Transforming Generic Coder LLMs to Effective Binary Code Embedding Models for Similarity Detection

Litao Li[*]
Queen's University
litao.li@queensu.ca

Leo Song[*]
Queen's University
leo.song@queensu.ca

Steven H. H. Ding
McGill University
steven.h.ding@mcgill.ca

Benjamin C.M. Fung
McGill University
ben.fung@mcgill.ca

Philippe Charland
Mission Critical Cyber Security Section, Defence R&D Canada
philippe.charland@drdc-rddc.gc.ca

Abstract

Cybersecurity and software research have crossed paths with modern deep learning research for a few years. The power of large language models (LLMs) in particular has intrigued us to apply them to understanding binary code. In this paper, we investigate some of the many ways LLMs can be applied to binary code similarity detection, as it is a significantly more difficult task compared to source code similarity detection due to the sparsity of information and less meaningful syntax. It also has great practical implications, such as vulnerability and malware detection. We find that pretrained LLMs are mostly capable of detecting similar binary code, even with a zero-shot setting. Our main contributions and findings are to provide several supervised fine-tuning methods that, when combined, significantly surpass zero-shot LLMs and state-of-the-art binary code similarity detection methods. Specifically, we up-train the model through data augmentation, translation-style causal learning, LLM2Vec, and cumulative GTE loss. With a complete ablation study, we show that our training method can transform a generic language model into a powerful binary similarity expert, and is also robust and general enough for cross-optimization, cross-architecture, and cross-obfuscation detection.

1 Introduction

LLMs are almost ubiquitous in everyday life, fueling many powerful applications. Cybersecurity and software researchers have utilized language models to solve complex tasks, such as code generation, summarization, and similarity detection. In this paper, we investigate the benefits that LLMs can bring to binary code modeling. In particular, binary code similarity detection (BCSD) and retrieval are difficult due to the sparsity of information, loss of semantics and structures, and its limited syntax compared to source code. Moreover, the same source code can be compiled into many formats of binary code using different compiler settings and environments, including optimization levels, machine architectures, compiler software, and obfuscation techniques. Existing LLMs [Roziere *et al.*, 2023; Radford, 2018] can somewhat perform binary code analysis with their large training data of trillions of natural language and code tokens, but the proportion of binary code is small. Downstream tasks like similarity detection and vulnerability detection are still difficult and require fine-tuning. Some open-source coder models like CodeT5 [Wang *et al.*, 2021] and GraphCodeBERT [Guo *et al.*, 2020] even lack the training of binary code, making zero-shot useless for binary analysis.

[*]Equal contribution.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

Many binary code models are trained from scratch [Yang *et al.*, 2021; Tian *et al.*, 2021; Yu *et al.*, 2020] and their sizes are much smaller compared to LLMs. They typically focus on cross-optimization retrieval and fail to generalize to diverse compiler settings. Other works take pre-trained LLMs [Tan *et al.*, 2024; Wang *et al.*, 2022, 2024] and apply custom fine-tuning techniques for binary code matching. However, most of these approaches rely on either closed-source models like GPT [Radford, 2018] or large model sizes. This leads to scalability issues when the computational resource is a constraint, which realistically is the case with small research labs or even companies. We want to explore effective and efficient LLM fine-tuning for binary code embedding and matching.

In this work, we address the aforementioned problems and propose **EBM** (**E**ffective **B**inary **M**atching), our novel training framework, including carefully chosen data augmentation and fine-tuning processes, to uptrain a generic LLM into a binary code embedding and matching expert. We show in our experiments that LLMs have become dominant enough that even zero-shot models can surpass well-trained binary code models. With our fine-tuning applied, EBM can significantly increase the mean reciprocal rank (MRR) by 10% to 70%, depending on the tasks. We also provide a comprehensive ablation study to prove and emphasize the importance of each training process. The dataset and code can be accessed on Github [1]. Our major contributions are:

- We propose a multi-training framework to utilize the power of generic LLMs specifically for BCSD. We target different compiler settings, including cross-optimization, cross-architecture, and cross-obfuscation, to build a general and effective similarity retriever.
- To combat the lack of assembly code LLMs, we uptrain the generic LLM to a binary code-specific model. Particularly in cross-architecture similarity detection, this allows for significantly better translation between different syntaxes.
- We utilize LLM2Vec to build a refinement of the assembly tokens, which encodes better semantics through a masked next token prediction task.
- In the downstream contrastive learning task, we propose an enhanced version of InfoNCE loss to utilize all available samples within the batch. This is particularly useful when computing resources are limited and large models are trained.
- We build and compare various baselines to evaluate the effect of different language/coder models and state-of-the-art BCSD models. In our similarity retrieval evaluation, our approach outperforms all benchmarks for both datasets.
- We conduct thorough ablation studies and in-depth analysis to investigate all our training tasks and how they contribute to the similarity retrieval result. We show that all our training tasks are essential and can improve retrieval performance.

2 Related Works

Traditional BCSD Without data-driven or learning-based methods, code similarity detection is traditionally conducted using static analysis, dynamic analysis, or code-based algorithms. Static analysis usually involves graph matching [Dullien and Rolles, 2005; Bourquin *et al.*, 2013], where control flow graphs are extracted from assembly code and compared using algorithms or user-defined heuristics. Dynamic analysis instead leverages runtime or symbolic execution to investigate program behavior [Pewny *et al.*, 2015; Xu *et al.*, 2020; Egele *et al.*, 2014; Moser *et al.*, 2007]. Execution traces or paths can be profiled for either manual or automated comparison using distance-based or statistics-based methods. Code-based algorithms rely on patterns of the actual binary code, which can be opcode or instruction code. Various ways of analyzing code strings include distance metrics (Smith-Waterman or Levenshtein distance) [Gao *et al.*, 2008], N-Gram matching [Rosenblum *et al.*, 2008], and frequency analysis [Santos *et al.*, 2013].

Machine Learning-based BCSD is effective when a large amount of training data is available. The methods can generally be categorized into text-based, structure-based, or combined. Text-based models treat assembly instructions or opcodes as tokens and feed them into various models such as unsupervised [Ding *et al.*, 2019], RNN-based [Massarelli *et al.*, 2019b; Yang *et al.*, 2021; Tian *et al.*, 2021], BERT-based [Koo *et al.*, 2021; Ahn *et al.*, 2022; Li *et al.*, 2023a], and LLM-based [Tan *et al.*, 2024; Wang *et al.*, 2022; Liu *et al.*, 2023; Wang *et al.*, 2024]. Such models focus on the semantics

[1] Github Link

and syntax of the binary code to encode binary programs. In structure-based models, some form of graph structures, such as abstract syntax trees, control flow graphs, or custom graphs, is parsed from binary code. Then appropriate machine learning techniques can be applied, including tree-LSTM [Tai *et al.*, 2015; Yang *et al.*, 2021] and graph neural network [Xu *et al.*, 2017; Li *et al.*, 2019]. Instead of utilizing complicated language models for word embedding, these models often apply Word2Vec-style [Mikolov *et al.*, 2013] representation learning to obtain node features. Other research works combine code and structure information into a streamlined learning process [Massarelli *et al.*, 2019a; Gao *et al.*, 2018; Yu *et al.*, 2020]. When training the networks, contrastive learning is the most popular technique for supervised learning models to differentiate positive and negative samples, which are fed into the network as a pair of input data. Many approaches also utilize a siamese architecture to simultaneously encode pairs under the same set of model parameters. Recently, many similarity detection methods have adopted the InfoNCE loss [Oord *et al.*, 2018] to enhance the discriminative power of negative pairs by maximizing the mutual information. It has been shown to be quite effective and can sometimes lead to large performance gains, especially for LLMs where batch size tends to be small. Readers may refer to Appendix. A for additional information about LLM architecture and foundation models.

