Instruction Alignment for Binary Code Representation Learning

Huaijin Wang*
Shandong University
huaijinwang@sdu.edu.cn

Shuai Wang
Hong Kong University of Science and Technology
shuaiw@cse.ust.hk

Abstract

Binary code representation learning is a fundamental problem in software security and reverse engineering. Existing methods mainly learn function-level embeddings that capture coarse-grained semantic relationships between binary functions, but they largely ignore fine-grained instruction-level correspondences. This limitation misses valuable supervision signals available from compiler debug information, which can support the learning of more accurate and interpretable binary code representations.

We propose to leverage instruction alignment knowledge to further improve binary code representation learning. Our preliminary study reveals that models finetuned for function-level binary code similarity exhibit substantially better instruction alignment than their pre-trained model, suggesting a strong correlation between instruction alignment and function-level embedding quality. Motivated by this observation, we design a training approach that explicitly incorporates instruction alignment as an auxiliary training objective. Our experiments show that instruction alignment training improves retrieval accuracy and provides more discriminative signal for the model's similarity judgments.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Representation Learning; Binary Code Similarity

ACM Reference Format:
Huaijin Wang and Shuai Wang. 2026. Instruction Alignment for Binary Code Representation Learning. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3832783.3837516

1 Introduction

Binary code representation learning trains models to map binary functions into fixed-dimensional vector embeddings. It has become a cornerstone of modern binary analysis [26, 34, 53]. High-quality binary code embeddings enable a wide range of downstream tasks, including vulnerability detection [9, 38, 75], malware analysis [32, 40, 65], plagiarism detection [68, 75], and software supply chain security [22, 32, 56, 57, 72]. The embeddings' quality directly determines the effectiveness of downstream applications: embeddings that faithfully capture binary code semantics yield better retrieval, classification, and matching performance across the board.

To train such embedding models, existing methods predominantly rely on function-level supervision [9, 16, 26, 39, 40, 53, 55, 58–60, 62, 64, 67, 71, 72]. They compile the same source project under multiple configurations, such as various compilers (e.g., GCC and Clang) and optimization flags (e.g., `O0`–`O3`), and match binary functions across variants using their debug symbols. Two binary functions sharing the same symbol form a positive pair, as they originate from the same source function; otherwise, they form a negative pair [9, 60, 67, 75]. Models are then trained with contrastive objectives, such as triplet loss, to produce embeddings that bring positive pairs close and push negative pairs apart in the embedding space.

While this function-level symbol matching has proven effective, it represents only the coarse-grained knowledge available from the compilation process. Modern compilers produce a wealth of additional information. Most notably, debug information that maps each assembly instruction back to its originating source line. This fine-grained mapping between source and binary code provides rich semantic knowledge that existing approaches have overlooked.

This paper proposes to exploit *instruction alignment* knowledge to improve binary code representation learning. Specifically, we leverage the debug information produced during compilation to establish fine-grained correspondences between individual assembly instructions across different binary variants of the same function. Two assembly instructions originating from the same source line are considered semantically aligned, providing a supervision signal at a much finer granularity than function-level symbol matching.

We first conduct a measurement study to evaluate how well existing models capture instruction-level semantics. We formulate instruction alignment as a retrieval task: given a pair of binary functions compiled from the same source, and given an instruction in one function, we measure the model's ability to retrieve the semantically corresponding instruction in the other function. Our measurement reveals that models finetuned with function-level contrastive learning exhibit substantially better instruction alignment than their pre-trained version, suggesting a strong correlation between instruction-level understanding and embedding quality.

Motivated by this observation, we propose `InsnAlign`, a training approach that explicitly incorporates instruction alignment as an auxiliary objective alongside function-level contrastive learning. By augmenting the standard training pipeline with an InfoNCE loss [43] on instruction-level correspondences, we improve embedding quality while providing interpretable, instruction-level evidence for similarity judgments. The resulting model not only produces more accurate function-level embeddings but also provides interpretable evidence for similarity judgments by identifying which specific instructions correspond to each other. Our contributions are summarized as follows:

*Corresponding author

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837516

- We identify that existing binary code representation learning methods only exploit coarse-grained function-level knowledge and overlook the instruction-level semantic correspondences available from compiler debug information.
- To our best knowledge, we are the first to propose an instruction alignment measurement to evaluate how well binary code embedding models capture fine-grained instruction semantics, formulated as a retrieval task for quantitative evaluation.
- We design a training approach that leverages instruction alignment knowledge to improve binary code embeddings. Our extensive experiments demonstrate that our method improves retrieval performance while providing inspectable evidence that is more discriminative on hard-to-distinguish candidates.

2 Background and Preliminaries

2.1 ML-based BCSA

Binary code similarity analysis (BCSA) is often adopted to measure the binary code representation quality. It aims to determine whether two binary code snippets share similar functionality. This task is essential in scenarios where source code is unavailable, such as analyzing proprietary software [42], detecting known vulnerabilities in firmware [22, 32], and identifying code reuse across binaries [57]. A common definition [53, 55, 60, 64] of the BCSA task is as follows:

Definition. *Given a binary function q and a pool of candidate functions $P = \{f_1, f_2, \ldots, f_N\}$, the goal is to rank all candidates based on their similarity to q, ideally placing the true positive (i.e., the function compiled from the same source) at the top of the list.*

Because the pool of candidate functions can be very huge in real-world usage (e.g., millions of functions across firmware images or software repositories), fast similarity computation is critical [22]. Conventional approaches that rely on symbolic execution [3, 36, 37, 42, 59] or dynamic testing [12, 62] require heavyweight per-pair analysis, making them impractical at scale [9, 55]. In contrast, representation learning techniques support fast comparison inherently: the embedding model $\mathcal{M}$ maps a binary function f to a fixed-dimensional vector $\mathbf{v} = \mathcal{M}(f) \in \mathbb{R}^d$, and similarity between any two functions reduces to a single vector distance computation (e.g., cosine similarity). The sufficiently close embeddings indicate that the two functions are likely compiled from the same source function, while distant embeddings suggest dissimilarity. This paradigm enables efficient retrieval over large candidate pools, as embeddings can be precomputed and indexed [22, 57, 72]. To train such models, a large dataset of binary functions with known semantic relationships is required [26, 60].

2.2 Ground Truth Construction via Compilation

As aforementioned, the core of ML-based BCSA methods is to learn high-quality binary code representations, which requires a large dataset of binary functions with known semantic relationships for training and evaluation. To construct reliable ground truth for training and evaluation, existing works [9, 17, 28, 40, 53, 60, 62, 67] employ open-source software and compile the same source project under diverse configurations. Given a project P with source functions $\{s_1, s_2, \ldots, s_k\}$, the project is compiled with multiple configurations $C = \{c_1, c_2, \ldots, c_m\}$, where each configuration represents a unique combination of compiler (e.g., GCC, Clang), optimization level (e.g., `O0`, `O3`), and target architecture (e.g., x86_64, AArch64).

