Advancing Binary Code Similarity Detection via Context-Content Fusion and LLM Verification

Chaopeng Dong[1,2,*], Jingdong Guo[1,2,*], Shouguo Yang[3], Yi Li[4], Dongliang Fang[1,2,†], Yang Xiao[1,2,†], Yongle Chen[5], Limin Sun[1,2]

[1]*Institute of Information Engineering, CAS*, China
[2]*School of Cyber Security, University of Chinese Academy of Sciences*, China
[3]*Zhongguancun Laboratory*, China
[4]*Nanyang Technological University*, Singapore
[5]*College of Computer Science and Technology, Taiyuan University of Technology*, China

{dongchaopeng, guojingdong, fangdongliang, xiaoyang, sunlimin}@iie.ac.cn,
yangshouguo@outlook.com, yi_li@ntu.edu.sg, chenyongle@tyut.edu.cn

***Abstract*—Binary Code Similarity Detection (BCSD), essential for binary-code related tasks like vulnerability detection, has attracted increasing attention in recent years. However, existing methods frequently fall short of achieving both high precision and recall at scale, and their results often lack interpretability due to the neglect of function context and reliance on purely similarity-driven outputs. Our key insights are twofold: *1) Binary functions are not self-contained; they depend on other code and data beyond their content to fulfill their functionalities. 2) Large language models (LLMs) excel not only at analyzing code but also at generating reasonable explanations.* Motivated by these insights, we propose a general BCSD framework, Co^2FuLL. We first systematically select stable and representative code and data features, along with their corresponding dependencies on the functions, to construct the function context. Then, by fusing function context with content similarities computed by the existing BCSD approach, we substantially narrow down the search space. Ultimately, we employ LLMs with a carefully designed prompt to verify the remaining candidates and produce clear, human-readable explanations. We conduct comprehensive experiments on a large function pool under varying compilation settings and after binary stripping. The results show that Co^2FuLL based on HermesSim and DeepSeek-V3 achieves 80.5% precision and 94.4% recall, improving the baseline HermesSim by 142.5% and 42.2%, respectively, providing an accurate and interpretable solution for BCSD.**

***Index Terms*—vulnerability detection, binary code similarity detection, large language model, function context**

I. Introduction

Binary code similarity detection (BCSD) is a core technique underpinning various downstream security tasks, including vulnerability detection [1]–[7], malware detection and clustering [8]–[10], software plagiarism detection [11], software supply chain analysis [12]–[14], and patch analysis [15], [16]. With the rapid advancements in artificial intelligence (AI) and natural language processing (NLP), BCSD research is shifting from traditional rule-based techniques [1], [17] to learning-based approaches [2], [4], [18], [19], which offer better scalability and accuracy. Figure 1 illustrates a typical BCSD pipeline, which includes a learning-based matching step followed by manual verification by security practitioners. Given a query function and a pool of candidate functions: ❶ Features are extracted from appropriate code representations such as control flow graphs (CFGs), producing a variety of input formats; ❷ A neural network encoder (e.g., GGNN) maps these features into high-dimensional embeddings, and similarity scores are computed between the query and each candidate. The top-K or threshold-filtered candidates are retained; ❸ Finally, analysts manually inspect the results to identify true positives (i.e., target functions).

Limitations of Existing Approaches. Researchers have long pursued more representative binary features and advanced neural network models to enhance embeddings and improve BCSD performance [2], [4], [19], [20]. Despite these advancements, current approaches still suffer from two limitations:

- **L1: Difficulty in achieving both high precision and recall at scale.** Modern software systems typically consist of thousands of binaries and millions of functions. Within such vast search spaces, numerous irrelevant functions may exhibit high similarity to a given query, either due to shared structural patterns or superficial feature overlap. This forces analysts into an unfavorable trade-off: adopt stricter matching thresholds and risk missing true positives, or apply looser thresholds and incur a high false positive rate. As demonstrated by our preliminary study in Section II-A, even the best result achieved by HermesSim yielded only 33.2% precision and 66.4% recall.
- **L2: The output lacks interpretability**. The function embeddings generated by existing methods are optimized for computations but remain semantically opaque to humans.

[*] These authors contributed equally to this work.
[†] Corresponding authors
This Work was done while the first author was at Nanyang Technological University.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00033

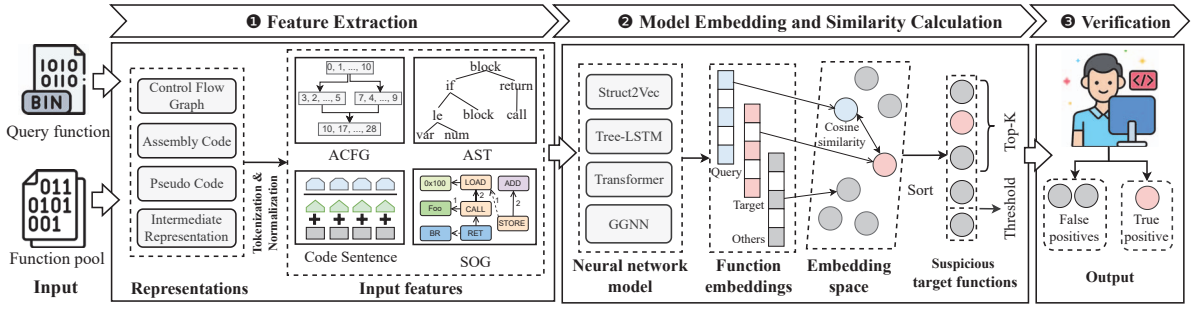

Fig. 1: Pipeline of the BCSD task. ❶ and ❷ are automated processes, while ❸ is a manual process.

As a result, security analysts must scrutinize suspicious target functions from the ground up, without any interpretive assistance from the output, making the process both time-intensive and error-prone.

In the era of large language models (LLMs), numerous researchers have integrated LLMs into a wide array of software engineering tasks, including code generation [21], [22], code summarization [23], [24], and program repair [25], [26]. Motivated by these advancements, we seek to incorporate LLMs into the BCSD task to classify suspicious functions and generate explanatory outputs, addressing both limitations. However, directly using LLMs is impractical due to **L3: the large volume of candidate functions causing significant LLM time and financial costs overhead.** Based on our preliminary study Section II-A, although current BCSD methods remove many irrelevant functions, they still produce hundreds of thousands of false positives needing verification.

Our Approach. The analysis of binary functions reveals that they do not exist in isolation; their associated code, data, and dependencies are vital for reducing both false positives and false negatives. In light of this, we define the code, data, and their dependencies associated to the function as the context, and propose a general BCSD framework, Co^2FuLL (Context–Content Fusion and LLM Verification). The core idea of Co^2FuLL is to *fuse function content and context information to efficiently narrow down the search space and utilize LLM to verify the remaining candidates.* Its workflow consists of two main stages: **1) Candidate Retrieval**. We build the function context with extracted stable binary features and their code and data dependencies. In parallel, we generate function embeddings with the existing BCSD tool as the function content. Content and context similarities are subsequently computed and fused to rank candidate functions, with the top-K results forwarded to the next stage. **2) LLM Verification**. For each top-K candidate, we extract its code snippets and pair them with the query function, accompanied by an LLM instruction crafted through a carefully designed prompt. This composite input is submitted to the LLM, which assesses whether the candidate is the target function and generates an explanation to facilitate manual verification. It is important to emphasize that our goal is not to create a new model that outperforms existing BCSD methods in embedding quality. Instead, *we aim to build a general BCSD framework that improves the effectiveness of current methods and reduces the manual verification effort.*

Evaluation. To evaluate the impact of function context on improving existing BCSD methods across different compilation settings, we compare original baselines with their context-augmented versions on a large function pool. Context improves retrieval MRR by 55.9% on average. The combination of HermesSim and context recalls 98.2% of target functions in the top-5, greatly reducing false positives and lowering LLM time and cost (resolve **L3**).

To evaluate LLM verification accuracy, we apply 4 prompting techniques with 6 prompts, testing 5 LLM settings across 7 LLMs (general, task-specific, and reasoning). Results show LLMs achieve high precision and recall, provide reasonable explanations, showing great utility and reasonability, and improve manual verification accuracy by 6.5%, and reduce time cost by 34.9% (resolve **L2**).

