, and control flow. Data transfer occurs when one block writes data to a shared memory location, which can then be read by another block. Shared variable addresses enable blocks to access and manipulate the same data, facilitating communication and coordination. Additionally, assembly blocks can call each other, either passing control to a different block or invoking specific functionalities within other blocks. Control flow mechanisms, such as branching and looping, allow blocks to make decisions and execute different sequences of instructions based on specific conditions, further enabling complex interactions and collaborations between assembly blocks.

An evident example of such interactions can be observed in the correlation between block 4 and 5 in the DAA and simple calculator programs, as shown in Figures 6.12 and 6.13, respectively. In these two programs, blocks 4 and 5 are responsible for registering and deregistering threads for operations, and they work together to achieve their task. In Figure 6.13, blocks 4 and 5 exhibit a high correlation value of 78.17, indicating a high degree of similarity in correspondence when one is shifted relative to the other at a given delay ∂ . However, blocks 4 and

5 in Figure 6.13 do not have the highest correlation value due to differences in instructions when registering and deregistering threads, despite sharing many common right operands as address targets. Conversely, blocks 1 and 4 in Figure 6.13 demonstrate a high correlation value of 93.80 because the two blocks utilize very similar right operands, indicating that they manipulate many common addresses in their block operations.

The analysis facilitated by the AHC algorithm and CDFGs provides valuable insights into the correlation and functionality of assembly code. As assembly codes can quickly become complex, these tools enable researchers to conduct thorough analyses, uncovering intricate relationships and patterns within the codebase. Further exploration of the correlations and functionalities identified by these methods can lead to a deeper understanding of program behavior and facilitate effective optimization and debugging processes. Complete diagrams and tables can be found in the “charts” and “clusters” folder in this research repository.

CHAPTER VII

CONCLUSION

The methods described in this paper are particularly valuable when there is no prior knowledge of the code structure, and the disassembled code is highly complex. In such scenarios, discerning the relationships and functionalities of code blocks through traditional code tracing, while simultaneously monitoring variable and register changes, becomes challenging. The application of dimension reduction effectively eliminates non-essential data flow indicators, focusing on critical instructions and operands. The block characterization methods discussed facilitate the grouping of blocks into functional classes, enabling the labeling of functional code blocks based on distinct operand and instruction types and their frequency of use. Regarding data flow analysis, the construction of highly correlated Control Data Flow Graphs (CDFGs) enhances the binary data flow analysis by simplifying the graphs, thus enabling more focused very busy and available expression analysis.

In real-life applications, the Agglomerative Hierarchical Clustering (AHC) algorithm is anticipated to perform better with larger and more complex programs, and even more so with programs exhibiting operational irregularities. The Silhouette Coefficient evaluation metric shows that the AHC algorithm performs exceptionally well on large datasets, such as the CSV parser program, with an average coefficient of 0.71. However, its performance is only average on smaller datasets, such as the simple calculator and dynamic array allocator, with average coefficients of 0.35 and 0.33, respectively.

Considering user security concerns, the source code of widely used applications distributed by major companies such as WhatsApp, VK, and various operating systems, is often not available for public review. This lack of transparency can raise suspicions about user privacy. Binary static analysis can alleviate these concerns by enabling the visualization of blocks with intensive operations, based on the diversity of instructions, operands, and their entropy levels. These blocks can be clustered by functionality without requiring prior knowledge

or a labeled dataset, making the tool accessible and efficient for both regular users and developers. Consequently, this approach promotes enhanced security analysis and trust in widely used applications.

REFERENCES

- Liu, Q., Liu, J. & Chen, Y., 2019. On the theoretical basis of memory-free approaches for fractional differential equations. *Engineering Computations*, 36(4), pp. 1201-1218. Available at: <https://doi.org/10.1108/ec-08-2018-0389>.
- Ahmad Zabidi, M.N., 2012. *Malware Analysis with Multiple Features*.
- Borda, M., 2011. *Fundamentals in Information Theory and Coding*. Springer, Berlin, Heidelberg.
- Bragen, S.R., n.d. *Malware detection through opcode sequence analysis using machine learning*.
- Ferwana, I., 2019. Clustering Arabic tweets for Saudi National Vision 2030. *International Journal of Advanced Trends in Computer Science and Engineering*, 8(1.4), pp. 243-248.
- Huang, S.-Y., Lee, M.-H., Hsiao, C.K., 2009. Nonlinear measures of association with kernel canonical correlation analysis and applications. *Journal of Statistical Planning and Inference* 139, pp. 2162–2174.
- Ladas N, Borchert F, Franz S, et al. (2023). Programming techniques for improving rule readability for rule-based information extraction natural language processing pipelines of unstructured and semi-structured medical texts. *Health Informatics Journal*, 29(2), 14604582231164696.
- Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., & Jagadish, H. (2008). Regular expression learning for information extraction. *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, Honolulu, Hawaii, pp. 21-30.
- Liu, Q., Liu, J. & Chen, Y., 2019. On the theoretical basis of memory-free approaches for fractional differential equations. *Engineering Computations*, 36(4), pp. 1201-1218.