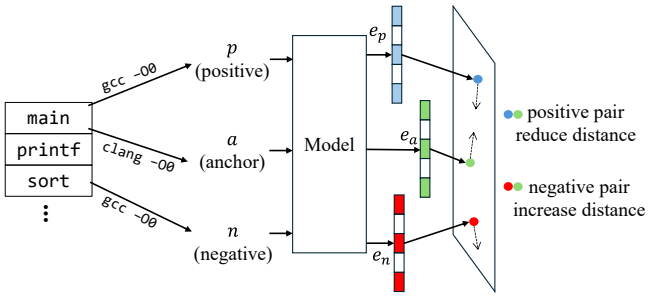


Figure 1: Ground truth construction and training objective. p and a are compiled from "`main`" function, while n is compiled from a different function "`sort`". The training objective encourages the model to learn representations that bring the embeddings of p and a closer together while pushing the embeddings of a and n apart.

Each configuration c_i produces a set of binary functions (i.e., $\{b_1^{c_i}, b_2^{c_i}, \ldots, b_k^{c_i}\}$), where $b_j^{c_i}$ is compiled from source function s_j. The debug symbol of $b_j^{c_i}$ serves as the identifier linking it back to s_j. Two binary functions $b_i^{c_1}$ and $b_i^{c_2}$ sharing the same symbol (i.e., originating from the same source function s_i) form a *positive pair*, while functions with different symbols form *negative pairs* [9, 29, 38, 54, 55, 60, 72]. Fig. 1 shows that the positive pairs (p and a) are built from the same source function "`main`", while the negative pair (a and n) is formed by two functions compiled from different source functions ("`main`" and "`sort`"). Apparently, the positive pairs share the same semantics, while negative pairs are likely dissimilar. Thus, the training objective is to learn representations that bring positive pairs closer together in the embedding space while pushing negative pairs apart.

Although the functions of positive pairs are compiled from the same source function, different configurations can lead to various compiler optimizations for produced assembly instructions, resulting in significant variations in the produced binary code, making the similarity analysis challenging [39, 61]. Thus, a careful design of the training objective is required to learn robust representations that capture the underlying semantics despite the syntactic differences.

2.3 Contrastive Learning for BCSA

With positive and negative pairs established, models are often trained using contrastive learning to produce high-quality representations (i.e., embeddings). Both loss functions described below measure the similarity between embeddings using cosine similarity [9, 53, 60, 67, 72], i.e., $\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$.

The embedding distance is typically defined using the cosine similarity, such as $d(\mathbf{u}, \mathbf{v}) = 1 - \cos(\mathbf{u}, \mathbf{v})$. The training objective encourages the model to produce embeddings where positive pairs have high cosine similarity (i.e., low distance) and negative pairs have low cosine similarity (i.e., high distance).

A commonly used loss function is the triplet loss [48]:

$$\mathcal{L}_{\text{triplet}} = \max\left(0,\ d(e_a, e_p) - d(e_a, e_n) + \alpha\right), \tag{1}$$

where a is an anchor, p is a positive sample, n is a randomly selected negative sample, and α is a margin hyperparameter. The triplet loss penalizes cases where the anchor is closer to a negative sample than to the positive sample by at least a margin α.

2.4 Debug Information and Src-Bin Mapping

When compiling a program with debug flags (e.g., -g), the compiler generates debug information following standards such as DWARF [11]. This debug information contains, among other data, a mapping from each assembly instruction to the source line that produced it. Fig. 2 illustrates this mapping with a simple example. The instructions of compiled binary functions in (b) and (c) can be mapped back to the source lines in (a) using the debug information.

Formally, for a binary function $B = [b_1, b_2, \ldots, b_n]$ compiled from source function s with source lines $\{l_1, l_2, \ldots, l_m\}$, the debug information provides a mapping $\phi : b_i \mapsto l_j$, indicating that assembly instruction b_i was generated from source line l_j.

Given two binary functions B^{c_1} and B^{c_2} compiled from the same source function s under different configurations c_1 and c_2, instructions $b_i^{c_1}$ and $b_j^{c_2}$ are *semantically aligned* if they map to the same source line, i.e., $\phi_{c_1}(b_i^{c_1}) = \phi_{c_2}(b_j^{c_2})$. This provides a fine-grained semantic correspondence at the instruction level, which is significantly richer than the coarse function-level matching used by existing methods.

3 Observation and Motivation

In this section, we present a preliminary study of instruction alignment and the key observations that motivate our approach. We conduct this study on `jTrans` [60], a Transformer-based binary code embedding model that provides both a pre-trained checkpoint and a checkpoint finetuned for function-level BCSA using contrastive learning. We evaluate both checkpoints on an instruction alignment task over function pairs compiled from the same source code under different compilation configurations. §3.1 and §3.2 describe our instruction alignment evaluation methodology, and the results presented in §3.3 motivate this study.

3.1 Instruction Embedding

Fig. 3 illustrates the instruction embedding process. A binary code snippet is represented as a sequence of tokens fed into the transformer, which produces hidden states $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_S] \in \mathbb{R}^{S \times d}$, where S is the sequence length and d is the hidden dimension. Since each assembly instruction typically spans multiple tokens (e.g., opcode and operands), we aggregate token-level representations into instruction-level embeddings. We define an instruction index mapping $\psi : \{1, \ldots, S\} \to \{1, \ldots, K\} \cup \{-1\}$, where K is the number of instructions in the function and -1 indicates special or padding tokens not belonging to any instruction. The embedding of the k-th instruction is computed by mean-pooling the hidden states of all tokens belonging to that instruction:

$$\mathbf{e}_k = \frac{1}{|\mathcal{T}_k|} \sum_{t \in \mathcal{T}_k} \mathbf{h}_t, \quad \text{where} \quad \mathcal{T}_k = \{t \mid \psi(t) = k\}. \tag{2}$$

This process introduces no additional learnable parameters, and the instruction embeddings are derived directly from the transformer backbone.

Given two functions A and B, we obtain their instruction embeddings $\{\mathbf{e}_1^A, \ldots, \mathbf{e}_n^A\}$ and $\{\mathbf{e}_1^B, \ldots, \mathbf{e}_m^B\}$, where n and m are the number of instructions in A and B, respectively. The similarity between any instruction pair $(\mathbf{e}_i^A, \mathbf{e}_j^B)$ can be computed using cosine similarity, which serves as the basis for instruction alignment evaluation.

3.2 Instruction Alignment Measurement

Before leveraging instruction-level knowledge for training, we first investigate how well existing binary code embedding models capture instruction-level semantics. Since binary functions are composed of assembly instructions, a natural question arises: given a pair of functions with identical symbols (i.e., compiled from the same source), can the model's instruction-level embeddings explain *why* these two functions are considered similar?

We formulate instruction alignment as a retrieval task. Given two binary functions $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_m]$ compiled from the same source function under different configurations, we establish an instruction alignment matrix $\mathbf{M} \in \{0, 1\}^{n \times m}$ using the debug information: $\mathbf{M}_{i,j} = 1$ if a_i and b_j are semantically aligned (i.e., $\phi(a_i) = \phi(b_j)$), and $\mathbf{M}_{i,j} = 0$ otherwise. Fig. 2(d) shows an example of such a matrix, where the colored cells indicate the aligned instruction pairs from Fig. 2(b) and 2(c).

