UNIVERSITY OF THE AEGEAN

ENGINEERING SCHOOL

INFORMATION AND COMMUNICATION SYSTEMS ENGINEERING DEPARTMENT

Authorship Verification in Binary Files

DIPLOMA THESIS

by

Adam Kyriakos

Supervisor: Stamatatos Efstathios

Members of the Examination Committee: Kavallieratou Ergina, Kostoulas Theodoros

Samos, September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΑΚΩΝ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Επαλήθευση Συγγραφικής Πατρότητας σε Δυαδικά Αρχεία

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αδάμ Κυριάκου

Επιβλέπων: Σταματάτος Ευστάθιος

Μέλη Εξεταστικής Επιτροπής: Καβαλλιεράτου Εργίνα, Κωστούλας Θεόδωρος

Σάμος, Σεπτέμβριος 2025

Acknowledgments

I would like to express my sincere gratitude to my supervising professor, Dr. Stamatatos Efstathios, for his guidance and for giving me the opportunity to work on such an interesting and challenging topic for my diploma thesis. I would also like to thank my family for their unwavering support throughout all my years of study.

© 2025

Adam Kyriakos

Information and Communication Systems Engineering Department

UNIVERSITY OF THE AEGEAN

Table of Contents

1 Introduction 12

1.1 Artificial Intelligence and Binary Code Analysis 12

1.2 Diploma Thesis Subject: Authorship Verification from Binary Code 13

1.3 Structure of the Diploma Thesis 13

2 Supporting Technologies 14

2.1 Neural Networks and Deep Learning 14

2.1.1 Siamese Neural Networks 14

2.1.2 Multi-Layer Perceptron (MLP) 15

2.1.3 Recurrent Neural Networks (RNN) 17

2.1.4 Long Short-Term Memory (LSTM) 18

2.1.5 Optimization Algorithms 20

2.2 Natural Language Processing 22

2.2.1 Byte-Pair Encoding (BPE) and SentencePiece 22

2.2.2 Token Normalization 23

2.3 Reverse Engineering and Code Disassembly 24

2.3.1 Interactive Disassembler (IDA Pro) 24

2.3.2 Radare2 27

2.4 Compilers and Obfuscation Techniques 28

2.4.1 GCC/G++ Compiler 28

2.4.2 Clang Compiler 29

2.4.3 Obfuscator-LLVM 30

3 Related Work 32

3.1 VeriBin 32

3.2 SepBIN 34

4 Proposed System 37

4.1 Overall System Architecture 37

4.2 Pre-Processing Pipeline 38

4.3 Architecture of the Siamese Model 41

4.4 Training and Optimization Process 42

5 Experimental Evaluation 44

5.1 Datasets and Experimental Setup 44

5.2 Evaluation Metrics 47

5.3 Analysis of Results 48

6 Conclusions and Future Work 51

6.1 Summary of Conclusions 51

6.2 Future Extensions and Research Directions 52

References 53

List of Figures

Figure 1. Structure of a Siamese Network 15
Figure 2. Structure of a Multi-Layer Perceptron Network 16
Figure 3. Structure of a Recurrent Neural Network 17
Figure 4. Visualization of the Vanishing and Exploding Gradient Problems 18
Figure 5. Structure of a Long Short-Term Memory block 19
Figure 6. Structure of a BiLSTM Network 20
Figure 7. Average Optimizer Performance across 8 optimization problems 21
Figure 8. Visualization of the Byte-Pair Encoding (BPE) tokenization algorithm 22
Figure 9. Example of the Stemming and Lemmatization techniques 23
Figure 10. Example of normalized token sequences from the original source codes 24
Figure 11. IDA Pro's Dummy Name representation dialog 25
Figure 12. IDA Pro's GUI Overview 26
Figure 13. Radare2's iaito GUI Overview 27
Figure 14. The GCC Compilation Process 28
Figure 15. High-Level architecture of LLVM/Clang Compilation 29
Figure 16. Example of obfuscated result using OLLVM 30
Figure 17. Overview of the VeriBin model 33
Figure 18. Visualization of the Control Flow Graphs used in SepBIN 35
Figure 19. Overview of the SepBIN framework 35
Figure 20. Detailed Architecture of the Proposed System 38
Figure 21. Example of JSONL function entries with token normalization (left) and without (right) 39
Figure 22. Example of the CSV entries 40
Figure 23. Overview of the Proposed System's Preprocessing Pipeline 40
Figure 24. Overview of the Proposed System's Siamese model 41
Figure 25. Overview of the Proposed System's full training process 42

List of Tables

Table 1. Statistics of Experiment Datasets - Part 1 45
Table 2. Statistics of Experiment Datasets - Part 2 46
Table 3. Evaluation Metrics - Test 1 48
Table 4. Evaluation Metrics - Test 2 48
Table 5. Evaluation Metrics - Test 3 49
Table 6. Evaluation Metrics - Test 4 49
Table 7. Evaluation Metrics - Test 5 49
Table 8. Evaluation Metrics - Test 6 50
Table 9. Evaluation Metrics - Test 7 50

Acronyms

MLP	Multi-Layer Perceptron
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
BiLSTM	Bidirectional LSTM
SGD	Stochastic Gradient Descent
Adam	Adaptive Moment Estimation
Nadam	Nesterov-accelerated Adaptive Moment Estimation
BPE	Byte-Pair Encoding
GCC	GNU Compiler Collection
LLVM	Low Level Virtual Machine
OLLVM	Obfuscator-LLVM
GAN	Generative Adversarial Network
CFG	Control Flow Graph
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
JSON	JavaScript Object Notation
CSV	Comma-Separated Values
ROC	Receiver Operating Characteristic
AUROC	Area Under the ROC Curve
ReLU	Rectified Linear Unit
CUDA	Compute Unified Device Architecture
GCJ	Google Code Jam

Abstract

This diploma thesis concerns the development of an artificial intelligence system for the authorship verification of binary files. The system's purpose is to be able to accurately determine if two executable programs originate from the same programmer, a task that is challenging due to the removal of obvious stylistic elements during the source code compilation process.

The methodology combines reverse engineering and deep learning techniques. Initially, during the preprocessing stage, the binary files are disassembled into assembly code and a filtering and normalization process is applied to replace specific values such as memory addresses with generic identifiers. Subsequently, the tokenized functions are passed into a Siamese neural network, which utilizes a deep learning encoder to generate a stylistic representation for each code sample and then compares them using a Multi-Layer Perceptron.

The findings of the research indicate that the system can achieve high performance in verifying programmer identity. The most significant finding is the model's remarkable adaptability to different compilers, optimizations and new datasets. These findings indicate that the system's resilience makes it applicable in realistic software analysis scenarios. The innovation of this work is the effective preprocessing method for disassembled assembly code and the deep learning architectures designed for comparison, contributing a powerful and thorough solution to the field.

Keywords: *artificial intelligence, deep learning, authorship verification, binary files, reverse engineering, coding style, Siamese neural network*

Περίληψη

Η παρούσα διπλωματική εργασία αφορά την ανάπτυξη ενός συστήματος τεχνητής νοημοσύνης για την επαλήθευση πατρότητας δυαδικών αρχείων. Σκοπός του συστήματος είναι να μπορεί να προσδιορίζει με ακρίβεια εάν δύο εκτελέσιμα προγράμματα προέρχονται από τον ίδιο προγραμματιστή, κάτι που είναι απαιτητικό λόγω της αφαίρεσης των προφανών υφολογικών στοιχείων κατά τη διαδικασία μεταγλώττισης του πηγαίου κώδικα.

Η μεθοδολογία συνδυάζει τεχνικές αντίστροφης μηχανικής και βαθιάς μάθησης. Αρχικά στη διαδικασία προεπεξεργασίας αποσυναρμολογούνται τα δυαδικά αρχεία σε κώδικα assembly και εφαρμόζεται μια διαδικασία φιλτραρίσματος και κανονικοποίησης, όπου συγκεκριμένες τιμές όπως οι διευθύνσεις μνήμης αντικαθίστανται με γενικά αναγνωριστικά. Στη συνέχεια, οι τμηματοποιημένες συναρτήσεις τροφοδοτούνται σε ένα σιαμαίο νευρωνικό δίκτυο, το οποίο αξιοποιεί έναν κωδικοποιητή βαθιάς μάθησης για να παράγει μια υφολογική αναπαράσταση για κάθε δείγμα κώδικα και ύστερα τα συγκρίνει με ένα νευρωνικό δίκτυο εμπρόσθιας τροφοδότησης.

Τα ευρήματα της έρευνας καταδεικνύουν ότι το σύστημα μπορεί να επιτυγχάνει υψηλή απόδοση στην επαλήθευση ταυτότητας προγραμματιστή. Το σημαντικότερο εύρημα είναι η αξιοσημείωτη προσαρμοστικότητα του μοντέλου σε διαφορετικούς μεταγλωττιστές, βελτιστοποιήσεις και νέα σύνολα δεδομένων. Τα ευρήματα αυτά υποδηλώνουν ότι η ανθεκτικότητα του συστήματος το καθιστά εφαρμόσιμο σε ρεαλιστικά σενάρια ανάλυσης λογισμικού. Η καινοτομία της εργασίας έγκειται στην αποτελεσματική μέθοδο προεπεξεργασίας του αποσυναρμολογημένου κώδικα assembly και στις αρχιτεκτονικές βαθιάς μάθησης σχεδιασμένες για σύγκριση, προσφέροντας ως συνεισφορά μια ισχυρή και ολοκληρωμένη λύση στο πεδίο.

Λέξεις Κλειδιά: *τεχνητή νοημοσύνη, βαθιά μάθηση, επαλήθευση πατρότητας, δυαδικά αρχεία, αντίστροφη μηχανική, ύφος κώδικα, σιαμαίο νευρωνικό δίκτυο*

1

Introduction

1.1 Artificial Intelligence and Binary Code Analysis

Binary code analysis is a fundamental task in several areas, particularly in fields like cybersecurity where accessing the original source code is often difficult. Traditionally, this analysis has been a time-consuming process performed by specialized experts for tasks such as malware detection, vulnerability discovery and reverse engineering. However, the complexity of software nowadays has made manual analysis a rather time-consuming and ineffective task.

This has resulted in the increasing use of Artificial Intelligence in the field to automate and enhance the analysis of binary code. By training machine learning models on binary code, they can learn to find code patterns that are difficult for humans to detect. These models can be trained to classify programs, find similar code patterns or detect malicious behavior. Regardless, the compilation process continues to discard much of the obvious stylistic information from the source code, leaving a low-level representation that is difficult to interpret. Therefore, the main problem is the extraction of consistent and meaningful enough features from this noisy data for interpretation.

1.2 Diploma Thesis Subject: Authorship Verification from Binary Code

This diploma thesis addresses the specific problem of authorship verification from binary code. The goal is to develop a system that can determine whether two different binary executables were compiled from source code written by the same author. This task is technically challenging because the compilation process removes the most obvious indicators of a programmer's style, such as variable names, comments and code formatting.

Similar to existing systems in the field, this thesis work revolves around the fact that specific coding patterns are left as subtle but detectable stylistic information in the final compiled code. Previous work has demonstrated that while compilation removes explicit identifiers, a programmer's coding style can still be recovered from the resulting binary [1]. The proposed approach is to use a deep learning model, specifically a Siamese neural network, which will be trained to analyze pairs of functions from filtered and tokenized disassembled binaries. By learning to distinguish between code from the same author and code from different authors, the system becomes a reliable tool for authorship verification, even without access to the original source code.

1.3 Structure of the Diploma Thesis

The supporting technologies used in this thesis are presented in Chapter 2. Chapter 3 discusses the related work in the field of binary authorship verification. In Chapter 4, the architecture of the proposed system is developed, detailing both the data preprocessing pipeline and the training model. Chapter 5 describes the experimental setup and the datasets used and an analysis of the results obtained from the system's evaluation. Finally, Chapter 6 summarizes the conclusions of this work and proposes directions for future research and extensions.

2

Supporting Technologies

2.1 Neural Networks and Deep Learning

2.1.1 Siamese Neural Networks

In machine learning a standard design for learning to compare two inputs to determine their similarity is the Siamese network architecture which was first introduced for signature verification [2]. The Siamese network architecture takes advantage of two identical encoders that share weights with each other to map a pair of inputs into a shared vector space. From this vector space the distance between the two vectors can be measured to estimate their similarity, making it a simple yet powerful system for verification tasks.

The Siamese network structure contains two subnetworks, and that part is the main strength of its architecture. The two subnetworks are in essence encoders that are identical and share the weights between each other. The system's inputs are pairs where each part of a pair is passed into its own encoder. This weight sharing technique makes sure that the network learns a consistent mapping function. If the two encoders had different weights, then it would not be guaranteed that even two identical inputs would produce the same output vector so a useful comparison between the two would be impossible. Therefore, by enforcing shared weights the network is forced to learn to extract the same features from both parts of the input pair in the same manner.