- Lyda, R., Hamrock, J., 2007. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy*, 5, pp. 40–45.
- Maulana, H. & Al-Khowarizmi, A., 2022. Analysis of the effectiveness of online learning using EDA data science and machine learning. *Sinkron*, 7(1), pp. 222-231.
- McLachlan, G.J., 1992. *Discriminant Analysis and Statistical Pattern Recognition*. John Wiley & Sons, Ltd.
- Naveen, G. & Nedungadi, P., 2014. Query-based multi-document summarization by clustering of documents. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1973-1978.
- Noroozi, M. & Favaro, P., 2016. Unsupervised learning of visual representations by solving jigsaw puzzles. In *Computer Vision – ECCV 2016* (pp. 69-84). Springer, Cham.
- Obert, J., Loffredo, T., 2021. Efficient Binary Static Code Data Flow Analysis Using Unsupervised Learning, in: *2021 4th International Conference on Artificial Intelligence for Industries (AI4I)*. Presented at the 2021 4th International Conference on Artificial Intelligence for Industries (AI4I), pp. 89–90.
- Obert, J., Loffredo, T., 2020. Efficient Binary Static Code Data Flow Analysis Using Unsupervised Learning (No. SAND--2019-14311R, 1592974, 682701).
- Pappas, V., Polychronakis, M., & Keromytis, A. (2012). Smashing the gadgets: hindering return-oriented programming using in-place code randomization. *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, San Francisco, California, pp. 601-615.
- Rabiner, L. A., 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), pp. 257-286.

- Schueren, G. (2017). TCPSnitch: dissecting the usage of the socket API. arXiv, no. 1711.00674.
- Song, J., Zhu, Z. & Price, C., 2014. Feature grouping for intrusion detection system based on hierarchical clustering. In *Advances in Intelligent and Soft Computing* (pp. 270-280). Springer, Berlin, Heidelberg.
- Subedi, K., Budhathoki, D., & Dasgupta, D. (2018). Forensic analysis of ransomware families using static and dynamic analysis. *Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW)*, San Francisco, California, pp. 71-76.
- Tamura, Y., Obara, N. & Miyamoto, S., 2012. A method of two-stage clustering with constraints using agglomerative hierarchical algorithm and one-pass k-means. *2012 Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS)*, pp. 1414-1419.
- Verbeek, F., Bockenek, J., & Ravindran, B. (2020). Highly automated formal proofs over memory usage of assembly code. In *Computer Aided Verification* (pp. 98-117). Springer, Cham.
- Wang, P. (2021). Demystifying regular expression bugs: A comprehensive study on regular expression bug causes, fixes, and testing. arXiv, no. 2104.09693.
- Wang, P., Brown, C., Jennings, J., & Stolee, K. (2021). Demystifying regular expression bugs. *Empirical Software Engineering*, 27(1), 1-55.
- Wang, Z. & Srinivasan, R., 2015. Classification of household appliance operation cycles: a case-study approach. *Energies*, 8(9), pp. 10522-10536.
- Webb, Z., Nnadili, M., Seghers, E., Briceno-Mena, L. & Romagnoli, J., 2022. Optimization of multi-mode classification for process monitoring. *Frontiers in Chemical Engineering*, 4, Article 900083.

- Yang, L. & Li, C., 2023. Identification of vulnerable lines in smart grid systems based on improved agglomerative hierarchical clustering. *IEEE Access*, 11, pp. 13554-13563.
- Zhao, Y. and Karypis, G., 2002. Evaluation of hierarchical clustering algorithms for document datasets. *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pp. 515-524.
- Zhao, Y., Karypis, G., & Fayyad, U., 2005. Hierarchical clustering algorithms for document datasets. *Data Mining and Knowledge Discovery*, 10(2), 141-168.
- Zhu, R., Tan, Y., Zhang, Q., Wu, F., Zheng, J., & Xue, Y. (2016). Determining image base of firmware files for arm devices. *IEICE Transactions on Information and Systems*, E99.D(2), 351-359.

APPENDIX

Appendix A (Entropy Calculation)

```
import csv
from collections import defaultdict
import math

# Function to calculate entropy
def calculate_entropy(probabilities):
    entropy = 0
    for prob in probabilities:
        if prob != 0:
            entropy -= prob * math.log2(prob)
    return entropy

# Function to calculate probability distribution and entropy
def calculate_probabilities_and_entropy(blocks):
    with open(
        "entropy\dynamic_array_calculator_entropy.csv", "w", newline=""
    ) as csvfile:
        fieldnames = ["Block_ID", "Type", "Assembly", "Probability", "Entropy"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        for block_id, block in blocks.items():
            total_count = len(block)
            variables = defaultdict(int)

            # Count occurrences of each variable type
            for line in block:
                variables[(line["Instruction"], "Instruction")] += 1
                if line["Left Operand"]:
                    variables[(line["Left Operand"], "Left Operand")] += 1
                if line["Right Operand"]:
                    variables[(line["Right Operand"], "Right Operand")] += 1

            # Calculate probabilities and entropies
            for (variable, variable_type), count in variables.items():
                probability = count / total_count
                entropy = calculate_entropy([probability])
                writer.writerow(
                    {
                        "Block_ID": block_id,
                        "Type": variable_type,
                        "Assembly": variable,
                        "Probability": probability,
                        "Entropy": entropy,
                    }
                )

# Function to read the assembly CSV file
def read_assembly_csv(file_path):
    blocks = defaultdict(list)
    with open(file_path, newline="") as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            block_id = row["Block_ID"]
            blocks[block_id].append(
```

```

        {
            "Instruction": row["Instruction"],
            "Left Operand": row["Left Operand"],
            "Right Operand": row["Right Operand"],
        }
    )
    return blocks

# Main function
def main():
    assembly_file_path = "compiled\dynamic_array_allocator_disassembled\dynamic_array_allocator_diassembly.csv"
    blocks = read_assembly_csv(assembly_file_path)
    calculate_probabilities_and_entropy(blocks)

if __name__ == "__main__":
    main()