We first identify all instructions in A that have at least one semantically aligned counterpart in B:

$$\mathcal{I}^{A \to B} = \{a_i \mid \sum_{j=1}^{m} \mathbf{M}_{i,j} > 0\}. \tag{3}$$

This filtering is necessary because compiler optimizations may eliminate certain instructions or inline callee semantics, leaving some instructions without a counterpart in the other binary function. For each $a_i \in A'$, we rank all instructions in B by the cosine similarity between their embeddings and a_i's embedding, yielding a ranked list B_{a_i}. We define the rank of a_i as the highest position of any aligned instruction:

$$\text{rank}(B_{a_i}) = \min \left\{ \text{pos}(b_j, B_{a_i}) \mid \mathbf{M}_{i,j} = 1 \right\} \tag{4}$$

where $\text{pos}(b_j, B_{a_i})$ denotes the position of b_j in the ranked list B_{a_i}. Intuitively, when an aligned instruction b_j has a similar embedding to a_i, it ranks near the top, resulting in a small $\text{rank}(B_{a_i})$.

We evaluate the alignment quality using standard retrieval metrics, i.e., Mean Reciprocal Rank (MRR) and Recall@1:

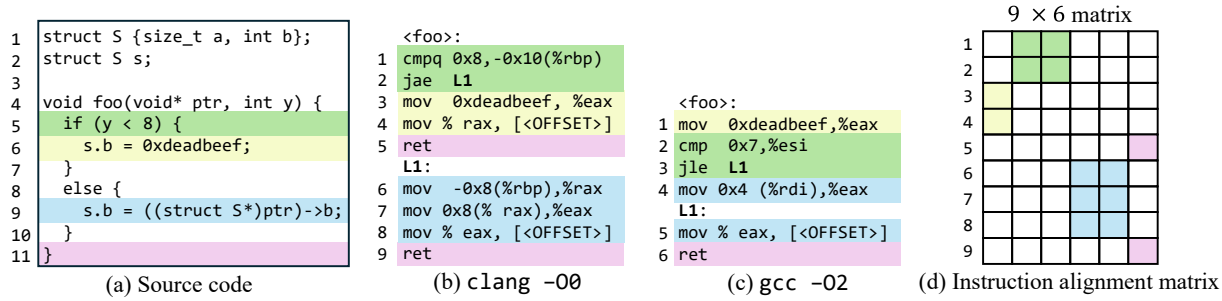
$$\text{Recall}^{\text{insn}}@1 = \frac{1}{|\mathcal{I}^{A \to B}|} \sum_{a_i \in \mathcal{I}^{A \to B}} \mathbb{1}\big(\text{rank}(B_{a_i}) = 1\big) \tag{5}$$

$$\text{MRR}^{\text{insn}} = \frac{1}{|\mathcal{I}^{A \to B}|} \sum_{a_i \in \mathcal{I}^{A \to B}} \frac{1}{\text{rank}(B_{a_i})} \tag{6}$$

where $\mathbb{1}(\cdot)$ is the indicator function, which returns 1 if the condition holds and 0 otherwise. Higher values of Recall@1 and MRR indicate better instruction alignment accuracy.

3.3 Preliminary Instruction Alignment Study

Table 1 presents the instruction alignment performance of both the pre-trained and finetuned `jTrans` models across different optimization level pairs. The finetuned model consistently outperforms the pre-trained model in both MRR and Recall@1, indicating that the contrastive training at the function level has indeed improved



(a) Source code (b) `clang -O0` (c) `gcc -O2` (d) Instruction alignment matrix

Figure 2: Debug information provides a mapping from assembly instructions to source lines. (b) and (c) are compiled from the source code in (a) under different configurations. The colored instructions in (b) and (c) are originated from the source lines with the same color, thus they are semantically aligned. (d) is the instruction alignment matrix of (b) and (c), where the colored cells indicate the aligned instruction pairs with value of 1. Other white cells have value of 0.

Table 1: Instruction alignment performance of pre-trained and finetuned `jTrans` models on their test dataset.

Model	MRR^{insn}		$Recall^{insn}@1$	
	O0→O2	O0→O3	O0→O2	O0→O3
Pre-trained	0.635	0.630	0.524	0.519
Finetuned	0.684	0.681	0.574	0.573
	O2→O0	O3→O0	O2→O0	O3→O0
Pre-trained	0.716	0.708	0.627	0.619
Finetuned	0.731	0.722	0.641	0.632

the model's ability to capture instruction-level semantics. The improvement in instruction alignment correlates with the finetuned model's superior performance on function-level BCSA, suggesting that better instruction-level understanding contributes to more accurate function-level similarity judgments.

Additionally, the results between non-optimized and lower optimized functions (i.e., O0→O2 and O2→O0) are better than the alignment between higher optimized settings (i.e., O0→O3 and O3→O0), which is expected since aggressive optimizations can largely alter the instruction sequence, making alignment more challenging. This phenomenon further underscores the importance of instruction-level understanding for robust binary code similarity analysis, especially in scenarios involving heavily optimized binaries.

3.4 Motivation

The above observations lead to our central research question: *Can we explicitly incorporate instruction alignment knowledge into the training process to further improve binary code embeddings?*

If function-level training already implicitly improves instruction alignment, then directly optimizing for instruction alignment should provide an even stronger supervisory signal. This fine-grained objective encourages the model to learn precise semantic correspondences at the instruction level, which should in turn produce higher-quality function-level embeddings.

Furthermore, explicit instruction alignment training may offer an additional benefit: *interpretability*. A model trained with instruction alignment can not only determine that two functions are similar with function-level embedding distance, but also pinpoint *which specific instructions* correspond to each other, giving inspectable evidence alongside the similarity judgment.

4 Methodology

Previous §3.1 has shown the instruction embedding mechanism, and this section explains the overview (§4.1) of `InsnAlign`, the design of instruction alignment loss (§4.2), the combined training objective (§4.3), and the training data construction (§4.4).

4.1 Overview

`InsnAlign` extends existing Transformer-based binary code embedding models, including `jTrans` [60] and `CLAP` [53], by augmenting standard function-level contrastive training with an instruction-level alignment objective derived from fine-grained source-line correspondences (§2.4). Although designed for Transformer-based models, the framework is also applicable to other architectures that can produce fine-grained embeddings.

Fig. 3 illustrates the overall training framework. Given a pair of binary functions (A, B) compiled from the same or overlapping source code, our framework first constructs an instruction alignment matrix $\mathbf{M}$ using debug information. It then encodes both functions with a shared Transformer to obtain token-level hidden states and function-level embeddings. Initial instruction embeddings are derived by mean-pooling the hidden states of the tokens belonging to each instruction (§3.1). Based on these instruction embeddings, an instruction similarity matrix $\mathbf{S}$ is computed using cosine similarity. We then formulate an InfoNCE loss [43] over $\mathbf{M}$ and $\mathbf{S}$ to optimize instruction alignment. This loss is jointly combined with the function-level similarity objective, implemented as the triplet loss [48], to improve overall embedding quality.