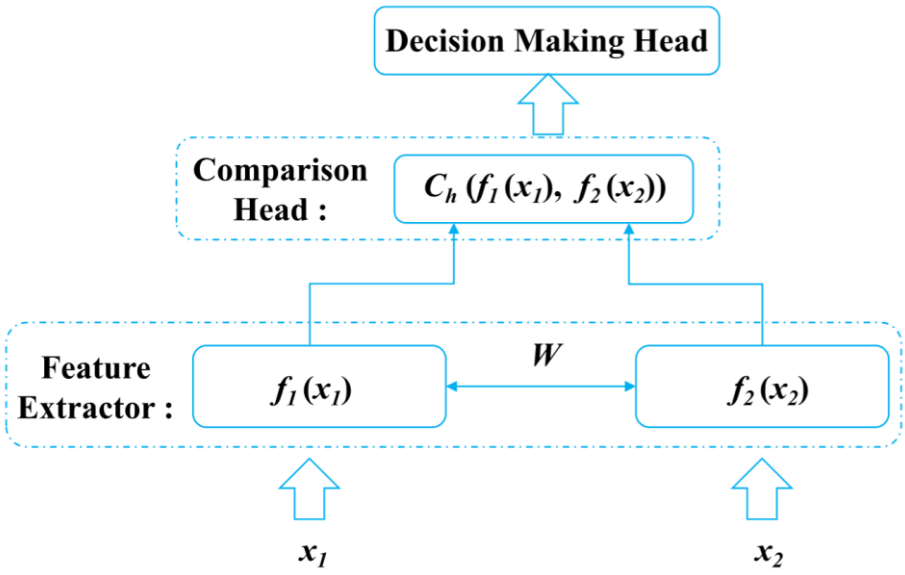

Figure 1. Structure of a Siamese Network

Source: *https://www.researchgate.net/publication/363605765*

The training process of a Siamese network is quite different from that of typical classification. The training objective in Siamese networks is not to extrapolate the classification loss from a single input, but instead it is based on the pairwise relationship between the output vectors of the input pair. The contrastive loss function is a common method for calculating this as it minimizes the distance between the two vectors of a positive pair while it makes sure that the distance between the vectors of a negative pair is larger than a particular predefined value [3]. As a result, with this method the network is trained to place similar items close to each other. Siamese networks are used for various applications and therefore the inputs of the encoders can range from images for face verification, to sentences for semantic similarity or vector representations of individual code blocks for software analysis.

2.1.2 Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP) is a fundamental class of feedforward neural network that consists of a series of connected Dense layers. It represents one of the earliest and simplest forms of a deep neural network. Its structure includes an input layer, some hidden layers and an output layer. In this architecture, the neurons of each layer are connected to the neurons of the preceding layer with the information flowing only from input to output.

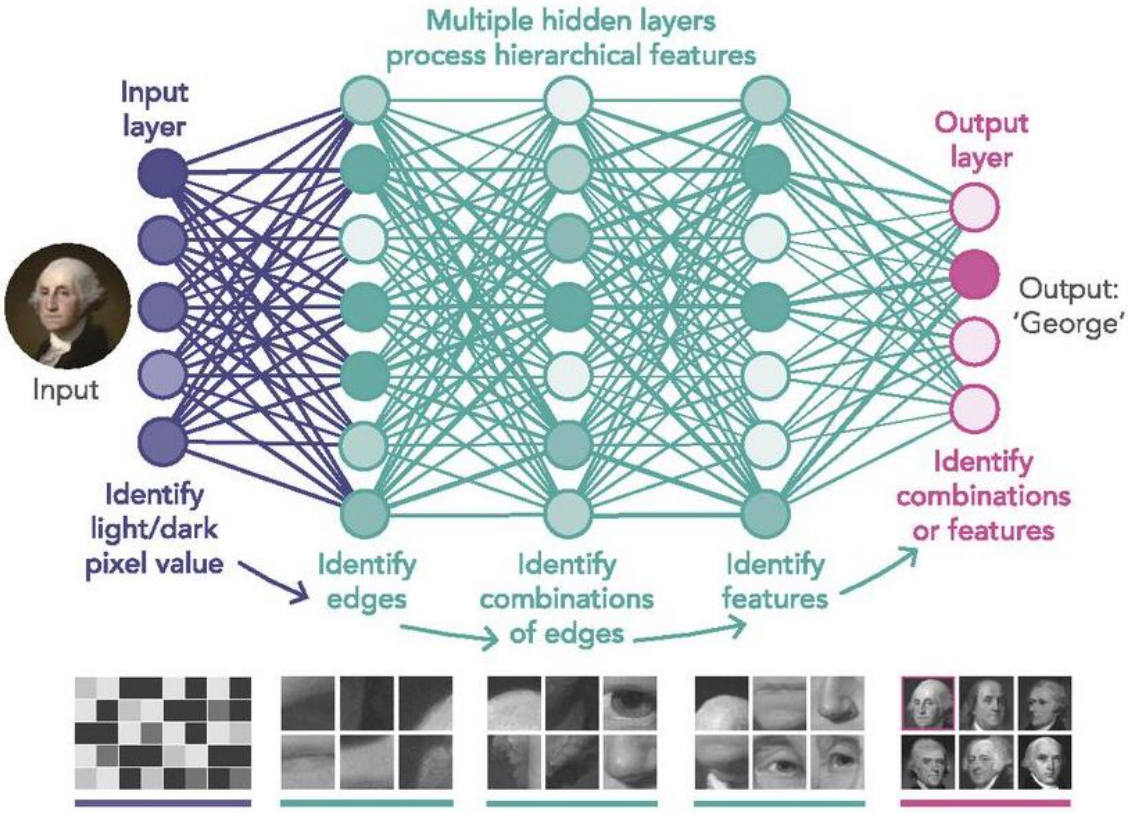

Figure 2. Structure of a Multi-Layer Perceptron Network

Source: *https://www.pnas.org/doi/10.1073/pnas.1821594116*

The power of a MLP comes from the combination of these layered connections and the use of non-linear activation functions (such as ReLU or sigmoid) at each neuron. Without non-linearity, a large stack of layers would be the same as a single layer. The non-linear activations allow MLPs to learn complex relations between input and output. In fact, a MLP with just one hidden layer can in theory approximate any continuous function, which is known as the universal approximation theorem [4].

As universal function approximators, MLPs are often used as the final classification component of a larger deep learning model, capable of making a prediction based on a set of learned features [5]. In similarity learning tasks the MLP can be made more potent by passing it not just the output vectors from the encoders, but also engineered features derived from them such as their element-wise difference or product to learn more complex decisions.

2.1.3 Recurrent Neural Networks (RNN)

Standard feedforward networks assume that the inputs processed are entirely independent of each other. This assumption is invalid for data such as natural language or source code, where the order of elements is critical to the overall meaning. Recurrent Neural Networks (RNNs) are neural networks designed to consider this dependence, which allows them to deal with such sequential data [6]. As a result, they are highly suitable for tasks involving natural language, time-series data and sequences of code instructions [7].

The main feature of a RNN is its recurrent connection. At each step of a sequence, the network's calculations take into account both the current input and the hidden state from the previous step. This hidden state is essentially the network's memory since it stores information about the previous inputs that it has processed. By sharing parameters across each step, RNNs can operate efficiently and most importantly can successfully apply what they have learned to sequences of any length.

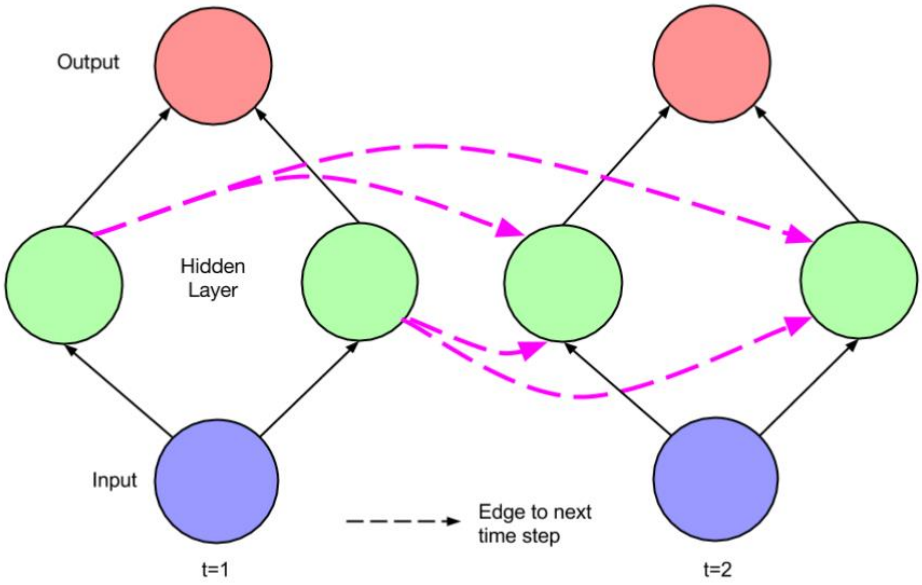

Figure 3. Structure of a Recurrent Neural Network

Source: *https://arxiv.org/abs/1506.00019*

This recurrent architecture however has a significant flaw. During the training process, the gradients must be propagated back through the same recurrent connections at each step, which for long sequences can mean that the gradients may eventually either vanish or explode due to their constant multiplication. The vanishing gradient problem especially is a problem that occurs quite often and can severely limit the ability of RNNs to learn long-term dependencies [6]. As a result, they are not effective for tasks that require long-term memory, such as for the understanding of a lengthy document or a complex code block.

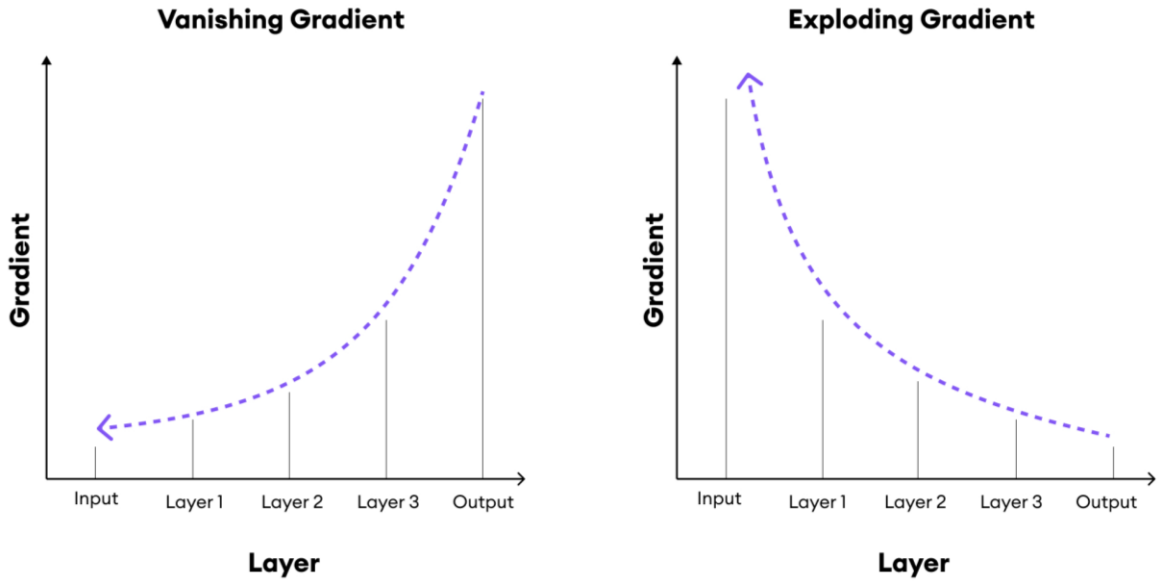

Figure 4. Visualization of the Vanishing and Exploding Gradient Problems

Source: *https://www.superannotate.com/blog/activation-functions-in-neural-networks*

2.1.4 Long Short-Term Memory (LSTM)

Although RNNs can successfully learn short-term dependencies, they can easily fail to learn connections between distant elements in a sequence mainly due to the vanishing gradient problem. Long Short-Term Memory (LSTM) networks are advanced types of RNNs that address this exact limitation through a more complex internal structure which allows them to effectively learn and remember information over long sequences [8]. This capability has made them one of the most popular and effective architectures for sequence modeling tasks.

The key innovation of LSTMs is the cell state and gating mechanism, which creates a dedicated path for long-term memory to flow through. The cell state is like a transporter of information that moves through the entire sequence. As a result, information from earlier steps is kept over long distances thus mitigating the vanishing gradient problem. The flow of data in and out of this cell state is regulated by three gates [6]. The forget gate decides what is no longer relevant and should be discarded from the previous cell state. The input gate decides what information from the current input should be saved in the cell state. The output gate decides what part of the cell state should be used to produce the network's output and hidden state for the current step.

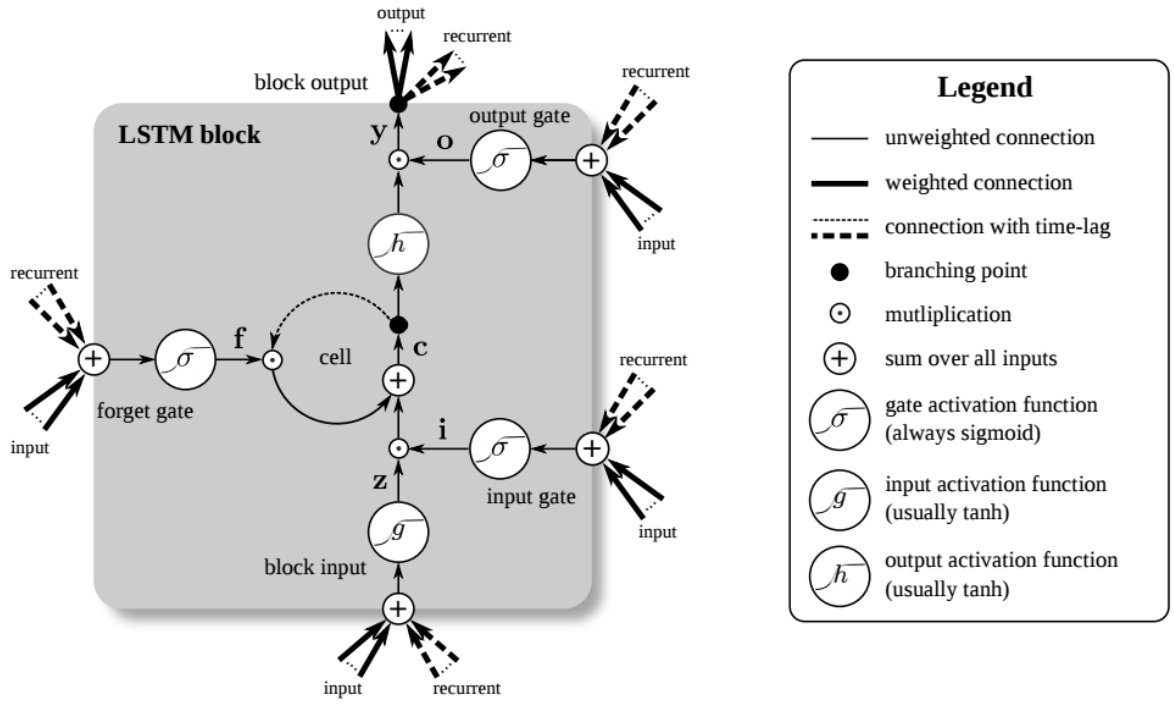

Figure 5. Structure of a Long Short-Term Memory block

Source: *https://arxiv.org/abs/1503.04069*

Each of these gates is basically a small neural network that learns to open or close based on the context thus allowing the LSTM to selectively remember or forget information. This explicit memory management mechanism is what allows LSTMs to capture long-term relations. To further enhance the contextual understanding of a sequence, a bidirectional architecture can be implemented. A Bidirectional LSTM consists of two separate LSTMs, with one processing the sequence in a forward direction and the other in a reverse direction. At each step the outputs of these two networks are concatenated, creating a representation that includes both past and future context [9]. The final hidden states from both directions are often combined to produce a single vector that is essentially a summary representation of the entire input sequence.

