LC3: Long Cross-Language Code Clone Detection Enhanced by Opcode Sequences and Affinity Aggregation

Xilin Lan, Huan Zhang, Yang Yang, Chengwu Xue, Li Kuang*

School of Computer Science and Engineering, Central South University, Changsha, China
{xilinlan, z_huan, yang978, 8209240610, kuangli}@csu.edu.cn

Abstract

Cross-language code clone detection, which identifies functionally similar code across programming languages, is critical for ensuring synchronized evolution and reducing maintenance costs in multi-platform software development. While zero-shot approaches have emerged as a practical solution to data scarcity, state-of-the-art methods still face two major limitations: an insufficiency in learning language-agnostic representations and information loss during the processing of long code. To address these challenges, we propose LC3, a novel framework for robust zero-shot cross-language code clone detection. To overcome the language-agnostic representation insufficiency, LC3 fuses source code with its underlying opcode sequences, leveraging a bimodal architecture and adversarial training to learn a language-agnostic representation. To resolve long-code information loss, LC3 introduces a semantic affinity aggregation strategy. This strategy synthesizes a robust clone score from a complete pairwise similarity matrix computed between segmented code blocks, overcoming the limitations of both simple truncation and aggregation. Extensive experiments show that LC3 significantly outperforms state-of-the-art zero-shot baselines, especially in challenging long-code scenarios.

Introduction

In the development and maintenance of modern multi-platform software, cross-language code clones, which are code snippets with identical functionality implemented in different programming languages, are increasingly common. For example, large-scale applications targeting diverse user groups often require simultaneous development and maintenance of Java versions for Android, Objective-C versions for iOS, and C/C# versions for Windows (Cheng et al. 2016; Nafi et al. 2019). If these functionally equivalent but linguistically distinct codebases fail to evolve synchronously, subtle defects may easily arise, significantly increasing software maintenance costs. Therefore, automated techniques for cross-language code clone detection are essential to ensure software quality and reduce development costs.

In practical software engineering, cross-language code clone detection is commonly approached as a large-scale retrieval problem (Li et al. 2023b): given a code snippet in one language as the query, the task is to retrieve semantically equivalent snippets from candidate codebases in another language.

Early approaches primarily relied on lexical and syntactic features extracted from source code (Cheng et al. 2017; Vislavski et al. 2018; Nafi et al. 2019). With the advancement of Pre-trained Language Models (PLMs), researchers have fine-tuned these models on parallel corpora of cross-language clone pairs (Tao et al. 2022; Fang et al. 2023). However, it is difficult to obtain large-scale, manually annotated parallel data. Recently, researchers have focused on the zero-shot setting by only training on more readily available monolingual data. Specifically, state-of-the-art zero-shot method ZC3 (Li et al. 2023b) pioneers this direction by employing a joint training strategy with adversarial alignment and cycle consistency loss, eliminating the need for parallel clone data. Despite its effectiveness, this reliance on source code introduces two inherent limitations.

Limitation 1: Insufficiency in Language-Agnostic Representation The core challenge in zero-shot cross-language code clone detection lies in learning a language-agnostic functional representation from monolingual data that generalizes to unseen language pairs. Existing methods that rely on token-based representations of source code (Tao et al. 2022; Li et al. 2023b) are inherently limited, as they tend to capture superficial lexical and syntactic patterns (Liu et al. 2023; Zhang et al. 2024) rather than deep functional equivalence. To address this, researchers have explored Intermediate Representations (IRs). However, while approaches leveraging ASTs (Mou et al. 2016; Xu et al. 2024) or fusing source code with assembly (Li et al. 2024) are effective in monolingual contexts, their transferability to the cross-language challenge is hindered by the heterogeneity of these IRs across different language ecosystems. Therefore, the fundamental bottleneck remains the absence of a framework capable of aligning these heterogeneous operational representations while effectively integrating them with high-level source code semantics.

Limitation 2: Information Loss in Long Code Processing PLMs typically limit the input length to 512 tokens. However, Hu et al. observed that a substantial portion of real-world functions and methods in CodeSearchNet exceed 512 tokens (Hu et al. 2024). PLM-based code clone detection

*Corresponding author.

models often resort to simple truncation (Tao et al. 2022; Li et al. 2023b), which risks the loss of essential logic not located at the beginning of a function. An alternative approach involves aggregating block representations through pooling. However, this pooling-based strategy is suboptimal for clone detection. Simple aggregation methods, such as pooling, merge all code block representations into a single vector, uniformly weighting each part of the function. This approach is problematic as a critical cloned segment may constitute only a small fraction of the entire function. Thus, averaging the representations inevitably dilutes the distinct semantic characteristics of this potentially small cloned segment within the larger, non-cloned context, ultimately impairing detection accuracy.

To address these challenges, we propose LC3, a novel framework for robust zero-shot code clone detection composed of two core components: Bimodal Functional Representation Learning and Semantic Affinity Aggregation (SAA). To address the insufficiency in language-agnostic representation, LC3 fuses source code with its corresponding opcode sequences (Jeon and Moon 2020). This fusion is achieved through a dedicated bimodal encoder that first employs cross-attention to integrate the low-level opcode sequence with the high-level source code representation, followed by an adaptive gating mechanism to dynamically balance their contributions. The resulting representation is then optimized via adversarial alignment to ensure language-agnostic properties, effectively capturing deep functional logic while suppressing superficial linguistic patterns. To tackle the limitation of long code processing, the framework introduces a semantic affinity aggregation strategy. This strategy first constructs a complete block-to-block similarity matrix to circumvent truncation-induced information loss. It then derives the final detection score by identifying the maximum similarity value and its contextual support through matrix analysis. This approach overcomes the critical signal dilution problem inherent in simple aggregation methods, thus providing a more reliable judgment.