4.2 Instruction Alignment Loss

Given two binary functions A and B compiled from the same source function under different configurations, we obtain their instruction embeddings $\{\mathbf{e}_1^A, \ldots, \mathbf{e}_n^A\}$ and $\{\mathbf{e}_1^B, \ldots, \mathbf{e}_m^B\}$ through the instruction pooling module. Using the debug information, we construct a binary match matrix $\mathbf{M} \in \{0,1\}^{n \times m}$, where $\mathbf{M}_{ij} = 1$ if instructions $\mathbf{e}_i^A$ and $\mathbf{e}_j^B$ originate from the same source line.

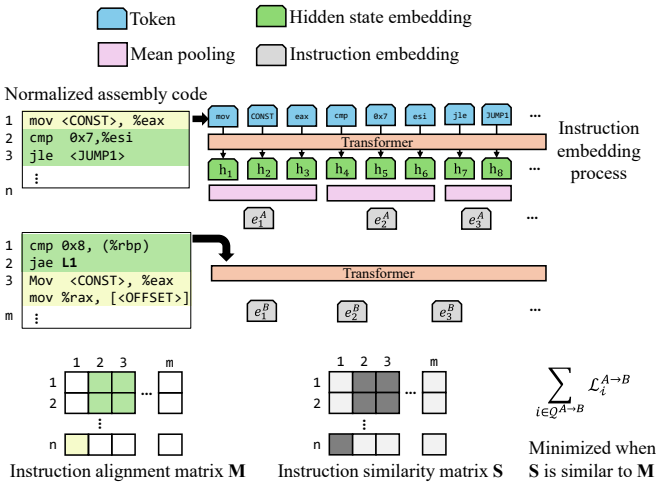


Figure 3: Instruction alignment overview. Colored $\mathbf{M}_{i,j}$ denotes instruction a_i and b_j originates from the same source line. Deep gray $\mathbf{S}_{i,j}$ denotes $\cos(\mathbf{e}_i^A, \mathbf{e}_j^B)$ is close to 1.

Why InfoNCE? Unlike function-level BCSA, where negatives can be drawn from the entire dataset (e.g., millions of candidate functions), instruction alignment operates within a single function pair: for a query instruction a_i, the candidate set is simply the m instructions of the paired function B. Because a binary function contains only a bounded number of instructions, this candidate set is small enough to serve directly as the InfoNCE [43] denominator, casting alignment as an m-way classification that pulls each query toward its matches and away from the remaining instructions without any negative sampling. Moreover, instruction alignment is inherently many-to-many; a single source line may expand into multiple instructions (e.g., through macro expansion), and multiple lines may share instructions. Thus, we adopt a multi-positive generalization inspired by supervised contrastive learning [25]: all instructions b_j with $\mathbf{M}_{ij} = 1$ are treated as positives, and the loss (Eq. 8) maximizes the aggregate probability mass over this positive set rather than forcing a single hard match.

Loss Formulation. We compute the instruction-to-instruction similarity matrix:

$$\mathbf{S}_{ij} = \cos(\mathbf{e}_i^A, \mathbf{e}_j^B). \tag{7}$$

The alignment loss is computed symmetrically in both directions. In the forward direction ($A \to B$), for each instruction $\mathbf{e}_i^A$ that has at least one match in B (i.e., $\sum_j \mathbf{M}_{ij} > 0$), we compute:

$$\mathcal{L}_i^{A\to B} = \log \sum_{j=1}^{m} \exp(\mathbf{S}_{ij}/\tau) - \log \sum_{j:\mathbf{M}_{ij}=1} \exp(\mathbf{S}_{ij}/\tau), \tag{8}$$

where τ is a temperature parameter. Intuitively, Eq. 8 is minimized when the model assigns all probability mass to the positive matches $\{j : \mathbf{M}_{ij} = 1\}$ among the m candidates. When there is exactly one positive match, Equation 8 reduces to the standard InfoNCE loss [43]; when multiple positives exist, it generalizes to a multi-positive variant that maximizes the aggregate softmax probability over the entire positive set.

The backward direction ($B \to A$) is computed analogously by transposing $\mathbf{S}$ and $\mathbf{M}$. The total instruction alignment loss is the average over all valid queries in both directions:

$$\mathcal{L}_{\text{align}} = \frac{1}{|\mathcal{I}^{A\to B}| + |\mathcal{I}^{B\to A}|} \left(\sum_{i\in\mathcal{I}^{A\to B}} \mathcal{L}_i^{A\to B} + \sum_{j\in\mathcal{I}^{B\to A}} \mathcal{L}_j^{B\to A} \right), \tag{9}$$

where $\mathcal{I}^{A\to B}$ and $\mathcal{I}^{B\to A}$ are the sets of query instructions with at least one positive match defined by Eq. 3. The symmetric computation ensures that the alignment is bidirectional: instructions in A are encouraged to find their counterparts in B, and vice versa.

4.3 Combined Training Objective

Combined Loss. The overall training objective combines the whole function-level triplet loss with the instruction-level alignment loss:

$$\mathcal{L} = \mathcal{L}_{\text{triplet}} + \lambda \cdot \mathcal{L}_{\text{align}}, \tag{10}$$

where λ is a weighting coefficient that controls the relative importance of the alignment loss.

The function-level triplet loss $\mathcal{L}_{\text{triplet}}$ operates on the function-level embeddings (§2.3), encouraging semantically similar functions to have close embeddings while pushing dissimilar ones apart. The instruction alignment loss $\mathcal{L}_{\text{align}}$ operates on the per-instruction embeddings, enforcing fine-grained semantic correspondences between individual instructions.

Layer Freezing. To reduce computational cost while preserving the pre-trained knowledge, we adopt a layer freezing strategy: the embedding layer and the first L encoder layers of the BERT backbone are frozen during training, and only the upper layers are finetuned. This behavior is consistent with the finetuning stage of the original `jTrans` model, which also freezes the lower layers during contrastive training [60]. The freezing strategy allows the model to retain the general binary code understanding acquired during pre-training while adapting the upper representations for instruction-level alignment.

4.4 Training Data Construction

We construct training pairs from the compiled binaries with debug information, where each positive pair (A, B) consists of two binary functions that share the same source function but are compiled under different configurations (e.g., different optimization levels or compilers). Since both functions originate from the same source, their match matrix $\mathbf{M}$ is constructed by aligning all instructions that share the same source line (e.g., the $\mathbf{M}$ of Fig. 3). Positive pairs drive both the function-level triplet loss (as the positive example) and the instruction alignment loss (over all shared source lines). A negative function is randomly sampled from a different symbol group to provide the negative example for the triplet loss. Specifically, we reuse the binaries of BinaryCorp (i.e., the training data) of `jTrans`, for fair comparison.

4.5 Baselines

To evaluate the effectiveness of instruction alignment, we compare our newly trained model directly against the models finetuned for BCSA of `jTrans` [60] and `CLAP` [53].

jTrans is a customized Transformer for learning jump-aware binary code embedding. It outperforms prior models, including Genius [14], Gemini [67], SAFE [40], Asm2vec [9], OrderMatters [71]. To perform a fair comparison, we continue training the finetuned `jTrans` checkpoint with the auxiliary instruction alignment loss, using the same dataset, tokenization, and training settings as in the original `jTrans` paper.