```

Appendix B (Simple Calculator)

```
#include <iostream>

int main() {
    char op;
    double num1, num2;

    // Input from the user
    std::cout << "Enter an operator (+, -, *, /): ";
    std::cin >> op;

    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;

    // Switch statement to perform operations
    switch (op) {
        case '+':
            std::cout << num1 << " + " << num2 << " = " << (num1 + num2) << std::endl;
            break;
        case '-':
            std::cout << num1 << " - " << num2 << " = " << (num1 - num2) << std::endl;
            break;
        case '*':
            std::cout << num1 << " * " << num2 << " = " << (num1 * num2) << std::endl;
            break;
        case '/':
            if (num2 == 0) {
                std::cout << "Error: Division by zero" << std::endl;
            } else {
                std::cout << num1 << " / " << num2 << " = " << (num1 / num2) << std::endl;
            }
            break;
        default:
            std::cout << "Invalid operator" << std::endl;
    }

    return 0;
}
```

Appendix C (Simple Calculator Assembly)

```

00000000000010b0 <__cxx_global_var_init>:
10b0: 55                push    %rbp
10b1: 48 89 e5          mov     %rsp,%rbp
10b4: 48 8d 3d a6 2f 00 00 lea     0x2fa6(%rip),%rdi
10bb: e8 c0 ff ff ff    call    1080 <_ZNSt8ios_base4InitC1Ev@plt>
10c0: 48 8b 3d 31 2f 00 00 mov     0x2f31(%rip),%rdi
10c7: 48 8d 35 93 2f 00 00 lea     0x2f93(%rip),%rsi
10ce: 48 8d 15 83 2f 00 00 lea     0x2f83(%rip),%rdx
10d5: e8 66 ff ff ff    call    1040 <__cxa_atexit@plt>
10da: 5d                pop     %rbp
10db: c3                ret
10dc: 0f 1f 40 00       nopl    0x0(%rax)

00000000000010e0 <_GLOBAL__sub_I_simple_calculator.cpp>:
10e0: 55                push    %rbp
10e1: 48 89 e5          mov     %rsp,%rbp
10e4: e8 c7 ff ff ff    call    10b0 <__cxx_global_var_init>
10e9: 5d                pop     %rbp
10ea: c3                ret
10eb: 0f 1f 44 00 00    nopl    0x0(%rax,%rax,1)

00000000000010f0 <_start>:
10f0: f3 0f 1e fa       endbr64
10f4: 31 ed             xor     %ebp,%ebp
10f6: 49 89 d1          mov     %rdx,%r9
10f9: 5e                pop     %rsi
10fa: 48 89 e2          mov     %rsp,%rdx
10fd: 48 83 e4 f0       and     $0xfffffffffffff0,%rsp
1101: 50                push    %rax
1102: 54                push    %rsp
1103: 45 31 c0          xor     %r8d,%r8d
1106: 31 c9             xor     %ecx,%ecx
1108: 48 8d 3d d1 00 00 00 lea     0xd1(%rip),%rdi
110f: ff 15 b3 2e 00 00 call    *0x2eb3(%rip)
1115: f4                hlt
1116: 66 2e 0f 1f 84 00 00 cs nopw 0x0(%rax,%rax,1)
111d: 00 00 00

0000000000001120 <deregister_tm_clones>:
1120: 48 8d 3d 39 2f 00 00 lea     0x2f39(%rip),%rdi
1127: 48 8d 05 32 2f 00 00 lea     0x2f32(%rip),%rax
112e: 48 39 f8          cmp     %rdi,%rax
1131: 74 15             je      1148 <deregister_tm_clones+0x28>
1133: 48 8b 05 9e 2e 00 00 mov     0x2e9e(%rip),%rax
113a: 48 85 c0          test    %rax,%rax
113d: 74 09             je      1148 <deregister_tm_clones+0x28>
113f: ff e0            jmp     *%rax
1141: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
1148: c3                ret
1149: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

0000000000001150 <register_tm_clones>:
1150: 48 8d 3d 09 2f 00 00 lea     0x2f09(%rip),%rdi
1157: 48 8d 35 02 2f 00 00 lea     0x2f02(%rip),%rsi
115e: 48 29 fe          sub     %rdi,%rsi
1161: 48 89 f0          mov     %rsi,%rax
1164: 48 c1 ee 3f       shr     $0x3f,%rsi
1168: 48 c1 f8 03       sar     $0x3,%rax