Our main contributions can be summarized as follows:

- We propose a novel bimodal representation learning framework that fuses source code with opcode sequences. This approach establishes a rich semantic representation, thereby enhancing the effectiveness of adversarial training in achieving zero-shot generalization.
- We introduce a semantic affinity aggregation strategy to address the challenges of long code processing. This strategy derives a robust similarity score from a complete matrix of local similarities, mitigating information loss from common truncation methods.
- We conduct comprehensive experiments demonstrating that LC3 significantly outperforms state-of-the-art zero-shot methods, especially in challenging long-code scenarios.

Related Work

Code Clone Detection

Code clone detection identifies functionally similar code snippets. We focus on the challenging Type-IV clones, characterized by functional equivalence despite differing implementations (Roy and Cordy 2007).

Significant progress exists for Type-IV detection in single-language contexts. These methods often rely on language-specific representations, such as Abstract Syntax Trees (ASTs) (Mou et al. 2016; Wei and Li 2017; Yu et al. 2019; Wang et al. 2020; Xu et al. 2024), assembly code analysis (Li et al. 2024), or large pre-trained models like CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2021). Other research, like AdaCCD (Du et al. 2024), focuses on adapting detection capabilities from resource-rich to resource-poor languages within a single-language setting. However, the deep reliance of these methods on language-specific tools or data structures limits their direct applicability to cross-language detection.

Consequently, cross-language clone detection has gained prominence, with the core challenge being to bridge semantic gaps across languages. Early rule-based methods like CLCMiner (Cheng et al. 2017), LICCA (Vislavski et al. 2018), and CLCDSA (Nafi et al. 2019) lacked robustness for complex clones. Recent deep learning approaches have shown more promise. Methods including C4 (Tao et al. 2022), TCCCD (Fang et al. 2023), FSD-CLCD (Zhang et al. 2024), and CCT-code (Sorokin et al. 2025) leverage contrastive learning but require large-scale annotated parallel data. To address this data dependency, ZC3 (Li et al. 2023b) pioneers a zero-shot approach by employing a joint training strategy with adversarial alignment and cycle consistency loss, eliminating the need for parallel clone data.

Relying on source code tokens alone is insufficient, and common IRs also pose challenges for cross-language alignment. Representations like ASTs are structurally heterogeneous across languages, while low-level ones like assembly are hardware-specific. Given these challenges, our approach employs opcode sequences. While opcodes themselves vary, their linear sequence format provides a shared structural representation of the underlying operational logic. Fusing source code with these operational semantics is intended to enrich the overall representation, complementing the high-level information from tokens with low-level execution details.

Long Text Sequence Processing

The limited input length of most PLMs, typically capped at 512 tokens, presents a significant challenge for processing real-world code, often leading to information loss. Two primary strategies have been developed to address this limitation.

The first modifies model architectures to extend context windows, using sparse attention (Child et al. 2019; Beltagy, Peters, and Cohan 2020; Kitaev, Kaiser, and Levskaya 2020; Ainslie et al. 2020), recurrence (Dai et al. 2019), hierarchical modeling (Lv et al. 2015; Gao and Callan 2022), or compressed attention (Chen, Ye, and Zhang 2019; Guo et al. 2019). While effective, these methods require costly retraining. The second, more common strategy retains the original PLM and segments long inputs into smaller blocks. These blocks are encoded independently, and their resulting outputs are aggregated. Aggregation can occur at the

score level, by computing and then combining similarity scores between block pairs, or at the representation level, by merging block embeddings into a global vector using methods like pooling or attention (Hu et al. 2024). This aggregation approach has been successfully applied in various tasks, including question answering through paragraph score normalization (Wang et al. 2019), document ranking via hierarchical sentence aggregation (Yang et al. 2020), and paragraph-level search using fusion techniques like max pooling and attention (Li et al. 2023a).

While segmentation-based techniques are a common approach for long code, they are limited in the fine-grained task of code clone detection. Traditional aggregation methods like pooling or averaging tend to dilute critical local clone signals. To address this, we propose a score-level strategy that focuses on the most significant similarities instead of simply averaging them. Our approach constructs a complete similarity matrix to prevent information loss, then actively locates the strongest similarity peak and its contextual support for a more robust judgment of long code clones.

Methods

Our LC3 framework is designed with a modular architecture to systematically address the two primary limitations in zero-shot cross-language code clone detection. As illustrated in Figure 1, the framework comprises two core components: Bimodal Functional Representation Learning, which learns language-agnostic representations by fusing source code with its corresponding opcode sequences, and the Semantic Affinity Aggregation Strategy, which robustly aggregates similarity information to handle arbitrarily long code snippets during inference. We refer to this core bimodal encoder component as LC3-Dual.