CLAP uses the RoBERTa base architecture [31], a different architecture compared with `jTrans`. It also employs a cross-modality training paradigm to align embeddings between binary code and natural language descriptions. With the high-quality natural language embeddings, `CLAP` achieves the state-of-the-art performance on BCSA.

4.6 Datasets

To evaluate the effect of instruction alignment fairly, we use datasets adopted by prior BCSA studies, including BinaryCorp [60] for training and BinKit [26] for evaluation. All binaries are stripped before extracting assembly instructions.

BinaryCorp Dataset. To enable a fair comparison with `jTrans`, we reuse its training dataset—the training partition of BinaryCorp dataset—for training. However, not all binaries in BinaryCorp were compiled with debug information, which is required to construct ground truth for instruction alignment training. Therefore, we use only the subset of BinaryCorp that contains debug information. This training dataset contains 1,655,011 binary functions from 1,544 projects, forming 2,326,328 positive function pairs.

To evaluate the model trained with the auxiliary instruction alignment loss, we do not use the original BinaryCorp test set from the `jTrans` paper, in order to avoid potential data leakage. Specifically, the original test set contains binaries compiled from the same projects as those in the training set, so some test functions originate from the same source lines as training functions, which could inflate instruction alignment performance. Instead, we construct the test set from BinKit [26]. We have manually checked the projects in BinKit and confirmed that none of them overlap with the projects in BinaryCorp, ensuring a clean evaluation of instruction alignment without data leakage.

BinKit Dataset. BinKit [26] is a large-scale BCSA dataset containing 213,400 binaries compiled from 51 open-source projects under diverse compiler versions and compilation settings. All binaries in BinKit include debug information, making the dataset suitable for evaluating instruction alignment. However, as noted in prior work [55, 60], BinKit suffers from substantial duplication, where many binary functions are compiled from the same source lines. For example, the Coreutils project accounts for 42.1% of all binaries in the dataset, while it merely contributes nearly 2,700 source functions. To mitigate this issue, we follow the deduplication preprocess of vSim [55]. Specifically, for each compilation configuration, functions from the same project that share the same symbol are treated as duplicates; one function is retained for evaluation, and the remaining duplicates are removed. We also remove 18 projects (e.g., gawk) that exist in the training set of BinaryCorp. After the preprocess, we use the binary functions of Coreutils project for validation, and the binary functions of the remaining 32 projects as the test set. We use the binaries produced by the latest and oldest versions of GCC and Clang (gcc-11, clang-13, gcc-4.9, clang-4) to cover a wide range of compiler behaviors and avoid duplications.

4.7 Metrics

For direct comparison with prior work on binary code similarity analysis (BCSA), we adopt the widely used Recall@1 [9, 17, 39, 53, 60], which measures the proportion of queries whose true match is ranked first.

Let the query set be $\mathcal{Q} = \{q_1, q_2, \ldots, q_N\}$, and let $\text{rank}(q_i)$ denote the rank of the ground-truth match for query function q_i in the candidate pool $\mathcal{P}$. Then, $\text{Recall}^{\text{func}}@1$ is defined as

$$\text{Recall}^{\text{func}}@1 = \frac{1}{N} \sum_{f \in \mathcal{Q}} \mathbb{1}(rank(f) = 1), \tag{11}$$

where $\mathbb{1}(\cdot)$ is the indicator function. A higher $\text{Recall}^{\text{func}}@1$ indicates better retrieval accuracy and higher-quality function-level embeddings. Previous studies [9, 53–55, 60] also report mean reciprocal rank (MRR); we omit it for simplicity, because MRR is strongly positively correlated with $\text{Recall}^{\text{func}}@1$ in the BCSA.

4.8 Implementation Details

All experiments are conducted on a server with an NVIDIA A6000 GPU (48 GB), 256 GB RAM, and an AMD Threadripper 3970X CPU. For fair comparison, we continue training the finetuned `jTrans` and `CLAP` checkpoints with the auxiliary instruction alignment loss, using the same disassembler (i.e., IDA Pro [19]), the training dataset (§4.6), and training settings as in the original `jTrans` paper [60].

We set the learning rate to $1e-5$, the frozen layer count L to 10, the weight λ of the instruction alignment loss to 0.001, and train the models two epochs. The former two hyperparameters are consistent with the original `jTrans` finetuning settings, while the last one is chosen empirically based on preliminary experiments to balance the two objectives effectively. We choose a small λ because the finetuned `jTrans` checkpoint already provides strong function-level similarity performance. In practice, the original triplet loss is extremely small (below 0.001), whereas the instruction alignment loss is much larger (around 1.5). A small λ is therefore necessary to balance the two objectives and prevent instruction-level alignment from overwhelming function-level representation learning [4].

5 Evaluation

To measure the effectiveness of instruction alignment training, we conduct a comprehensive evaluation addressing the following research questions (RQs):

- **RQ1**: How does instruction alignment training converge, and how does it affect instruction-level alignment quality?
- **RQ2**: How does instruction alignment training impact function-level BCSA performance?
- **RQ3**: Does the instruction-level alignment signal provide stronger discriminability than function-level embeddings?
- **RQ4**: How reliable are compiler-generated labels as a training signal, and how resilient is our method to label noise?
- **RQ5**: How effective is the auxiliary objective when combined with hard negative mining?

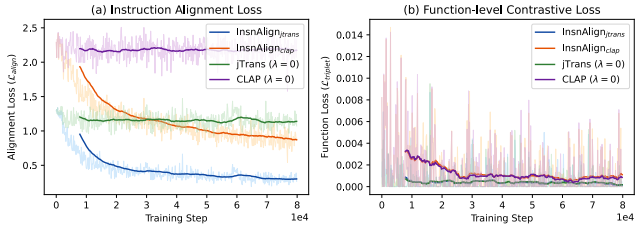


Figure 4: Training loss curves. (a) Instruction alignment loss $\mathcal{L}_{align}$ decrease when $\lambda > 0$. (b) Function-level contrastive loss $\mathcal{L}_{triplet}$ remain near zero, indicating negligible degradation. Note that the instruction alignment loss is much greater than the function-level loss.

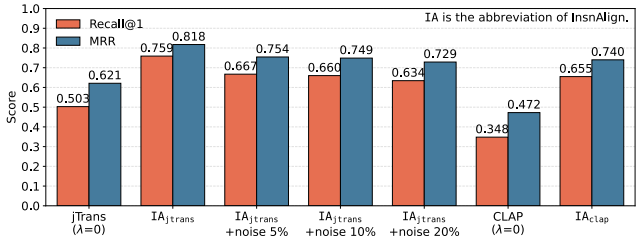


Figure 5: Average Recall^{insn}@1 and MRR^{insn}.

Moreover, we present case studies (§5.6) to understand InsnAlign's difficult situations and show the potential for patch presence detection, a challenging application for function-level representations.