3 Methodologies

We define the task of BCSD as retrieving the most similar function from a pool of binary functions. Similar binary functions are created from the same source code with different compiler settings. In our case, the settings can be optimization-based, obfuscation-based, architecture-based, and compiler-based. The input is a binary function x and the output is a list of functions, ranked by cosine similarity. For evaluation purposes, the pool contains exactly one similar function. The complete training undergoes four processes: data augmentation, translation learning, embedding training, and contrastive learning.

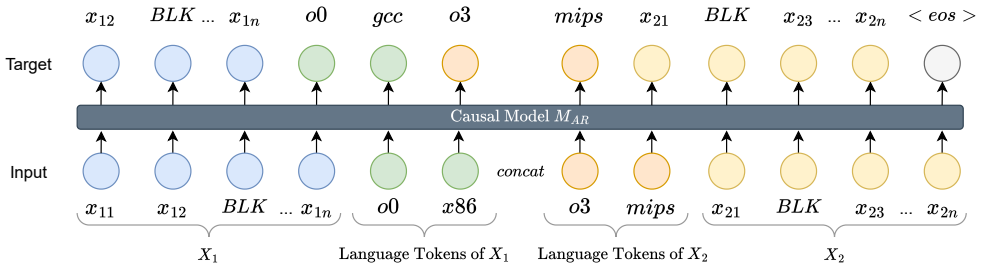


Figure 1: The input and target of causal uptraining are shown here. The input contains sampled pairs of semantically identical assembly functions, where **BLK** tokens are added for structure awareness, and language tokens are added for translation awareness. The training follows an autoregressive manner, where previous tokens are used to predict the next unseen token.

3.1 Enabling Translation and Structure Awareness

Assembly code is an intermediate code acquired from a disassembler, such as IDAPRO [2]. To clean the data, all addresses, strings, and bytes are replaced by special tokens: **addr, byte, str**. The special tokens help reduce noise when addresses or bytes are arbitrary and also reduce context length after tokenization. We flatten the original data structure of assembly code, which contains basic blocks and instruction streams, into a single sentence. Doing this removes the requirement for hierarchical architecture and layered attention to process the inputs. This allows us to directly use existing pre-trained language and coder models. The drawback is that a single sentence no longer contains structural information regarding the semantically bound sections of the code (i.e., basic blocks). We propose to simply add a special **BLK** token in between basic blocks to increase structure awareness during training. This enables the model to consider assembly structure, which can be easily learned in our uptraining processes.

[2]https://hex-rays.com/ida-pro

Cross-architecture similarity detection is a difficult task that, to our knowledge, has not been extensively studied along with other compiler settings like cross-optimization. Assembly code can have a completely different syntax based on the machine architecture during compilation. Different architectures can be seen as different “languages" in this context. Inspired by [Conneau and Lample, 2019], we add several additional tokens to indicate the “language" information of each binary function, which corresponds to the optimization level, compiler, obfuscation, and architecture. Adding such information has proven to be effective, especially for cross-architecture detection.

3.2 Binary Translation Continual Training

Modern coder models aim to assist humans in coding tasks like code generation and summarization. Although assembly code is included in some of the existing models, it is not as well understood to the extent of source code languages like Python or Java. A cheap and effective way to uptrain such coder models with assembly code is to use an autoregressive setup, which trains the model to predict the next tokens.

We treat the task similarly to translation in natural language, as different compilers produce various syntaxes depending on the settings. This is especially true in cross-architecture similarity detection, where it is often not possible to retrieve similar code between machines with architectures like x86 and PowerPC. We only sample semantically identical function pairs (X_1, X_2) and concatenate them into a single sequence $X = concat(X_1, X_2)$ as input to the model. A pre-trained coder AR model is initialized with weights W^0 and fine-tuned with the causal objective and update rule:

$$\mathcal{L}_{\mathcal{AR}} = -\sum_{i=1}^{n} \log P(x_i \mid x_{<i}) \tag{1}$$

$$W^{(t+1)} = W^{(t)} - \eta \cdot \nabla_W \mathcal{L}_{\mathcal{AR}}(X; W^{(t)}) \tag{2}$$

Where η is the learning rate. As discussed in section 3.1, we manually add “language" tokens to all assembly functions as an indication of their compiler settings. For training the translation task, we place these tokens in between the concatenation of function pairs, acting as a transition. We give an illustration of the input and output format in Figure 1. It is important to let the model access these tokens during training before predicting the second sentence, X_2, as it provides generalization due to knowing what settings to expect and predict. It is important to reiterate that these language tokens are unavailable during the inference phase.

3.3 LLM2Vec

BERT [Devlin, 2018] models have been proven effective in embedding tasks, largely due to their bidirectional attention mechanism, which enables them to capture contextual information from the whole sequence [Reimers, 2019; Shi *et al.*, 2023]. However, the landscape of NLP has shifted significantly in recent years to LLMs trained with next token prediction, such as GPT-2 [Radford *et al.*, 2019], Llama [Touvron *et al.*, 2023], and their successors. These models, typically based on unidirectional architectures, have achieved state-of-the-art performance across a wide range of tasks, including embedding tasks, often surpassing traditional bidirectional models in scalability and generalization [Wang *et al.*, 2023a].

$$\mathcal{L}_{\text{MNTP}} = -\sum_{i}^{masked} P(x_i) \log P(\hat{x}_i | x_1, ..., x_n) \tag{3}$$

To bridge the gap between the strengths of causal models and the benefits of bidirectional architectures, we adopt an innovative approach called LLM2Vec [BehnamGhader *et al.*, 2024]. As depicted in Figure 2, this method leverages the pre-trained knowledge of causal models, while enabling bidirectional attention and being trained through a masked next token prediction task (MNTP). Specifically, we initialize the model with the same weights from a pre-trained causal model and enable bidirectional attention during forward propagation. The model is then fine-tuned using MNTP loss, defined in Equation 3, where a random token in the sequence is masked and predicted based on the surrounding context.

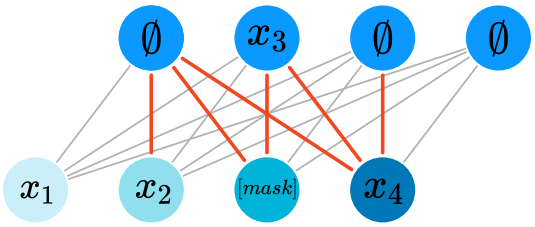


Figure 2: An overview of the LLM2Vec approach. The method adds bidirectional attention (red lines) based on future context to the original causal model (gray lines). It is fine-tuned with masked next token prediction (MNTP). The model can be further fine-tuned with contrastive learning approaches such as InfoNCE and GTE losses, for specific downstream tasks.

