# CodeBuster: A Hybrid Similarity Model for Detecting C++ Code Plagiarism

Aaron
*University of Western Ontario*
ayi9@uwo.ca

Arjun
*University of Western Ontario*
akrish73@uwo.ca

Munadir
*University of Western Ontario*
mkha473@uwo.ca

Zhantore
*University of Western Ontario*
zboranga@uwo.ca

Riley Wong
*University of Western Ontario*
rwong366@uwo.ca

Khilan Desai
*University of Western Ontario*
kdesai58@uwo.ca

***Abstract*—With the advent of Artificial Intelligence, academic dishonesty in programming related study has risen dramatically as students use AI tools to mask plagiarism through variable renaming, structural refactoring, and logical changes. We present CodeBuster, a multilayered plagiarism detection system designed to combat these forms of plagiarism using a trained model that computes similarity between two C++ programs more effectively than traditional token only methods. Our approach combines a token based similarity check, a GraphCodeBERT semantic embedding model, and an output similarity check, aggregated into a feature vector and passed through a Multilayer Perceptron (MLP). On our evaluation set, the model achieves an overall accuracy of 0.798 with weighted precision, recall, and F1 scores of 0.801, 0.798, and 0.798, respectively. The confusion matrix (TN=1267, FP=392, FN=247, TP=1253) indicates balanced detection performance across both plagiarized and non plagiarized classes, with strong true positive identification and controlled false negative rates. These results suggest that incorporating semantic and behavioral analysis can have a significant positive detection of sophisticated plagiarism, showing a potential multi layered framework that could bridge the gap between traditional token based detection and semantic-aware, behaviour-aware analysis.**

## I. Introduction

### A. Motivation

With the rapid proliferation of generative artificial intelligence (GenAI) tools in academic settings, maintaining integrity in computer science programming education has become an increasingly urgent challenge. Tools such as ChatGPT, GitHub Copilot, and Claude Code enable students to instantly generate syntactically correct, functional code, creating a new and accessible mechanism for submitting work that is not their own [1]. Evidence confirms the scale of this problem as a 2024 ACM study found a measurable increase in plagiarism in an introductory CS course following the public release of ChatGPT, with students shifting away from traditional plagiarism hubs toward AI as their primary cheating tool, and those who did so showing statistically significant learning losses proportional to the amount of plagiarized work submitted [2].

What makes this form of academic dishonesty particularly difficult to address is that it extends far beyond simple copy and paste. A student can submit a stolen solution to a GenAI model and receive a functionally identical program with renamed variables, restructured control flow, reordered logic, and altered syntax, allowing them to fully preserve runtime behavior while obscuring the code's origin. These techniques, once requiring significant manual effort, are now automated and available to any student. This highlights a critical need for plagiarism detection systems capable of reasoning not just about the textual surface of code, but about its deeper semantic and behavioral properties. This paper presents **CodeBuster**, a multilayered plagiarism detection system for C++ assignments designed to meet this need.

### B. Related Works

The two most widely deployed plagiarism detection tools in academic CS settings are Measure of Software Similarity (MOSS) [3] and JPlag [4]. Both rely on token based comparison: MOSS employs document fingerprinting via the Winnowing algorithm to identify shared substrings between submissions, while JPlag converts programs into token sequences and applies the Greedy String Tiling algorithm to compute pairwise similarity. These tools perform well against direct copying and superficial edits, but they treat code as text rather than as computation. Consequently, they are fully vulnerable to structural obfuscation such as dead code insertion and statement reordering, and in one documented case, MOSS flagged 93% of honest submissions as suspicious simply because the optimal solutions to a short problem converged on the same structure [5]. AI generated code compounds this failure: because it is novel and does not match any previously submitted work, it evades token based detection entirely.

To address these limitations, we combine three complementary layers of analysis in CodeBuster. First, Term Frequency-Inverse Document Frequency (TF-IDF) cosine similarity captures lexical overlap and flags direct or lightly modified copying. Second, a fine tuned GraphCodeBERT model produces semantic embeddings that encode both source code tokens and data flow graphs, enabling detection of functionally equivalent code that has been structurally refactored. Third, an output based comparison layer executes both submissions against a shared test suite and measures the proportion of identical outputs, catching cases where even semantic analysis may be

insufficient. These three signals are combined into a feature vector fed through a Multilayer Perceptron (MLP) trained on a labeled dataset of original and obfuscated C++ code pairs to produce a final plagiarism probability score.