```

```

116c: 48 01 c6          add    %rax,%rsi
116f: 48 d1 fe          sar    %rsi
1172: 74 14             je     1188 <register_tm_clones+0x38>
1174: 48 8b 05 75 2e 00 00 mov    0x2e75(%rip),%rax
117b: 48 85 c0          test   %rax,%rax
117e: 74 08             je     1188 <register_tm_clones+0x38>
1180: ff e0            jmp    *%rax
1182: 66 0f 1f 44 00 00 nopw   0x0(%rax,%rax,1)
1188: c3               ret
1189: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)

0000000000001190 <__do_global_dtors_aux>:
1190: f3 0f 1e fa      endbr64
1194: 80 3d c5 2e 00 00 00 cmpb   $0x0,0x2ec5(%rip)
119b: 75 2b            jne    11c8 <__do_global_dtors_aux+0x38>
119d: 55              push   %rbp
119e: 48 83 3d 12 2e 00 00 cmpq   $0x0,0x2e12(%rip)
11a5: 00
11a6: 48 89 e5          mov    %rsp,%rbp
11a9: 74 0c            je     11b7 <__do_global_dtors_aux+0x27>
11ab: 48 8b 3d a6 2e 00 00 mov    0x2ea6(%rip),%rdi
11b2: e8 e9 fe ff ff    call   10a0 <__cxa_finalize@plt>
11b7: e8 64 ff ff ff    call   1120 <deregister_tm_clones>
11bc: c6 05 9d 2e 00 00 01 movb   $0x1,0x2e9d(%rip)
11c3: 5d              pop    %rbp
11c4: c3               ret
11c5: 0f 1f 00          nopl   (%rax)
11c8: c3               ret
11c9: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)

00000000000011d0 <frame_dummy>:
11d0: f3 0f 1e fa      endbr64
11d4: e9 77 ff ff ff    jmp    1150 <register_tm_clones>
11d9: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)

00000000000011e0 <main>:
11e0: 55              push   %rbp
11e1: 48 89 e5          mov    %rsp,%rbp
11e4: 48 83 ec 30       sub    $0x30,%rsp
11e8: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
11ef: 48 8b 3d da 2d 00 00 mov    0x2dda(%rip),%rdi
11f6: 48 89 7d d0       mov    %rdi,-0x30(%rbp)
11fa: 48 8d 35 1b 0e 00 00 lea     0xe1b(%rip),%rsi
1201: e8 4a fe ff ff    call   1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
1206: 48 8b 3d d3 2d 00 00 mov    0x2dd3(%rip),%rdi
120d: 48 89 7d d8       mov    %rdi,-0x28(%rbp)
1211: 48 8d 75 fb       lea     -0x5(%rbp),%rsi
1215: e8 56 fe ff ff    call   1070
<_ZStl-
sISt11char_traitsIcEERSt13basic_istreamIT_T0_ES6_RS3_@plt>
121a: 48 8b 7d d0       mov    -0x30(%rbp),%rdi
121e: 48 8d 35 18 0e 00 00 lea     0xe18(%rip),%rsi
1225: e8 26 fe ff ff    call   1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
122a: 48 8b 7d d8       mov    -0x28(%rbp),%rdi
122e: 48 8d 75 f0       lea     -0x10(%rbp),%rsi
1232: e8 f9 fd ff ff    call   1030 <_ZNSirsERd@plt>
1237: 48 89 c7          mov    %rax,%rdi
123a: 48 8d 75 e8       lea     -0x18(%rbp),%rsi
123e: e8 ed fd ff ff    call   1030 <_ZNSirsERd@plt>