Bimodal Functional Representation Learning

This component aims to learn a representation that captures the language-agnostic functionality of a function. To this end, we augment source-code representations with opcode sequences. By representing the functionality of the program as uniform atomic operations, opcodes expose its fundamental computational steps, offering a view that is more robust to syntactic variations than solely using source code. The extraction of these opcode sequences is a static process that utilizes standard toolchains to transform high-level source code into a lower-level representation (e.g., Python's opcodes, Java's bytecode, or C/C++'s assembly instructions). While the specific opcode sets differ across platforms, this abstraction consistently minimizes high-level syntactic variations. Further details on the extraction process for each language are provided in Appendix A.

Bimodal Encoder Architecture Our encoder contains two parallel feature extraction branches. The first branch uses a pre-trained model to extract a contextual representation, H_{cls}, from the source code. The second branch employs a bidirectional long short-term memory network (BiLSTM) (Hochreiter and Schmidhuber 1997) to capture the temporal information from the opcode sequence, yielding H_{op}. These two heterogeneous representations are then fused via two key mechanisms. These fusion mechanisms are optimized through the subsequent contrastive learning objective. This process also drives the entire bimodal encoder to generate effective functional representations.

Cross-Attention Fusion To achieve a meaningful fusion, we employ a cross-attention mechanism, a standard technique for integrating information from two different sequences. The core idea is to generate a summary of the opcode sequences H_{op} that is specifically focused on the aspects most relevant to the overall function of the source code H_{cls}.

In this mechanism, we use the global representation of the source code H_{cls} as the Query. The sequence of representations from the opcode sequences H_{op} serves as both the Key and Value. The Multi-Head Attention (MHA) module then calculates a weighted average of the Value vectors, with the weights being determined by the similarity between the source code's Query and each Key vector in the opcode sequences as specified in Equation 1:

$$H_{\text{attn}} = \text{MHA}(Q = H_{\text{cls}}, K = H_{\text{op}}, V = H_{\text{op}}) \quad (1)$$

where H_{attn} is the opcode sequence embedding vector that focuses on the key execution logic of the source code, which serves as a complement to the source code representation at the execution level.

Adaptive Gating Mechanism The relative importance of the high-level source code abstraction and the low-level execution details may vary between code snippets. Therefore, we introduce an adaptive gating mechanism to dynamically balance the contributions of the source code representation H_{cls} and the attention-weighted execution representation H_{attn}. This gating unit learns a weight vector g based on the two spliced representations, defined as in Equation 2:

$$g = \sigma\left(W_g\left[H_{cls}, H_{attn}\right] + b_g\right) \quad (2)$$

where g is a gating weight between 0 and 1, σ is a Sigmoid function, W_g and b_g are trainable parameters. The final fused feature H_{fused} is a weighted sum of the two types of information according to the gating weight g, as defined in Equation 3:

$$H_{\text{fused}} = g \odot H_{\text{cls}} + (1 - g) \odot H_{\text{attn}} \quad (3)$$

where $\odot$ represents the element-by-element product. This approach ensures that the two modal information can be combined in an optimal ratio.

The fused representation H_{fused} is then optimized via a composite training objective, designed to learn a representation space that is both highly discriminative and language-agnostic.

Contrastive Learning Objective We use contrastive learning to train our model. The goal is to create a representation space where similar code snippets are close and dissimilar ones are distant. This is achieved by minimizing the InfoNCE loss function, using the fused feature H_{fused} as input. In Equation 4, $sim(q, c)$ denotes the cosine similarity. The loss for a single positive pair is:

$$\mathcal{L}_{cl} = -\sum_{i=1}^{N} \log \frac{\exp\left(sim\left(q_i, c_i^{+}\right)/\tau\right)}{\sum_{j=1}^{M} \exp\left(sim\left(q_i, c_j\right)/\tau\right)} \quad (4)$$