### C. Problem Definition

CodeBuster faces several challenges that must be addressed to ensure accurate and reliable plagiarism detection. Ensuring data quality and diversity is critical, as the system must perform consistently across varying coding styles, skill levels, and obfuscation techniques. Maintaining model performance across these conditions is difficult, as students may employ combinations of variable renaming, logic reordering, and structural refactoring to disguise plagiarized work. We also need to ensure that the three layers of analysis are well calibrated so that no single signal dominates the final classification.

Designing a reliable pipeline for CodeBuster involves constructing a labeled dataset of original and obfuscated C++ code pairs, which adds significant complexity to the development process. Addressing these challenges while delivering a system that is both accurate and practical is central to the success of CodeBuster.

## II. Methodology

Our system employs a multimodal pipeline for detecting plagiarism and similarity between C++ submissions even after extensive refactoring. We compute three distinct similarity measures, token, semantic, and output similarity, and then classify the combined signal using an MLP.

### A. Token Similarity via TF–IDF and Cosine Similarity

Token similarity captures lexical overlap: shared keywords, operators, and identifiers. Each submission is transformed into a vector using TF–IDF.

Term frequency–inverse document frequency is defined as

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t), \tag{1}$$

where $\text{TF}(t, d)$ counts occurrences of token $t$ in document $d$, and $\text{IDF}(t)$ down weights common tokens (e.g., `int`, `return`) while emphasizing distinctive ones.

We compute cosine similarity between the TF–IDF vectors $\mathbf{A}$ and $\mathbf{B}$:

$$\cos(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|}, \tag{2}$$

### B. Semantic Similarity via GraphCodeBERT

To test whether program logic is preserved under refactoring, we compute semantic similarity using a fine tuned GraphCodeBERT. Unlike CodeBERT, which is primarily token/sequence based, GraphCodeBERT integrates data flow information to better represent procedural code. We first decompose each file into function level units, embed those units with GraphCodeBERT, and then aggregate chunk level similarities into a file level score. Figure 1 illustrates the workflow for computing function level semantic similarity using GraphCodeBERT.

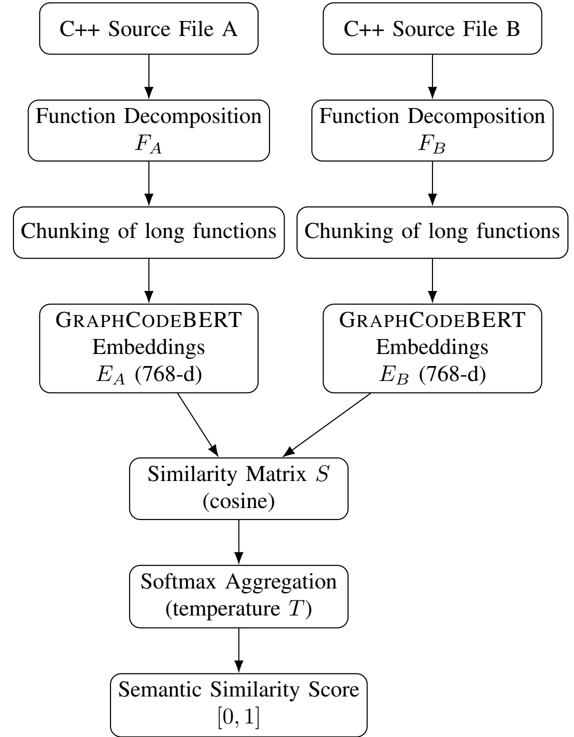


Fig. 1. Workflow for function level semantic similarity using GraphCodeBERT.

*1) Function Level Decomposition:* Source files often exceed the maximum input length of transformer models and contain multiple independent semantic units. To address this, each C++ file is parsed into individual functions using the `tree-sitter-languages` module, which provides an abstract syntax tree (AST) representation of the program. Let a file $F$ be decomposed into a set of functions:

$$F = \{f_1, f_2, \ldots, f_n\}. \tag{3}$$

Each function is then tokenized. Functions whose token length exceeds a fixed threshold (400 tokens in our implementation) are further split into smaller overlapping chunks. This ensures compatibility with the model input size while preserving local semantic coherence. This results in a collection of chunks per file, where each chunk represents a semantically meaningful unit of code.