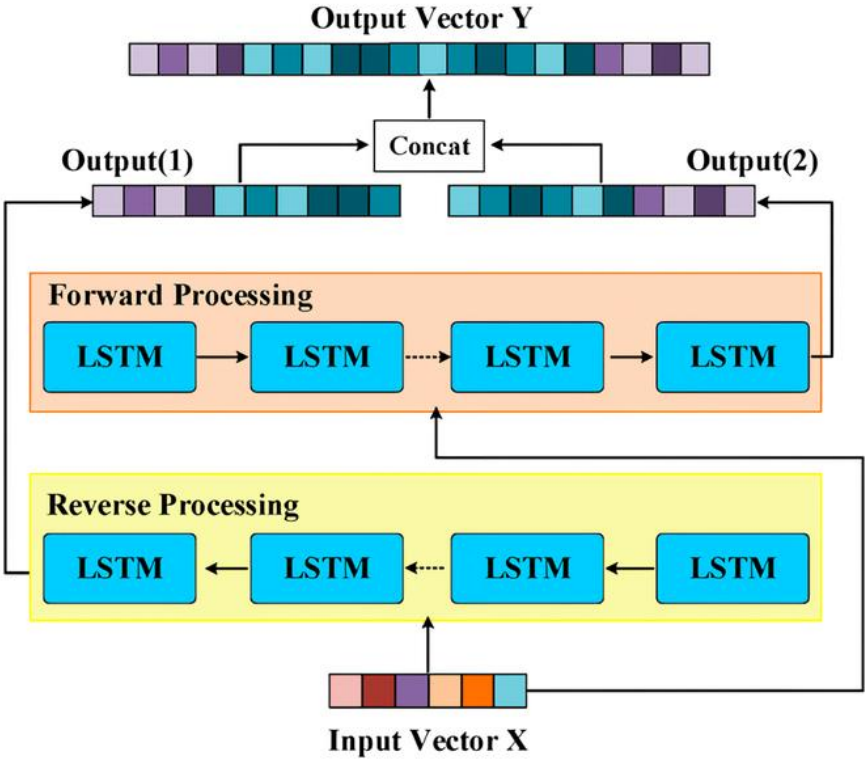

Figure 6. Structure of a BiLSTM Network

Source: *https://www.mdpi.com/2072-4292/16/17/3124*

2.1.5 Optimization Algorithms

The process of training a neural network is essentially a search for a set of weights that minimizes the given loss function. This search is performed using an optimization algorithm. The optimizer iteratively updates the network's weights based on the gradients of the loss function, effectively guiding the model to find a set of weights that can achieve the best performance.

The first fundamental methods that were developed, like Stochastic Gradient Descent (SGD), update weights using a fixed learning rate. However, modern adaptive optimizers have become standard practice as they adjust the learning rate for each parameter individually. Adam (Adaptive Moment Estimation) is among the most popular adaptive optimizers as it computes individual learning rates for each parameter by using momentum and past squared gradients. The Nadam optimizer further enhances Adam by incorporating Nesterov momentum, a technique that allows the optimizer to look ahead and can lead to faster convergence during training [10].

Choosing the best optimizer and tuning its hyperparameters (such as the learning rate) can become a bloated task. However, recent studies have shown that there is no single optimizer that consistently outperforms all others across all tasks and that testing multiple optimizers with their default parameters (as provided by the deep learning libraries) works about as well as extensively tuning the hyperparameters of just one optimizer [11]. Optimizers from the Adam family are generally considered safe and reliable choices, providing robust performance without the need for extensive tuning for most deep learning tasks.

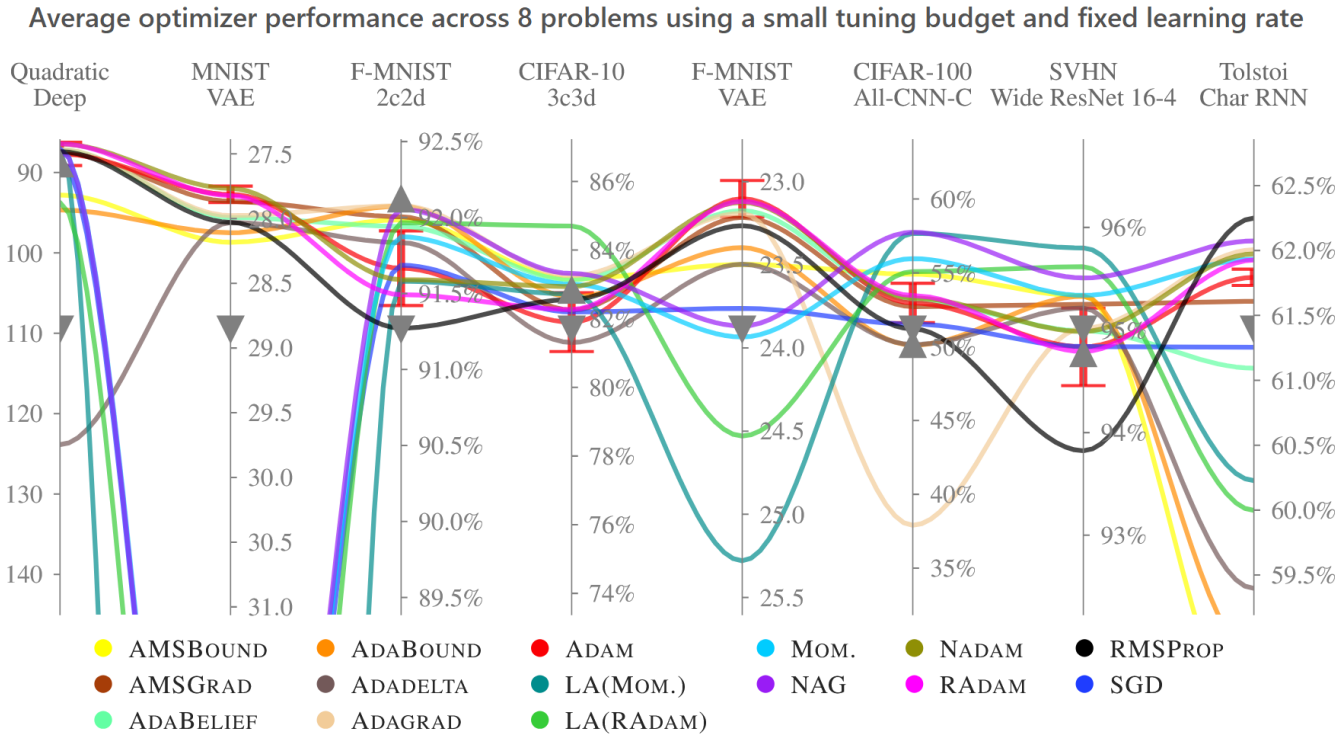

Figure 7. Average Optimizer Performance across 8 optimization problems

Source: *https://arxiv.org/abs/2007.01547*

2.2 Natural Language Processing

2.2.1 Byte-Pair Encoding (BPE) and SentencePiece

Tokenization is the process of dividing a sequence into smaller parts and it is a critical step in processing text data for machine learning. Methods that tokenize based on words or spaces have two significant drawbacks. They can create a huge vocabulary and they cannot handle out of vocabulary words, which are words that were not encountered during training. When an unknown word is encountered it is only mapped to a generic "unknown" token, resulting in loss of information. Subword tokenization algorithms, such as Byte-Pair Encoding (BPE), were developed to address these issues by creating tokens based on the most common sequences of characters in the data [7].

The BPE algorithm is simple but powerful. First, it initializes the vocabulary with all individual characters present in the training corpus. It then iteratively counts all adjacent pairs of tokens in the corpus and merges the most frequent pair into a single token which is added to the vocabulary. This process is repeated a set number of times, creating a vocabulary that contains individual characters and common subwords. That way when tokenizing new text, BPE can represent even unknown words by breaking them down into a sequence of these known subword tokens. This approach eliminates the out of vocabulary problem while maintaining a manageable final vocabulary size.

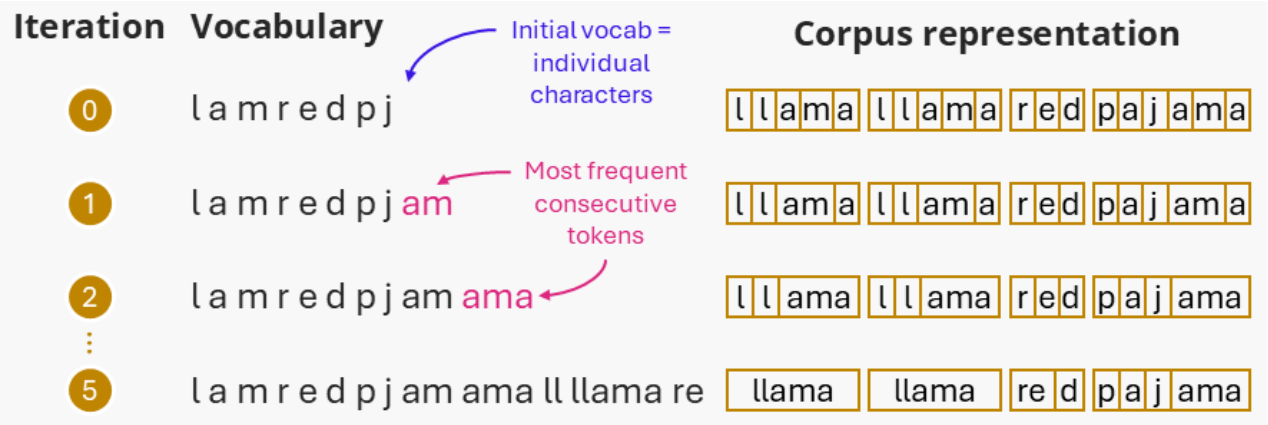

Figure 8. Visualization of the Byte-Pair Encoding (BPE) tokenization algorithm

Source: *https://oneminutenlp.com/p/byte-pair-encoding-algorithm*

This technique is highly effective for specialized data, such as source code or assembly language. In assembly the vocabulary consists of opcodes, mnemonics, registers and an almost infinite number of possible immediate values and memory addresses. BPE can learn to represent common opcodes as single tokens while breaking down less common instructions or large numbers into smaller and reusable pieces.

SentencePiece [12] is an open-source subword tokenizer and detokenizer that offers optimized implementations of algorithms such as BPE. It uses several optimization techniques to provide a significant improvement in computational efficiency, such as by utilizing a binary heap to reduce the complexity of BPE segmentation from O(N^2) down to O(N log(N)). This results in a substantial increase in performance, with the authors noting a 380 times faster segmentation speed on raw Japanese data compared to that of the standard subword-nmt implementation. Furthermore, SentencePiece is designed to contain all the necessary rules and parameters in the model file and thus does not rely on external language specific tools like traditional BPE approaches. Lastly, SentencePiece is developed as an end-to-end system able to directly convert the raw text input into a sequence of vocabulary IDs without dependencies, which greatly simplifies the development of machine learning datasets.

2.2.2 Token Normalization

Normalization is often a beneficial step to perform before tokenization, as it reduces the complexity of the data and helps the model generalize better. In traditional NLP, this includes techniques like converting to lowercase or stemming and lemmatization, which reduce words to their root form [13].

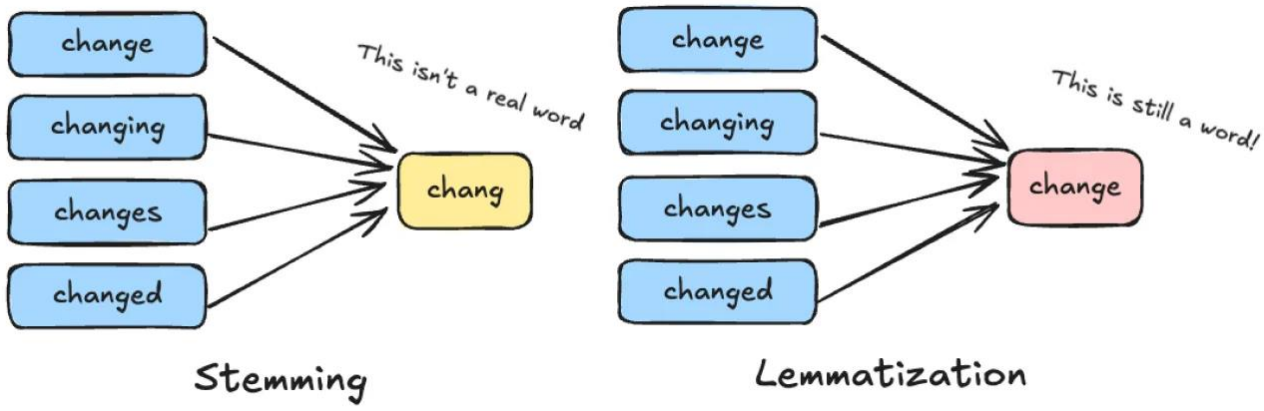

Figure 9. Example of the Stemming and Lemmatization techniques

Source: *https://sendbird.com/blog/what-is-natural-language-processing*

While these specific techniques are not necessarily applicable to code, the underlying principle of abstraction remains crucial. The main point is to reduce data sparsity and superficial variability, which can confuse a machine learning model.