To evaluate Co^2FuLL on BCSD, we employ the optimal configuration (HermesSim+context, DeepSeek-V3 LLM, Few-shot prompt) against baseline methods. The results demonstrate that Co^2FuLL attains 80.5% precision and 94.4% recall, outperforming the original BCSD method HermesSim by 142.5% in precision and 42.2% in recall, while incurring only minimal additional time and financial cost (resolve **L1**).

Contributions. Our contributions are summarized as follows:

- We systematically illustrate the BCSD task workflow and highlight the limitations of existing approaches through a preliminary study on a large function pool.
- We propose Co^2FuLL, a novel BCSD framework that combines function context and content with LLM-based verification to accurately identify target functions at scale. The framework is general, lightweight, and compatible with all existing BCSD methods. We open-source Co^2FuLL at [27] to facilitate the following research.
- We conduct extensive experiments to evaluate Co^2FuLL on the BCSD task, systematically exploring LLM performance across prompting techniques and settings. Results show Co^2FuLL achieves high precision and recall, strengthens existing methods, and greatly reduces manual effort.

II. Preliminary Study and Motivation

In this section, we present a preliminary study and vivid example to underscore the limitations of existing approaches and to motivate our proposed work.

A. Preliminary Study of Existing Approaches

To reveal the limitations of existing BCSD approaches, we conduct a study on 714,084 functions across 24 projects within the BinKit dataset [28]. We randomly select 1,000 query functions, each paired with a function pool comprising 10,000 candidate functions compiled under diverse settings. The details of the dataset construction are presented in Section IV-A. We then select four state-of-the-art BCSD approaches, GMN [18], Trex [6], Asteria [4], and HermesSim [19], which have been extensively evaluated in prior studies, to compute similarities between the query and candidate functions and rank the candidates accordingly. Finally, we vary the threshold from 0 to 1 and adjust the Top-K value from 1 to 50 to compute the precision (P) and recall (R) of approaches. Given our stronger emphasis on recalling target functions, a critical concern in real-world scenarios such as vulnerability detection, we introduce F_2 score as a comprehensive metric as follows:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}, F_2 = \frac{5 \cdot P \cdot R}{4 \cdot P + R} \quad (1)$$

TP, FP, and FN represent the number of correctly recalled target functions, non-target functions incorrectly recalled, and target functions that are not recalled, respectively.

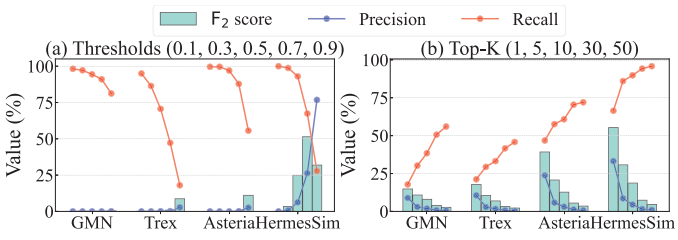

Fig. 2: Results across approaches. The left figure shows results by thresholds, while the right shows results by Top-K values.

Figure 2 presents the results of the four approaches. As observed, using a looser strategy, i.e., lower thresholds or higher Top-K values, improves recall. However, this gain comes at the cost of a significant rise in false positives and a corresponding drop in precision, increasing the burden of manual verification (step ❸ in Figure 1). Even the most effective approach, HermesSim, only attains its best F_2 score of 55.3%, with a precision of 33.2% and a recall of 66.4%, when the top-K is set to 1. In short, current BCSD methods fail to achieve both high precision and recall at scale, forcing analysts to review many candidates and incurring heavy manual effort and time costs.

B. Motivating Example

We select three function pairs $(A_i, B_i), i \in \{1, 2, 3\}$ from the project *gsl-2.5* [29], compiled with different settings to illustrate our motivation. We apply four BCSD methods to compute similarities, shown in red above the CFGs in Figure 3a. Negative pairs (A_1, B_1) and (A_3, B_3) yield high similarities from similar code and structure, causing false positives. The positive pair (A_2, B_2) shows low similarities

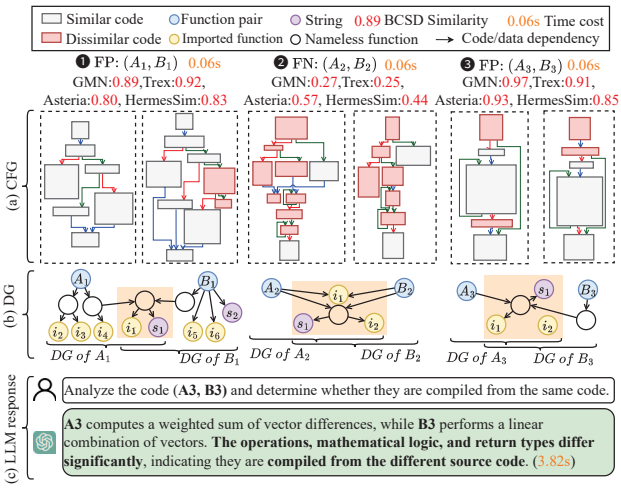

Fig. 3: A motivating example. The figure shows BCSD similarities across four methods, followed by the functions' control-flow graphs (CFGs), dependency graphs (DGs), and the LLM response for the function pair (A_3, B_3). The common regions of the DGs are highlighted with an orange background.

due to compilation differences, resulting in false negatives across all methods.

Since these function pairs are hard to distinguish by content alone, we focus on their dependency graphs (DGs), as shown in Figure 3b. The central DG section highlights shared nodes and edges. The negative pair (A_1, B_1) differs: A_1 indirectly depends on imports i_2, i_3, i_4 via nameless functions, while B_1 directly depends on imports i_5, i_6, and string s_2. In contrast, the positive pair (A_2, B_2) shows strong DG similarity, with both connected to imports i_1, i_2, and string s_1 through shared dependencies. Thus, these two pairs can be easily distinguished by analyzing DG similarities and differences.

For the false positive pair (A_3, B_3), which shows high BCSD similarity and shares most DG nodes and edges, we fed their code snippets into an LLM for classification (Figure 3c). The LLM accurately summarized the functions, correctly classified the pair as negative, and provided a reasonable explanation. However, it was inefficient, taking 3.82 seconds, over 50 times longer than BCSD similarity computations.

Building on the above observations, we distill our motivation into two key insights.

Insight-1. Binary functions are not isolated in binary. The related code and data dependencies in DGs offer a valuable alternative perspective for assessing function similarity.

Insight-2. LLMs can accurately identify functions and provide reasonable explanations, yet are limited by inefficiency.

III. Methodology

Figure 4 illustrates the workflow of Co^2FuLL, which consists of two main stages: ① **Candidate Retrieval** and ② **LLM Verification**. The input consists of a query function and a function pool. The output is the target functions for the query.

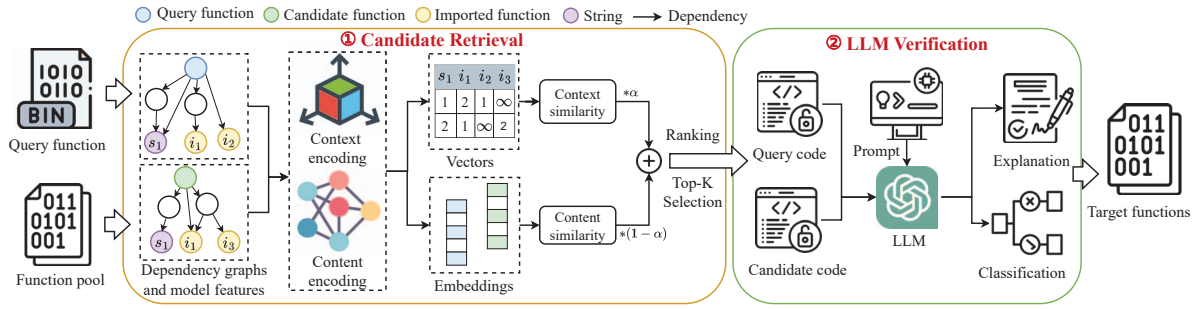

Fig. 4: Overall workflow of Co^2FuLL

The goal of the Candidate Retrieval stage is to swiftly narrow down the search space while ensuring a high recall rate of the target functions. We begin by constructing dependency graphs for both query and candidate functions and extracting features for the BCSD model. Next, we generate context vectors and content embeddings independently to compute two similarity scores, which are then fused to rank the candidates, with the Top-K subsequently forwarded to the next stage.