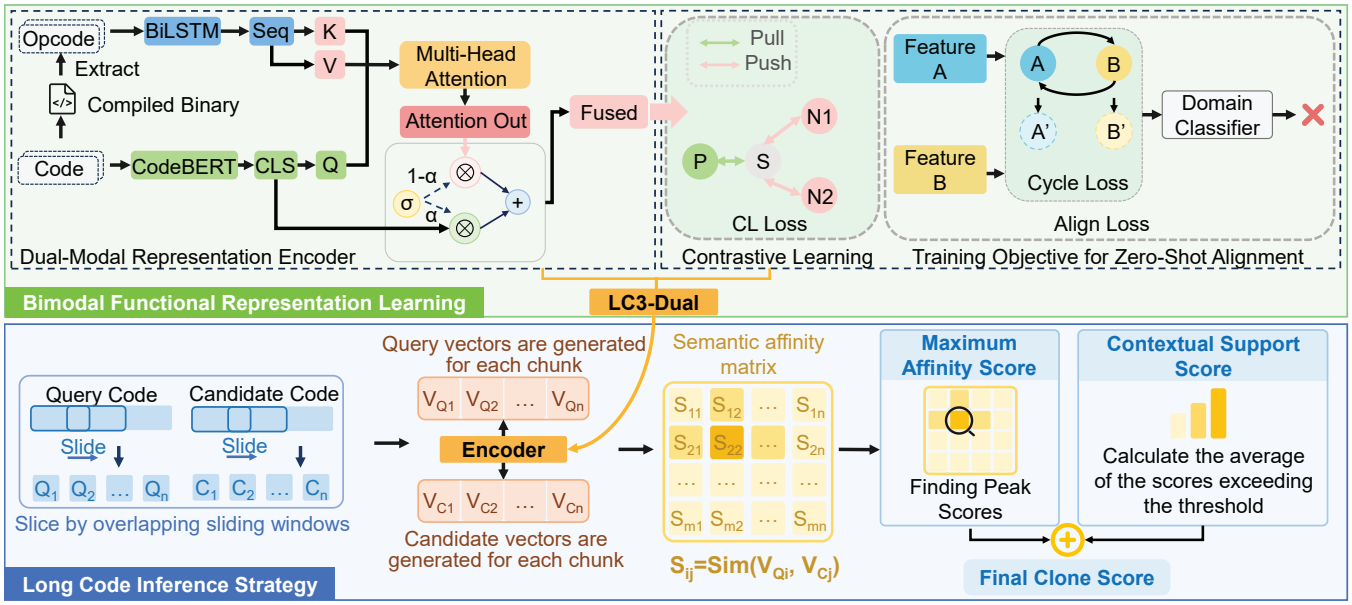

Figure 1: The Overall Framework of the LC3 Method.

This equation calculates the loss for an anchor sample q_i and its positive sample c_i^+. The denominator normalizes this score against the scores of M negative samples. The temperature τ is a hyperparameter that scales the scores to control the model's discriminative sensitivity.

Training Objective for Zero-Shot Alignment To enable the model to generalize from monolingual language clone pairs to the cross-lingual setting, we adopt the joint training strategy from prior work (Li et al. 2023b), incorporating two auxiliary loss functions to remove language-specific features.

Adversarial Alignment We employ an adversarial alignment mechanism to encourage the encoder G_f to produce language-agnostic representations. A domain discriminator G_d is trained to identify the source language from the fused representation H_{fused}, while the encoder is trained to generate features that confuse the discriminator. This process forces the representations to be free of language-specific artifacts. The objective is optimized using the loss shown in Equation 5, where $\mathcal{L}_{CE}$ denotes the cross-entropy loss, and d is the ground-truth domain label for the input x:

$$\mathcal{L}_{\text{align}}(\theta_f, \theta_d) = \mathrm{E}_{x\sim(X_S\cup X_T)}\Big[\mathcal{L}_{\text{CE}}\big(G_d(G_f(x;\theta_f));\theta_d\big), d\Big] \quad (5)$$

Cycle Consistency To further regularize the alignment, we add a cycle consistency loss. This loss ensures that the geometric structure of the embedding spaces for different languages remains consistent by penalizing reconstruction errors when a representation is mapped from a source language to a target and back again. The reconstruction error is measured using the L1 distance, as defined in Equation 6:

$$\mathcal{L}_{\text{cycle}} = \mathrm{E}_{x_s\sim X_S}\left[\|G_{T\to S}\left(G_{S\to T}\left(x_s\right)\right) - x_s\|_1\right] + \mathrm{E}_{x_t\sim X_T}\left[\|G_{S\to T}\left(G_{T\to S}\left(x_t\right)\right) - x_t\|_1\right] \quad (6)$$

The mapping functions $(G_{S\to T}, G_{T\to S})$ are implemented as lightweight Adapter modules, which consist of a bottleneck architecture (down-projection, ReLU, up-projection) with a residual connection.

Total Loss The final training objective is a weighted combination of the primary contrastive loss and the two auxiliary alignment losses, balanced by hyperparameters α and β, as shown in Equation 7:

$$\mathcal{L}\text{total} = \mathcal{L}cl + \alpha\mathcal{L}\text{align} + \beta\mathcal{L}\text{cycle} \quad (7)$$

Semantic Affinity Aggregation Strategy

To handle long code snippets that exceed the input limits of pre-trained models without information loss, we introduce the Semantic Affinity Aggregation (SAA) strategy. This approach avoids simple truncation or averaging by systematically analyzing the local similarities between two pieces of code.

The process begins by segmenting the long source code of a query Code Q and a candidate Code C into smaller, manageable chunks. We use a sliding window of 512 tokens with a specified stride to create a series of overlapping blocks for each snippet: $\{Q_1, Q_2, ..., Q_n\}$ and $\{C_1, C_2, ..., C_m\}$. Each of these blocks is then passed through our pre-trained LC3-Dual encoder to generate a corresponding sequence of embedding vectors: $\{V_{Q_1}, V_{Q_2}, ..., V_{Q_n}\}$ and $\{V_{C_1}, V_{C_2}, ..., V_{C_m}\}$. With these block-level embeddings, we compute the cosine similarity between every possible pair of blocks (Q_i, C_j) to construct an $m \times n$ affinity matrix M. This matrix provides a complete map of all local similarities between the two code snippets. From this matrix, we derive the final clone score using a two-part algorithm:

Maximum Affinity Score (MAS): We assume that the most reliable clone evidence is embedded in the strongest local similarity between two long code snippets. Therefore,

we directly extract the maximum value in the affinity matrix M to identify the most significant local similarity, as shown in the Equation 8, which is used to capture the most convincing semantic similarity score in the potential clone relationship.