*2) Chunk Embedding with GraphCodeBERT:* Each chunk is encoded using GraphCodeBERT, a transformer based model pre-trained on large scale source code corpora with structural information derived from program graphs. We fine tuned this model on our labeled plagiarism dataset to adapt it to the specific characteristics of C++ code similarity. Given a chunk $c_i$, GraphCodeBERT produces a dense embedding:

$$\mathbf{e}_i \in \mathbb{R}^{768}. \tag{4}$$

For a file with $n$ chunks, we obtain a set of embeddings:

$$E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}. \tag{5}$$

These embeddings capture semantic properties such as control flow, data dependencies, and functional intent, rather than relying solely on surface level token similarity.

*3) Similarity Matrix Computation:* For a pair of files $A$ and $B$ with chunk embeddings $E^A$ and $E^B$, we compute a pairwise cosine similarity between all chunk pairs:

$$S_{ij} = \frac{\mathbf{e}_i^A \cdot \mathbf{e}_j^B}{\|\mathbf{e}_i^A\| \, \|\mathbf{e}_j^B\|}. \tag{6}$$

This produces a similarity matrix:

$$S \in \mathbb{R}^{|A| \times |B|}. \tag{7}$$

Each entry $S_{ij}$ measures the semantic similarity between chunk $i$ from file $A$ and chunk $j$ from file $B$. The matrix captures fine grained relationships between all functional components of both programs.

*4) Softmax Weighted Aggregation:* A naive average of all values in $S$ would heavily dilute strong plagiarism signals, especially when two files share only a small number of similar functions. To address this, we adopt a softmax weighted aggregation strategy. First, we compute normalized weights:

$$w_{ij} = \frac{\exp(S_{ij}/T)}{\sum_{k,\ell} \exp(S_{k\ell}/T)}, \tag{8}$$

where $T$ is a temperature hyperparameter controlling the sharpness of the distribution. The final similarity score is computed as:

$$\text{Score} = \sum_{i,j} w_{ij} S_{ij}. \tag{9}$$

Smaller values of $T$ emphasize the highest similarities, allowing strong matches to dominate the final score, while larger values distribute weight more evenly. This produces a single file level similarity score in the range $[0, 1]$.

*5) Multiple Instance Learning (MIL) Formulation:* Each file pair is treated as a bag of chunk pairs, rather than a single instance. Although similarity labels (0 or 1) are available only at the file level, the model implicitly learns which chunk pairs contribute most to plagiarism. Each code file has multiple instances (chunks), and each file pair has one label. This setting corresponds to a Multiple Instance Learning (MIL) problem, which is well suited for scenarios where only coarse grained labels are available but fine grained patterns exist. MIL allows the model to capture partial plagiarism, where only a subset of functions may be copied.

*6) Fine Tuning Strategy:* We fine tune GRAPHCODEBERT using a dataset of 1500 code pairs annotated with binary similarity labels. For each pair: extract and chunk functions, encode chunks, compute the similarity matrix, aggregate similarities using softmax weighted aggregation, and compute the loss. We use mean squared error (MSE) between predicted similarity score $\hat{y}$ and ground truth label $y$:

$$\mathcal{L} = (y - \hat{y})^2. \tag{10}$$

Optimization is performed using the Adam optimizer with the learning rate selected on a validation set of 300 code pairs. Training is conducted for multiple epochs, and the model with the lowest validation loss is selected. Fine tuning adapts the pre-trained representations to the specific characteristics of C++ plagiarism patterns. Figure 2 summarizes the fine tuning workflow for adapting GraphCodeBERT to code plagiarism detection.