The LLM Verification stage classifies the Top-K candidates and provides clear explanations. Each candidate is paired with the query function and fed into an LLM through a carefully crafted prompt. The LLM then determines whether the candidate matches the query and generates a corresponding explanation to substantiate its judgment.

A. Candidate Retrieval

LLMs cannot be directly applied to large function pools owing to prohibitive time and financial costs. Candidate retrieval aims to eliminate irrelevant functions by leveraging two principal sources of information: context and content.

1) Function Context: Function context offers an alternative perspective for capturing function semantics, distinct from that conveyed by function content. As highlighted in Section II-B, functions within a binary are not self-contained; they rely on other code and data to achieve their functionality, with such dependencies often propagating recursively. Therefore, we aim to capture the function context by leveraging features in binaries alongside their associated dependency relationships.

We first provide the rationale for context feature selection following two principles: *(1) The extraction and computation of features should remain lightweight to ensure scalability. (2) The features should be stable across different compilation settings and resilient to binary stripping.* Thus, we select five features that allow lightweight extraction and computation [3], [30](i.e., satisfying Principle 1): imports (imp), strings (str), constants (const), the number of CFG edges (edge), and the number of CFG nodes (node). We then further evaluate their stability (i.e., whether they satisfy Principle 2) by conducting experiments on binaries. To this end, we extract these features from stripped binaries and group them by projects and file names. For each group, we randomly select two binaries compiled under different settings, align the source feature set S_{src} with the target set S_{tgt}, and measure stability with the proportion of matched features as in Equation (2).

$$\rho_{\text{feat}} = \frac{|S_{src} \bigcap S_{tgt}|}{|S_{src}|}, \text{feat} \in \{\text{str}, \text{imp}, \text{const}, \text{node}, \text{edge}\} \quad (2)$$

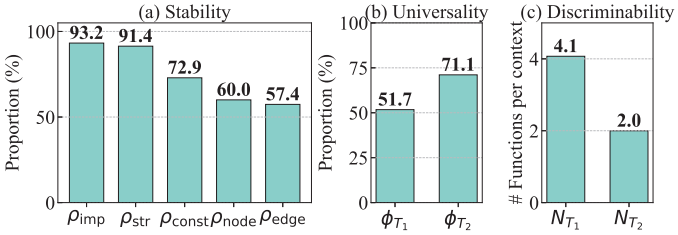

Fig. 5: Average results of the stability of context features, universality, and discriminability of the context studies. The y-axis in Figures (a) and (b) represents the proportion, whereas the y-axis in Figure (c) indicates the average number of functions per distinct context.

Figure 5a shows the stability of candidate features. Imports (93.2%) and strings (91.4%) remain highly consistent across compilation settings and after stripping. In contrast, constants, the number of nodes, and the number of edges show lower stability due to compilation effects (e.g., substitutions such as "> 5" vs. "≥ 6", and loop unrolling). Thus, we only select imports and strings to form the function context.

Based on the selected features, we quantify their relationship strength with functions by measuring the shortest dependency path length (minimum distance) between them in the DG. We then define the function context as follows:

Definition 1 (Function Context): Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ denote the set of functions, $\mathcal{N} = \{n_1, n_2, \ldots, n_k\}, n_i \in \mathcal{I} \bigcup \mathcal{S}$ denote the set of feature nodes in DG. $\mathcal{I}$ and $\mathcal{S}$ are the sets of imports and strings, respectively. The function context is:

$$\mathcal{C}(f_i) = [d(f_i, n_1), d(f_i, n_2), \ldots, d(f_i, n_k)] \quad (3)$$

where $d(f_i, n_j)$ denotes the minimum distance (set to ∞ if unreachable) between function f_i and feature node n_j in DG. Since each function may contain feature nodes absent in the other, we assign a value of ∞ to missing nodes when computing context similarity.

Algorithm 1: Function Context Construction

Input: The set of functions $\mathcal{F}$, imports $\mathcal{I}$ and strings $\mathcal{S}$
Output: Function context $\mathcal{C}$

```
1  Initialize C, D, and DG;
2  for n_i ∈ F ⋃ S do
3  |   for S_f ∈ GetDependFuncs(n_i) do
4  |   |   DG.add_edges(n_i, S_f)
5  |   end
6  end
7  for n ∈ I ⋃ S do
8  |   D(n) ← Dijkstra(DG, n)
9  end
10 for f ∈ F do
11 |   C(f) ← [D(f, n_1), ..., D(f, n_k)], n_i ∈ I ⋃ S
12 end
13 return C;
```

For example, the function context for the query function f_q and the candidate function f_c in Figure 4 are $\mathcal{C}(f_q) = [1, 2, 1, \infty]$ and $\mathcal{C}(f_c) = [2, 1, \infty, 2]$, respectively.

To improve function retrieval, the function context must satisfy two attributes. *1) Universality: it should be common across functions; 2) Discriminability: it should make functions easily distinguishable, with few sharing the same context*. To demonstrate these, we conduct two studies:

Universality Study. To evaluate context universality, we compute the proportion of functions that contain context with:

$$\phi = \frac{M_t}{N}, t \in \{T_1, T_2\} \tag{4}$$

Here, M_t and N are the number of functions with context and the total functions, respectively. To assess the impact of indirect dependencies, we group the associations between functions and features into two types. T_1: the function directly depends on the feature ($d(f_i, n_j) = 1$); T_2: the function depends on the feature via a dependency path in the DG ($d(f_i, n_j) \geq 1$ and $d(f_i, n_j) \neq \infty$). For example, in Figure 3b, B_1 directly depends on i_5, i_6, s_2 (T_1), while A_1 depends on i_2, i_3, i_4 via paths (T_2). Figure 5b shows universality results: with T_1, only 51.7% (ϕ_{T_1}) of functions have context, whereas in T_2, the proportion increases to 71.1% (ϕ_{T_2}), demonstrating strong universality, as the majority of functions benefit from contextual enhancement.

Discriminability Study. To evaluate context discriminability, we first filter functions in Section II-A by project, file, and function name, yielding 29,099 unique functions. We then group them by distinct contexts and count the number of functions for each context as N_t, where $t \in T_1, T_2$. Figure 5c presents the discriminability results: under T_1, an average of 4.1 functions share the same context, whereas with T_2, the number falls to 2.0, reflecting a strong discriminability.

Algorithm 1 shows the procedure for constructing the function context, which can be divided into three main steps:

1) Dependency Graph (DG) Construction (lines 1–6): Following initialization, for each function and string node (n_i), we identify all dependent functions S_f and add the edges in DG accordingly.
2) Minimum Distance Computation (lines 7–9): Each context feature is taken as a starting point, and Dijkstra's algorithm [31] is employed to compute the minimum distances between features and all reachable functions, with the results stored in $\mathcal{D}$.
3) Context Generation (lines 10–12): For each function, its context is generated by querying the corresponding distances between the function and features from $\mathcal{D}$.

2) Context-Content Fusion: The context-content fusion combines the strengths of both content and context information to produce a unified similarity score for BCSD. To achieve that, the content similarity s_{content} is first computed by generating function embeddings using an existing BCSD tool (e.g., HermesSim). Then, the context similarity between functions f_1 and f_2 is calculated using the following equation.

$$\mathbf{s} = \max\{\frac{1}{|\mathcal{N}'|}\sum_{n\in\mathcal{N}'}[1 - \mathcal{F}(\mathcal{C}(f_1, n), \mathcal{C}(f_2, n))], 0\} \tag{5}$$

$$\mathcal{F} = \begin{cases} \frac{|\mathcal{C}(f_1,n) - \mathcal{C}(f_2,n)|}{\max[\mathcal{C}(f_1,n), \mathcal{C}(f_2,n)]} & \text{if } \mathcal{C}(f_1,n) \neq \infty \wedge \mathcal{C}(f_2,n) \neq \infty \\ 1 + \frac{1}{\mathcal{C}(f_1,n)} & \text{if } \mathcal{C}(f_1,n) \neq \infty \wedge \mathcal{C}(f_2,n) = \infty \\ 1 + \frac{1}{\mathcal{C}(f_2,n)} & \text{if } \mathcal{C}(f_1,n) = \infty \wedge \mathcal{C}(f_2,n) \neq \infty \end{cases} \tag{6}$$

Here, $\mathcal{N}'$ denotes the union set of feature nodes associated with the two functions. $\mathcal{C}(f_1)$ and $\mathcal{C}(f_2)$ represent the function contexts of f_1 and f_2, respectively. $\mathcal{C}(f, n)$ indicates the distance between function f and the feature node n. Ultimately, we fuse the context and content similarities with the equation:

$$\mathbf{s}_{\text{fused}} = \alpha \cdot \mathbf{s}_{\text{content}} + (1 - \alpha) \cdot \mathbf{s}_{\text{context}} \tag{7}$$

where $\alpha \in [0, 1]$ is a fusion weight hyperparameter that balances the contributions of content and context similarities.