$$S_{MAS} = \max(\mathrm{M}) \tag{8}$$

Contextual Support Score (CSS): To improve the robustness of the MAS, we introduce the Contextual Support Score (CSS). CSS provides contextual evidence by measuring the local support for the MAS. It is calculated as the average similarity of the block pairs immediately adjacent to the MAS's cell in the affinity matrix, considering only those pairs whose similarity exceeds a predefined confidence threshold θ. This approach checks if the surrounding block similarities also surpass the confidence threshold, making the final clone measure more robust. Its definition is as follows in Equation 9 :

$$S_{CSS} = \frac{1}{|N'_\theta(p_{max})|} \sum_{p \in N'_\theta(p_{max})} M_p \tag{9}$$

where p_{max} is the position of the MAS, and $N'_\theta(p_{max})$ is the set of its neighboring positions whose similarity scores exceed the threshold θ.

The final clone score S_{clone} is defined as the weighted sum of these two components, as shown in Equation 10:

$$S_{clone} = \begin{cases} \lambda \cdot S_{MAS} + (1-\lambda) \cdot S_{CSS} & \text{if } S_{MAS} > \theta \\ 0 & \text{if } S_{MAS} \leq \theta \end{cases} \tag{10}$$

First, we need to ensure that S_{MAS} itself is meaningful. It is only calculated when it exceeds the threshold θ. If S_{MAS} in the matrix fails to exceed the threshold θ, the code pair is considered non-clone and its score is 0. This prevents code pairs with low global similarity from obtaining a non-negligible score. The final score is the weighted sum of S_{MAS} and S_{CSS}, balanced by the hyperparameter λ. This composite score represents a holistic measure of the clone relationship, which combines the maximum similarity value with the overall quality of the high-similarity region.

Experiment

Experimental Setup

Datasets Our experimental evaluation relies on two datasets, both derived from the comprehensive CodeNet dataset (Puri et al. 2021). To ensure a fair comparison with prior work, we use the same training data as ZC3 (Li et al. 2023b), the Java and Python monolingual clone pairs from the $\mathrm{CSN_{CC}}$ dataset (Guo et al. 2022). For evaluation, we use the Java and Python pairs within the $\mathrm{CSN_{CC}}$ test set. A small number of code snippets that failed to compile were filtered from both sets. To specifically evaluate performance on long code, a primary challenge we address, we also constructed a new dataset, CodeNet-VL (Variable-Length). This dataset, curated from CodeNet, contains code snippets from language pairs such as Java, Python, C, and C#, with their lengths divided into four intervals. The construction details of CodeNet-VL are provided in Appendix B. Detailed statistics for these datasets are provided in Table 1.

Dataset	Range	Python	Java	C	C#
$\mathrm{CSN_{CC}}$	Total	3,190	2,885	–	–
CodeNet-VL	(0, 256]	2,035	1,992	1,978	1,315
	(256, 512]	1,874	2,081	1,507	1,370
	(512, 1024]	933	1,115	543	708
	(1024, ∞)	263	422	171	317
	Total	5,105	5,610	4,199	3,710

Table 1: Number of different programming languages in the datasets used for the experiments.

Evaluation Metrics We use Mean Average Precision (MAP) (Musgrave, Belongie, and Lim 2020) as the primary evaluation metric (Ye et al. 2020). The reported MAP score is the arithmetic mean of the scores from all retrieval directions. For $\mathrm{CSN_{CC}}$, we report the MAP over bidirectional retrieval (Python↔Java). For the CodeNet-VL, we report the mean MAP for both the X→Y and Y→X retrieval directions, where X, Y∈ Python, Java, C, C#. Notably, C and C# were never seen during the training procedure.

Baseline Models Our baseline models are divided into three categories:

- Pre-trained Language Models: including CodeBERT (Feng et al. 2020), GraphCodeBERT (Guo et al. 2021), and UniXcoder (Guo et al. 2022), used to establish the performance baseline without task optimization.
- Fine-tuned Baselines: For fair comparison, we fine-tune the above models using contrastive learning on the same monolingual training data as LC3.
- SOTA Baseline: ZC3 (Li et al. 2023b) is our most direct and important baseline for comparison, as it follows the same zero-shot training paradigm as ours and represents the current state-of-the-art level.

Implementation Details Experiments were conducted on a server equipped with two Intel Xeon Gold 6248 CPUs and two Tesla V100 GPUs running CentOS7. Models were trained for 100 epochs using AdamW with a batch size of 32 and a learning rate of 5e-5. In the loss function, hyperparameters α and β were set to 1.0, and the contrastive temperature τ was 0.05. For long code, we used a sliding window of 512 tokens with a step size of 384, setting aggregation parameters λ and θ to 0.85 and 0.5. These values were determined by a parameter sensitivity analysis detailed in Appendix C.

Main Results and Analysis

Encoder Performance Comparison To verify the effectiveness of our bimodal encoder, we compare it against baselines using the standard truncation strategy to ensure a fair comparison of their core representation learning abilities.