In essence, this approach constructs a BERT-like model from a pre-trained causal model rather than training a BERT model from scratch. This strategy offers several key advantages. First, it allows us to leverage the existing pre-trained LLMs, reducing the need for costly and time-consuming pre-training. Most importantly, it achieves better performance than training a BERT-style model from scratch.

3.4 Cumulative GTE Loss

The InfoNCE loss (Equation 4) is a widely used objective function in contrastive learning. It encourages the model to learn distinctive representations by contrasting one positive pair against in-batch negative ones. Incorporating additional contrastive pairs can further enhance the learning process [Li *et al.*, 2023b; Wang *et al.*, 2022]. We denote k_i to be the embedding of the source binary function (key) in batch i, q_i to be the embedding of the target binary function (query) in batch i, and τ as a hyperparameter for temperature.

$$\mathcal{L}_{\text{InfoNCE}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\text{sim}(q_i,k_i)/\tau)}{\sum_{j=1}^{N}\exp(\text{sim}(q_i,k_j)/\tau)} \tag{4}$$

The GTE (general text embedding) loss generalizes the InfoNCE loss by introducing additional contrastive terms, which include query-key pairs as well as query-query, key-key, and key-query pairs. This broader set of contrasts helps the model capture richer relationships within the data. The GTE loss is defined as:

$$\begin{aligned}\mathcal{L}_{\text{GTE}} &= -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\text{sim}(q_i,k_i)/\tau)}{Z}\\ Z &= \sum_{j} e^{\text{sim}(q_i,k_j)/\tau} + \sum_{j} e^{\text{sim}(q_i,q_j)/\tau}\\ &+ \sum_{j\neq i} e^{\text{sim}(q_j,k_i)/\tau} + \sum_{j\neq i} e^{\text{sim}(k_j,k_i)/\tau}\end{aligned} \tag{5}$$

Based on the GTE loss, we propose the cumulative GTE loss (cGTE), which aggregates embedding representations from multiple distributed models and accumulated inputs. The cGTE loss is formulated as:

$$\begin{aligned}q &= q^{(1)}||q^{(2)}||...||q^{(n)}\\ k &= k^{(1)}||k^{(2)}||...||k^{(n)}\\ \mathcal{L}_{\text{cGTE}} &= \text{GTE}(q,k)\end{aligned} \tag{6}$$

where the superscript denotes accumulated vector representation from distributed models or batches. The backpropagation is halted until a number of inputs are met. The cGTE loss syncs gradient backpropagation generated from asynchronous input batches and distributed models to the same model weights. cGTE creates more contrastive pairs with limited computing resources.

4 Training and Experiments Setup

4.1 Backbone Model and Baselines

We use Qwen2.5-Coder-0.5B [Hui *et al.*, 2024] as the backbone model for all training processes. We compare against existing non-LLM methods, including PalmTree [Li *et al.*, 2021], SAFE [Massarelli *et al.*, 2019b], OrderMatters [Yu *et al.*, 2020], and Asm2Vec [Ding *et al.*, 2019]. We also evaluate several state-of-the-art large coder models, such as GraphCodeBERT [Guo *et al.*, 2020], CodeT5+ [Wang *et al.*, 2023b], Qwen2.5-Coder-1.5B [Hui *et al.*, 2024], Qwen3 [Yang *et al.*, 2025], and CodeGemma [Team *et al.*, 2024]. All baseline models, except for Qwen2.5-Emb, are fine-tuned on the same training set using contrastive learning. Qwen2.5-Emb, being a pre-trained embedding model, is included as a baseline to indicate the differences between programming language code and binary code. As our model is trained on Qwen2.5-Coder-0.5B, the Qwen2.5-Coder-1.5B baseline is a good candidate for an ablation study.

4.2 Dataset

We use two datasets for training and evaluation. The first contains multiple libraries written in C and is used by other existing works [Ding *et al.*, 2019]. We manually compile each library using different optimization levels (O0, O1, O2, and O3), compilers (GCC and Clang), architectures (x86, PowerPC, Arm, and MIPS), and obfuscations (none, substitution, flatten, bogus control flow, and all). The training libraries used include BusyBox, Coreutils, Curl, ImageMagick, PuTTY, and SQLite. The evaluation libraries are GMP, LibTomCrypt, and OpenSSL. We separate the libraries for the purpose of out-of-domain evaluation, which is a common practice in binary code retrieval. The pool size for the retrieval evaluation is 1,000. The second dataset is BinaryCorp [Wang *et al.*, 2022], which is constructed based on ArchLinux[3] and Arch User Repository[4]. BinaryCorp only contains cross-optimization functions. We use the same training and testing splits and pool size of 10,000 as the original paper for comparison.

4.3 Training Details

We train our models on a machine with 4 NVIDIA RTX 6000 GPUs (24 GB). Each computing card can contain one batch of 4 entries with 512 tokens as the sequence length. We use AdamW as the training optimizer and a learning rate of 10^{-5}. Every model is trained with 5 epochs and warmed up for $1000/[\text{batch size}]$ steps. The best model weights are based on the best validation contrastive loss.

5 Evaluation

In this section, we evaluate both datasets with LLM-based and non-LLM baselines. The evaluation metrics are mean reciprocal rank (MRR) and recall@1. Due to limited space and a large number of combinations for our evaluation setup, we only include a subset of the retrieval tasks. The remaining evaluation details, including additional tasks, training, and inference time, are in the Appendix B.

Models	**MRR**						**Recall@1**					
	O0,O3	O0,O1	O0,O2	O1,O3	O2,O3	Avg.	O0,O3	O0,O1	O0,O2	O1,O3	O2,O3	Avg.
SAFE	0.189	0.189	0.200	0.218	0.171	0.193	0.063	0.000	0.063	0.063	0.000	0.038
PalmTree	0.023	0.020	0.019	0.314	0.878	0.251	0.008	0.006	0.007	0.184	0.676	0.176
Asm2Vec	0.444	0.494	0.460	0.535	0.563	0.499	0.234	0.290	0.252	0.343	0.376	0.299
OrderMatters	0.006	0.006	0.008	0.006	0.006	0.006	0.000	0.001	0.002	0.001	0.000	0.001
GraphCodeBERT (125M)	0.636	0.757	0.673	0.792	0.920	0.756	0.560	0.694	0.602	0.722	0.895	0.695
CodeT5+ (110M)	0.604	0.650	0.629	0.830	0.893	0.721	0.532	0.572	0.552	0.783	0.869	0.662
Qwen2.5-Emb (1.5B)	0.569	0.648	0.573	0.773	0.907	0.694	0.498	0.578	0.505	0.699	0.875	0.631
Qwen2.5-Coder (1.5B)	0.758	0.881	0.807	0.864	0.936	0.849	0.706	0.842	0.757	0.810	0.912	0.805
CodeGemma (2B)	0.763	0.888	0.833	0.866	0.931	0.856	0.696	0.840	0.778	0.821	0.905	0.808
EBM (0.5B)	**0.850**	**0.942**	**0.902**	**0.933**	**0.955**	**0.916**	**0.793**	**0.903**	**0.850**	**0.887**	**0.929**	**0.872**

Table 1: Evaluation on cross-optimization settings (O0, O1, O2, and O3) with a pool size of 1,000.