B. LLM Verification

LLM verification focuses on resolving the remaining candidate functions from the retrieval stage. Leveraging the powerful code analysis capabilities of LLMs, candidate functions are classified as positive or negative by feeding paired query and candidate pseudocode snippets into the LLM using carefully designed prompts. To reduce variability caused by different instruction sets across architectures, we use decompiled pseudocode instead of raw binaries as input for verification. A thorough evaluation of LLM selection, prompt design, and configuration settings is presented in Section IV-A.

IV. Evaluation

Co^2FuLL comprises two stages: candidate retrieval and LLM verification. Accordingly, we evaluate the performance of each stage, as well as the overall effectiveness on the BCSD task, by addressing the following research questions.

- **RQ1**. How effective is the function context in enhancing the retrieval result of existing BCSD approaches? To what extent does each component in context impact the overall contribution (i.e., Ablation study)?

- **RQ2**. How do various prompting techniques and LLM settings affect the accuracy, efficiency, and associated financial expenditure of LLMs?
- **RQ3**. Can the explanations generated by LLMs assist in the manual verification process? Are the explanations offered by LLMs reasonable?
- **RQ4**. How effective is Co^2FuLL in the BCSD task compared to baselines?

A. Experimental Setup

Dataset. We evaluate our method and baselines on the public BinKit dataset [28], which is more diverse than Dataset-1 [32] and BinaryCorp [33], covering binaries from many projects, architectures, compilers, and optimization levels. We split BinKit into training and testing sets: 25 projects with 422,592 functions for training (used to retrain baseline BCSD models and select fusion weight α) and 24 projects with 714,084 functions for testing. To evaluate BCSD robustness under different compilation settings, we design three sub-tasks: XA (cross-architecture and bitness), XC (cross-compiler and optimization levels), and XM (cross all settings). Prior work [3], [19], [20] generally assumes that the target function is always present in the pool, which does not always hold in real-world scenarios, as binaries may lack the queried function. Therefore, for each sub-task, we randomly select 1,000 query functions and sample 10,000 candidates per query from the test set to form dataset D_1, where only half the queries have corresponding targets. In our experiments, HermesSim with function context achieved the best performance, retrieving 98.2% of target functions within the Top-5. Thus, we select the Top-5 candidates per query for LLM verification, forming dataset D_2. Dataset details are shown in Table I.

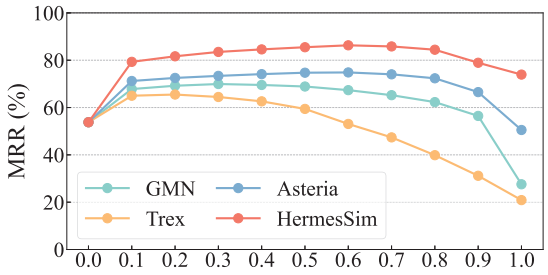

Fig. 6: The selection of fusion weight α.

Implementation. We use IDA Pro [34] to extract features, dependencies, and code snippets, NetworkX [35] to generate dependency graphs and implement Dijkstra's algorithm, and scikit-learn [36] with pandas [37] for similarity computation and ranking. We construct the query and candidate function pool (1,000 queries and 10,000 candidates per query) from the training set, and tune the fusion weight α by varying it from 0 to 1 in increments of 0.1, selecting the value that yields the highest MRR for each BCSD method. As shown in Figure 6, $\alpha = 0$ and $\alpha = 1$ denote performance based entirely on context and the base model, respectively. As α increases, the contribution of the content score to the fusion score gradually grows. Stronger methods, such as HermesSim and Asteria, achieve their highest MRR at α of 0.6, while weaker methods like GMN and Trex achieve their best MRR at low α of 0.3 and 0.2, respectively. The variation of α across methods arises because stronger methods generate more reliable content embeddings and can therefore place greater emphasis on them. In contrast, weaker methods lack sufficient capability in function embedding representation and must therefore rely more heavily on context. All experiments were conducted on a server with an Intel Xeon Gold 5218 CPU @2.30GHz, 512 GB RAM, and dual Tesla V100 GPUs (32 GB each). LLMs are accessed via API services provided by their respective vendors.

TABLE I: Dataset Overview. "QF" and "CF" denote the query and candidate functions, respectively. "Pos." and "Neg." indicate positive and negative function pairs.

Name	Sub-tasks	# QF	# CF	# Pos.	# Neg.
D_1	XC, XA, XM	1,000	10,000	500	9,999.5k
D_2	XM	1,000	5	491	4,510

Baselines. We select five state-of-the-art methods as baselines:

- *GMN* [18]: A GNN variant that jointly reasons over a pair of CFGs, shown to outperform other GNN-based methods in prior work [32].
- *Trex* [6]: Uses a hierarchical Transformer to extract execution semantics from micro traces for embedding.
- *Asteria* [4]: Employs a Tree-LSTM to encode abstract syntax trees (ASTs) derived from pseudocode. Its enhanced variant, *Asteria-Pro* [3], further incorporates imports and exports to filter and re-rank candidates on this basis.
- *HermesSim* [19]: Normalizes binary code by lifting it into Toy IR and uses a semantics-oriented graph (SOG) to generate function embeddings.

Methods such as jTrans [20] and Asm2Vec [5] are excluded due to their lack of cross-architecture support. All baseline methods were evaluated using their original implementations and default configurations.

LLM Selection. We group LLMs into categories and select representatives from each, yielding 7 LLMs in total as follows:

- *General Models*. These models are pre-trained on large text corpora to understand and generate human-like language across diverse topics. We select three series of advanced LLMs: Qwen [38], DeepSeek [39], and GPT [40]. To evaluate the impact of LLM size, we use three models from Qwen-2.5: 7B, 14B, and 72B. For the other two series, we use DeepSeek-V3 and GPT-4o.
- *Task-specific Models*. These models are designed to excel at a single, well-defined task (e.g., code generation, code fixing). Since our task is code-related, we include Qwen2.5-Coder-14B [41], a Code-Specific LLM built upon Qwen2.5.
- *Reasoning Models*. These models are designed to solve complex problems by mimicking human-like logical thinking, using step-by-step deduction and causal inference. We use DeepSeek-R1, an advanced reasoning model.

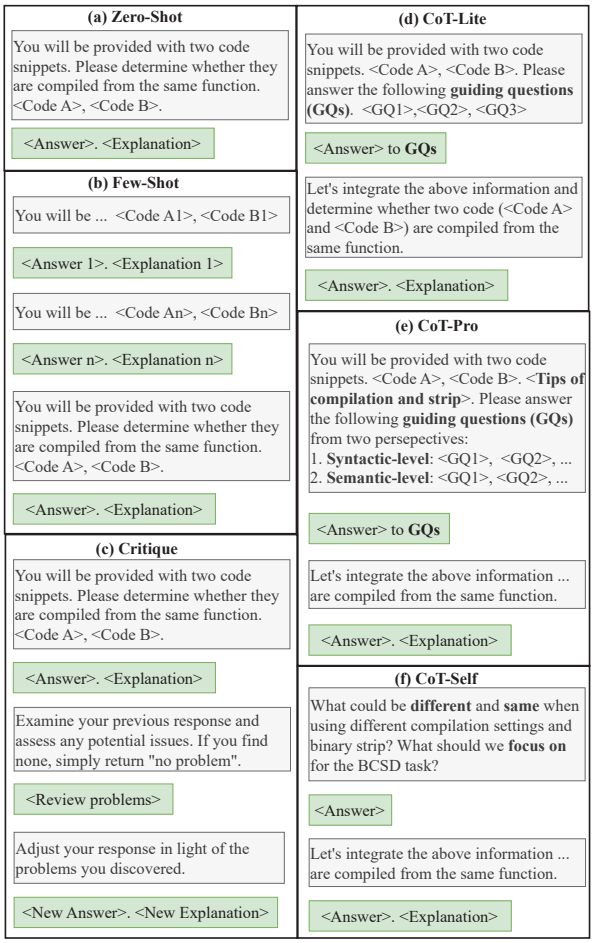

Fig. 7: Prompt techniques. The grey boxes represent user inputs, while the green boxes display the LLM responses.