The results in Table 2 show that LC3-Dual consistently outperforms all baselines, including the state-of-the-art ZC3,

Methods	CSN_{CC}		CodeNet-VL	
	Python→ Java	Java→ Python	(0, 256]	(256, 512]
CodeBERT	1.25	0.84	0.95	0.97
GraphCodeBERT	4.55	1.79	4.97	3.41
UniXcoder	19.46	15.99	27.96	14.58
CodeBERT-FT	58.67	57.96	69.79	67.05
GraphCodeBERT-FT	61.39	56.86	72.03	67.63
UniXcoder-FT	52.02	50.72	64.69	59.41
ZC3	63.16	62.96	74.13	70.79
LC3-Dual	**65.86(+4.28%)**	**66.06(+4.92%)**	**75.94(+2.44%)**	**75.30(+6.37%)**

Table 2: Performance comparison on standard benchmarks using the truncation strategy. The MAP for CodeNet-VL is the average across all language pairs and retrieval directions. All results are reported with 3 random seeds. All improvements over the strongest baseline, ZC3, are statistically significant (p-value < 0.05, based on paired t-tests over 3 random seeds).

Methods	(512, 1024]	(1024,∞)
ZC3 (Trunc)	72.17	46.15
ZC3 (SAA)	76.75 (+6.34%)	61.53 (+33.32%)
LC3-Dual (Trunc)	73.13	49.29
LC3-Dual (SAA)	**76.83 (+5.06%)**	**63.13 (+28.08%)**

Table 3: Performance comparison on long code scenarios. The numerical MAP is the average value of the model in each retrieval direction for all cross-language pairs of CodeNet-VL.

Methods	Python→Java	Java→Python	Avg
LC3-Dual	**65.86**	**66.06**	**65.96**
w/o L_{align}	65.59	65.55	65.57
w/o L_{cyc}	63.33	64.10	63.72
w/o Gate	62.35	62.94	62.65
w/o opcode	60.74	60.02	60.38
w/o Attention	59.92	56.61	58.27
w/ only opcode	3.09	2.55	2.82

Table 4: Ablation study of the LC3-Dual encoder on the CSN_{CC} dataset. "w/o" denotes removing a component. All values are percentages (%).

on both the CSN_{CC} and CodeNet-VL datasets for short code. This indicates that our bimodal architecture, which fuses source code with opcode sequences, provides a richer semantic foundation than models relying solely on source code, thus enhancing the learning of semantic representations.

Effectiveness and Generalizability of the SAA Strategy To evaluate our aggregation strategy, we compare its performance against the baseline truncation strategy on long code snippets. To demonstrate the general applicability of our strategy, the experiments were conducted on both our own LC3-Dual and ZC3, the current state-of-the-art encoder.

As shown in Table 3, the standard truncation strategy (Trunc) leads to a severe performance drop on long code snippets. In contrast, our proposed SAA strategy effectively mitigates this issue. When applied to the ZC3 encoder, SAA yields a significant relative improvement of over 30% for code exceeding 1024 tokens, suggesting its general applicability. Furthermore, when combined with our more powerful LC3-Dual encoder, the complete LC3 model achieves the best performance across all long-code intervals.

Ablation Experiment

Analysis of LC3-Dual Encoder To systematically evaluate the contribution of each component in our LC3-Dual encoder, we conduct a series of ablation experiments. The results are presented in Table 4.

The ablation study reveals each component's distinct contribution. Removing the cross-attention mechanism (w/o Attention) or the opcode sequence modality (w/o opcode) causes the most significant performance degradation, confirming that fusing these information sources is the cornerstone of the model. The near-total performance collapse in the w/ only opcode experiment underscores that the opcode sequences complement rather than replace source code. Performance also drops when removing the adaptive gate (w/o Gate), highlighting the importance of dynamic fusion. For alignment losses, removing cycle consistency (w/o L_{cyc}) induces a substantial drop, confirming its crucial role in maintaining consistent geometric structure across language representations. The subtler yet still significant impact of removing adversarial alignment (w/o L_{align}) suggests it provides fine-tuning that further refines the language-agnosticism of the learned representations.

Analysis of Aggregation Strategies We further analyze the design of our aggregation strategy by comparing different aggregation methods on long code snippets. All methods are based on the LC3-Dual encoder.

As shown in Table 5, representation-level aggregation (rep-agg) methods perform poorly, underperforming even the simple truncation baseline. This is because the representation of a small, critical cloned section is diluted by the representations of numerous irrelevant code blocks, prevent-