Normalization encourages the model to learn generalizable structural patterns of the code rather than memorizing specific values, thereby reducing the risk of overfitting. By clearing away this noise, we can help a model focus for example on the programmer's abstract stylistic choices while also drastically reducing the size of the vocabulary.

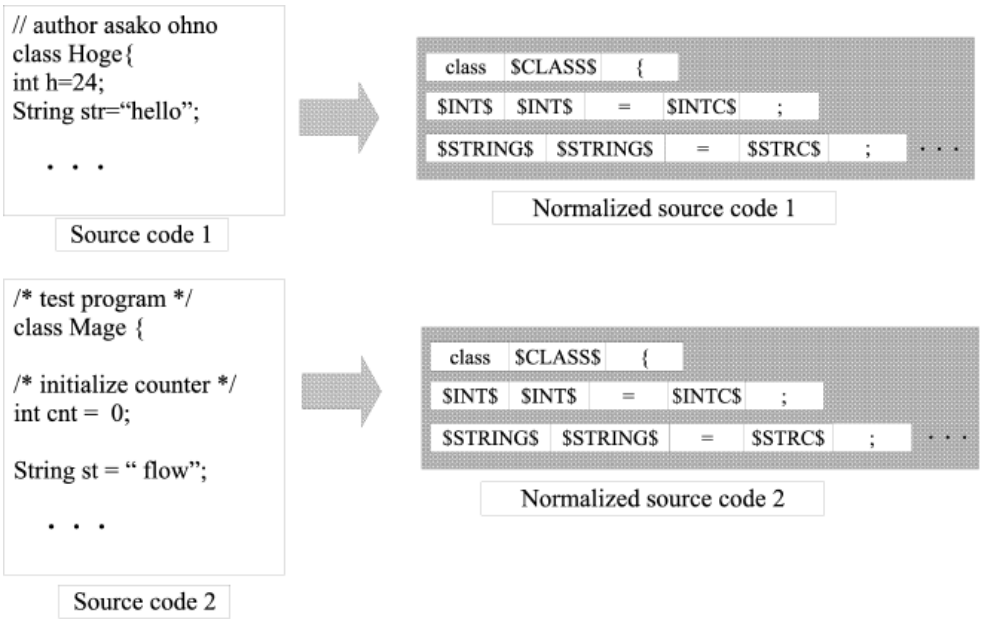

Figure 10. Example of normalized token sequences from the original source codes

Source: *https://www.researchgate.net/publication/224647168*

2.3 Reverse Engineering and Code Disassembly

2.3.1 Interactive Disassembler (IDA Pro)

Interactive Disassembler (IDA Pro) is widely regarded as the industry standard tool for reverse engineering. It is a closed-source and subscription-based software with features for both static and dynamic analysis of binary files. Its main strength is its powerful analysis engine, which can automatically identify functions, data structures and cross-references within a binary. One of its most known features is its ability to visualize the control flow of a function as an intuitive graph, making it much easier for the user to understand the logic behind it.

IDA is not just a passive disassembler, it automatically analyzes and annotates the disassembled code. For example, it replaces specific raw memory addresses with human readable "dummy names" (such as sub_73906D75 for a function, loc_40721B for a location) to provide additional context [1].

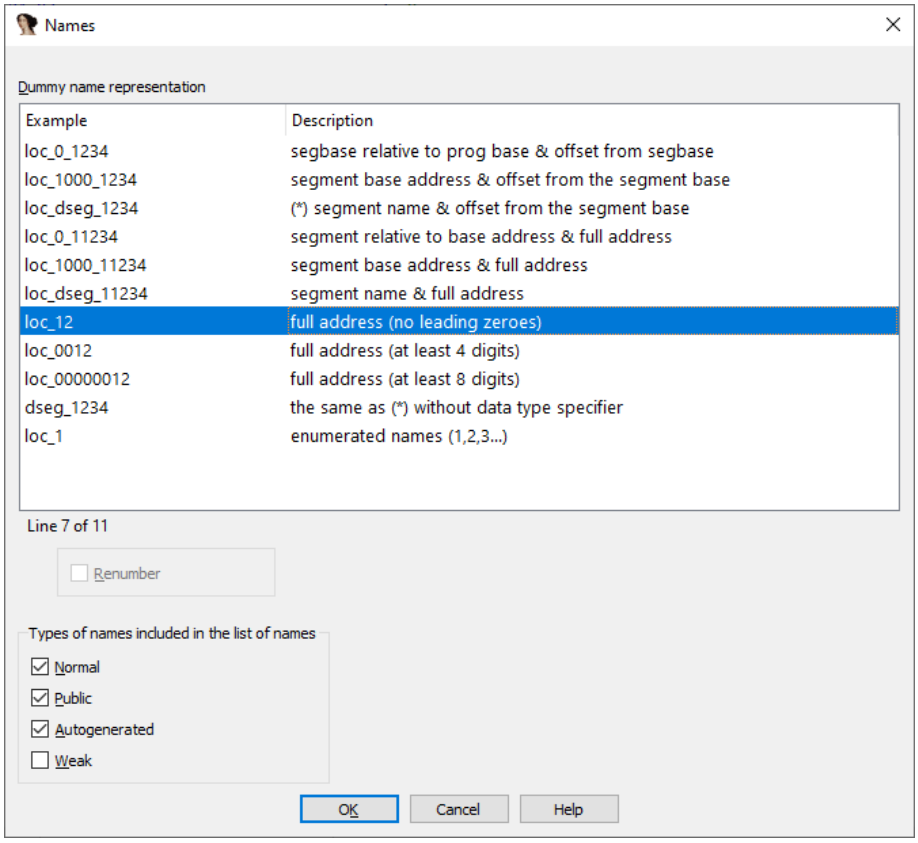

Figure 11. IDA Pro's Dummy Name representation dialog

Source: *https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names*

The program analysis can then be enhanced by building upon this, as the user can rename variables, add comments and define structures. Although its architecture and scripting capabilities allow for some automation, it is mainly designed as an interactive GUI-based tool for expert reverse engineering analysts and is therefore less suitable for large-scale and automated analysis pipelines that need to process thousands of files.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names

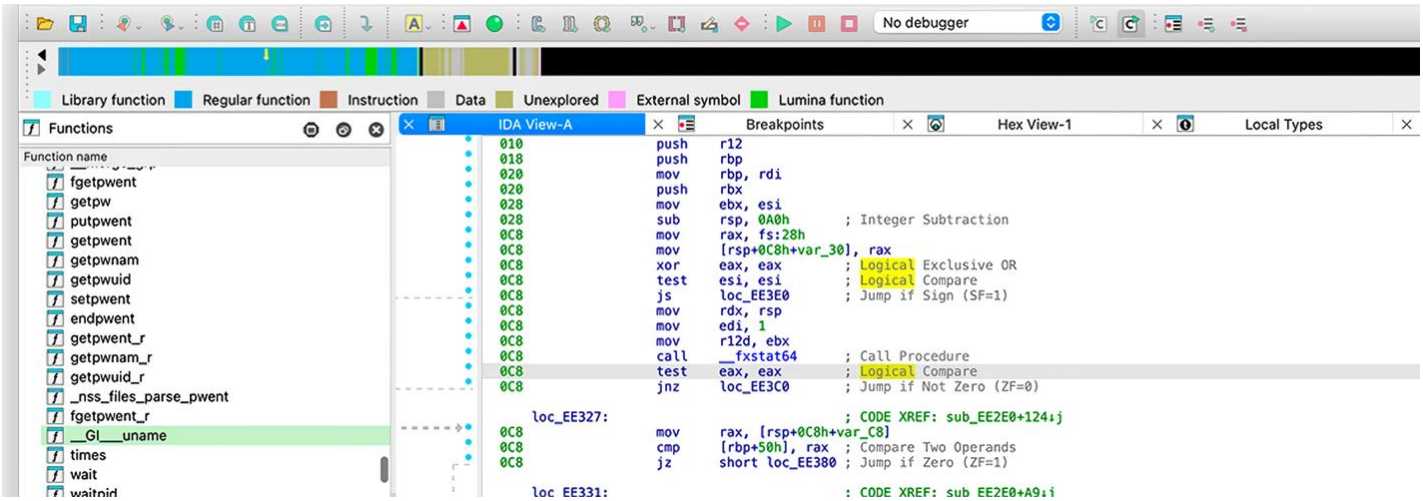

Figure 12. IDA Pro's GUI Overview

Source: *https://hex-rays.com/ida-pro*

2.3.2 Radare2

The Radare2 framework is an open-source alternative to commercial options like IDA Pro and has become increasingly popular in the reverse engineering community. It is a command-line driven package of tools designed for scriptability. While it is a bit more challenging to learn than GUI-based tools, its powerful command-line interface and compatibility with various scripting languages make it ideal for automating complex analysis tasks on binary files [2].

This scriptability is the key feature that makes Radare2 ideal for machine learning applications. The user can easily write a script that programmatically instructs the Radare2 core to perform a series of analysis commands on thousands of files. The output of these commands can be returned in structured formats such as JSON, which can be easily parsed and integrated into a larger data processing pipeline. Radare2's analysis capabilities allow for the automatic identification of all functions within an executable, the extraction of raw opcodes for each instruction and the generation of a complete disassembly for any given function. This ability to perform repeatable and automated function-level extraction is necessary for creating the large datasets required to train machine learning models for binary analysis. Furthermore, its ability to provide detailed and structured information for each instruction makes it ideal for a subsequent normalization step.

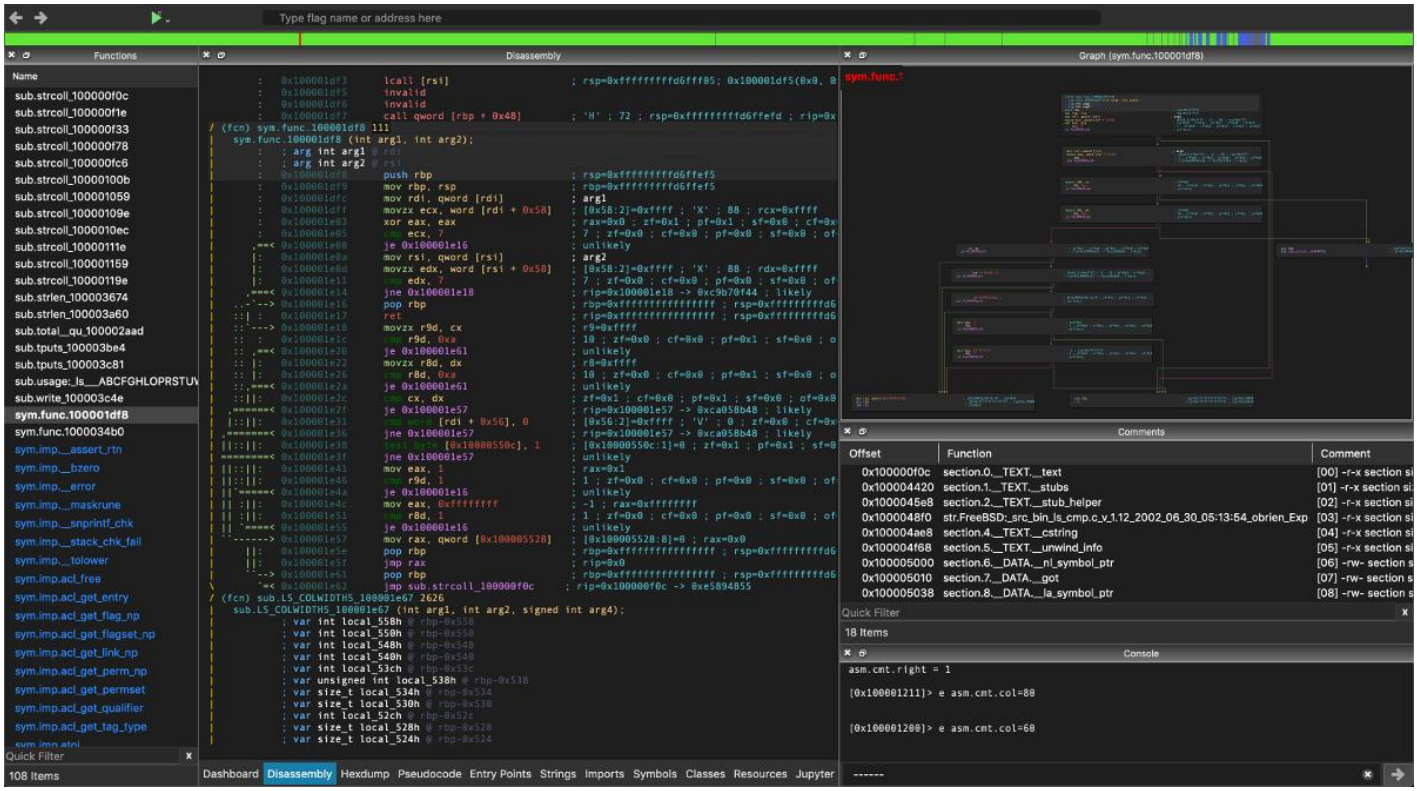

Figure 13. Radare2's iaito GUI Overview

Source: *https://www.radare.org/n/*

[2] https://book.rada.re/

2.4 Compilers and Obfuscation Techniques

2.4.1 GCC/G++ Compiler

The GNU Compiler Collection (GCC), which includes the G++ compiler for C++, is one of the most widely used and foundational open-source compiler collections in software development. It has a long history and is renowned for its stability as well as its support for a vast number of programming languages and target architectures. It is the default choice on many operating systems, particularly in Linux.

GCC has support for multiple optimization levels which can be specified using command-line flags. These optimization levels instruct the compiler to apply various code transformations to ultimately improve the program's execution speed and reduce its size. By default, on many systems GCC links against shared libraries dynamically. This means that the code for standard library functions is not included in the executable itself but is instead loaded from the operating system at runtime, thus resulting in smaller binary files.