LLM Settings. We explore two major LLM settings.

- `top_p` (nucleus sampling) controls output diversity by limiting next-word choices to the smallest set with a cumulative probability above the threshold (e.g., 0.8 includes only the top 80% most likely words).
- `temperature` regulates the randomness of the model's output. Lower values yield more deterministic and consistent responses, whereas higher values introduce greater variability, resulting in more diverse and inventive outputs.

Prompting Techniques. Inspired by prior works [23], [42], we adopt four commonly used prompting techniques as follows:

- *Zero-Shot* [43] prompt instructs the model to perform a task without providing examples or extra context, as in Figure 7a.
- *Few-Shot* [44] provides several input-output examples (typically 2–5) before the actual task to illustrate the expected response, as demonstrated in Figure 7b.
- *Critique* [45] encourages LLMs to identify potential problems in their initial responses and refine their answers accordingly, as shown in Figure 7c.
- *Chain-of-Thought (CoT)* [46] prompt adapts LLMs by incorporating intermediate reasoning steps, thereby guiding the model toward more accurate and structured responses. We reference [47] and propose three CoT variants. 1) *CoT-Lite*: Pose guiding questions to the LLM, then use both questions and responses as task context (Figure 7d). 2) *CoT-Pro*: Provide background on compilation settings and binary stripping, with guiding questions at syntactic and semantic levels (Figure 7e). 3) *CoT-Self*: Instead of expert-crafted questions, instruct the LLM to independently identify compilation effects, stripping impacts, and key task points (Figure 7f).

TABLE II: Results of the function retrieval experiment. "Base" and "+Context" denote methods without and with context similarity, respectively. Scores are reported as MRR (%).

Method	XC		XA		XM	
	Base	+Context	Base	+Context	Base	+Context
Trex	43.7	(↑71.8) 75.1	16.2	(↑301.4) 64.9	25.8	(↑176.2) 71.1
GMN	34.9	(↑99.2) 69.5	33.6	(↑117.6) 73.1	24.7	(↑185.6) 70.4
Asteria	55.1	(↑40.9) 77.7	81.5	(↑10.3) 89.9	52.2	(↑52.7) 79.6
HermesSim	80.6	(↑12.0) 90.3	90.1	(↑4.8) 94.4	74.7	(↑23.4) 92.2
Average	53.6	(↑45.8) 78.1	55.3	(↑45.6) 80.6	44.3	(↑76.8) 78.3

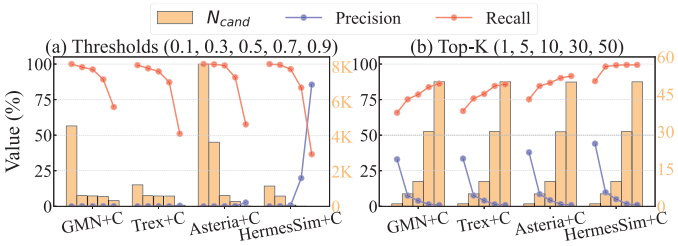

Fig. 9: Impact of different thresholds and Top-K for methods with context. "+C" is short for "+Context". The bar charts are plotted against the right y-axis.

B. RQ1: Enhancement of Context for Function Retrieval

Following prior work [3], [19], we employ Mean Reciprocal Rank (MRR $= \frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}}\frac{1}{r_q}$) to show the impact of context enhancement for function retrieval, where $\mathcal{Q}$ is the set of queries and r_q is the rank of the targets to query q.

Table II presents the function retrieval results of the original BCSD approaches and their enhanced versions incorporating the function context across XC, XA, and XM sub-tasks. On the one hand, the integration of context boosts the performance of baselines substantially across all methods and sub-tasks. On average, the four BCSD methods augmented with context surpass their original version (Base) by 45.7%, 45.6%, and 76.4% across the three sub-tasks, respectively.

On the other hand, the improvement of scores varies across different methods, owing to their capabilities. For example, GMN and Trex exhibit remarkable gains of 185.6% and 176.2%, respectively, when enriched with context in the XM task. In contrast, methods such as Asteria and Hermessim show relatively modest increases in MRR for the XM task, with improvements of 52.7% and 23.4%, respectively.

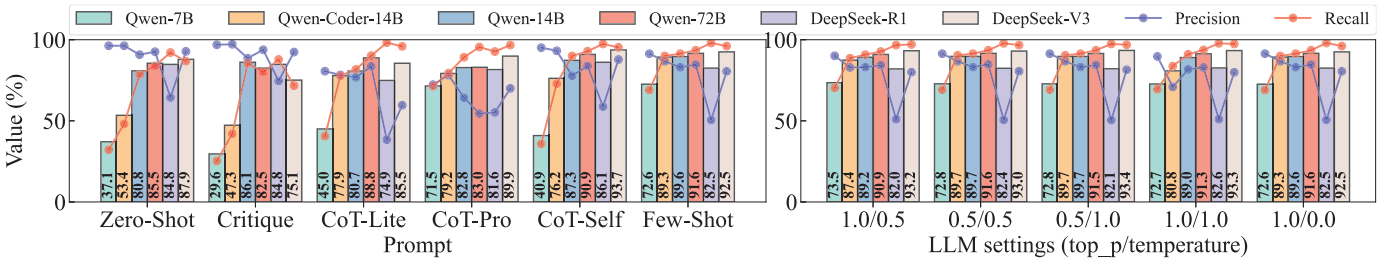

Fig. 8: The impact of LLM verification across varying prompts and LLM settings. The bar charts denote the F_2 score.

To evaluate the tradeoff between two filtering strategies, we vary the threshold from 0 to 1 and the Top-K value from 1 to 50, recording precision, recall, and N_{cand} (the number of candidates for LLM verification per query after filtering) as evaluation metrics. *The goal of filtering is to reduce the number of candidates while preserving the majority of target functions.* As shown in Figure 9, increasing Top-K or lowering the threshold improves the recall but also raises N_{cand} (i.e., lower precision), resulting in greater time and financial overhead for subsequent LLM verification. Top-K-based filtering outperforms threshold-based filtering, as the latter often yields more than 100 candidates per query to achieve a high recall comparable to that of the former. Specifically, with Top-K set to 5, HermesSim+C attains 98.2% recall while limiting the candidate set to just 5 per query, thereby striking an optimal balance between recall and computational cost.

Ablation Study. To showcase the contributions of each component in the function context, we established five distinct configurations for the ablation study:

- Base: The base methods without any context.
- Base+Context (direct): Methods enhanced with context composed of features directly relied upon by the functions.
- Base+Context (imp): Methods enhanced with context composed solely of the imports.
- Base+Context (str): Methods enhanced with context composed solely of the strings.
- Base+Context: Methods enhanced with complete context.

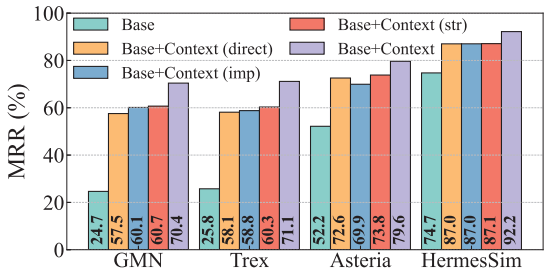

Fig. 10: Ablation study of the function context.

Figure 10 presents the results of the ablation study. As illustrated, methods incorporating the complete context consistently outperform the other four configurations, with average improvements of 76.8% over Base, 13.9% over Base+Context (direct), 13.4% over Base+Context (imp), and 11.2% over Base+Context (str), respectively. The superior performance of the complete context stems from its ability to leverage comprehensive information, encompassing all dependencies and features associated with the target function. In contrast, alternative configurations, such as relying solely on strings, imports, and direct dependencies, inevitably result in a performance loss.

> **Answer to RQ1**. Context integration boosts MRR for function retrieval by 55.9% across the sub-tasks on average, while removing any component in context drops it by 11.2–13.9%, highlighting their importance.