```

```

1243: 0f be 45 fb      movsbl -0x5(%rbp),%eax
1247: 83 c0 d6      add $0xffffffffd6,%eax
124a: 89 c1      mov %eax,%ecx
124c: 48 89 4d e0      mov %rcx,-0x20(%rbp)
1250: 83 e8 05      sub $0x5,%eax
1253: 0f 87 db 01 00 00      ja 1434 <main+0x254>
1259: 48 8b 45 e0      mov -0x20(%rbp),%rax
125d: 48 8d 0d a0 0d 00 00      lea 0xda0(%rip),%rcx
1264: 48 63 04 81      movslq (%rcx,%rax,4),%rax
1268: 48 01 c8      add %rcx,%rax
126b: ff e0      jmp *%rax
126d: f2 0f 10 45 f0      movsd -0x10(%rbp),%xmm0
1272: 48 8b 3d 57 2d 00 00      mov 0x2d57(%rip),%rdi
1279: e8 12 fe ff ff      call 1090 <_ZNSoIsEd@plt>
127e: 48 89 c7      mov %rax,%rdi
1281: 48 8d 35 c9 0d 00 00      lea 0xdc9(%rip),%rsi
1288: e8 c3 fd ff ff      call 1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
128d: 48 89 c7      mov %rax,%rdi
1290: f2 0f 10 45 e8      movsd -0x18(%rbp),%xmm0
1295: e8 f6 fd ff ff      call 1090 <_ZNSoIsEd@plt>
129a: 48 89 c7      mov %rax,%rdi
129d: 48 8d 35 b1 0d 00 00      lea 0xdb1(%rip),%rsi
12a4: e8 a7 fd ff ff      call 1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
12a9: 48 89 c7      mov %rax,%rdi
12ac: f2 0f 10 45 f0      movsd -0x10(%rbp),%xmm0
12b1: f2 0f 58 45 e8      addsd -0x18(%rbp),%xmm0
12b6: e8 d5 fd ff ff      call 1090 <_ZNSoIsEd@plt>
12bb: 48 89 c7      mov %rax,%rdi
12be: 48 8b 35 fb 2c 00 00      mov 0x2cfb(%rip),%rsi
12c5: e8 96 fd ff ff      call 1060 <_ZNSoIsEPFRSoS_E@plt>
12ca: e9 87 01 00 00      jmp 1456 <main+0x276>
12cf: f2 0f 10 45 f0      movsd -0x10(%rbp),%xmm0
12d4: 48 8b 3d f5 2c 00 00      mov 0x2cf5(%rip),%rdi
12db: e8 b0 fd ff ff      call 1090 <_ZNSoIsEd@plt>
12e0: 48 89 c7      mov %rax,%rdi
12e3: 48 8d 35 6f 0d 00 00      lea 0xd6f(%rip),%rsi
12ea: e8 61 fd ff ff      call 1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
12ef: 48 89 c7      mov %rax,%rdi
12f2: f2 0f 10 45 e8      movsd -0x18(%rbp),%xmm0
12f7: e8 94 fd ff ff      call 1090 <_ZNSoIsEd@plt>
12fc: 48 89 c7      mov %rax,%rdi
12ff: 48 8d 35 4f 0d 00 00      lea 0xd4f(%rip),%rsi
1306: e8 45 fd ff ff      call 1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
130b: 48 89 c7      mov %rax,%rdi
130e: f2 0f 10 45 f0      movsd -0x10(%rbp),%xmm0
1313: f2 0f 5c 45 e8      subsd -0x18(%rbp),%xmm0
1318: e8 73 fd ff ff      call 1090 <_ZNSoIsEd@plt>
131d: 48 89 c7      mov %rax,%rdi
1320: 48 8b 35 99 2c 00 00      mov 0x2c99(%rip),%rsi
1327: e8 34 fd ff ff      call 1060 <_ZNSoIsEPFRSoS_E@plt>
132c: e9 25 01 00 00      jmp 1456 <main+0x276>
1331: f2 0f 10 45 f0      movsd -0x10(%rbp),%xmm0
1336: 48 8b 3d 93 2c 00 00      mov 0x2c93(%rip),%rdi
133d: e8 4e fd ff ff      call 1090 <_ZNSoIsEd@plt>
1342: 48 89 c7      mov %rax,%rdi
1345: 48 8d 35 11 0d 00 00      lea 0xd11(%rip),%rsi