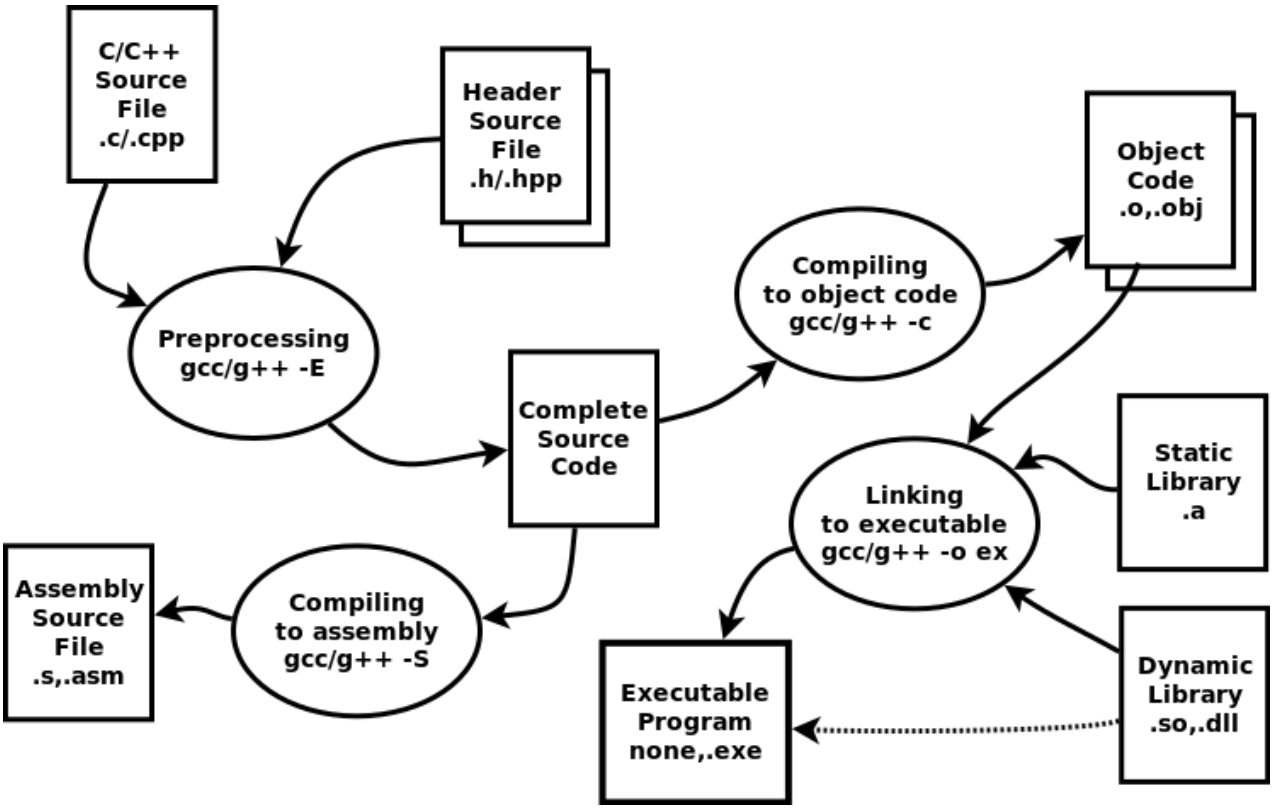

Figure 14. The GCC Compilation Process

Source: *https://www.cs.nmsu.edu/~jcook/posts/c-compiling-linking/*

2.4.2 Clang Compiler

Clang is a modern compiler front-end designed as a high-performance replacement for GCC, particularly for C-family languages. It is the main front-end for the larger LLVM (Low Level Virtual Machine) compiler infrastructure, which is known for its clean and modular design as well as its powerful optimization capabilities. Compared to GCC, Clang is known for its faster compilation speeds and its more user-friendly error and warning messages.

A substantial distinction in its behavior, especially on operating systems like Windows, is its default linking strategy. While GCC defaults to dynamic linking, Clang on Windows may default to static linking. When static linking is used, the code for all required library functions is copied directly into the final executable file. This creates a binary with no external dependencies, which can simplify distribution. However, it also results in a significantly larger file size, as the executable is bloated with standard library code that is not written by the programmer. This behavior can typically be changed by passing specific parameters to the compiler to enforce dynamic linking, however it remains that it is not the default behavior. Even when using dynamic linking similar to GCC, the binaries produced have very different characteristics from those produced by GCC from the exact same source code. The fact that compilers produce such different outputs provides a perfect test case for measuring the reliability of any binary analysis system that tries to identify author characteristics.

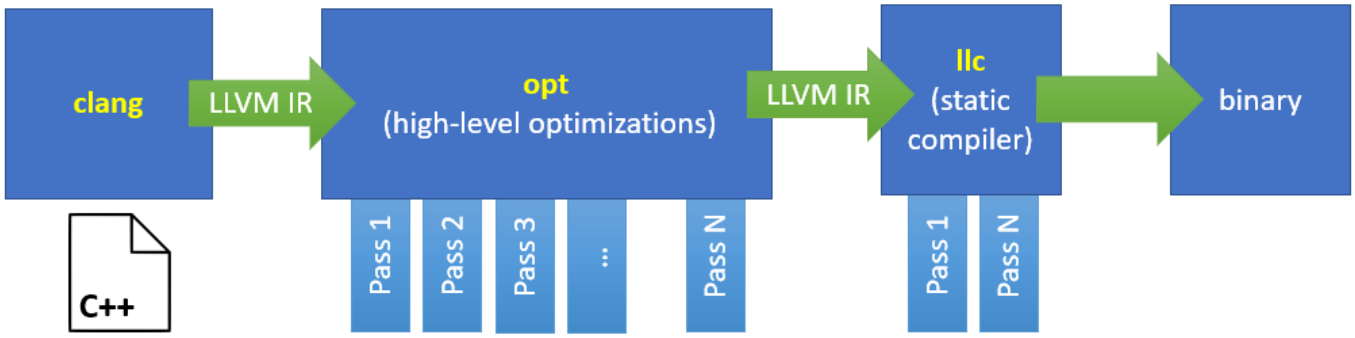

Figure 15. High-Level architecture of LLVM/Clang Compilation

Source: *https://egorbo.com/opt-for-llvm-guide.html*

2.4.3 Obfuscator-LLVM

Code obfuscation regards specific techniques applied to a program that, while not changing its functional behavior, make it more difficult for humans to understand through software analysis. This is often used as a form of software protection to confuse reverse engineering attempts and intellectual property theft.

Obfuscator-LLVM (OLLVM) is a well-known and open-source project based on the LLVM infrastructure that provides several common obfuscation passes that can be applied during the compilation process. These include techniques like instruction substitution (replacing simple operators with equivalent but more complex sequences), bogus control flow (adding opaque predicates and fake branches to obscure the control flow graph) and control flow flattening (rewriting a function's logic as a dispatcher-driven state machine) [14].

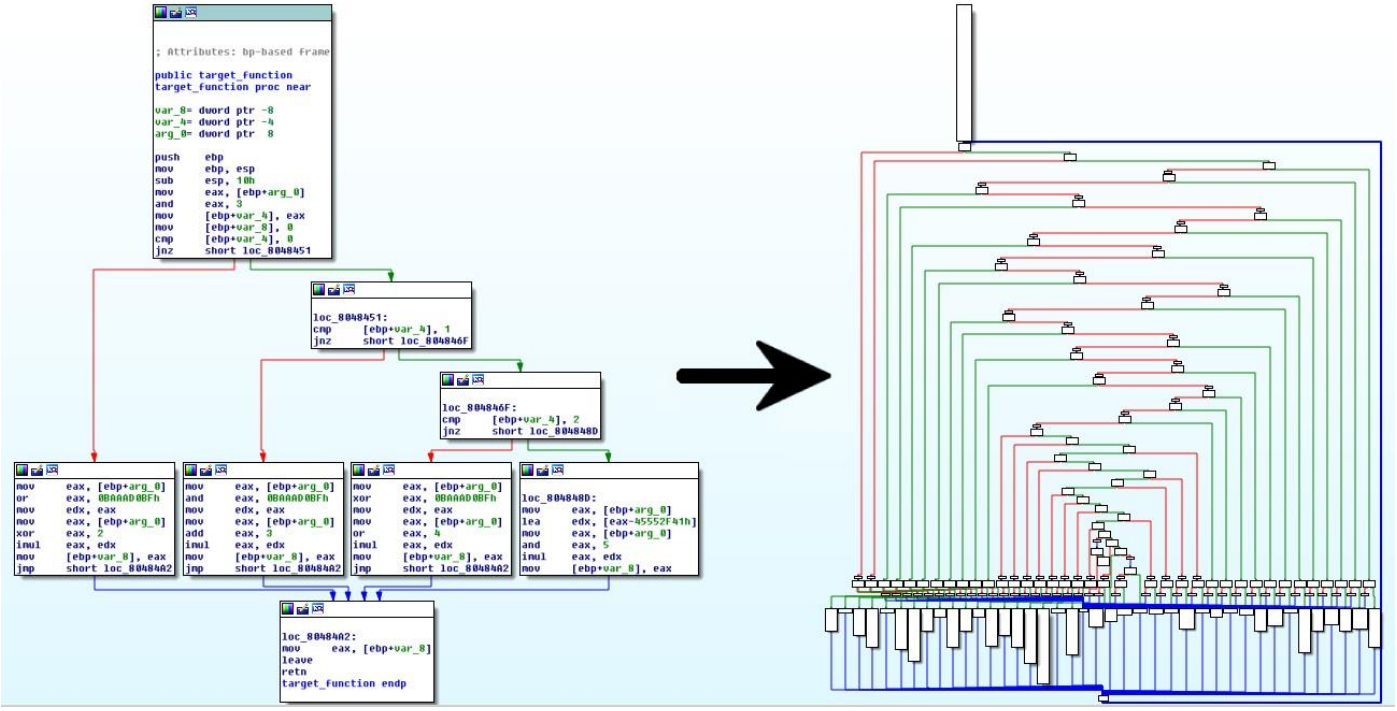

Figure 16. Example of obfuscated result using OLLVM

Source: *https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html*

While many obfuscation techniques exist in theory, only a few are widely used in practice and even popular open-source projects like the original OLLVM are often no longer actively maintained. Modern versions of these obfuscators are typically community-driven forks or reimplementations that have been adjusted to work with newer versions of the LLVM framework.

Dedicated deobfuscators have also been continuously developed to automatically reverse the effects of the most common obfuscation techniques. Nevertheless, obfuscated binaries can create a challenging test case for evaluating the resilience of any code analysis system, as they test the system's ability to see through such transformations and identify the core underlying logic.

3

Related Work

3.1 VeriBin

VeriBin [15] is a system proposed for the binary authorship verification problem, which is designed to determine if two arbitrary binary files were written by the same author in an open-world setting. The authors of VeriBin previously developed SCS-Gan [16], a similar adversarial system designed for source code authorship verification upon which VeriBin builds.

The system's processing pipeline starts by disassembling binaries into assembly code using the commercial tool IDA Pro. The system has a token filtering mechanism that removes certain tokens such as compiler-generated or functionality-specific identifiers. This filtering is implemented by removing tokens that start with specific prefixes and it is highly dependent on IDA Pro's output, which automatically generates these "dummy names". Adapting this filtering logic to other disassemblers like Radare2, which do not follow the same naming conventions, would be a non-trivial task. After filtering, the assembly code is tokenized using SentencePiece, a subword tokenization system used to mitigate the out of vocabulary issue and reduce the total number of unique tokens. These tokens are converted using a pretrained Skip-gram model into vectors that are further adjusted later in the training process.

For the training stage the core architecture is a Siamese network structure. Each encoder is identical and consists of a Bidirectional LSTM (BiLSTM) with 128 units, followed by a multi-head self-attention mechanism with 16 heads. This architecture processes input token sequences (up to a length of 1200) to generate a stylometric representation. The similarity between the two binaries is then calculated using the cosine similarity of the resulting style vectors.

The innovative part of the VeriBin system is the Generative Adversarial Network (GAN) structure utilized for functionality debiasing. The adversarial component is a functionality debiasing discriminator, which is a two-layer dense neural network that attempts to predict the functional similarity of the input binaries based on the generated style vectors. The encoder is trained to fool the discriminator by generating representations that contain no functional information. This adversarial training forces the stylometric representations to be functionality-agnostic, ensuring the model focuses only on the consistency of the author's coding styles. The model is trained to optimize the resulting scores against the ground truth labels using a binary cross-entropy loss function and a Nadam optimizer with a learning rate of 0.01.