[3]https://archlinux.org/packages/

[4]https://aur.archlinux.org/

5.1 Cross-optimization Evaluation

Table 1 highlights the performance comparison between various baselines and our approach for cross-optimization retrieval. LLMs, even with a zero-shot setting, can outperform existing non-LLM methods, which have been state-of-the-art in the past for binary similarity detection. However, they still struggle with more difficult retrieval tasks like **[O0, O3]**. EBM can consistently outperform other models across all optimization settings, achieving the highest MRR and Recall@1 scores, while also improving **[O0, O3]** significantly.

5.2 Cross-architecture

Cross-architecture has been shown to be the most difficult task for both LLMs and non-LLM baselines, as shown in Table 2. Even with contrastive fine-tuning, Qwen-2.5-Coder, which has 1.5 billion parameters, fails to effectively retrieve cross-architecture functions. Due to the difference in syntax, matching the semantics across several languages is nearly impossible with limited training. Our method outperforms all baselines by a large margin. This further proves the significance of the causal training process that contributes to the improvement in cross-architecture generalization. We will show firm results in ablation studies.

Models	**MRR**				**Recall@1**			
	Arm, x64	PowerPC, x64	MIPS, x64	Avg.	Arm, x64	PowerPC, x64	MIPS, x64	Avg.
SAFE	0.239	0.187	0.196	0.208	0.063	0.063	0.063	0.063
PalmTree	0.037	0.036	0.018	0.031	0.031	0.013	0.007	0.017
Asm2Vec	0.242	0.293	0.417	0.317	0.085	0.113	0.231	0.143
OrderMatters	0.007	0.007	0.007	0.007	0.002	0.000	0.001	0.001
GraphCodeBERT (125M)	0.067	0.269	0.495	0.277	0.037	0.204	0.419	0.220
CodeT5+ (110M)	0.056	0.303	0.462	0.274	0.035	0.227	0.392	0.218
Qwen2.5-Emb (1.5B)	0.039	0.059	0.409	0.169	0.031	0.035	0.331	0.132
Qwen2.5-Coder (1.5B)	0.256	0.481	0.548	0.428	0.179	0.380	0.442	0.334
CodeGemma (2B)	0.293	0.581	0.548	0.474	0.208	0.479	0.432	0.373
EBM (0.5B)	**0.783**	**0.792**	**0.859**	**0.811**	**0.675**	**0.703**	**0.784**	**0.721**

Table 2: Evaluation on cross-architecture settings (Arm, x86-64, PowerPC, and MIPS) with a pool size of 1,000.

5.3 Cross-obfuscation

Obfuscation techniques can introduce complex and confusing variants to a binary function and are considered the most difficult task for binary retrieval. In Table 3, all models have shown undesirable results for **[all, none]**, which retrieves vanilla functions from functions with all three obfuscation techniques applied. Our method still outperforms all other baselines. While investigating the functions in this task, we often found that the obfuscated function can be 10x as large as the vanilla function. With limited sequence length, this result is somewhat expected. We suspect a much longer sequence length can alleviate this constraint and improve performance.

5.4 BinaryCorp

Table 4 shows the evaluation of the BinaryCorp dataset. Among all benchmarks, jTrans [Wang *et al.*, 2022] and CLAP [Wang *et al.*, 2024] are recent LLM-based models for BCSD. They have shown great results compared to the other methods. CLAP has the closest performance compared to our method, though has a few drawbacks. Firstly, CLAP relies on GPT3.5, a closed-source model, to generate natural language explanations. It is also costly and difficult to scale the model. CLAP uses Llama 13B and 30B as the backbone model for supervised fine-tuning, which can become a constraint for resource-limited scenarios.

In conclusion, EBM is a general approach that can adapt any pre-trained coder model, such as the 0.5B model used in our study, for binary embedding matching tasks. This methodology eliminates the need for external expert models, such as GPT, or auxiliary inputs, such as control flow graphs. Thus, it simplifies the data pipeline and training process and makes it more accessible and easy to use.

Models	MRR				Recall@1			
	all, none	none, bcf	sub, fla	Avg.	all, none	none, bcf	sub, fla	Avg.
SAFE	0.256	0.181	0.264	0.234	0.0625	0.0625	0.125	0.083
PalmTree	0.122	0.289	0.215	0.209	0.060	0.200	0.083	0.114
Asm2Vec	0.200	0.181	0.264	0.215	0.069	0.063	0.125	0.086
OrderMatters	0.008	0.006	0.007	0.007	0.001	0.001	0.001	0.001
GraphCodeBERT (125M)	0.230	0.648	0.479	0.452	0.163	0.557	0.391	0.370
CodeT5+ (110M)	0.176	0.619	0.372	0.389	0.118	0.539	0.291	0.316
Qwen2.5-Emb (1.5B)	0.288	0.630	0.466	0.461	0.213	0.538	0.375	0.375
Qwen2.5-Coder (1.5B)	0.391	0.719	0.580	0.563	0.301	0.637	0.491	0.476
CodeGemma (2B)	0.454	0.796	0.571	0.607	0.356	0.735	0.473	0.521
EBM (0.5B)	**0.531**	**0.815**	**0.784**	**0.710**	**0.454**	**0.738**	**0.713**	**0.635**

Table 3: Evaluation on cross-obfuscation settings (all obfuscations, bogus control flow, flattened, and substitution) with a pool size of 1,000. The proposed EBM model outperforms all baselines by an absolute margin of over 15% in both MRR and Recall@1 metrics.