```



```

134c:  e8 ff fc ff ff      call    1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
1351:  48 89 c7            mov     %rax,%rdi
1354:  f2 0f 10 45 e8      movsd  -0x18(%rbp),%xmm0
1359:  e8 32 fd ff ff      call    1090 <_ZNSoIsEd@plt>
135e:  48 89 c7            mov     %rax,%rdi
1361:  48 8d 35 ed 0c 00 00 lea     0xcd(%rip),%rsi
1368:  e8 e3 fc ff ff      call    1050 <_ZStlsISt11char_trait-
sIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
136d:  48 89 c7            mov     %rax,%rdi
1370:  f2 0f 10 45 f0      movsd  -0x10(%rbp),%xmm0
1375:  f2 0f 59 45 e8      mulsd  -0x18(%rbp),%xmm0
137a:  e8 11 fd ff ff      call    1090 <_ZNSoIsEd@plt>
137f:  48 89 c7            mov     %rax,%rdi
1382:  48 8b 35 37 2c 00 00 mov     0x2c37(%rip),%rsi
1389:  e8 d2 fc ff ff      call    1060 <_ZNSoIsEPFRSoS_E@plt>
138e:  e9 c3 00 00 00      jmp     1456 <main+0x276>
1393:  f2 0f 10 45 e8      movsd  -0x18(%rbp),%xmm0
1398:  0f 57 c9            xorps  %xmm1,%xmm1
139b:  66 0f 2e c1         ucomisd %xmm1,%xmm0
139f:  0f 85 2d 00 00 00   jne     13d2 <main+0x1f2>
13a5:  0f 8a 27 00 00 00   jp      13d2 <main+0x1f2>
13ab:  48 8b 3d 1e 2c 00 00 mov     0x2c1e(%rip),%rdi
13b2:  48 8d 35 a8 0c 00 00 lea     0xca8(%rip),%rsi
13b9:  e8 92 fc ff ff      call    1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
13be:  48 89 c7            mov     %rax,%rdi
13c1:  48 8b 35 f8 2b 00 00 mov     0x2bf8(%rip),%rsi
13c8:  e8 93 fc ff ff      call    1060 <_ZNSoIsEPFRSoS_E@plt>
13cd:  e9 5d 00 00 00      jmp     142f <main+0x24f>
13d2:  f2 0f 10 45 f0      movsd  -0x10(%rbp),%xmm0
13d7:  48 8b 3d f2 2b 00 00 mov     0x2bf2(%rip),%rdi
13de:  e8 ad fc ff ff      call    1090 <_ZNSoIsEd@plt>
13e3:  48 89 c7            mov     %rax,%rdi
13e6:  48 8d 35 8c 0c 00 00 lea     0xc8c(%rip),%rsi
13ed:  e8 5e fc ff ff      call    1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
13f2:  48 89 c7            mov     %rax,%rdi
13f5:  f2 0f 10 45 e8      movsd  -0x18(%rbp),%xmm0
13fa:  e8 91 fc ff ff      call    1090 <_ZNSoIsEd@plt>
13ff:  48 89 c7            mov     %rax,%rdi
1402:  48 8d 35 4c 0c 00 00 lea     0xc4c(%rip),%rsi
1409:  e8 42 fc ff ff      call    1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
140e:  48 89 c7            mov     %rax,%rdi
1411:  f2 0f 10 45 f0      movsd  -0x10(%rbp),%xmm0
1416:  f2 0f 5e 45 e8      divsd  -0x18(%rbp),%xmm0
141b:  e8 70 fc ff ff      call    1090 <_ZNSoIsEd@plt>
1420:  48 89 c7            mov     %rax,%rdi
1423:  48 8b 35 96 2b 00 00 mov     0x2b96(%rip),%rsi
142a:  e8 31 fc ff ff      call    1060 <_ZNSoIsEPFRSoS_E@plt>
142f:  e9 22 00 00 00      jmp     1456 <main+0x276>
1434:  48 8b 3d 95 2b 00 00 mov     0x2b95(%rip),%rdi
143b:  48 8d 35 3b 0c 00 00 lea     0xc3b(%rip),%rsi
1442:  e8 09 fc ff ff      call    1050 <_ZStl-
sISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
1447:  48 89 c7            mov     %rax,%rdi
144a:  48 8b 35 6f 2b 00 00 mov     0x2b6f(%rip),%rsi
1451:  e8 0a fc ff ff      call    1060 <_ZNSoIsEPFRSoS_E@plt>
1456:  31 c0              xor     %eax,%eax