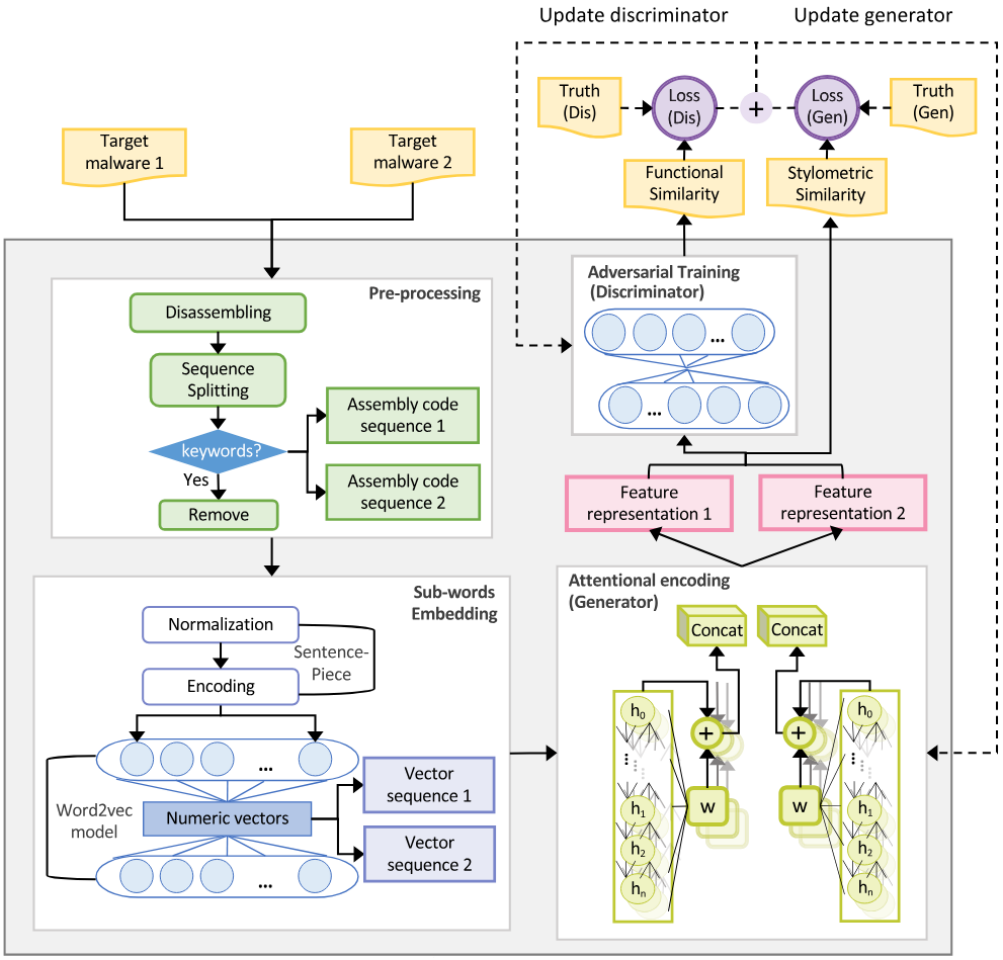

Figure 17. Overview of the VeriBin model

Source: [15]

In their evaluation, the authors used datasets from Google Code Jam (GCJ) and a collection of APT malware samples, all compiled only with GCC. On the GCJ dataset, VeriBin achieved high performance, with an AUROC score of 0.95 for known authors and 0.91 for unseen authors. However, the system's performance degrades significantly when faced with packed binaries. The results show that packing has a dominant effect, causing binaries packed with the same packer to appear highly similar, regardless of the author. This indicates that the model is heavily influenced by the packer's code rather than the actual author's style. The experimental results also show that the improvement gained from the GAN component is not always significant relative to the added complexity of the GAN architecture. Finally, a significant practical limitation is reproducibility as although the authors claim that the code has been released, the provided GitHub repository appears to only contain draft code and not the full functional implementation. The reliance on the closed-source and subscription-based IDA Pro disassembler also significantly hinders accessibility and further research.

3.2 SepBIN

SepBIN [17] is a multi-task framework designed for simultaneous binary semantic comparison and binary authorship verification. The authors argue that for semantic comparison, style is noise and for authorship verification, semantics are noise. Therefore, SepBIN is designed as a multi-task learning system that explicitly attempts to separate these two feature types to improve performance on both tasks.

The system's preprocessing pipeline begins by disassembling binaries using the open-source tool Radare2 to extract Control Flow Graphs (CFGs). A graph-based representation is selected because it captures the program's logical structure, which is more robust to simple obfuscations like instruction substitution than linear sequences. All function CFGs within a binary are treated as a single disconnected graph. Each basic block within a CFG is a node and its instructions are tokenized and passed through a 1D-Convolutional layer to produce a single vector representation for that block.

The training stage of SepBIN begins with a Backbone Binary Encoder, a two-layer Graph Isomorphism Network (GIN), which processes the entire binary graph to create an initial graph-level vector representation. This vector contains both semantic and stylistic information and is passed into a Preliminary Decomposition Module. This module uses three independent Multi-Layer Perceptron (MLP) subnets to perform a rough separation of the features into semantic, stylistic and common (relevant to both tasks) features.

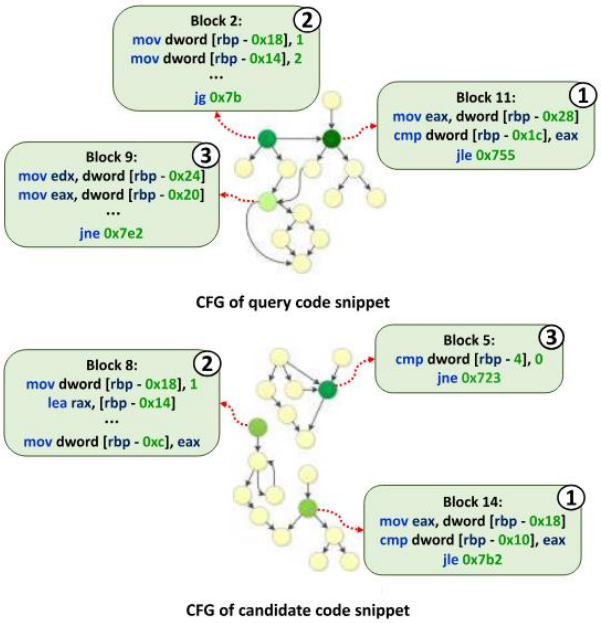

Figure 18. Visualization of the Control Flow Graphs used in SepBIN

Source: [17]

The system then uses two Refinement Modules to further refine this separation, one for semantics and one for style. These modules use a gating mechanism to learn a weighted selection of the raw feature patterns to effectively remove noise. After this refinement, the vectors are then processed by two final MLPs to produce representations for the semantic comparison and authorship verification tasks. This separation process is regularized with a final Reconstruction Module which attempts to reconstruct the initial binary representation from the two refined vectors to avoid information loss from the original input.

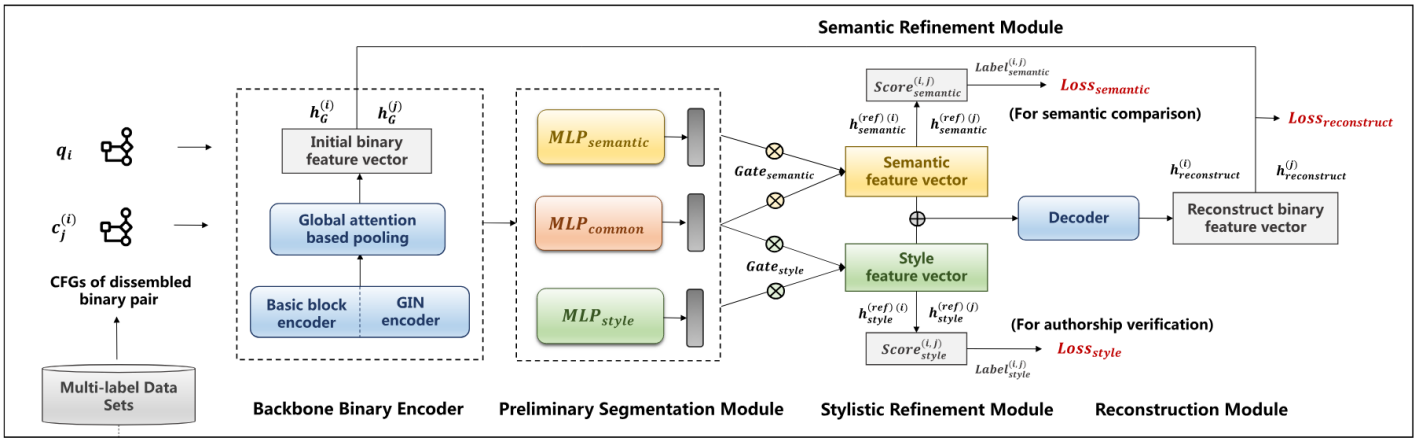

Figure 19. Overview of the SepBIN framework

Source: [17]

SepBIN is trained using a multi-objective joint optimization strategy with the Adam optimizer, where the total loss is a weighted sum of the semantic comparison loss, the authorship verification loss and the reconstruction loss. For the semantic comparison and authorship verification, the loss is derived using the binary cross-entropy loss function, while for the reconstruction the loss is calculated with the mean squared error function. The Adam optimizer has different learning rates for the two tasks, with 1e-3 for semantic comparison and 1e-4 for authorship verification.

For evaluation on datasets like Google Code Jam, the authors use a ranking-based approach with metrics such as Precision@K and Recall@K. SepBIN demonstrates strong performance, reaching a Precision@1 of 90.4% for authorship verification on the GCJ dataset. The system also demonstrates high resilience against code obfuscation techniques for semantic comparison, with the fully finetuned SepBIN achieving Precision@1 values reaching 98.48% on single obfuscation and 96.16% on combined obfuscation datasets. The authors also highlight the ability to finetune the pretrained model on new single-task datasets which allows for quick adaptation to new data.

The primary disadvantage of SepBIN is its high complexity. The multi-stage architecture, combined with the difficulty of constructing and processing graph-based representations, makes the system significantly more challenging to implement and train than a system based on linear sequences. Similar to the VeriBin paper, the authors do not provide a link to a functional open-source implementation, hindering reproducibility. Furthermore, the multi-task learning approach requires a dataset with both semantic and authorship labels for its initial training, which can be difficult to create for real-world applications and represents a significant data dependency.

4

Proposed System

4.1 Overall System Architecture

The proposed system, similar to the existing systems, is designed as a two-stage process. The two primary stages include a comprehensive preprocessing pipeline followed by a deep learning model for similarity analysis.

The main purpose of the preprocessing pipeline is to convert a corpus of C/C++ source code files from multiple authors into numerical data that the deep learning model can then be trained on. This involves a multi-step process that starts by compiling the C/C++ source code files using a variety of compilers and settings to generate the binary executables. The pipeline then disassembles these binaries to extract individual assembly functions, which are normalized and tokenized into sequences of integers. The final output of this stage is a structured dataset of function pairs with a ground truth label about whether they were originally from the same author or not.

The second stage of the proposed system is a deep learning model specifically designed for similarity analysis. In this stage, the output structured dataset of function pairs from the preprocessing pipeline is used to train a Siamese neural network. The deep learning model consists of identical encoders that generate style vectors and a decision head that learns to compare them to produce the final similarity score. The system is trained to optimize this score against the ground truth labels to create a system that can accurately verify authorship from unseen binary files.

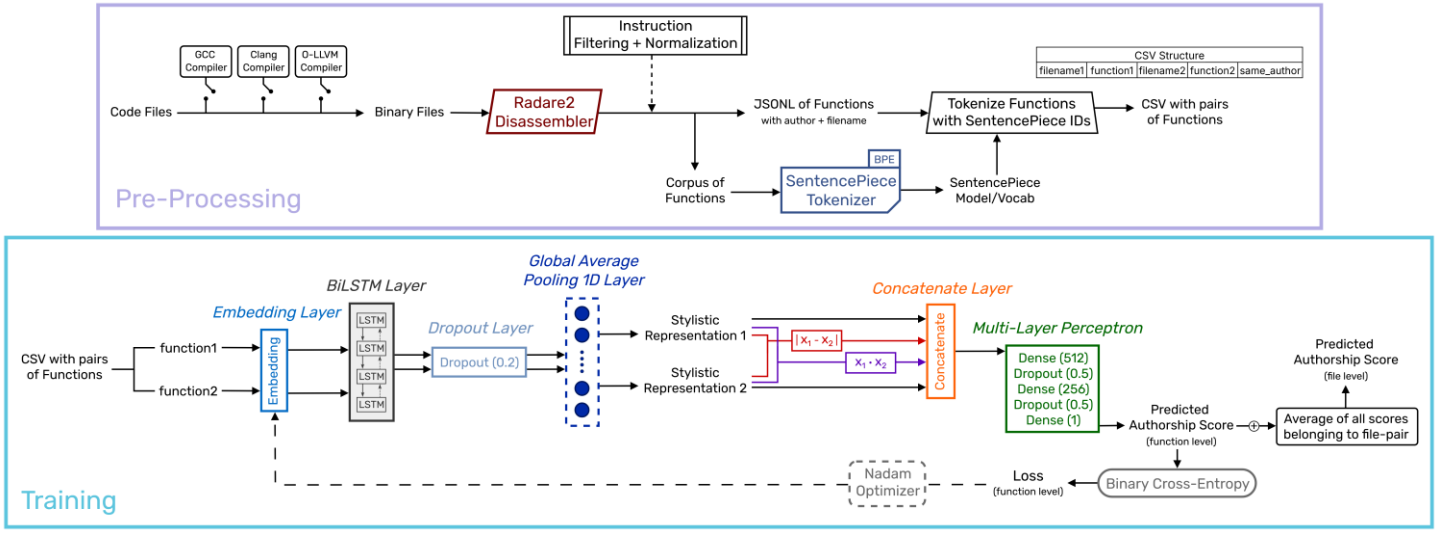

Figure 20. Detailed Architecture of the Proposed System

4.2 Pre-Processing Pipeline

The preprocessing pipeline is a modular and configurable system designed to transform raw source code into a structured dataset ready for training. The execution flow of the preprocessing script is controlled by a series of user-defined boolean flags that decide which stage will be executed. This modularity is beneficial for development and experimentation, as it avoids the need to rerun the entire pipeline if only a later stage needs to be executed.

The first stage of the pipeline is the compilation of C/C++ source files (.c/.cpp) into binary executables (.exe). The script can be configured to compile only a specified percentage of the available files or to ignore authors from a predefined list. These features are especially useful for creating smaller test datasets or for separating training and testing authors. The compilation is performed in parallel using a pool of threads and it supports a multitude of compilers to generate a diverse set of binaries. This includes GCC/G++, Clang and a version of Clang for obfuscation based on Obfuscator-LLVM. Support for additional compilers can be implemented easily, with minor changes to the script. For Clang compilation, the /MD parameter is used to apply dynamic linking. This is done to avoid bloating the binaries with compiler code, such as library functions, and to be consistent with the default behavior of GCC compilation. However, this inability to properly handle binaries bloated with compiler code represents a shortcoming of the script. For obfuscated compilation, the script is designed to apply several obfuscation passes, including instruction substitution (-sub), bogus control flow (-bcf) and control flow flattening (-fla).

The second stage is the disassembly of the generated binaries into assembly code. This process uses the r2pipe library to interface with the Radare2 reverse engineering framework. For each binary, the script performs an analysis to identify function boundaries and then extracts the assembly instructions for each function. A critical part of this stage is the normalization function which processes the assembly tokens. It replaces specific information such as memory calls and hexadecimal addresses with generic placeholders like mem and addr. This normalization process is highly effective as it allows the entire vocabulary of the assembly language corpus to be captured by a SentencePiece model with just about 1000 tokens. The extracted and normalized functions are then streamed into two separate files: a raw text corpus (.txt) for tokenizer training and a structured JSON Lines file (.jsonl) that contains the author and filename metadata for each function. The script also includes an optional step to generate a list of all authors present in the .jsonl file, which can then be used by the author exclusion feature in subsequent runs.