Models	MRR						Recall@1					
	O0,03	O1,O3	O2,O3	O0,Os	O2,Os	**Avg.**	O0,03	O1,O3	O2,O3	O0,Os	O2,Os	**Avg.**
Gemini	0.037	0.161	0.416	0.049	0.195	0.172	0.024	0.122	0.367	0.030	0.151	0.139
SAFE	0.127	0.345	0.643	0.147	0.377	0.328	0.068	0.247	0.575	0.079	0.283	0.250
OrderMatters	0.062	0.319	0.600	0.075	0.233	0.258	0.040	0.248	0.535	0.040	0.158	0.204
Asm2Vec	0.072	0.449	0.669	0.083	0.510	0.357	0.046	0.367	0.589	0.052	0.426	0.296
PalmTree	0.130	0.403	0.677	0.152	0.496	0.372	0.080	0.326	0.609	0.097	0.420	0.306
jTrans (Zero Shot)	0.137	0.490	0.693	0.182	0.513	0.403	0.088	0.412	0.622	0.122	0.430	0.335
jTrans (Finetune)	0.475	0.663	0.731	0.539	0.664	0.614	0.376	0.580	0.661	0.443	0.585	0.529
CLAP	0.764	0.903	0.941	**0.813**	0.877	0.860	0.719	0.875	0.920	0.774	0.847	0.827
EBM	**0.779**	**0.911**	**0.955**	0.808	**0.909**	**0.872**	**0.725**	**0.882**	**0.942**	**0.808**	**0.889**	**0.849**

Table 4: Evaluation on the BinaryCorp-3M dataset with pool size=10,000

5.5 Similarity Threshold

In the preceding sections, we assumed the presence of at least one matching pair in the search pool. However, in real-world scenarios, top-1 selection often yields high false positive rates. To address this limitation, we introduce a similarity threshold based on the distribution of similarity scores in the training data. Table 5 reports the mean and standard deviation for both similar and non-similar optimization pairs. We set the threshold at 'mean - std' for each model. The accuracy is then computed as the proportion of queries without matching pairs that correctly fall below this threshold, formulated as "# of queries with no matching pair / total number of test queries." EBM continues to outperform the best baseline model. The results for the remaining experiments are in the Appendix.

Source	Dest	Accuracy		Mean ± STD of Similar pairs		Mean ± STD of Non-similar pairs	
		CodeGemma	EBM	CodeGemma	EBM	CodeGemma	EBM
o0	o1	0.472	0.719	0.904 ± 0.136	0.910 ± 0.106	0.207 ± 0.220	0.180 ± 0.180
o0	o2	0.387	0.676	0.868 ± 0.155	0.879 ± 0.120	0.208 ± 0.212	0.178 ± 0.172
o0	o3	0.236	0.458	0.830 ± 0.195	0.833 ± 0.173	0.210 ± 0.213	0.184 ± 0.174
o1	o2	0.675	0.831	0.937 ± 0.092	0.952 ± 0.076	0.176 ± 0.213	0.164 ± 0.173
o1	o3	0.438	0.675	0.899 ± 0.151	0.916 ± 0.134	0.168 ± 0.221	0.170 ± 0.176
o2	o3	0.618	0.793	0.955 ± 0.139	0.965 ± 0.133	0.183 ± 0.217	0.174 ± 0.173

Table 5: Model Accuracy for non-match and Similarity Metrics Comparison

5.6 Comparison of Base and EBM-trained Models

To further assess the effectiveness of EBM, we perform the same evaluation on the base model for similarity detection with Qwen2.5-0.5B and Qwen3-1.7B models to understand the difference between pre- and post-finetuning. Table. 6 illustrates the average MRR and Recall@1 across cross-optimization, cross-architecture, and cross-obfuscation. Without training, the base model struggles to

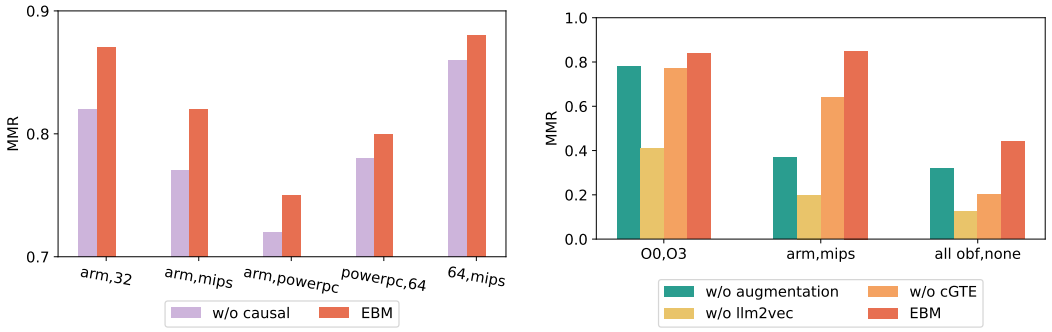
understand the semantics of assembly code, especially for cross-architecture and cross-obfuscation tasks. EBM clearly enables significant uplift for similarity detection and boosts the MRR up to 16X and Recall@1 up to 30X.

Models	Average MRR			Average Recall@1		
	Cross-Optimization	Cross-Obfuscation	Cross-Architecture	Cross-Optimization	Cross-Obfuscation	Cross-Architecture
Qwen2.5-Coder-0.5B (Base)	0.485	0.226	0.050	0.444	0.184	0.026
Qwen2.5-Coder-0.5B (EBM)	0.924	0.738	0.816	0.883	0.668	0.733
Qwen3-1.7B (Base)	0.284	0.023	0.028	0.228	0.011	0.015
Qwen3-1.7B (EBM)	0.917	0.700	0.826	0.884	0.627	0.761

Table 6: Evaluation on the Base Model and Finetuned Model

6 Ablation

We illustrate ablation studies that investigate the effects of data augmentation and fine-tuning processes. All parts of our method must be included, as they provide either generalization or significant performance improvement for a tuned LLM to understand the context, semantics, and structures of assembly code.



(a) Causal training ablation study

(b) Data augmentation, LLM2Vec, and cGTE ablation.

Figure 3: (3a) shows the MRR for two versions of EBM, with and without causal training. Cross-architecture retrieval is particularly sensitive to causal training, as it enables better translation between different languages. In (3b), EBM significantly improves the MRR compared to ablated models. LLM2Vec contributes the most to the increase.

6.1 Assembly Code Data Augmentation

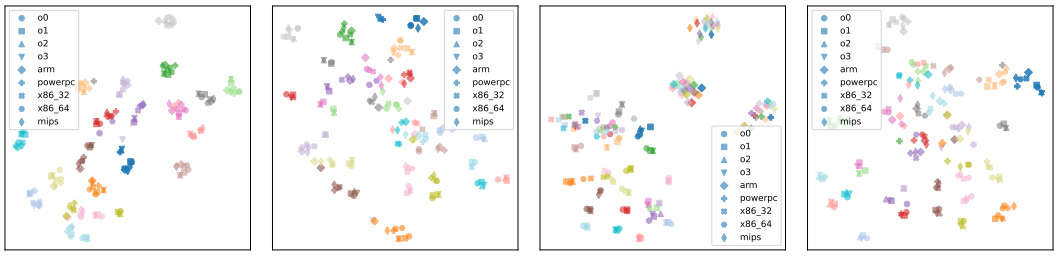
We provide an ablation study on the data augmentation process, which includes data cleaning and adding special tokens for translation and structure awareness. The details can be found in Section 3.1. In Figure 3b, we plot the MRR when data augmentation is removed. It has a major improvement in cross-architecture and cross-obfuscation retrieval. Since these tasks are considered more difficult than cross-optimization retrieval, enhancing the quality of input data with structural and language information is a key process. The assembly code also becomes less noisy after the tokenization, which is often a constraint for cross-obfuscation detection.

6.2 Causal Training

Causal uptraining is the first process to fine-tune a generic LLM. In our analysis, it has an evident impact on improving cross-architecture retrieval and less impact on other retrieval tasks. In Figure 3a, we plot the cross-architecture evaluation for the two versions of our model, with and without causal uptraining. The MRR difference varies from 2% to 6%, which is a significant improvement.