```

1458:	48 83 c4 30	add	\$0x30,%rsp
145c:	5d	pop	%rbp
145d:	c3	ret	

Appendix D (Dynamic Array Allocator)

```
#include <iostream>

template <typename T> class DynamicArray {
public:
    DynamicArray(int initialSize = 10)
        : size(initialSize), capacity(initialSize) {
        data = new T[capacity];
    }

    ~DynamicArray() { delete[] data; }

    void push_back(T value) {
        if (size == capacity) {
            resize(capacity * 2);
        }
        data[size++] = value;
    }

    T &operator[](int index) { return data[index]; }

    int getSize() const { return size; }

private:
    void resize(int newCapacity) {
        T *newData = new T[newCapacity];
        for (int i = 0; i < size; i++) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }

    T *data;
    int size;
    int capacity;
};

int main() {
    DynamicArray<int> myArray;
    for (int i = 0; i < 20; i++) {
        myArray.push_back(i * 5);
    }

    for (int i = 0; i < myArray.getSize(); i++) {
        std::cout << myArray[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Appendix E (CSV Parser)

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>

// A simple structure to hold a row of CSV data
struct CSVRow {
    std::vector<std::string> fields;
};

int main() {
    const char *filename = "csv_parser_test.csv";
    std::ifstream file(filename);

    if (!file.is_open()) {
        std::cout << "Error opening file: " << filename << std::endl;
        return 1;
    }

    std::vector<CSVRow> rows;
    std::string line;
    while (std::getline(file, line)) {
        CSVRow row;
        std::stringstream lineStream(line);
        std::string field;

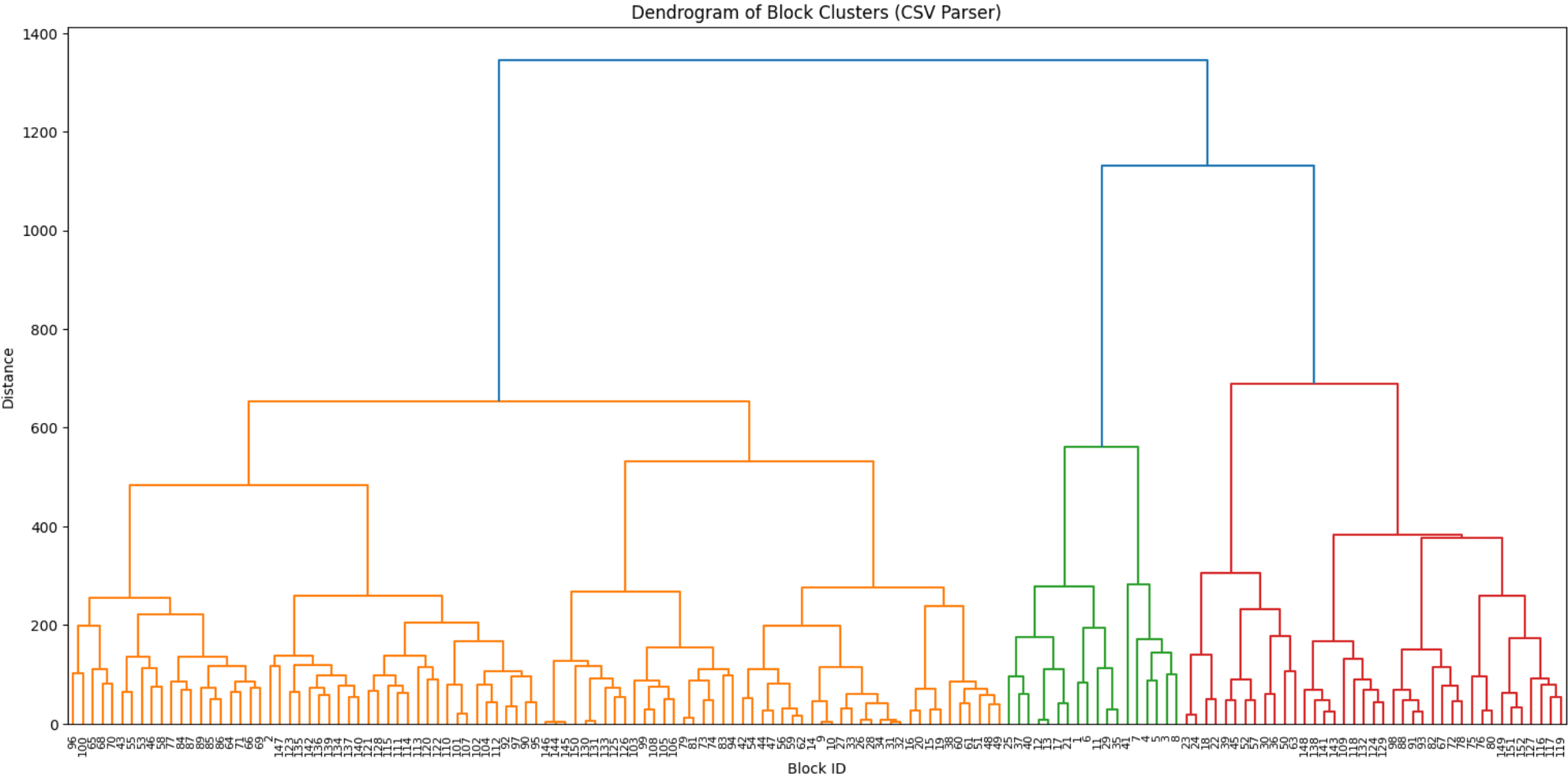
        while (std::getline(lineStream, field, ',')) {
            row.fields.push_back(field);
        }

        rows.push_back(row);
    }

    // Accessing the parsed data
    for (auto &row : rows) {
        for (auto &field : row.fields) {
            std::cout << field << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Appendix F (Dendrogram of CSV Parser Block Clusters)



Appendix G (Coefficient Value of Simple Calculator Blocks)

Block #	Cluster	Coefficient
1	1	0.2295101493
2	1	0.4046322572
3	2	0.2819449670
4	3	0.6346495543
5	3	0.6376666930
6	1	0.3861224461
7	4	0.0
8	1	0.1870311707