C. RQ2: Impact of Prompt and LLM Settings on LLM

Figure 8 shows the precision, recall, and F_2 score for various prompt techniques and LLM settings across different LLMs on D_2. Owing to the significantly higher cost of GPT-4o, it is excluded from the prompt and settings experiments. The results reveal the following findings.

Finding 1. *The size of LLMs significantly impacts their accuracy and sensitivity to prompt engineering, with this influence progressively diminishing as model scale increases.* Larger LLMs tend to yield higher accuracy than smaller LLMs. Using Zero-Shot, the F_2 scores of Qwen-7B, Qwen-14B, and Qwen-72B are 37.1%, 80.8%, and 85.5% respectively. The corresponding gain in F_2 decreases from 117.8% (Qwen-14B over Qwen-7B) to 5.8% (Qwen-72B over Qwen-14B). When shifting from Zero-Shot to Few-Shot, the F_2 score of Qwen-7B rises by 95.7% (37.1% to 72.6%), whereas Qwen-72B shows only a modest gain of 7.1% (85.5% to 91.6%).

Finding 2. *Prompting techniques exert markedly different levels of influence across LLMs*. Few-Shot is the most reliable, consistently achieving high accuracy across all models. Except for Qwen-72B, CoT-Pro outperforms CoT-Lite across all LLMs, showing that detailed guidance and refined task representation provide clear benefits. CoT-Self suits high-capacity models like DeepSeek-V3 (R1) but performs worse on smaller models (e.g., Qwen-7B) than other CoT variants. Critique performs worst, even below Zero-Shot, as it often mislabels correct answers despite occasionally fixing errors.

Finding 3. *General LLMs outperform task-specific and reasoning models*. Qwen-Coder-14B and DeepSeek-R1 generally excel in specialized tasks such as code generation and mathematical analysis. Yet, they underperform their general-purpose counterparts, Qwen-14B and DeepSeek-V3, by 33.9% and 3.5% under Zero-Shot, respectively.

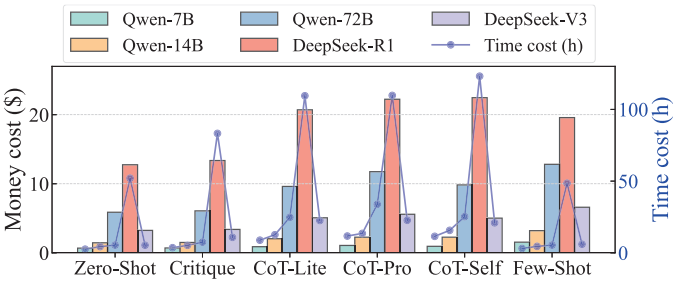

Fig. 11: The time and money cost across varying prompts and LLMs on D_2. The bar charts denote the money costs.

Finding 4. *LLM settings have minimal impact on accuracy*. Except for Qwen-Coder-14B, which experiences a 9.5% decline in F_2 score when shifting from `1.0/0.0` to `1.0/1.0`, other models show differences under 1% across LLM settings. **Time and Money Cost**. We report the time and money costs of LLMs under different prompting techniques in Figure 11. Zero-Shot, Critique, and Few-Shot incur lower latency than CoT, which is slowed by intermediate reasoning. Few-Shot is generally the most expensive due to example overhead, except for DeepSeek-R1, which shows the highest cost and latency overall because of its lengthy reasoning output.

> **Answer to RQ2**. Advanced prompts greatly enhance LLM accuracy, especially for smaller models. Conversely, LLM settings present minimal impact.

D. RQ3: Utility and Reasonability of LLM Explanations

For the utility and reasonability studies of LLM explanations, we randomly selected 50 samples (25 positives and 25 negatives) from D_2 and invited 6 participants with over 4 years of reverse engineering and BCSD experience. They were evenly divided into two groups (G1 and G2), with the utility study conducted first to ensure G1 was not exposed to explanations before the reasonability study.

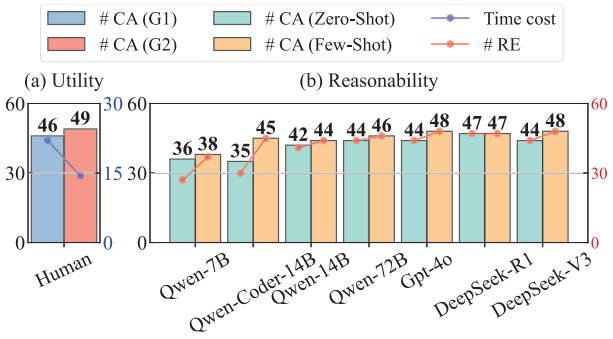

Fig. 12: Results of the utility and reasonability studies for LLM explanations. "# CA" and "# RE" denote the number of correct answers and reasonable explanations.

Utility Study. This study aims to illustrate the utility of explanations in aiding manual verification. For the 50 samples, G1 receives only the code snippets, whereas G2 is provided with both the code snippets and the corresponding explanations generated by DeepSeek-V3 using the Few-Shot prompt. We ask the participants to classify the samples and record the analysis time cost. As observed in Figure 12a, G2 demonstrates higher accuracy in classifying the samples and reduces analysis time by 34.9% compared to G1. With the support of LLM-generated explanations, participants in G2 can swiftly discern the differences (or similarities) between the two code snippets and validate the findings by focusing on the corresponding code segments. In contrast, participants in G1 must analyze the snippets from scratch, a time-consuming process that risks overlooking critical details. Furthermore, LLMs often summarize the functionalities of the code snippets, aiding participants in rapidly grasping their intended purposes.

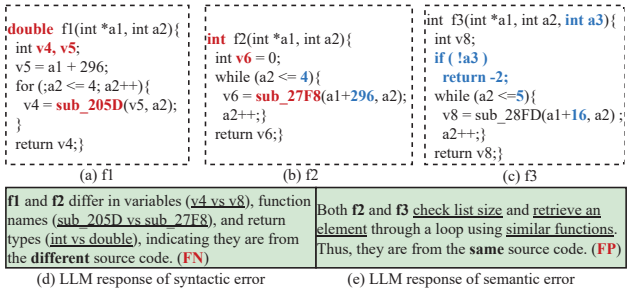

Fig. 13: LLM failure cases. Figures (a) to (c) present the code of three functions, where f1 and f2 are compiled from the same code, while f3 originates from a different source. Figures (d) to (f) display the corresponding LLM responses. Syntactic differences between f1 and f2 are highlighted in red, whereas semantic differences between f2 and f3 are marked in blue.

Reasonability Study. This study evaluates the reasonability of explanations. We select two prompting techniques, Zero-Shot and Few-Shot, across 7 LLMs, resulting in 700 explanations to be examined. For 50 samples, code snippets together with their explanations are presented to G1, and participants are asked to determine whether each explanation is reasonable.

As observed, *LLMs with greater capacity demonstrate superior classification accuracy and produce more coherent, reasoned explanations under identical prompts*. For instance, under the Zero-Shot prompt, Qwen-7B correctly classifies 36 samples, of which only 27 are considered reasonable. In contrast, Qwen-14B accurately identifies 42 samples, with 41 accompanied by reasonable explanations. Moreover, *Few-Shot prompt significantly enhances the reasonability of explanations generated by LLMs, thereby improving overall accuracy*. Leveraging Few-Shot, Qwen-7B successfully addresses shortcomings observed under the Zero-Shot prompt, where compilation settings and binary stripping-related variations were often misinterpreted as functional differences. Notably, even advanced models such as GPT-4o and DeepSeek-V3 benefit from Few-Shot prompting, with the number of reasonable explanations rising from 44 to 48.

Failure Cases Analysis. We categorize the common failure cases (92 out of 700 test cases in RQ3) and present represen-

TABLE III: Results of BCSD methods and Co^2FuLL with different LLMs on D_1. N_v = TP + FP denotes the number of functions requiring human verification (i.e., functions reported as positives), and FG is short for feature generation. Time is measured using a single process. T and K denote the threshold and Top-K values that yield the highest F_2 score, with the corresponding metrics reported under these settings.