5.1 RQ1: Training Loss Convergence

Fig. 4 shows the training loss curves over the course of two epochs (approximately 80K training steps). We observe several key trends:

Alignment Loss Decreases Significantly. When $\lambda > 0$, the instruction alignment loss $\mathcal{L}_{align}$ drops sharply during the first epoch (Fig. 4(a)). The rapid initial decline indicates that the models quickly learn to align instruction-level embeddings across function pairs compiled from the same source. The loss continues to decrease in the second epoch but at a slower rate, suggesting convergence of the alignment objective using those pre-trained models. Moreover, we also finetune jTrans and CLAP with $\lambda = 0$ for isolating the effect of instruction alignment training. As shown in Fig. 4, their $\mathcal{L}_{align}$ remains high.

Function-level Loss Remains Stable. The function-level contrastive loss $\mathcal{L}_{func}$ (Fig. 4(b)) remains near zero throughout training, consistent with the fact that the models are initialized from well-trained checkpoints that have already converged on the function-level objective. The loss values of InsnAlign$_{jtrans}$ and jTrans ($\lambda = 0$) are stable, and the loss values of InsnAlign$_{clap}$ and CLAP ($\lambda = 0$) decrease slightly. This stability confirms that the auxiliary instruction alignment loss does not degrade the model's existing function-level similarity capability. The following evaluation uses the models further trained with $\lambda = 0$ as baselines, since they slightly outperform their original versions.

Table 2: Function-level Recall^{func}@1 across 16 cross-compiler, cross-optimization settings (O0 vs. O3).

O0	O3	jTrans ($\lambda = 0$)	InsnAlign$_{jtrans}$	CLAP ($\lambda = 0$)	InsnAlign$_{clap}$
GCC-11	GCC-11	0.5011	**0.5269**	0.6507	**0.6675**
	Clang-13	0.4525	**0.4729**	0.6490	**0.6702**
	GCC-4	0.4768	**0.5082**	0.6296	**0.6472**
	Clang-4	0.4386	**0.4630**	0.6563	**0.6745**
Clang-13	GCC-11	0.3768	**0.4088**	0.6371	**0.6548**
	Clang-13	0.3687	**0.3966**	0.6467	**0.6661**
	GCC-4	0.3543	**0.3886**	0.6223	**0.6450**
	Clang-4	0.3526	**0.3879**	0.6523	**0.6717**
GCC-4	GCC-11	0.4803	**0.5089**	0.6324	**0.6539**
	Clang-13	0.4395	**0.4509**	0.6380	**0.6587**
	GCC-4	0.4909	**0.5159**	0.6474	**0.6623**
	Clang-4	0.4274	**0.4451**	0.6395	**0.6585**
Clang-4	GCC-11	0.3233	**0.3542**	0.6288	**0.6480**
	Clang-13	0.3180	**0.3442**	0.6389	**0.6593**
	GCC-4	0.3023	**0.3375**	0.6175	**0.6384**
	Clang-4	0.3142	**0.3409**	0.6470	**0.6673**
Average		0.4011	**0.4282**	0.6396	**0.6590**

Instruction Alignment Results. We evaluate the instruction alignment quality using the MRR^{insn} and Recall^{insn}@1 metrics defined in §3.2. Fig. 5 presents the results between binaries compiled with trivial optimization and aggressive optimization of BinKit dataset. The models trained with our auxiliary task significantly outperform those without by 50.9% for jTrans and 88.2% for CLAP on Recall^{insn}@1, demonstrating that our training method can significantly improve the instruction alignment performance.

5.2 RQ2: Impact on Binary Function Embedding

This RQ investigates the impact of instruction alignment training on function-level representation learning. We measure the Recall^{func}@1 on the BinKit [26] dataset across 16 cross-compiler, cross-optimization-level settings (4 compilers at O0 vs. O3).

5.2.1 Setup. We compare four model variants: the jTrans and CLAP models trained with $\lambda = 0$, and the models being trained with instruction alignment task (InsnAlign uses $\lambda > 0$). We evaluate on a pool size of 10,000 functions per comparison setting, consistent with the most challenging setting in prior work [53, 55, 60].

5.2.2 Results. Table 2 presents the Recall^{func}@1 for each model variant across all 16 comparison settings.

Improvements on Function Embeddings. Compared with the models trained with $\lambda = 0$, instruction alignment training can further improve the function-level embeddings, achieving higher Recall^{func}@1 across all settings. These results demonstrate that the auxiliary instruction alignment objective not only improves instruction-level representation quality (RQ1) but also benefits function-level representation learning, despite the function-level contrastive loss remaining stable during training.

Consistent Improvement Across Compiler Pairs. The improvement holds consistently across all 16 compiler configuration pairs, including both recent compilers (e.g., gcc-11 and clang-13) and outdated compilers (e.g., gcc-4.9 and clang-4). As shown in Table 2, jTrans performs much better on GCC-11 compilers, indicating the training data used in this study was compiled with GCC [55], while the improvement on the comparison with unseen compilers

(Clang-4 and Clang-13) is still significant, and $\texttt{InsnAlign}_{\texttt{clap}}$ shows even increases over CLAP across different settings. This experiment demonstrates our approach is resilient to diverse compiler behaviors and optimization levels, denoting its robustness to distribution shift.

5.3 RQ3: Discriminability of Similar Functions

Beyond retrieval accuracy, instruction alignment offers inspectable evidence: because similarity is examined at the instruction level, the aligned instruction pairs constitute concrete, inspectable evidence for why two functions are judged similar or dissimilar. For such evidence to be trustworthy, the instruction-level signal must itself be discriminative. In this RQ, we therefore quantify to what extent it separates positive function pairs (compiled from the same source) from negative pairs (compiled from different sources). This evaluation is distinct from the instruction retrieval task in RQ1: there, MRR^{insn} and $\text{Recall}^{\text{insn}}$@1 measure the instruction embedding quality, whereas here we ask whether finer-grained similarities provide a more discriminative signal than the function-level embedding.

We define a *per-pair instruction alignment score* to aggregate instruction-level similarities into a single scalar that characterizes the alignment quality of a function pair. We hypothesize that this finer-grained score can offer more discriminative and interpretable evidence for similarity judgments than the coarse-grained function embedding similarity, and test this hypothesis below.

Mean Alignment Score (MAS). Given two binary functions $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ with instruction similarity matrix $\mathbf{S}$ defined in Eq. 7, we compute the mean alignment score as:

$$\text{MAS}(A, B) = \frac{1}{2}\left(\frac{1}{n}\sum_{i=1}^{n}\max_{1\le j\le m}\mathbf{S}_{i,j} + \frac{1}{m}\sum_{j=1}^{m}\max_{1\le i\le n}\mathbf{S}_{i,j}\right). \tag{12}$$

The MAS captures the average best-match similarity between instructions of the two functions in both directions, providing a symmetric measure of how well the instructions align overall. For positive pairs, we expect MAS to be high because many instructions originate from the same source lines and should have similar embeddings. For negative pairs, instructions lack true semantic correspondences, so the best-match similarities should be lower on average.

Similar to the cosine similarity between function-level embeddings, MAS itself is not a binary indicator of similarity; rather, it provides a continuous measure of how well the instructions align, which can serve as interpretable evidence for a similarity judgment. Additionally, we can measure the discriminability of both function embedding cosine similarity and MAS in distinguishing positive vs. negative pairs using the following metrics.