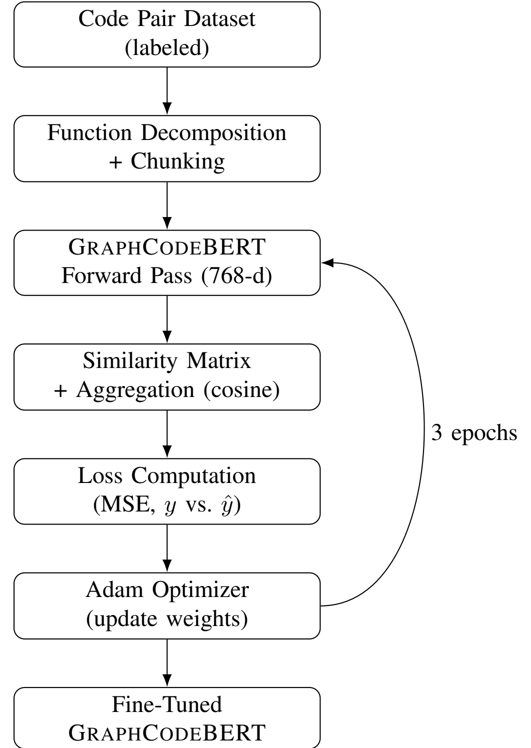


Fig. 2. Fine-tuning workflow for adapting GRAPHCODEBERT to code plagiarism detection.

*7) Inference Procedure:* Given a new pair of files, we:

1. Parse each file into functions.
2. Chunk long functions into overlapping segments.
3. Encode chunks using the fine-tuned GRAPHCODEBERT.
4. Compute the chunk level similarity matrix.
5. Apply softmax weighted aggregation.
6. Output a similarity score in $[0, 1]$.

*8) Advantages of the Proposed Approach:* The proposed method measures semantic similarity rather than surface level token overlap, enabling detection of logically equivalent code despite syntactic differences. Function level decomposition allows the framework to scale to long source files, while the MIL formulation supports partial plagiarism detection when only subsets of functions are shared. GRAPHCODEBERT embeddings provide robustness to code obfuscation techniques such as renaming and reformatting, and the softmax weighted aggregation emphasizes highly similar function pairs so that strong plagiarism signals are not diluted by unrelated code.

### *C. Output Similarity via Program Execution*

Some students rewrite the code but execute it in the same manner. In other words, the two programs produce the same

output, yet are different upon visual inspection. This, however, is a challenging aspect, as two programs may seek to solve the same issue whilst not being directly plagiarized from one another. For this reason, output similarity serves as a complementary signal that strengthens detection confidence rather than acting as definitive proof of plagiarism.

*1) Sand boxed environment:* The code pairs are handled in a sandboxed environment using a temporary directory. Ensuring clean up when the program is done running. The C++ code pairs are first written into text files in the environment. Allowing each pair is to be compiled using the g++ command. An output similarity score of [0, 1] is returned if one file fails to compile while the other compiles, and both are unable to compile, respectively. On the successful compilation of both files, both programs are executed on a fixed set of deterministic test cases. Each test case contains a set of inputs fed to the program through stdin.

$$T = \{t_1, t_2, ..., t_n\} \tag{11}$$

Each program is force terminated after a certain period of time (0.5 seconds in this case), to catch infinite loops and hanging programs, due to expecting more inputs. For a file run with n inputs, we obtain a list of outputs, where one is a tuple: (raw, decoded) containing the raw and decoded output given tn :

$$O = [o_1, o_2, ..., o_n] \tag{12}$$

*2) Output Comparison:* For a pair of files A and B, with outputs $O^A$ and $O^B$, we perform a pairwise output similarity, comparing $o_n^A$ with $o_n^B$. First a check on the raw outputs of the pair is performed:

$$R = \sum_{0}^{n} o_i^A.raw == o_i^B.raw \tag{13}$$

If the raw outputs do not match, we proceed to compare the decoded outputs. We compare the similarity of the data structures of the outputs, breaking them down into three fundamental types (floats, strings, and none types). To compare floats we use a normalized distance equation to return a score between [0, 1]:

$$D = 1 - \left(\frac{distance}{abs(max(a, b))}\right) \tag{14}$$

To compare strings we use the Levenshtein algorithm to calculate the minimum number of single characters to transform one string into another, with an output between [0, 1]. To compare none types, we check if one output is a nonetype, if the other is an empty string or 0, they are given a high score (0.9). In the case both are none types return a score of 1.0. If an output is a nonetype and the other output is none of the previous cases, return a score of 0. This leaves us with a list of output similarity scores, where sn is the output similarity score of the pair $o_n^A$ , $o_n^B$ :

$$S = [s_1, s_2, ..., s_n] \tag{15}$$

The final score is calculated by taking the average of the list:

$$Score = \frac{\sum_0^n s_i}{n} \tag{16}$$

Returning a single final output similarity score in the range of [0, 1].