Method	LLM	Metric (%)										Time (h)			Money ($)
		T	P@T	R@T	F_2@T	N_v	K	P@K	R@K	F_2@K	N_v	FG	Matching	Total	
GMN	N/A	0.95	0.2	67.0	0.8	196,075	1	8.9	17.8	14.8	1,003	153.7	0.2	153.9	N/A
Trex		0.95	14.1	9.2	9.9	326	1	10.6	21.2	17.7	1,002	163.2	0.2	163.4	
Asteria		0.95	26.1	37.8	34.7	725	1	24.9	50.4	41.8	1,011	2242.8	0.2	2243.0	
Asteria-Pro		0.95	0.7	48.0	0.8	33,278	1	30.5	60.0	50.3	984	393.5	0.2	393.7	
HermesSim		0.70	26.3	67.4	51.4	1,279	1	33.2	66.4	55.3	1,000	199.2	0.2	199.4	
HermesSim+Context		0.80	48.4	61.0	58.0	630	1	44.0	88.0	73.3	1,000	199.9	0.3	200.2	
Co^2FuLL (HermesSim+Context)	Qwen-7B	N/A					5	91.4	67.8	71.5	371	199.9	3.3	203.3	1.5
	Qwen-Coder-14B							86.7	88.4	88.1	510		4.7	204.6	6.4
	Qwen-14B							83.0	89.8	88.4	541		4.8	204.7	3.2
	Qwen-72B							84.7	91.8	90.3	542		5.7	205.6	12.8
	GPT-4o							82.3	90.4	88.7	549		5.9	205.8	55.3
	DeepSeek-R1							50.6	96.2	81.5	951		48.8	248.7	19.6
	DeepSeek-V3							80.5	94.4	**91.2**	586		6.3	206.2	6.6

* N_v exceeds 1,000 due to tied rankings in Top-K results, while for Asteria-Pro, it's below 1,000 as some queries yield no candidates owing to its filtering module.

tative examples for each category as follows:

- *Syntactic error* (47 cases): Due to binary stripping and the limitation of decompilation, the code exhibits syntactic variations, such as differing variable and function names. In such cases, these differences are misinterpreted by LLMs as functional mismatches, resulting in false negatives. This type of error occurs predominantly with smaller LLM models, such as Qwen-7B, and is seldom observed with larger models such as Qwen-72B and DeepSeek-V3. Figure 13d illustrates a failure case: the LLM interprets differences such as variable names and function names as substantive code differences, resulting in a false negative.
- *Semantic error* (45 cases): LLMs excel at capturing the overall semantics of functions, yet they may overlook subtle differences (e.g., loop bounds or numerical constants), especially when the structure and logic of two pieces of code appear highly similar. Moreover, code optimizations such as function inlining may partly alter a function's semantics, leading to deviations in the LLMs' interpretation. Figure 13e illustrates a failure case: while the LLM correctly identifies the core logic shared by the two code snippets, it overlooks subtle differences, such as the constants used in addition, the return values, and the check on `a1` in `f4` that is absent from `f3`), resulting in a false positive.

Answer to RQ3. The LLM-generated explanations facilitate manual verification, which improves both accuracy and time cost.

E. RQ4: Effectiveness of Co^2FuLL in the BCSD task

Table III shows results of baselines and Co^2FuLL with different LLMs on D_1. To reduce output randomness and ensure fair comparison, we use the Few-Shot prompt with `top_p`=1.0 and `temperature`=0. We then vary the similarity threshold (0–1) and Top-K (1–50), reporting the best metrics. The first two columns list the method and LLM, while the others present metrics, time, and cost for each approach.

Metric Analysis. All baselines yield low F_2 scores even with optimal thresholds and Top-K values. In contrast, Co^2FuLL, built on HermesSim with function context, achieves a 73.3% F_2, 32.5% higher than HermesSim. Using a high-capacity LLM (DeepSeek-V3) to verify the Top-5 candidates from HermesSim+Context further improves performance to 80.5% precision, 94.4% recall, and 91.2% F_2, surpassing HermesSim by 64.9%. These results show the importance of context and LLMs in BCSD. With LLM support, manual verification (N_v) drops significantly, for instance, Co^2FuLL with HermesSim and DeepSeek-V3 reduces N_v from 1,000 to 586 (41.4%) compared to HermesSim alone.

Scalability Analysis. Benefiting from candidate retrieval, the LLM needs only to verify a small set of candidates (Top-5), incurring minimal additional time and financial cost. As shown in Table III, the total runtime of Co^2FuLL with context and DeepSeek-V3 on the large-scale dataset D_1 (10 million function pairs) is 206.2 hours, merely 6.8 hours (3.4%) longer than HermesSim's 199.4 hours. The additional financial cost is only $6.6, making the approach both economical and practical. Since the LLM is only used to analyze the Top-5 candidates, Co^2FuLL's cost remains constant regardless of the size of the function pool. Thus, Co^2FuLL exhibits strong scalability and is highly suitable for large-scale BCSD tasks.

Answer to RQ4. Co^2FuLL achieves both high precision and recall in the BCSD task, especially when using model HermesSim and LLM DeepSeek-V3.

V. Discussion

A. Takeaways

LLM and Prompt Selection. Large-scale general models are generally better for BCSD but require higher financial and computational resources. Thus, participants should weigh accuracy against cost. Beyond DeepSeek-V3, Qwen-14B offers strong accuracy at very low cost, making it attractive for small companies. Few-shot prompting proves to be the most practical strategy, balancing efficiency and accuracy across LLMs. For Chain-of-Thought (CoT), the choice should align with model

capacity: smaller models (e.g., Qwen-7B) benefit from CoT-Pro's explicit reasoning guidance, while larger reasoning-optimized models (e.g., DeepSeek-R1) perform best with CoT-Self, autonomously generating reasoning steps to exploit their advanced capabilities.

Core Idea Migration. For computationally intensive tasks such as BCSD, practitioners must carefully balance efficiency, accuracy, and financial cost when employing LLMs. While LLMs offer powerful capabilities in code understanding and reasoning, they require substantially more resources and time compared to lighter models like GNNs. The core idea of Co^2FuLL, delegating the majority of easily identifiable cases to lightweight models or contextual features while using LLMs for the minority of difficult cases, can be extended to similar tasks, thereby enhancing both efficiency and accuracy with minimal additional cost.

Future Work. LLMs excel at analyzing code and understanding its purpose and core logic. Yet, as our failure case analysis reveals, they may overlook subtle distinctions when the overall code logic appears similar. A promising next step, therefore, is to leverage LLMs for high-level logic analysis while integrating external tools, such as symbolic execution [48], [49] and the SMT Solver Z3 [50] for formal verification, thereby capturing the subtle differences that LLMs may overlook.

B. Threats to Validity and Limitations

While Co^2FuLL shows promising results, we acknowledge several potential threats to validity and inherent limitations: (1) LLM responses may vary for identical inputs due to inherent randomness. To mitigate this, we first evaluated different LLM settings and observed that the impact on classification results was negligible. We then also fixed `temperature` at 0 to ensure consistent outputs. (2) We employ IDA Pro [34] to construct dependency graphs and extract code snippets. However, in stripped binaries, IDA may fail to accurately detect function boundaries. In future work, we plan to integrate additional binary analysis tools (e.g., Ghidra [51]) or adopt function identification techniques [52]–[54] to achieve more precise function detection and thereby overcome this limitation. (3) Binary obfuscation transforms code into harder-to-analyze forms while preserving functionality, using techniques such as control-flow flattening [55], opaque predicates [56], and string encryption [57], which will impact the effectiveness of both context and LLM. Accordingly, several works, such as SATURN [58], AutoSimpler [59], and FLOSS [60], have proposed effective techniques for recovering obfuscated information in binaries, which can be employed as a preprocessing step when applying Co^2FuLL to obfuscated binaries.

VI. Related Work

We divide the existing BCSD methods based on the features they use into two categories:

Text-Based. These approaches treat binary code as textual data and employ natural language processing (NLP) models to derive function semantics. Both Asm2Vec and Trex extract execution traces from binaries as input; the former utilizes the PV-DM model [61], whereas the latter employs a hierarchical Transformer architecture to generate embeddings, enabling cross-architecture detection and achieving higher accuracy owing to the capabilities of the stronger model. jTrans segments assembly code based on jump instructions and embeds the resulting sequences using the BERT architecture [62]. It introduces two novel pre-training tasks tailored for BCSD to capture function semantics, followed by fine-tuning with supervised learning, thereby enhancing BCSD performance.