5.3.1 Metrics. We treat the distinction between positive and negative pairs as a binary classification problem and report two metrics. *AUC-ROC* [2] measures how well MAS (or cosine similarity) separates positives from negatives: a value close to 1.0 indicates near-perfect discrimination, while 0.5 indicates no discriminability [47]. *Cohen's d* [5] quantifies the effect size between the score distributions of positive and negative pairs:

$$d = \frac{\mu_+ - \mu_-}{\sqrt{(\sigma_+^2 + \sigma_-^2)/2}}, \tag{13}$$

Table 3: AUC and Cohen's d.

Model	Negative sampling	AUC		Cohen's d	
		MAS	*cos*	MAS	*cos*
jTrans ($\lambda = 0$)	Top-5	0.644	0.573	0.435	0.111
$\texttt{InsnAlign}_{\texttt{jtrans}}$	Top-5	0.720	0.584	0.778	0.144
CLAP ($\lambda = 0$)	Top-5	0.646	0.650	0.236	0.190
$\texttt{InsnAlign}_{\texttt{clap}}$	Top-5	0.702	0.698	0.622	0.526
jTrans ($\lambda = 0$)	Random	0.986	0.996	3.606	4.487
$\texttt{InsnAlign}_{\texttt{jtrans}}$	Random	0.974	0.996	2.974	4.621
CLAP ($\lambda = 0$)	Random	0.986	0.987	4.373	4.393
$\texttt{InsnAlign}_{\texttt{clap}}$	Random	0.991	0.995	4.250	4.998

[1] The reported AUC and Cohen's d are the average over seeds 3, 5, 7, 42.

where μ_+, σ_+ and μ_-, σ_- are the mean and standard deviation for positive and negative pairs, respectively. A larger Cohen's d indicates clearer separation.

5.3.2 Setup. For each compilation configuration pair (e.g., O0→O3), we construct positive pairs from functions sharing the same source symbol and sample an equal number of negative pairs. We evaluate AUC-ROC and Cohen's d for both MAS and function embedding cosine similarity (*cos*).

Negative Sampling Strategy. In real-world BCSA, positive and negative pairs are inherently imbalanced: each query has limited true match but potentially thousands of negatives. Random negative sampling therefore produces mostly trivially dissimilar pairs, inflating AUC for all models (Table 3) [2, 15, 47]. We adopt a harder strategy: for each query, negatives are drawn from the top-5 candidates by function-level cosine similarity that do not share the same symbol, stress testing discriminability against hard negatives that are close in the embedding space but semantically different.

5.3.3 Results and Analysis. Table 3 illustrates the AUC and Cohen's d for both MAS and function-level cosine similarity (*cos*) under the different negative sampling strategies.

Instruction Alignment (MAS) vs. Function Embedding (*cos*). Under random negative sampling, both MAS and cosine similarity achieve high AUC for all models and are comparable (*cos* is even marginally higher), as trivially dissimilar negatives are easy to reject at either granularity. The distinction emerges only under the harder top-5 sampling, where both metrics drop: here MAS attains a higher Cohen's d than *cos* for every model and a higher AUC in most cases, indicating that the instruction-level signal is more discriminative when function embeddings are close.

Effect of Instruction Alignment Training. Comparing the models trained with $\lambda = 0$ and the models trained with InsnAlign, we observe that instruction alignment training improves the AUC and Cohen's d for both MAS and cosine similarity in the challenging setting. This observation is consistent with the evaluation of function-level BCSA performance in RQ2, where instruction alignment training also improves retrieval accuracy. The improvement in AUC and Cohen's d indicates that instruction alignment training not only enhances the quality of instruction embeddings but also yields clearer separation between similar and dissimilar pairs, making the instruction-level evidence a sharper and more reliable signal for explaining the model's similarity judgments.

Table 4: Recall^{func}@1 with synergy scoring.

O0	O3	jTrans ($\lambda = 0$)	InsnAlign$_{jtrans}$	CLAP ($\lambda = 0$)	InsnAlign$_{clap}$
GCC-11	GCC-11	0.5561	**0.5996**	0.6577	**0.6847**
	Clang-13	0.5147	**0.5757**	0.6574	**0.6857**
	GCC-4	0.5295	**0.5772**	0.6364	**0.6607**
	Clang-4	0.5085	**0.5715**	0.6643	**0.6893**
Clang-13	GCC-11	0.4599	**0.5448**	0.6429	**0.6715**
	Clang-13	0.4574	**0.5373**	0.6537	**0.6778**
	GCC-4	0.4445	**0.5236**	0.6305	**0.6569**
	Clang-4	0.4479	**0.5352**	0.6618	**0.6850**
GCC-4	GCC-11	0.5370	**0.5800**	0.6411	**0.6698**
	Clang-13	0.5022	**0.5606**	0.6425	**0.6733**
	GCC-4	0.5418	**0.5860**	0.6547	**0.6746**
	Clang-4	0.4954	**0.5571**	0.6492	**0.6733**
Clang-4	GCC-11	0.4097	**0.5106**	0.6352	**0.6645**
	Clang-13	0.4131	**0.5040**	0.6484	**0.6749**
	GCC-4	0.3931	**0.4915**	0.6233	**0.6520**
	Clang-4	0.4102	**0.5059**	0.6522	**0.6795**
Average		0.4763	**0.5475**	0.6470	**0.6733**
Average improvement[1]		+18.7%	**+27.9%**	+1.16%	**+2.17%**

[1] The average improvements over the values shown in Table 2.

5.3.4 A Synergy Effect. Because MAS shows stronger discriminability in distinguishing similar from dissimilar pairs than function-level embeddings, especially when two functions share close embeddings, we investigate whether combining the two signals can further improve function-level retrieval performance. We define the *synergy score* as a weighted combination:

$$\text{Score}(A, B) = (1 - \gamma) \cdot \cos(\mathbf{f}_A, \mathbf{f}_B) + \gamma \cdot \text{MAS}(A, B), \qquad (14)$$

where $\mathbf{f}_A, \mathbf{f}_B$ are function embeddings, and $\gamma = 0.5$ balances the two signals. Note that instruction alignment is time-consuming to compute, so we only apply the synergy scoring to the challenging comparison. We first rank candidates based on function-level cosine similarity, then compute MAS for the top 100 candidates and re-rank them using the synergy score. This approach is consistent with previous re-ranking approaches [54, 59]. Table 4 presents the synergy results.

Synergy Consistently Improves Retrieval. For InsnAlign$_{jtrans}$, synergy scoring increases Recall^{func}@1 from 0.4282 to 0.5475, yielding a 27.9% improvement, whereas InsnAlign$_{clap}$ gains only 2.17%. This discrepancy is consistent with the difference of MAS and *cos* in discriminability shown in Table 3. With the negative sampling strategy, MAS and *cos* of InsnAlign$_{clap}$ are similar, while InsnAlign$_{jtrans}$'s MAS is significantly larger than *cos*. The original jTrans and CLAP models also benefit from synergy scoring, indicating that instruction-level alignment provides complementary information regardless of the training stage. Moreover, our instruction alignment training further strengthens this benefit. As shown in Table 4, the relative improvement increases from 18.7% to 27.9% for the jTrans-based model and from 1.16% to 2.17% for the CLAP-based model, even though the trained models already achieve higher base Recall^{func}@1.