*3) Inference Procedure:*

1. Compile and execute both files.
2. Compute output similarity for each input.
3. Compute the final output similarity score.
4. Output similarity score in range $[0, 1]$.

### *D. Feature Vector and MLP Classifier*

For each pair of submissions, we form a three dimensional feature vector

$$\mathbf{x} = [\text{token_sim}, \ \text{semantic_sim}, \ \text{output_sim}], \tag{17}$$

and feed it into a multilayer perceptron (MLP) classifier with two hidden layers with 16 neurons each (ReLU activations) and a binary output (0 = not plagiarized, 1 = plagiarized). This choice is motivated by the low dimensional input, potential non linear interactions between similarity measures, and fast training. Training uses Binary Cross-Entropy with Logits loss and the Adam optimizer (learning rate = 0.001). We employ early stopping with patience of 10 epochs, saving the model with lowest validation loss. The dataset is split 80/20 for training and validation. Code pairs with predicted probability greater than or equal to 0.5 are classified as plagiarized.

## III. Results

We evaluated CodeBuster on a held out test set of 3,159 C++ code pairs. Table I presents the evaluation metrics on the held-out test set.

TABLE I
Evaluation metrics on held out test set

| Metric | Score |
|---|---|
| Accuracy | 79.8% |
| Precision (class 1) | 76.2% |
| Recall (class 1) | 83.5% |
| F1 Score | 0.798 |

Table II presents the confusion matrix for the evaluation set.

TABLE II
Confusion Matrix

| | Predicted 0 | Predicted 1 |
|---|---|---|
| **Actual 0** | 1267 | 392 |
| **Actual 1** | 247 | 1253 |

The model successfully identifies 83.5% of plagiarized submissions while maintaining 76.2% precision, meaning most flagged submissions are genuine cases of plagiarism.

## IV. Conclusion

We presented CodeBuster, a plagiarism detection system that combines token level, semantic, and output analysis to identify similarity between two C++ code snippets. Our results demonstrate that this multilayered approach achieves 79.8% accuracy.

Key contributions of this work include: token-based similarity using TF-IDF and cosine similarity, function-level decomposition with GraphCodeBERT for semantic similarity, output-based behavioral comparison to catch structural refactoring, and an MLP ensemble that combines the three similarity signals into a unified feature vector, learning non linear interactions to produce a final plagiarism probability.

Future work includes expanding the test suite for output similarity, training on larger and more diverse datasets, adversarial testing against AI generated obfuscation, threshold calibration for different deployment contexts, and implementing stronger isolation mechanisms such as Docker containerization for safer code execution.

## REFERENCES


[1] D. Cotton, P. Cotton, and J. Shipway, "Chatting and cheating: Ensuring academic integrity in the era of ChatGPT," *Innovations in Education and Teaching International*, vol. 61, no. 2, pp. 228–239, 2023. doi: 10.1080/14703297.2023.2190148

[2] B. Chen, C. M. Lewis, M. West, and C. Zilles, "Plagiarism in the Age of Generative AI: Cheating Method Change and Learning Loss in an Intro to CS Course," in *Proc. 11th ACM Conf. Learning @ Scale (L@S '24)*, Atlanta, GA, pp. 75–85, 2024. doi: 10.1145/3657604.3662046

[3] A. Aiken, "MOSS: A System for Detecting Software Similarity," Stanford University, 2022. [Online]. Available: https://theory.stanford.edu/˜aiken/moss/

[4] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[5] Z. Hicks et al., "The Failure of Plagiarism Detection in Competitive Programming," arXiv preprint arXiv:2505.08244, 2025.

[6] D. Guo, S. Ren, S. Lu, L. Feng, D. Tang, N. Duan, A. Abdi, V. S. Sheng, and M. Zhou. *GraphCodeBERT: Pre-trained Code Representation with Data Flow*. 2021. arXiv:2009.08366.

[7] F. Feng, J. Guo, B. Tang, S. Li, and Z. Liu. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. Findings of EMNLP, 2020.