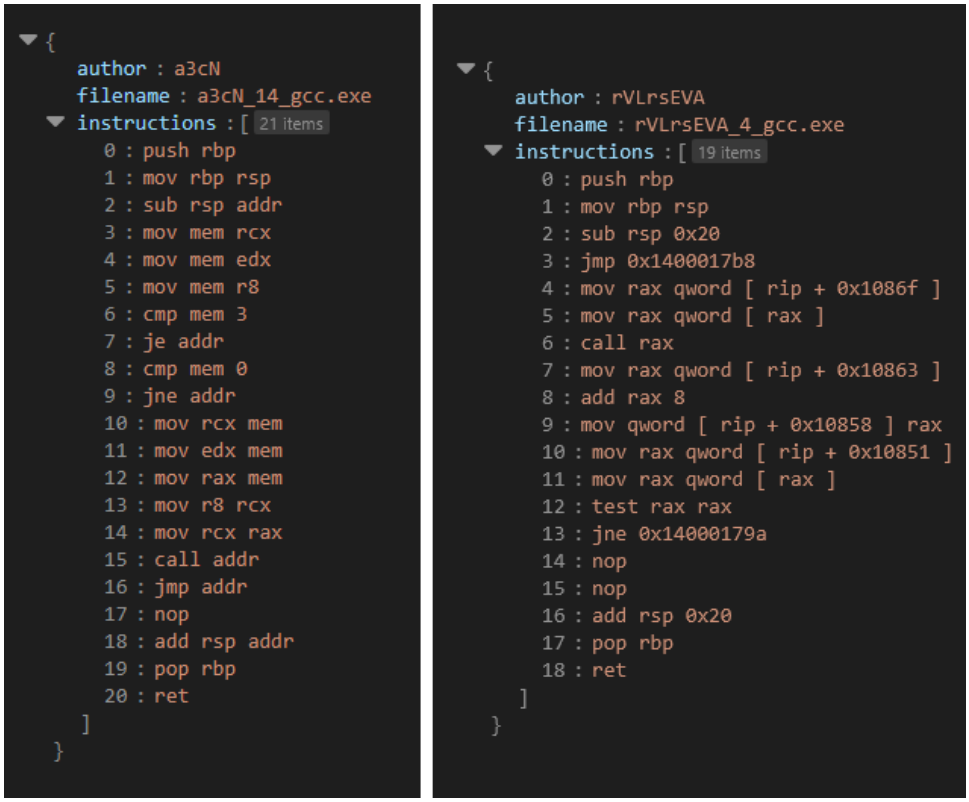

Figure 21. Example of JSONL function entries with token normalization (left) and without (right)

The third stage is the training of a SentencePiece tokenizer. Using the generated text corpus, the script trains a Byte-Pair Encoding (BPE) model (.model) that learns to convert the normalized assembly language into sequences of integer IDs. This step should only be performed when creating a training dataset. When preprocessing a testing dataset, this tokenizer training step must be skipped and the .model file generated from the training data must be used instead. This makes sure that the tokens in the testing data are correctly mapped to the integer IDs that the deep learning model was trained on.

The final stage of the pipeline is the creation of the dataset containing pairs of functions. To make this process memory-efficient, the script first processes the large .jsonl file into a temporary tokenized .jsonl file where each function's instructions are converted into SentencePiece IDs. The script then generates pairs using this indexed file. The logic for pair generation is designed to create a balanced dataset. For each author, a specified number of positive pairs is generated by randomly selecting two different functions from that author. Subsequently, an equal number of negative pairs is generated by pairing a function from the current author with a function from a different randomly chosen author. This makes sure that the final dataset has an equal amount of positive and negative pairs of functions. Each generated pair is stored as a row containing the tokenized sequences for both functions, the ground truth similarity label (1 for same author, 0 for different) and the original source filenames. After all pairs have been generated, the entire dataset is randomly shuffled to guarantee that the training process does not encounter the data rows in a structured order. This shuffled dataset is then saved as the final CSV file (.csv) which is the input for the training stage.

filename1	function1	filename2	function2	same_author
zToAhwZ1_3_gcc.exe	2 56 28 7 28 37 43 37 16 7 10 25 7 10 29 7 10 41 7 10 41 27 16 7 17 10 87 39 23 10 7 15 10 13 15 23 7 17 10 26 17 50 45 16 13 10 50 13 10 50 7 17 10 26 17 10 113 16 7 17 10 13 37 16 57 28 61 3	j55SwzWAPU3TUXmU8DC_4_gcc.exe	2 56 28 7 28 37 43 37 16 7 10 25 7 15 10 7 15 10 7 10 15 7 15 10 7 25 15 35 16 26 15 10 358 81 52 81 81 7 15 10 13 37 16 57 28 61 3	0
UsYPbze1DyetVEx_1_gcc.exe	2 56 28 7 28 37 43 37 16 7 10 44 7 17 10 26 17 70 53 16 7 29 10 7 15 23 67 15 70 13 15 23 67 15 79 39 23 10 13 15 23 7 25 15 7 15 10 35 15 62 13 37 16 57 28 61 3	UsYPbze1DyetVEx_3_gcc.exe	2 56 28 7 28 37 43 37 16 7 10 25 7 15 10 7 10 41 7 15 10 7 10 41 7 15 10 7 10 41 7 15 10 7 25 15 35 16 62 13 37 16 57 28 61 3	1

Figure 22. Example of the CSV entries

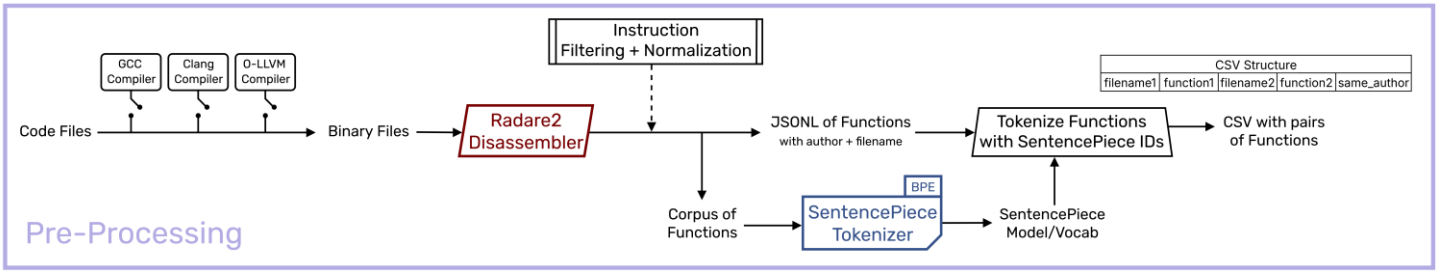

Figure 23. Overview of the Proposed System's Preprocessing Pipeline

4.3 Architecture of the Siamese Model

A Siamese neural network is designed to perform similarity analysis on the pairs of code functions. The architecture has two distinct inputs, each accepting a sequence of integer tokens representing a function. These two inputs are then processed by a shared encoder instance which is a fundamental part of the Siamese architecture, as it guarantees that both functions in a pair undergo the exact same transformation process.

The stylistic encoder is a sequential model that maps the tokenized function to a style vector. Its first layer is an embedding layer, which transforms the discrete integer tokens into dense vector representations. This layer is configured to mask padded zero values, which allows it to handle sequences of variable length and makes sure that padding does not affect the model's calculations. The resulting sequence of embeddings is then passed into a BiLSTM layer. This layer processes the sequence in both forward and reverse directions and its outputs are concatenated to produce a context-aware representation for each token. Afterwards, a dropout layer is applied to prevent overfitting. The final layer of the encoder is global average pooling, which reduces the sequence of vectors from the BiLSTM into a single vector by averaging across the sequence dimension. This vector is the final stylistic representation of the input function.

Once the shared encoder has produced the two style vectors, a decision head performs a detailed comparison to compute the final similarity score. This decision head first performs explicit feature extraction to provide the final classifier with enhanced comparative information. Specifically, it computes the absolute element-wise difference and the element-wise product of the two style vectors. These two new vectors are then concatenated with the original two style vectors. This feature vector is then processed by a Multi-Layer Perceptron (MLP). For this component, the same architecture used for classification in the SepBIN system is adopted. Specifically, the MLP consists of two dense layers with 512 and 256 neurons respectively, each using a ReLU activation function and followed by a dropout layer for regularization. The final output is produced by a dense layer with a single neuron and a sigmoid activation function, which yields a similarity score between 0 and 1 for the two input functions.

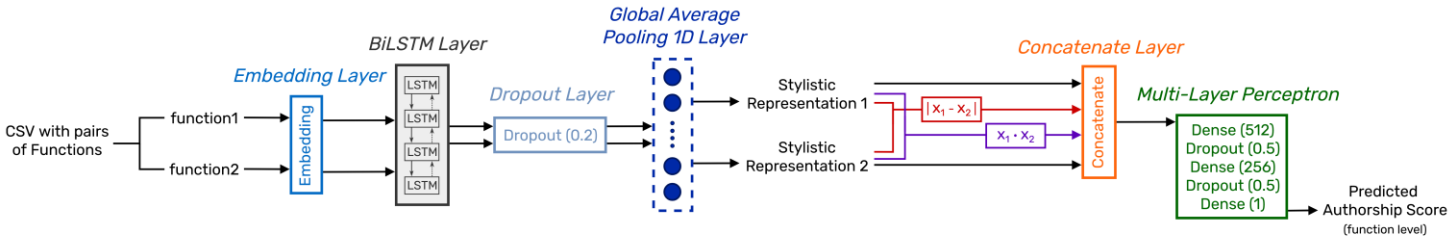

Figure 24. Overview of the Proposed System's Siamese model

4.4 Training and Optimization Process

The training phase starts with partitioning the data to prevent information leakage between the training and validation datasets. A file-based validation split is implemented, where the list of unique source filenames is first partitioned into training and validation sets at a ratio of 80/20. A pair of functions is then assigned to the validation dataset only if both of its original source files belong to the validation file list. This separation makes sure that the model's performance is evaluated on completely unseen file pairs.

For efficient data handling during training, a performant data pipeline is implemented. A generator function reads the dataset, converts the token sequences into integer arrays and applies padding to establish a uniform length for all sequences in a batch. This process is wrapped in a TensorFlow dataset object, which allows for asynchronous data loading. By prefetching data the CPU can prepare the next batch while the GPU is processing the current one, which minimizes bottlenecks and maximizes hardware utilization.

For training, the Nadam optimizer is chosen to update the model's weights. The loss function chosen is binary cross-entropy as it is the standard for binary classification tasks. The validation metric tracked during training is the Area Under the ROC Curve (AUROC), a reliable metric to measure the model's ability to distinguish between positive and negative pairs.

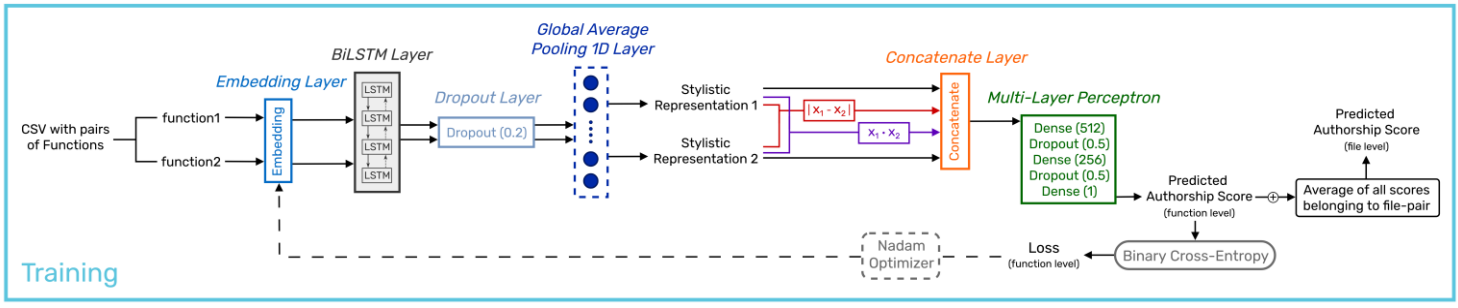

Figure 25. Overview of the Proposed System's full training process

The training process utilizes two crucial callbacks. An early stopping callback stops the training if the validation AUROC has stagnated for several epochs, which prevents the model from overfitting and automatically restores the model weights from the best performing epoch for the final model state. Simultaneously, a model checkpoint callback saves the model's weights to a file whenever the validation AUROC improves, which allows for training to resume from that state in subsequent runs.

At the end of the training process a file-level evaluation occurs. The model predicts similarity scores for all function pairs in the validation dataset and then these scores are aggregated by their original source file pair by taking the average score. The final performance result is the AUROC metric based on these aggregated file-level scores, providing a file-level evaluation of the system.

5

Experimental Evaluation

5.1 Datasets and Experimental Setup

For the implementation of the proposed system a specific set of open-source tools, libraries and hardware was used. All source code used for the implementation is publicly available on GitHub for reproducibility [3].

The preprocessing stage was executed on a local machine equipped with an AMD Ryzen 5 3600 CPU and 16GB of RAM. This stage utilized Python 3.12.10, Clang (version 19.1.5), GCC for MinGW-W64 (version 15.1.0), Radare2 (version 5.9.8) and SentencePiece (version 0.2.0). For the OLLVM obfuscation compiler, a community-driven patch for LLVM 19.1.5 was used [4]. The main Python libraries used for this stage were pandas (version 2.3.2) for data handling, r2pipe (version 1.9.6) for interfacing with Radare2 and tqdm (version 4.67.1) for progress monitoring.