Appendix H (Coefficient Value of Dynamic Array Allocator Blocks)

Block #	Cluster	Coefficient
1	3	0.4296547839
2	2	0.5873593221
3	3	0.3215548171
4	4	0.2475198615
5	4	0.3311968649
6	3	0.3166587670
7	5	0.0
8	3	0.0935459660
9	2	0.3715347471
10	2	0.2258940378
11	1	0.1239869220
12	1	0.2770959275
13	1	0.3789397572
14	2	0.4127225931

Appendix I (Coefficient Value of CSV Parser Blocks)

Block #	Cluster	Coefficient			
1	9	0	31	4	0.942159241
2	1	0	32	4	0.942159241
3	8	0	33	4	0.2365019815
4	8	0	34	4	0.942159241
5	8	0	35	9	1
6	9	0	36	3	0.75
7	8	0	37	9	0.4442042415
8	8	0	38	4	1
9	4	0.942159241	39	3	1
10	4	0.942159241	40	9	0.6614898697
11	9	0	41	8	0
12	9	0	42	4	0.942159241
13	9	0	43	2	1
14	4	0.942159241	44	4	0.942159241
15	4	0.6257308827	45	3	1
16	4	1	46	2	1
17	9	0.545233943	47	4	0.942159241
18	3	1	48	4	1
19	4	0.6257308827	49	4	1
20	4	1	50	3	0.75
21	9	0.545233943	51	4	1
22	3	1	52	3	1
23	3	1	53	2	0.6516236675
24	3	1	54	4	0.942159241
25	9	0.4442042415	55	2	1
26	4	0.942159241	56	4	0.942159241
27	4	0.2365019815	57	3	1
28	4	0.942159241	58	2	1
29	9	1	59	4	0.942159241
30	3	0.75	60	4	1
			61	4	1

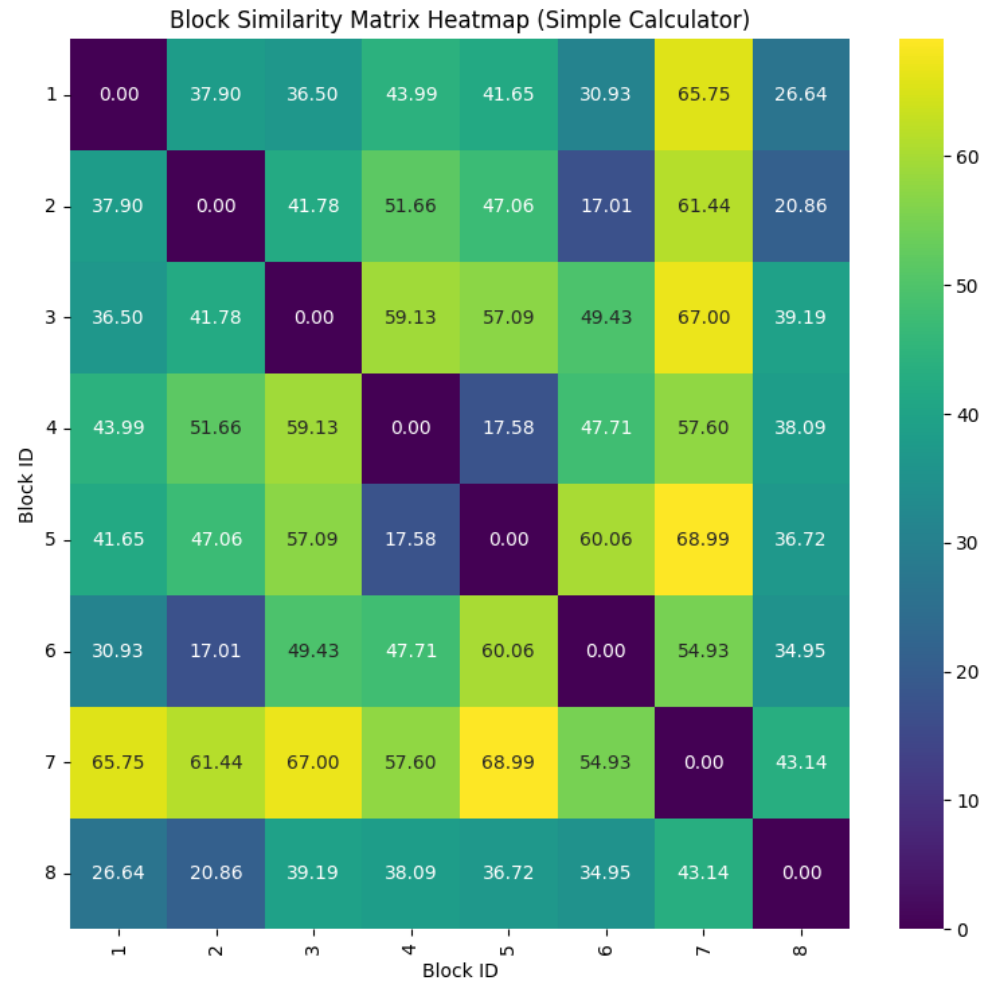
62	4	0.942159241
63	3	0.75
64	2	1
65	2	0.8335960393
66	2	1
67	7	1
68	2	0.5178260109
69	2	-0.2832463572
70	2	0.6015774142
71	2	1
72	7	1
73	5	0.942159241
74	5	0.5200139883
75	7	0
76	7	0.335790047
77	2	0.5934502851
78	7	1
79	5	0.942159241
80	7	0.335790047
81	5	0.942159241
82	7	1
83	5	0.6674020133
84	2	0.6585159413
85	2	1
86	2	0.3974326851
87	2	1
88	7	1
89	2	0.2355511206
90	2	1
91	7	1
92	2	1
93	7	1

94	5	0.942159241
95	2	1
96	2	0.8399340278
97	2	1
98	7	1
99	5	0.942159241
100	2	0
101	5	1
102	2	0
103	5	0.942159241
104	2	1
105	5	0.6257308827
106	5	1
107	5	1
108	5	0.942159241
109	6	-0.1968597201
110	5	0
111	1	0
112	2	1
113	1	0
114	1	0.545233943
115	1	1
116	7	1
117	7	1
118	6	1
119	7	1
120	1	0.5178260109
121	1	-0.1332203413
122	1	0.6015774142
123	1	1
124	6	1
125	5	0.942159241

126	5	0.5200139883
127	7	0
128	1	-0.178217599
129	6	1
130	5	0.942159241
131	5	0.942159241
132	6	1
133	5	0.6674020133
134	1	0
135	1	1
136	1	0.3974326851
137	1	1
138	6	1
139	1	0.2355511206

140	1	1
141	6	1
142	1	1
143	6	1
144	5	0.942159241
145	5	0.942159241
146	5	0.942159241
147	1	0
148	6	1
149	6	0
150	5	0.942159241
151	6	1
152	6	1

Appendix J (Block Similarity Heatmap of Simple Calculator)



Appendix K (Block Similarity Heatmap of Dynamic Array Allocator)