Graph-Based. These methods primarily leverage graph neural networks (GNNs) and graph-structured representations (e.g., CFGs) to generate function embeddings. Gemini is among the earliest works to adopt the Struct2Vec model [63] for function embedding, surpassing Genius, a prior method using the same features without deep learning, and thereby demonstrating the effectiveness of deep learning techniques in advancing BCSD. GMN employs a variant of GNN that jointly reasons over pairs of CFGs, achieving superior performance compared to Gemini. Asteria observed that abstract syntax trees (ASTs) derived from pseudocode decompiled by binary analysis tools (e.g., IDA Pro) exhibit greater robustness across architectures than CFGs, and utilized a Tree-LSTM [64] to encode them, yielding improved performance over GMN. HermesSim, the current state-of-the-art, emphasizes that prior works neglect critical code relationships, data (def-use), effect (execution order), and function conventions that remain robust across compilation settings. Accordingly, HermesSim normalizes binary code into Toy IR and constructs a semantics-oriented graph (SOG) based on these relationships, leveraging a Gated Graph Neural Network (GGNN) [65] to generate function embeddings, and outperforms all existing approaches.

Summarization. On the one hand, most existing methods primarily focus on the features of function content while either neglecting or failing to fully leverage the potential power of function context. On the other hand, they only employ small-scale deep learning networks to capture the variation patterns of functions under different compilation settings, resulting in inferior semantic understanding compared to LLMs.

VII. Conclusion

This work proposes a novel BCSD framework, Co^2FuLL, which integrates both content and contextual information of binary functions with LLM to advance existing BCSD methods. By systematically exploring diverse prompting strategies and LLM settings, Co^2FuLL identifies the optimal setup for effective LLM utilization. The results show that Co^2FuLL yields a precision of 80.5% and a recall of 94.4% on a large-scale BCSD, surpassing baselines by a significant margin.

VIII. ACKNOWLEDGMENT

We appreciate all the anonymous reviewers for their invaluable comments. This work is supported by National Natural Science Foundation of China (Grant No.92467201, No.62202462, No.62472302). Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Y. David and E. Yahav, "Tracelet-based code search in executables," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[2] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017, pp. 363–376.

[3] S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun, "Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge," *ACM Transactions on Software Engineering and Methodology*, vol. 33, pp. 1 – 40, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:255372539

[4] S. Yang, "Asteria: Deep learning-based for cross-platform binary code similarity detection," in *DSN*, 2021, p. 13.

[5] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," in *DIMVA*, 2019.

[6] K. Pei, Z. Xuan, J. Yang, S. S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *ArXiv*, vol. abs/2012.08680, 2020.

[7] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *ASE*, 2018, pp. 896–899.

[8] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *International Symposium on Recent Advances in Intrusion Detection*, 2005.

[9] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 46–54, 2007.

[10] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variant-detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 307–317, 2013.

[11] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 389–400.

[12] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *MSR*, 2022.

[13] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: Binary level partially imported third-party library detection via program modularization and semantic matching," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1393–1405, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248259104

[14] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 791–802, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267523849

[15] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220497409

[16] C. Dong, J. Guo, S. Yang, Y. Xiao, Y. Li, H. Li, Z. Li, and L. Sun, "Plocator: Fine-grained patch presence test in binaries via patch code localization," *ACM Transactions on Software Engineering and Methodology*, 2025.

[17] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *International Symposium on Software Testing and Analysis*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:6480274

[18] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International Conference on Machine Learning*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:139102215

[19] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *USENIX Security Symposium*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271325358

[20] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[21] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 982–994, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269128474

[22] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, "Teaching code llms to use autocompletion tools in repository-level code generation," *ArXiv*, vol. abs/2401.06391, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266977002

[23] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," *ArXiv*, vol. abs/2407.07959, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271097661

[24] R. Haldar and J. Hockenmaier, "Analyzing the performance of large language models on code summarization," *ArXiv*, vol. abs/2404.08018, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269137593

[25] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B.-C. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *ArXiv*, vol. abs/2310.08879, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264127977

[26] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, and Z. Chen, "A systematic literature review on large language models for automated program repair," *ArXiv*, vol. abs/2405.01466, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269502453

[27] "The official code and dataset of $co^2 full$." https://github.com/GentleCP/Co2FuLL-public, 2025.

[28] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, vol. 49, pp. 1661–1682, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:227127587

[29] "The gnu scientific library (gsl) is a numerical library for c and c++ programmers. it is free software under the gnu general public license." https://www.gnu.org/software/gsl/, 2025.

[30] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. X. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:24381176

[31] "Dijkstra's algorithm," https://en.wikipedia.org/wiki/Dijkstra's_algorithm, 2023.

[32] A. Marcelli, M. Graziano, and M. Mansouri, "How machine learning is solving the binary function similarity problem," 2022.

[33] "Binarycorp is built for binary similarity detection based on the archlinux official repositories and arch user repository," https://paperswithcode.com/dataset/binarycorp, 2022.

[34] "The best-of-breed binary code analysis tool, an indispensable item in the toolbox of world-class software analysts, reverse engineers, malware analyst and cybersecurity professionals." https://hex-rays.com/ida-pro/, 2022.

[35] "Networkx is a python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks." https://networkx.org/, 2025.

[36] "Simple and efficient tools for predictive data analysis." https://scikit-learn.org/stable/, 2025.

[37] "Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the python programming language." https://pandas.pydata.org/, 2025.

[38] "A family of large language models developed by alibaba cloud." https://chat.qwen.ai/, 2025.

[39] "A chinese artificial intelligence company that develops large language models (llms)." https://www.deepseek.com/, 2025.

[40] "A multimodal large language model trained and created by openai and the fourth in its series of gpt foundation models." https://chatgpt.com/, 2025.

[41] "An advanced code-specialized large language model (llm) developed by alibaba's qwen team, designed to excel in programming tasks such as code generation, debugging, completion, and explanation." https://github.com/QwenLM/Qwen2.5-Coder, 2025.

[42] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *ArXiv*, vol. abs/2402.07927, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267636769

[43] "Prompt engineering skills help to better understand the capabilities and limitations of large language models (llms)." https://www.promptingguide.ai/techniques, 2025.

[44] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. teusz Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:218971783

[45] G. Kim, P. Baldi, and S. M. McAleer, "Language models can solve computer tasks," *ArXiv*, vol. abs/2303.17491, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257834038

[46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, F. Xia, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *ArXiv*, vol. abs/2201.11903, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246411621

[47] Y. Wang, Z. Zhang, and R. Wang, "Element-aware summarization with large language models: Expert-aligned evaluation and chain-of-thought method," in *Annual Meeting of the Association for Computational Linguistics*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258841145

[48] "A symbolic virtual machine built on top of the llvm compiler infrastructure." https://github.com/klee/klee, 2024.

[49] "An open-source binary analysis platform for python. it combines both static and dynamic symbolic ("concolic") analysis, providing tools to solve a variety of tasks.." https://angr.io/, 2024.

[50] "A theorem prover from microsoft research." https://github.com/Z3Prover/z3, 2025.

[51] "A software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission." https://www.ghidra-sre.org/, 2024.

[52] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *USENIX Security Symposium*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:17022472

[53] X. Yin, S. Liu, L. Liu, and D. Xiao, "Function recognition in stripped binary of embedded devices," *IEEE Access*, vol. 6, pp. 75 682–75 694, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:56595927

[54] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling," *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2391–2408, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259267136

[55] T. László and A. Kiss, "Obfuscating c++ programs via control flow flattening," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:5061467

[56] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Information Security Conference*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:31112136

[57] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini, "Hidden in plain sight: Obfuscated strings threatening your privacy," *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211075800

[58] P. Garba and M. Favaro, "Saturn - software deobfuscation framework based on llvm," *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:202538819

[59] Y. Zhao, Z. Tang, G. Ye, X. Gong, and D. Fang, "Input-output example-guided data deobfuscation on binary," *Security and Communication Networks*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:245184281

[60] "Flare obfuscated string solver (flare-floss)." https://github.com/mandiant/flare-floss, 2025.

[61] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Neural Information Processing Systems*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:16447573

[62] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399

[63] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," *ArXiv*, vol. abs/1603.05629, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:2708270

[64] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *ArXiv*, vol. abs/1503.00075, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:3033526

[65] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," *arXiv: Learning*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:8393918