6.3 LLM2Vec and cGTE

LLM2Vec is our BERT-like uptraining that converts a pre-trained decoder model to an embedding model by enabling bidirectional attention. cGTE is our enhanced version of InfoNCE loss during



(a) EBM (b) w/o causal (c) w/o LLM2Vec (d) w/o cGTE

Figure 4: t-SNE visualization of embedding distributions. Each color corresponds to a binary function under different settings, represented by distinct shapes. That means, embeddings for the same function should cluster by color instead of shapes.

contrastive learning that introduces additional contrastive terms. Both are crucial to improving the retrieval performance, as shown in Figure 2. We show the difference in MRR for **[O0,O3]**, **[arm,mips]**, and **[all obf,none]**, as they represent cross-optimization, cross-architecture, and cross-obfuscation, respectively. Overall, LLM2Vec contributes the most to the performance gains for our training and can increase MRR by 2 to 4 times. cGTE is useful in complex tasks like cross-obfuscation, where introducing more negative contrastive pairs enables efficient training.

6.4 Embedding distributions

We provide visualizations for the embeddings of similar binary functions. The t-distributed Stochastic Neighbor Embedding (t-SNE) is used to reduce the embedding dimension to 2. In Figure 4, we plot 20 functions that are compiled using different optimization levels and architectures, represented in different shapes. Similar functions are grouped by colors, whereas the compilation setting is grouped by shapes. After dimensionality reduction, EBM produces tight groups that can be easily separated from the others. This illustrates that our final embeddings are effective and contain rich semantics.

7 Discussions and Limitations

LLMs have proven to be powerful for many learning tasks. To bridge such a technological gap in the binary software and security domain, we propose EBM, a multi-phase uptraining framework for binary code embedding and similarity detection. EBM is a flexible framework for any generic language or coder models, and is effective in outperforming all state-of-the-art benchmarks. Our data augmentation is carefully engineered to provide enhanced awareness of the language and structure information. The causal uptraining module leverages such information and pairs of data to translate different compilation settings to specifically enable cross-architecture learning phrased as a translation task. LLM2Vec performs a BERT-like task to learn the context and provide semantic-rich embeddings. Lastly, our custom cumulative GTE loss can efficiently capture more negative relationships during contrastive learning, significantly improving the semantics embedding in a resource-limited training environment. Our comprehensive evaluation shows that EBM outperforms all benchmarks in all tasks. The ablation study further indicates the effectiveness of all training processes.

Limitations Our limitations include the lack of evaluation on larger LLMs (due to our limited computational resources), relatively small training data compared to state-of-the-art coder models, and a lack of potential downstream tasks for binary code, such as malware and software vulnerability detection. We would like to lay a foundation for future work to apply our method or variants of it that address these limitations. We believe that such an approach has large potential in achieving the state-of-the-art going forward.

References

Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 361–374, 2022.

Parishad BehnamGhader, Vaibhav Adlakha, Marius Mosbach, Dzmitry Bahdanau, Nicolas Chapados, and Siva Reddy. Llm2vec: Large language models are secretly powerful text encoders. *arXiv preprint arXiv:2404.05961*, 2024.

Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.

K Clark. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. *Advances in neural information processing systems*, 32, 2019.

Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp)*, pages 472–489. IEEE, 2019.

Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20-22, 2008 Proceedings 10*, pages 238–255. Springer, 2008.

Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 896–899, 2018.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. Semantic-aware binary code representation with bert. *arXiv preprint arXiv:2106.05478*, 2021.

Zhenzhong Lan. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

Li Tao Li, Steven HH Ding, and Philippe Charland. Gental: Generative denoising skip-gram transformer for unsupervised binary code similarity detection. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2023.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*, 2023.

Guangming Liu, Xin Zhou, Jianmin Pang, Feng Yue, Wenfu Liu, and Junchao Wang. Codeformer: A gnn-nested transformer model for binary code similarity detection. *Electronics*, 12(7):1722, 2023.

Yinhan Liu. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 364, 2019.

Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, Roberto Baldoni, et al. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, pages 1–11, 2019.

Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, 2019.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 231–245. IEEE, 2007.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Alec Radford. Improving language understanding by generative pre-training. 2018.

N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. Learning to analyze binary computer code. In *AAAI*, pages 798–804, 2008.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *information Sciences*, 231:64–82, 2013.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2198–2210. IEEE, 2023.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Donghai Tian, Xiaoqi Jia, Rui Ma, Shuke Liu, Wenjing Liu, and Changzhen Hu. Bindeep: A deep learning approach to binary code similarity detection. *Expert Systems with Applications*, 168:114348, 2021.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*, 2023.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 503–515, 2024.

Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 363–376, 2017.

Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387, 2020.

Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236. IEEE, 2021.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1145–1152, 2020.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We believe the scope is accurate and the contributions are reflected in the subsequent sections.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: We discuss the limitations of the work in the Conclusion section.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We disclose the source code to reproduce the result. And the proposed methods are general enough and easy to reproduce.

   Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide instructions in the README.md file in the code repository.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We discuss about the details in Training Details section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All the results clearly show the statistical significance of the experiments.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We include the details about the training setup in this paper.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes, we do.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: There is no societal impact of the work performed.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We citep the original papers for the datasets used in this paper, and we comply with the open-source license.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should citep the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide README, comments alongside the published code.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: We did not use any large language models in this paper.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A More Related Work on Large Language Models

LLM Architectures. In the foundation of LLMs, transformer [Vaswani, 2017] is the architecture used to capture contextual relationships for long text sequences. The original transformer contains both encoder and decoder modules, where they work in tandem to encode input sequences into a fixed-length vector and output predictions for a new sequence. Such architecture can be used for machine translation, question answering, etc. Encoder models like BERT [Devlin, 2018] and its variants [Liu, 2019; Lan, 2019; Clark, 2020] perform pre-training using masked language modeling with a large amount of text data and generate contextual representations. BERT-style models use bi-directional information that conditions the entire sentence. The generated embeddings can be used for transfer learning or supervised fine-tuning in other downstream tasks. Decoder autoregressive (AR) models such as GPT [Radford, 2018] and Llama [Touvron *et al.*, 2023] perform the next token prediction based on previously seen tokens and generate new sequences.

Foundation Coder Models are trained on multiple programming languages and large codebases. They are capable of code generation, summarization, and translation. A list of modern open source coder models include Code Llama [Roziere *et al.*, 2023], codeT5 [Wang *et al.*, 2021], Qwen-coder [Hui *et al.*, 2024], GraphCodeBERT [Guo *et al.*, 2020], and many more. However, most of coder models are only trained using source code such as Python and Java. Assembly code has not attracted great attention from LLMs, thus some form of transfer learning is required to learn both the syntax and semantics of assembly code.