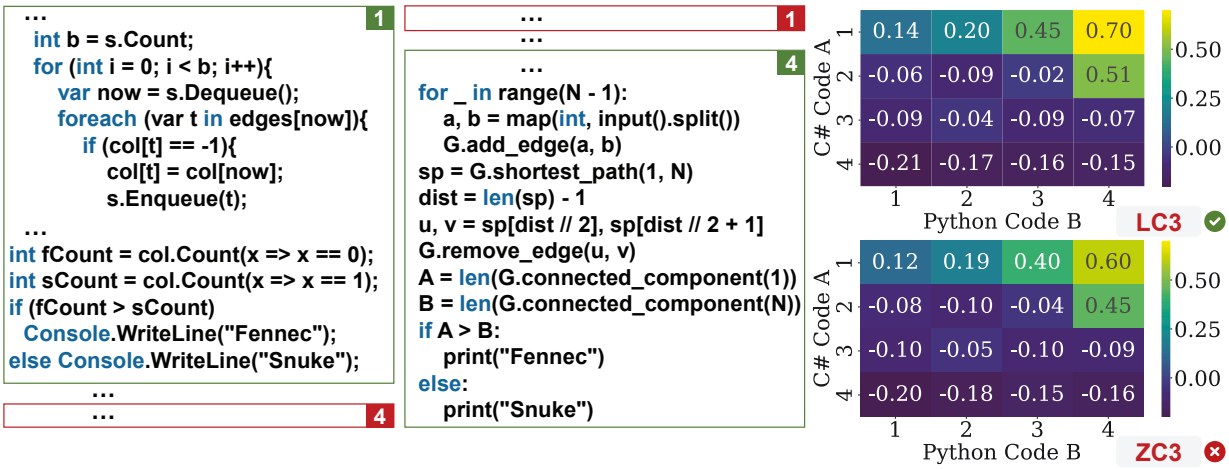

Figure 2: A long cross-language clone pair between C# and Python: C# uses BFS-based dynamic simulation, while Python uses graph-segmentation-based static analysis.

Strategy	Methods	MAP	Time
Baseline	Truncation	49.29	**24 ± 0.5**
rep-agg	Mean Pooling	46.20	145 ± 0.5
	Max Pooling	43.89	145 ± 0.5
score-agg	Mean Score	45.54	91 ± 0.5
	MAS	62.22	91 ± 0.5
	MAS + CSS	**63.13**	91 ± 0.5

Table 5: Analysis of aggregation strategies on CodeNet-VL (>1024 tokens), showing MAP (%) and average inference time (ms).

ing the final aggregated vector from reflecting the key local similarity.

Score-level aggregation (score-agg) methods yield different outcomes. The simple Mean-Score approach also suffers from the dilution problem, as a few high similarity scores are averaged down by a large number of low scores. In contrast, the MAS method, which focuses only on the maximum similarity, significantly outperforms both truncation and rep-agg, demonstrating the effectiveness of the score-aggregation paradigm. However, relying solely on MAS can be sensitive to outlier scores. Our final method, which combines MAS with the CSS, addresses this by verifying the peak similarity against the consistency of other high-scoring regions. This combination achieves the best and most stable performance, confirming the value of capturing not only the most salient similarity but also its contextual reliability.

Table 5 also presents the inference time for each strategy. While our score-agg methods lead to a higher computational cost than the Truncation baseline, we consider this a worthwhile trade-off for the substantial gain in MAP. We note that the time complexity is primarily determined by the number of blocks, and our analysis of the CodeNet-VL dataset shows that instances requiring a large number of block-to-block comparisons are infrequent. A more detailed discussion of the performance overhead is provided in Appendix D.

Case Study

To demonstrate the robustness of our framework, we analyze a C#-Python clone pair from CodeNet as shown in Figure 2. This case is very challenging because the two programs not only have different algorithmic logic, but also contain a large amount of code that is irrelevant to the core logic.

This example highlights the limitation of ZC3's truncation strategy, which only evaluates the first block from each code snippet. As shown in Figure 2, this block's similarity is too low, causing the method to overlook the clone. In contrast, our SAA strategy computes the full affinity matrix and successfully locates the true clone signal in later blocks. The heatmap reveals that both encoders were capable of identifying this strong similarity peak. This demonstrates that the baseline's failure was its flawed truncation strategy, not its encoder. SAA is therefore an essential mechanism for long code, capable of recovering signals that truncation misses.

Conclusion

In this paper, we proposed LC3, a framework that tackles two key challenges in cross-language code clone detection: learning language-agnostic representations and processing long code. Its core is a bimodal encoder that fuses source code with opcodes, leveraging adversarial and cycle-consistency training to capture deep functional semantics. For long code, LC3 introduces a semantic affinity aggregation strategy that integrates local similarities to prevent information loss from truncation. Our extensive experiments demonstrate that LC3 achieves state-of-the-art performance, with particularly significant gains in challenging long-code scenarios, thereby validating the effectiveness of our multi-faceted approach to code representation and analysis.

Acknowledgments

This work has been supported by the National Natural Science Foundation of China under grant No.62472447, Hunan Provincial Natural Science Foundation of China under grant No.2024JK2006, the Science and Technology Innovation Program of Hunan Province under grant No.2023RC1023, and the Fundamental Research Funds for the Central Universities of Central South University. This work was carried out in part using computing resources at the High Performance Computing Center of Central South University.

References

Ainslie, J.; Ontanon, S.; Alberti, C.; Cvicek, V.; Fisher, Z.; Pham, P.; Ravula, A.; Sanghai, S.; Wang, Q.; and Yang, L. 2020. ETC: Encoding Long and Structured Inputs in Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 268–284.

Beltagy, I.; Peters, M. E.; and Cohan, A. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Chen, L.; Ye, W.; and Zhang, S. 2019. Capturing source code semantics via tree-based convolution over API-enhanced AST. In *Proceedings of the 16th ACM international conference on computing frontiers*, 174–182.

Cheng, X.; Peng, Z.; Jiang, L.; Zhong, H.; Yu, H.; and Zhao, J. 2016. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 696–701.

Cheng, X.; Peng, Z.; Jiang, L.; Zhong, H.; Yu, H.; and Zhao, J. 2017. Clcminer: detecting cross-language clones without intermediates. *IEICE TRANSACTIONS on Information and Systems*, 100(2): 273–284.