The training stage was performed on a cloud-based machine from Paperspace. This machine was equipped with a Nvidia A6000 GPU, an Intel Xeon Gold 5315Y CPU and 45GB of RAM. The main libraries used for this stage were TensorFlow (version 2.19.0 with CUDA) with Keras (version 3.11.3) for the training and algorithms, scikit-learn (version 1.7.2) for evaluation metrics and numpy (version 2.3.3) for numerical operations.

[3] https://github.com/KyriakosAd/BinaryAuthorshipVerification

[4] https://github.com/DreamSoule/ollvm17/issues/38

Following the approach of previous work in this field, the proposed system was evaluated on data from the Google Code Jam (GCJ) programming competition, specifically using only C and C++ source files. Since Google has discontinued the competition, the official archives are no longer available. For this work, the data was sourced from a reliable third-party archive on GitHub [5]. Due to the different, ranking-based evaluation methodology of the SepBIN system, which makes a direct comparison difficult, the proposed system's performance was primarily compared against the results of the VeriBin system. All the tests were performed on code files from authors that had not appeared in the training data, replicating a realistic open-world evaluation scenario.

Table 1. Statistics of Experiment Datasets - Part 1

Test 1 Datasets				
Trained on GCJ 2017, 80% - GCC, Optimization Level 2				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Training	72679	89595	9563	3588344
Validation	14530	24490	6680	236856
Tested on GCJ 2017, 20% - GCC, Optimization Level 2				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	17960	25156	2292	916800

Test 2 Datasets				
Trained on GCJ 2017, 80% - GCC, Optimization Level 2				
Same training and validation statistics as in Test 1.				
Tested on GCJ 2017, 20% - GCC, Optimization Level 3				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	17960	25320	2292	916800

Test 3 Datasets				
Trained on GCJ 2017, 80% - GCC, Optimization Level 2				
Same training and validation statistics as in Test 1.				
Tested on GCJ 2017, 20% - GCC, Optimization Level 0				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	17958	23761	2292	916800

[5] https://github.com/Jur1cek/gcj-dataset

Table 2. Statistics of Experiment Datasets - Part 2

Test 4 Datasets				
Trained on GCJ 2017, 20% - GCC, Optimization Level 0				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Training	17958	23672	2292	861015
Validation	3579	6017	1621	55785
Tested on GCJ 2017, 80% - GCC, Optimization Level 2				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	72679	89622	9563	3825200

Test 5 Datasets				
Trained on GCJ 2017, 100% - GCC + Clang, Optimization Level 2				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Training	178963	182666	11951	4539255
Validation	35305	40999	10068	241145
Tested on GCJ 2016, 100% - GCC + Clang, Optimization Level 2				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	47662	40097	7525	3010000

Test 6 Datasets				
Trained on GCJ 2017, 100% - GCC + Clang, Optimization Level 2				
Same training and validation statistics as in Test 5.				
Tested on GCJ 2016, 10% - OLLVM, Optimization Level 3				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	4761	79669	718	287200

Test 7 Datasets				
Trained on GCJ 2017, 10% - GCC + Clang + OLLVM, Optimization Level 0				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Training	27072	93847	1242	472441
Validation	5111	8898	1110	24359
Tested on GCJ 2016, 10% - GCC + Clang + OLLVM, Optimization Level 3				
Dataset Type	Unique Files	Unique Functions	Authors	Pairs (Verification Cases)
Testing	27227	113479	1339	535600

For the preprocessing pipeline, the SentencePiece tokenizer was set to create a vocabulary size of 1000, which was sufficient due to the normalization process significantly reducing the total assembly tokens. For dataset creation, 200 positive function pairs and a corresponding 200 negative function pairs were generated for each author.

The training model was configured with a maximum sequence length of 256. This relatively short sequence length is sufficient because the system trains at function level and most individual functions in the dataset are well within this token limit after disassembly and tokenization. The embedding dimension was set to 128 and the batch size was set to 512. The Nadam optimizer was configured with the default learning rate of 0.001 after a few experiments of trying different learning rates with minimal impact. The model was trained for a maximum of 15 epochs, with an early stopping patience of 10 epochs.

5.2 Evaluation Metrics

The primary metrics used to evaluate the model's performance are Area Under the ROC Curve (AUROC), Accuracy and F1-Score. These metrics are calculated first at the function-level and then at the more practical file-level after aggregating the function pair scores. These file-level results directly measure the system's effectiveness on the overall authorship verification task.

Although metrics such as Precision and Recall are also calculated, they can be influenced by an imbalance between the number of positive and negative file pairs in the test dataset. This imbalance can appear due to the way the data generation process focuses on creating a balanced amount of function pairs for each author while not explicitly balancing the resulting number of unique file pairs.

5.3 Analysis of Results

The proposed system was subjected to seven tests to evaluate its performance and resilience under various conditions.

In the first test, the system was trained on 80% of the files from the GCJ 2017 dataset, compiled with GCC at optimization level -O2. It was then tested on the remaining 20% of files from the same dataset and compilation settings. The proposed system achieved a file-level AUROC of 0.992, significantly outperforming the 0.91 AUROC reported for the VeriBin system under similar conditions. To validate the importance of the assembly normalization step, the same test was run with the filtering process removed. Performance decreased significantly to an AUROC of 0.774, confirming that the normalization of tokens like memory addresses is a necessary preprocessing step for the model to effectively learn stylistic features.

Table 3. Evaluation Metrics - Test 1

System	AUROC	Accuracy	Precision	Recall	F1-Score
VeriBin	0.91	0.86	0.89	0.81	0.85
Proposed	0.992	0.976	0.887	0.979	0.930
Proposed without Filtering	0.774	0.866	0.665	0.378	0.482

The second test assessed the system's resilience to different optimization levels. The model was trained on 80% of the GCJ 2017 files compiled with GCC at optimization level -O2 and was tested on the remaining 20% of files from the same dataset, but this time compiled with the higher optimization level -O3. The system maintained excellent performance with an AUROC of 0.992, again outperforming VeriBin's reported AUROC of 0.92. This indicates that the stylistic features learned by the model are largely preserved even when more aggressive compiler optimizations are applied, showcasing the model's resilience in a cross-optimization scenario.

Table 4. Evaluation Metrics - Test 2

System	AUROC	Accuracy	Precision	Recall	F1-Score
VeriBin	0.92	0.85	0.90	0.78	0.83
Proposed	0.992	0.976	0.889	0.978	0.931

The third test assessed the impact of testing on unoptimized code. The model was trained on 80% of GCJ 2017 files compiled with GCC at optimization level -O2 and was tested on the remaining 20% of files compiled without any optimization (-O0). Performance decreased significantly, with the Accuracy dropping to 0.667 and the F1-score to 0.481. This suggests that the features learned from the concise code produced by optimization do not generalize properly to the structurally different code generated without any optimization.

Table 5. Evaluation Metrics - Test 3

AUROC	Accuracy	Precision	Recall	F1-Score
0.950	0.667	0.312	0.960	0.481

The fourth test assessed the reverse scenario, the model trained on the smaller 20% subset of the GCJ 2017 dataset compiled without optimization (-O0) and tested on the larger 80% subset of the same dataset compiled with optimization level -O2. Here, the system achieved excellent performance with an AUROC of 0.984, Accuracy of 0.978 and F1-Score of 0.931. This result suggests that training on unoptimized code allows the model to learn more fundamental and detailed stylistic patterns that are robust enough to persist even after the compiler applies aggressive optimizations.

Table 6. Evaluation Metrics - Test 4

AUROC	Accuracy	Precision	Recall	F1-Score
0.984	0.978	0.941	0.922	0.931

The fifth test evaluated the system's ability to handle different compilers and programming problems. The model was trained on the full GCJ 2017 dataset, with binaries generated from both GCC and Clang at optimization level -O2 and it was tested on the completely different GCJ 2016 dataset, also compiled with both GCC and Clang at optimization level -O2. The system demonstrated excellent performance, achieving an AUROC of 0.997. This result shows that the model successfully learns the programmer's underlying style, independent of the specific programming problem and can adjust to compiler differences as long as it has been exposed to them during training.

Table 7. Evaluation Metrics - Test 5

AUROC	Accuracy	Precision	Recall	F1-Score
0.997	0.997	0.989	0.985	0.987

The sixth test measured the system's resilience against obfuscation when it had not been trained on such data. The model was trained on the full GCJ 2017 dataset, compiled with GCC and Clang at optimization level -O2 and was tested on a 10% subset of the GCJ 2016 dataset that was compiled using OLLVM at optimization level -O3. The performance decreased significantly, with the AUROC dropping to 0.648. This demonstrates that the stylistic features learned from standard compilations are not resilient to obfuscation techniques like control flow flattening and bogus control flow, which are designed specifically to disrupt such patterns.

Table 8. Evaluation Metrics - Test 6

AUROC	Accuracy	Precision	Recall	F1-Score
0.648	0.738	0.259	0.572	0.356

The final test demonstrated the system's ability to adapt to obfuscation. The model was trained on a small 10% subset of the GCJ 2017 dataset compiled with GCC, Clang and OLLVM at optimization level -O0. This model was then tested on a 10% subset of the GCJ 2016 dataset, also compiled with all three compilers but at optimization level -O3. Performance improved to an AUROC of 0.886 which is significant, even if not as impressive as previous tests on non-obfuscated data. This shows that while obfuscation presents a challenge, the model can identify stylistic patterns in obfuscated data, provided that the specific obfuscation technique is included in the training data.

Table 9. Evaluation Metrics - Test 7

AUROC	Accuracy	Precision	Recall	F1-Score
0.886	0.874	0.909	0.680	0.778

6

Conclusions and Future Work

6.1 Summary of Conclusions

The experimental evaluation demonstrates that the proposed system successfully learns to identify stylistic patterns specific to the author from compiled C/C++ binaries. The system achieves a high performance in authorship verification, significantly outperforming the VeriBin system in a as close as possible comparison. The results confirm that the function-level approach combined with the assembly normalization and filtering pipeline is highly effective. The experimental evaluation further underscored the importance of this normalization step, with its removal drastically deteriorating the results.

The system's primary strength proved to be its adaptability. The experiments showed that the model is resilient to variations in compiler optimization levels, especially when trained on unoptimized code as it learns more fundamental stylistic patterns. Furthermore, it demonstrated excellent adaptability to different compilers and even entirely different programming problems, provided that the training data is representative.

However, the evaluation also revealed clear limitations with the system's main disadvantage being its vulnerability to code obfuscation techniques for which it has not been trained. Another potential weakness is the file-level imbalance in the dataset generation process, which may introduce subtle biases into the evaluation metrics and warrants further investigation.

6.2 Future Extensions and Research Directions

For future work the most important task is improving the system's resilience against sophisticated code obfuscation techniques. While including obfuscated samples in the training data proved effective, a more appropriate solution would involve learning features that are inherently resistant to such transformations. A promising approach would be to move beyond linear instruction sequences and incorporate graph-based features. By representing binaries as Control Flow Graphs (CFGs) and using Graph Neural Networks (GNNs) to process them, a future system could learn more complex structural patterns of an author's code that may be more difficult to alter with obfuscation techniques, potentially leading to a more resilient authorship verification model.

Future work should also consider refining the file pair generation logic to address the issue of imbalanced file pairs. An approach could be to select an exact number of functions from each file pair and have each file pair only appear once. However, this could disrupt the balance of positive and negative function pairs. This function-level imbalance could then potentially be resolved by adjusting the number of function pairs generated from negative file pairs to mirror the distribution of function pairs generated from the positive file pairs for each author. This would require careful implementation, as this part of the preprocessing pipeline is critical to the system's overall performance and any changes must be made carefully to avoid introducing new biases.

Regarding the preprocessing script's inability to properly handle binaries bloated with compiler code, future work can improve upon this by using simple techniques such as pattern matching to identify and remove library code from binaries that were created with static linking, which as shown in other work can be rather effective [18].

References

[1] Caliskan, Aylin, et al. "When coding style survives compilation: De-anonymizing programmers from executable binaries." arXiv preprint arXiv:1512.08546 (2015).

[2] Bromley, Jane, et al. "Signature verification using a "siamese" time delay neural network." Advances in neural information processing systems 6 (1993).

[3] Hadsell, Raia, Sumit Chopra, and Yann LeCun. "Dimensionality reduction by learning an invariant mapping." 2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06). Vol. 2. IEEE, 2006.

[4] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

[5] Goodfellow, Ian, et al. Deep learning. Cambridge: MIT press, 2016. Chapter 6 "Deep Feedforward Networks"

[6] Daniel Jurafsky and James H. Martin. 2025. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models, 3rd edition.

[7] Burkov, Andriy. The Hundred-Page Machine Learning Book. 2019.

[8] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[9] Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." IEEE transactions on Signal Processing 45.11 (1997): 2673-2681.

[10] Dozat, Timothy. "Incorporating nesterov momentum into adam." (2016).

[11] Schmidt, Robin M., Frank Schneider, and Philipp Hennig. "Descending through a crowded valley-benchmarking deep learning optimizers." International Conference on Machine Learning. PMLR, 2021.

[12] Kudo, Taku, and John Richardson. "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing." arXiv preprint arXiv:1808.06226 (2018).

[13] Lovins, Julie Beth. "Development of a stemming algorithm." Mech. Transl. Comput. Linguistics 11.1-2 (1968): 22-31.

[14] Junod, Pascal, et al. "Obfuscator-LLVM--software protection for the masses." 2015 ieee/acm 1st international workshop on software protection. IEEE, 2015.

[15] Ou, Weihan, et al. "VeriBin: A Malware Authorship Verification Approach for APT Tracking through Explainable and Functionality-Debiasing Adversarial Representation Learning." ACM Transactions on Privacy and Security 27.3 (2024): 1-37.

[16] Ou, Weihan, et al. "Scs-gan: Learning functionality-agnostic stylometric representations for source code authorship verification." IEEE Transactions on Software Engineering 49.4 (2022): 1426-1442.

[17] Song, Qige, et al. "SepBIN: Binary feature separation for better semantic comparison and authorship verification." IEEE Transactions on Information Forensics and Security 19 (2023): 1372-1387.

[18] Jacobson, Emily R., Nathan Rosenblum, and Barton P. Miller. "Labeling library functions in stripped binaries." Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools. 2011.