B Additional Results

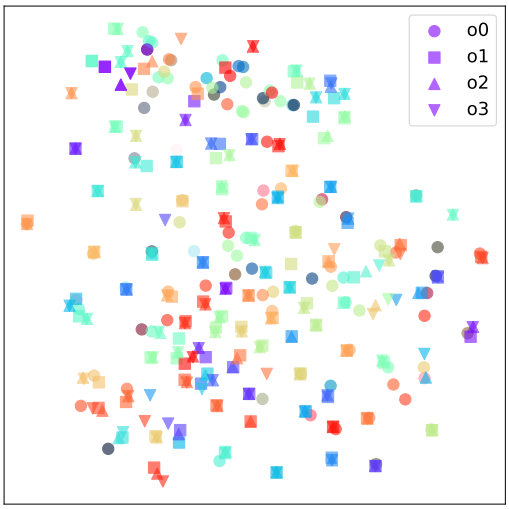
This section contains additional results for cross-compiler (Table 7), cross-architectures (Table 9), and cross-obfuscation (Table 11). We also plot the t-SNE distributions for all LLM models in Figure 5.

Models	**MRR**	**Recall@1**
	clang,gcc	clang,gcc
SAFE	0.200	0.063
PalmTree	0.423	0.178
Asm2Vec	0.523	0.328
OrderMatters	0.007	0.001
GraphCodeBERT (125M)	0.611	0.540
CodeT5+ (110M)	0.635	0.569
Qwen2.5-Emb (1.5B)	0.667	0.593
Qwen2.5-Coder (1.5B)	0.867	0.807
EBM (0.5B)	**0.946**	**0.912**

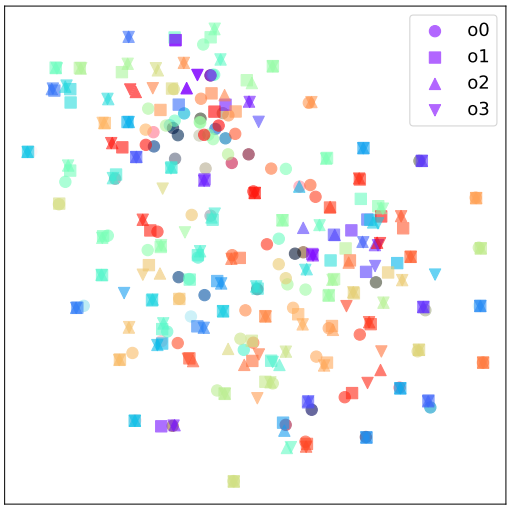
Table 7: Experimental results for cross-compiler between Clang and GCC. As a fairly easy task, Qwen2.5-Emb achieves 0.667 MRR with a zero-shot setting. EBM outperforms all the baselines by over 10%.

Models	**MRR**						
	Arm, PowerPC	Arm, x32	Arm, MIPS	PowerPC, x32	PowerPC, MIPS	x32, x64	x32, MIPS
SAFE	0.180	0.191	0.189	0.165	0.110	0.196	0.152
PalmTree	0.021	0.008	0.019	0.008	0.028	0.018	0.013
Asm2Vec	0.270	0.236	0.223	0.265	0.240	0.417	0.307
OrderMatters	0.008	0.008	0.007	0.008	0.006	0.007	0.008
GraphCodeBERT (125M)	0.094	0.075	0.064	0.256	0.215	0.495	0.239
CodeT5+ (110M)	0.057	0.043	0.048	0.287	0.259	0.461	0.259
Qwen2.5-Emb (1.5B)	0.014	0.030	0.032	0.142	0.095	0.409	0.079
Qwen2.5-Coder (1.5B)	0.269	0.319	0.374	0.490	0.480	0.757	0.522
EBM (0.5B)	**0.718**	**0.818**	**0.770**	**0.836**	**0.803**	**0.906**	**0.883**

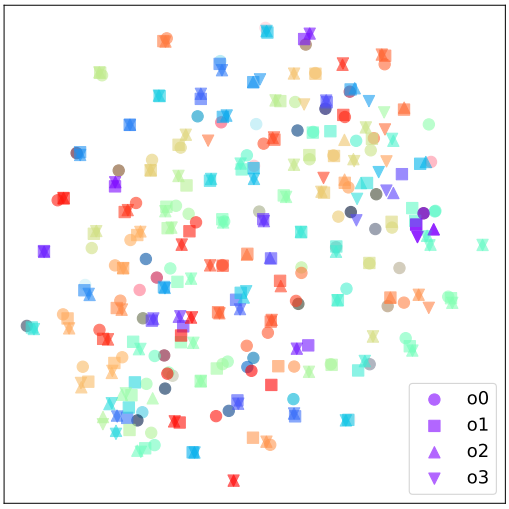
Table 8: Additional MRR results for cross-architecture settings.



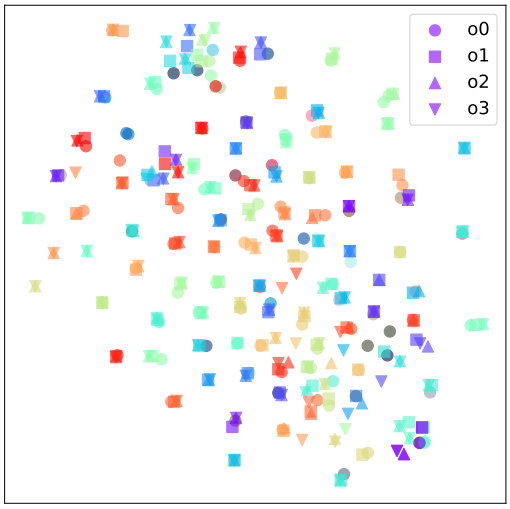
(a) CodeT5+



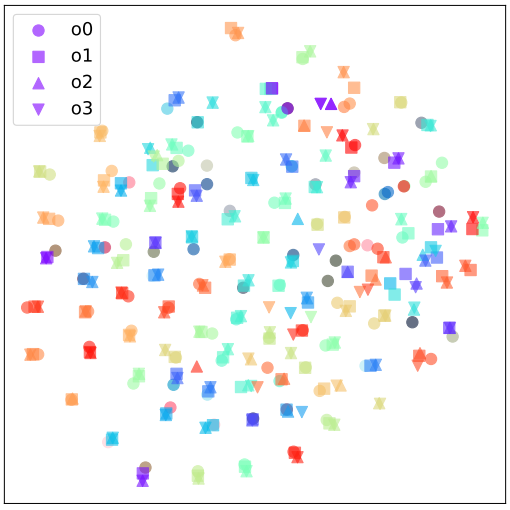
(b) Qwen2.5-Emb (zero-shot)



(c) GraphCodeBERT



(d) Qwen2.5-Coder



(e) EBM

Figure 5: t-SNE visualization of embedding distributions of 100 functions generated by LLM models. Each color corresponds to a binary function under different optimization settings, represented by distinct shapes. Embeddings for the same function should cluster by color instead of shapes. Note that some colors are visually similar but represent different functions. It is evident that the embedding representations generated by EBM distribute more evenly in the space than in all others.