Child, R.; Gray, S.; Radford, A.; and Sutskever, I. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J. G.; Le, Q.; and Salakhutdinov, R. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2978–2988.

Du, Y.; Ma, T.; Wu, L.; Zhang, X.; and Ji, S. 2024. AdaCCD: adaptive semantic contrasts discovery based cross lingual adaptation for code clone detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 17942–17950.

Fang, Y.; Zhou, F.; Xu, Y.; and Liu, Z. 2023. TCCCD: Triplet-Based Cross-Language Code Clone Detection. *Applied Sciences*, 13(21): 12084.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547.

Gao, L.; and Callan, J. 2022. Long document re-ranking with modular re-ranker. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2371–2376.

Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; and Yin, J. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7212–7225.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; LIU, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

Guo, Q.; Qiu, X.; Liu, P.; Shao, Y.; Xue, X.; and Zhang, Z. 2019. Star-Transformer. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 1315–1325.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Hu, F.; Wang, Y.; Du, L.; Zhang, H.; Zhang, D.; and Li, X. 2024. Tackling Long Code Search with Splitting, Encoding, and Aggregating. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, 15500–15510.

Jeon, S.; and Moon, J. 2020. Malware-detection method with a convolutional recurrent neural network using opcode sequences. *Information Sciences*, 535: 1–15.

Kitaev, N.; Kaiser, L.; and Levskaya, A. 2020. Reformer: The Efficient Transformer. In *International Conference on Learning Representations*.

Li, C.; Yates, A.; MacAvaney, S.; He, B.; and Sun, Y. 2023a. Parade: Passage representation aggregation fordocument reranking. *ACM Transactions on Information Systems*, 42(2): 1–26.

Li, H.; Wang, S.; Quan, W.; Gong, X.; Su, H.; and Zhang, J. 2024. Prism: Decomposing program semantics for code clone detection through compilation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Li, J.; Tao, C.; Jin, Z.; Liu, F.; and Li, G. 2023b. ZC 3: Zero-shot cross-language code clone detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 875–887. IEEE.

Liu, J.; Zeng, J.; Wang, X.; and Liang, Z. 2023. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 345–357. IEEE.

Lv, F.; Zhang, H.; Lou, J.-g.; Wang, S.; Zhang, D.; and Zhao, J. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 260–270. IEEE.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Musgrave, K.; Belongie, S.; and Lim, S.-N. 2020. A metric learning reality check. In *European Conference on Computer Vision*, 681–699. Springer.

Nafi, K. W.; Kar, T. S.; Roy, B.; Roy, C. K.; and Schneider, K. A. 2019. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1026–1037. IEEE.

Puri, R.; Kung, D.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Annual Conference on Neural Information Processing Systems*.

Roy, C. K.; and Cordy, J. R. 2007. A survey on software clone detection research. *Queen's School of computing TR*, 541(115): 64–68.

Sorokin, N.; Anton, T.; Abulkhanov, D.; Sedykh, I.; Piontkovskaya, I.; and Malykh, V. 2025. CCT-Code: Cross-Consistency Training for Multilingual Clone Detection and Code Search. In Ebrahimi, A.; Haider, S.; Liu, E.; Haider, S.; Leonor Pacheco, M.; and Wein, S., eds., *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*, 178–185. Albuquerque, USA: Association for Computational Linguistics. ISBN 979-8-89176-192-6.

Tao, C.; Zhan, Q.; Hu, X.; and Xia, X. 2022. C4: Contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*, 413–424.

Vislavski, T.; Rakić, G.; Cardozo, N.; and Budimac, Z. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, 512–516. IEEE.

Wang, W.; Li, G.; Ma, B.; Xia, X.; and Jin, Z. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 261–271. IEEE.

Wang, Z.; Ng, P.; Ma, X.; Nallapati, R.; and Xiang, B. 2019. Multi-passage BERT: A Globally Normalized BERT Model for Open-domain Question Answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 5878–5882.

Wei, H.; and Li, M. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, 3034–3040.

Xu, Z.; Qiang, S.; Song, D.; Zhou, M.; Wan, H.; Zhao, X.; Luo, P.; and Zhang, H. 2024. Dsfm: Enhancing functional code clone detection with deep subtree interactions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–12.

Yang, L.; Zhang, M.; Li, C.; Bendersky, M.; and Najork, M. 2020. Beyond 512 tokens: Siamese multi-depth transformer-based hierarchical encoder for long-form document matching. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 1725–1734.

Ye, F.; Zhou, S.; Venkat, A.; Marcus, R.; Tatbul, N.; Tithi, J. J.; Hasabnis, N.; Petersen, P.; Mattson, T.; Kraska, T.; et al. 2020. Misim: A neural code semantics similarity system using the context-aware semantics structure. *arXiv preprint arXiv:2006.05265*.

Yu, H.; Lam, W.; Chen, L.; Li, G.; Xie, T.; and Wang, Q. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 70–80. IEEE.

Zhang, L.; Luo, S.; Pan, L.; Wu, Z.; and Gong, K. 2024. FSD-CLCD: Functional semantic distillation graph learning for cross-language code clone detection. *Engineering Applications of Artificial Intelligence*, 133: 108199.