Models	Recall@1						
	Arm, PowerPC	Arm, x32	Arm, MIPS	PowerPC, x32	PowerPC, MIPS	x32, x64	x32, MIPS
SAFE	0.063	0.000	0.063	0.000	0.000	0.063	0.000
PalmTree	0.001	0.001	0.001	0.001	0.001	0.007	0.000
Asm2Vec	0.098	0.080	0.072	0.085	0.080	0.231	0.114
OrderMatters	0.001	0.002	0.000	0.001	0.000	0.000	0.000
GraphCodeBERT (125M)	0.059	0.034	0.029	0.187	0.146	0.419	0.183
CodeT5+ (110M)	0.023	0.013	0.019	0.223	0.198	0.392	0.205
Qwen2.5-Emb (1.5B)	0.003	0.010	0.013	0.093	0.054	0.331	0.050
Qwen2.5-Coder (1.5B)	0.189	0.230	0.280	0.395	0.379	0.682	0.414
EBM (0.5B)	**0.620**	**0.726**	**0.668**	**0.771**	**0.727**	**0.846**	**0.809**

Table 9: Additional recall@1 results for cross-architecture settings.

Models	MRR						
	all, sub	all, fla	all, bcf	none, sub	none, fla	sub, bcf	fla, bcf
SAFE	0.209	0.148	0.229	0.312	0.200	0.128	0.178
PalmTree	0.120	0.254	0.144	0.829	0.260	0.276	0.141
Asm2Vec	0.205	0.238	0.259	0.534	0.332	0.340	0.357
OrderMatters	0.008	0.008	0.007	0.007	0.008	0.008	0.008
GraphCodeBERT (125M)	0.239	0.331	0.290	0.932	0.464	0.621	0.447
CodeT5+ (110M)	0.174	0.324	0.263	0.873	0.378	0.585	0.465
Qwen2.5-Emb (1.5B)	0.277	0.337	0.355	0.900	0.466	0.618	0.463
Qwen2.5-Coder (1.5B)	0.374	0.417	0.422	0.953	0.583	0.687	0.540
EBM (0.5B)	**0.522**	**0.704**	**0.614**	**0.989**	**0.815**	**0.817**	**0.790**

Table 10: Additional MRR results for cross-obfuscation settings.

Models	Recall@1						
	all, sub	all, fla	all, bcf	none, sub	none, fla	sub, bcf	fla, bcf
SAFE	0.063	0.000	0.063	0.125	0.063	0.000	0.000
PalmTree	0.069	0.125	0.072	0.599	0.117	0.186	0.073
Asm2Vec	0.075	0.090	0.114	0.336	0.148	0.161	0.176
OrderMatters	0.001	0.001	0.001	0.000	0.002	0.001	0.002
GraphCodeBERT (125M)	0.186	0.268	0.230	0.901	0.384	0.540	0.375
CodeT5+ (110M)	0.127	0.251	0.196	0.839	0.301	0.511	0.374
Qwen2.5-Emb (1.5B)	0.207	0.262	0.284	0.867	0.375	0.540	0.380
Qwen2.5-Coder (1.5B)	0.292	0.340	0.340	0.921	0.504	0.603	0.454
EBM (0.5B)	**0.436**	**0.624**	**0.523**	**0.972**	**0.751**	**0.751**	**0.717**

Table 11: Additional recall@1 results for cross-obfuscation settings.

Model	Average Inference Time per Batch (100 samples) in seconds	Average Training Time per Batch (4 samples) in it/s
CodeT5P(110M)	0.007	13.90
GraphCodeBERT(125M)	0.005	17.78
Qwen2.5(1.5B)	2.000	2.78
CodeGemma2B	2.540	2.12
EBM(0.5B)	0.014	3.32

Table 12: Model Computational Performance Comparison

Source	**Dest**	**Accuracy**		**Mean ± STD of Similar pairs**		**Mean ± STD of Non-similar pairs**	
		CodeGemma	EBM	CodeGemma	EBM	CodeGemma	EBM
obf_all	obf_none	0.207	0.103	0.615 ± 0.232	0.580 ± 0.216	0.141 ± 0.189	0.244 ± 0.156
obf_all	obf_sub	0.179	0.097	0.607 ± 0.235	0.592 ± 0.206	0.149 ± 0.189	0.254 ± 0.158
obf_all	obf_fla	0.057	0.085	0.719 ± 0.185	0.765 ± 0.145	0.221 ± 0.192	0.386 ± 0.172
obf_all	obf_bcf	0.111	0.026	0.651 ± 0.227	0.690 ± 0.166	0.172 ± 0.189	0.347 ± 0.163
obf_none	obf_sub	0.838	0.890	0.943 ± 0.059	0.972 ± 0.065	0.181 ± 0.222	0.210 ± 0.179
obf_none	obf_fla	0.169	0.341	0.763 ± 0.167	0.746 ± 0.161	0.166 ± 0.206	0.230 ± 0.167
obf_none	obf_bcf	0.354	0.320	0.843 ± 0.149	0.774 ± 0.180	0.180 ± 0.218	0.233 ± 0.172
obf_sub	obf_fla	0.120	0.278	0.740 ± 0.172	0.734 ± 0.161	0.161 ± 0.202	0.235 ± 0.165
obf_sub	obf_bcf	0.231	0.265	0.810 ± 0.159	0.762 ± 0.173	0.169 ± 0.210	0.232 ± 0.171
obf_fla	obf_bcf	0.176	0.273	0.766 ± 0.167	0.735 ± 0.141	0.184 ± 0.195	0.291 ± 0.162
clang	gcc	0.529	0.849	0.911 ± 0.105	0.896 ± 0.097	0.200 ± 0.257	0.103 ± 0.164
arm	powerpc	0.014	0.478	0.727 ± 0.173	0.765 ± 0.168	0.278 ± 0.270	0.111 ± 0.160
arm	x86_32	0.014	0.624	0.732 ± 0.149	0.801 ± 0.137	0.290 ± 0.266	0.108 ± 0.166
arm	x86_64	0.011	0.526	0.724 ± 0.173	0.776 ± 0.163	0.273 ± 0.267	0.102 ± 0.161
arm	mips	0.034	0.587	0.757 ± 0.149	0.805 ± 0.148	0.283 ± 0.266	0.094 ± 0.159
powerpc	x86_32	0.111	0.529	0.820 ± 0.157	0.810 ± 0.157	0.307 ± 0.267	0.123 ± 0.173
powerpc	x86_64	0.102	0.424	0.818 ± 0.171	0.797 ± 0.195	0.279 ± 0.270	0.119 ± 0.169
powerpc	mips	0.104	0.474	0.803 ± 0.165	0.792 ± 0.181	0.302 ± 0.261	0.114 ± 0.165
x86_32	x86_64	0.276	0.712	0.902 ± 0.146	0.877 ± 0.148	0.318 ± 0.262	0.121 ± 0.179
x86_32	mips	0.136	0.663	0.797 ± 0.144	0.826 ± 0.140	0.280 ± 0.263	0.103 ± 0.166
x86_64	mips	0.124	0.624	0.800 ± 0.145	0.810 ± 0.143	0.286 ± 0.256	0.103 ± 0.165

Table 13: Additional results for Model Accuracy for non-match and Similarity Metrics Comparison