5.4 RQ4: Label Quality and Noise Resilience

InsnAlign relies on compiler-generated debug information to establish instruction-level correspondences for training. Although debug information provides valuable fine-grained supervision, it is not perfectly precise: aggressive compiler optimizations may attribute instructions to coarse or imprecise source lines [35, 74]. This RQ asks whether such silver labels remain useful when they contain noise. To assess the quality of the compiler-generated labels, we analyze two aspects: coverage and correctness.

```
1  else if(!(in=fopen(file_name,"r")))
2    return -1;
3  if(!use_stdin && fclose(in)!=0)
4    rc = -1;
5  return rc;
```

```
1    mov eax, 0FFFFFFFFh
2    jmp returnLabel
3  returnLabel:
4    ret
```

Figure 6: Example of a *PLAUSIBLE* label. Lines 2, 4, and 5 are absorbed into the common return sequence on the right; thus, the mov instruction can be attributed to both lines 2 and 4, although the assembly is labeled only with line 2.

Coverage. Because -O0 binaries undergo minimal optimization, we use them as the reference for measuring coverage in optimized binaries. We consider an -O0 instruction covered if it has at least one corresponding instruction in the optimized binary. Across our training data, 86.6% of instructions in -O0 functions are covered in their -O3 counterparts. This result suggests that the compiler-generated labels retain broad coverage even under aggressive optimization.

Correctness. We assess correctness on -O3 binaries, where aggressive optimization makes label misattribution most likely. Given recent evidence that LLMs can analyze binary code [24, 45, 50, 66], we use an LLM to check all validation-set mappings and manually audit 100 randomly sampled functions (24,004 mappings across 4,470 compiled source lines). The manual and LLM assessments are consistent.

The mappings are classified as *CORRECT* (86.8%), *PLAUSIBLE* (5.3%), *UNVERIFIABLE* (5.1%), or *SUSPICIOUS or WRONG* (2.8%). *PLAUSIBLE* mappings are likely or partially correct but cannot be confirmed from assembly alone (Fig. 6); *UNVERIFIABLE* mappings arise from imported glibc/gnulib headers or pure prologue/epilogue code. With only 2.8% classified as *SUSPICIOUS or WRONG*, the compiler-generated labels appear highly reliable.

Noise Resilience. We use InsnAlign$_{jtrans}$, the strongest instruction-alignment variant in §5.1, and corrupt 5%, 10%, and 20% of the instruction-to-source correspondences by randomly reassigning instructions to different source lines of the same function, then retrain and re-evaluate the model. As shown in Fig. 5, label noise degrades alignment quality only moderately: even with 20% injected noise, InsnAlign$_{jtrans}$ achieves 0.634 Recall^{insn}@1, still 26.0% higher than the baseline. Thus, the auxiliary objective does not require perfectly clean debug labels. Note that this random corruption is a stress-test proxy for noisy supervision; systematic mis-attribution caused by aggressive optimizations is separately bounded by the correctness analysis above.

5.5 RQ5: Effectiveness with Hard Negatives

Our main evaluation (RQ1-4) uses random negative sampling for the function-level triplet loss. This design keeps the training protocol consistent across all settings and isolates the contribution of instruction alignment training. This RQ further examines whether the auxiliary objective remains effective when the function-level contrastive objective is strengthened with hard negative mining [46].

Table 5: CLAP-based models under hard-negative training.

O0	O3	Non-synergy		Synergy	
		CLAP ($\lambda = 0$)	$\texttt{InsnAlign}_{\texttt{clap}}$	CLAP ($\lambda = 0$)	$\texttt{InsnAlign}_{\texttt{clap}}$
GCC-11	GCC-11	0.6791	**0.6939**	0.6937	**0.7075**
	Clang-13	0.6827	**0.6962**	0.6982	**0.7085**
	GCC-4	0.6604	**0.6698**	0.6753	**0.6841**
	Clang-4	0.6882	**0.7004**	0.7000	**0.7141**
Clang-13	GCC-11	0.6690	**0.6783**	0.6853	**0.6947**
	Clang-13	0.6790	**0.6876**	0.6930	**0.7039**
	GCC-4	0.6582	**0.6652**	0.6720	**0.6810**
	Clang-4	0.6868	**0.6956**	0.7006	**0.7091**
GCC-4	GCC-11	0.6697	**0.6777**	0.6822	**0.6930**
	Clang-13	0.6714	**0.6844**	0.6855	**0.6979**
	GCC-4	0.6739	**0.6837**	0.6857	**0.6990**
	Clang-4	0.6755	**0.6867**	0.6903	**0.6998**
Clang-4	GCC-11	0.6647	**0.6735**	0.6790	**0.6891**
	Clang-13	0.6711	**0.6839**	0.6874	**0.7004**
	GCC-4	0.6516	**0.6593**	0.6673	**0.6777**
	Clang-4	0.6809	**0.6907**	0.6947	**0.7051**
Average		0.6726	**0.6829**	0.6869	**0.6978**

We use CLAP-based model since it shows better performance in our function retrieval experiments.

Setup. To mine hard negatives, we first encode all functions in the training set with the current model and retrieve the top-5 functions for each anchor according to cosine similarity. During training, the negative sample in each triplet is drawn uniformly from this top-5 candidate set (true matches are excluded), and we refresh the mined top-5 candidates every 10,000 training steps. Hard negatives make $\mathcal{L}_{triplet}$ substantially larger than in random negative sampling, so we increase λ to 0.02 to keep $\mathcal{L}_{triplet}$ and $\mathcal{L}_{align}$ balanced.

Results. As shown by Table 5, with hard negative mining enabled, function-level retrieval performance improves, compared to the random negative sampling results (Table 2). Notably, InsnAlign continues to improve function-level retrieval over the hard-negative baseline, and synergy re-ranking (§5.3.4) provides a further gain. Using the 160,000 per-query top-1 results of $\texttt{InsnAlign}_{\texttt{clap}}$ with and without synergy re-ranking (underlying Table 5), a paired t-test confirms that the synergy gain is statistically significant ($t = 27.0$, $p = 5.2 \times 10^{-161}$). These results indicate that instruction alignment is complementary to hard negative mining: hard negatives sharpen the function-level decision boundary, while instruction alignment supplies fine-grained supervision that function-level triplets alone can hardly capture.

5.6 Case Study

We present concrete examples illustrating how instruction alignment provides interpretable evidence for BCSA. Fig. 7 shows the simplified assembly of blake2b from Coreutils compiled by gcc -O0 and gcc -O3, which exhibit significantly different instruction sequences and control flow structures.

Instruction Splitting Under Optimization. A single source line can be compiled into multiple non-contiguous instructions. For instance, source line 282 produces four instructions marked by blue and yellow in Fig. 7; under -O3, these are scattered across distant locations. $\texttt{InsnAlign}_{\texttt{jtrans}}$ nonetheless assigns close embeddings to these instructions across the two variants, demonstrating resilience

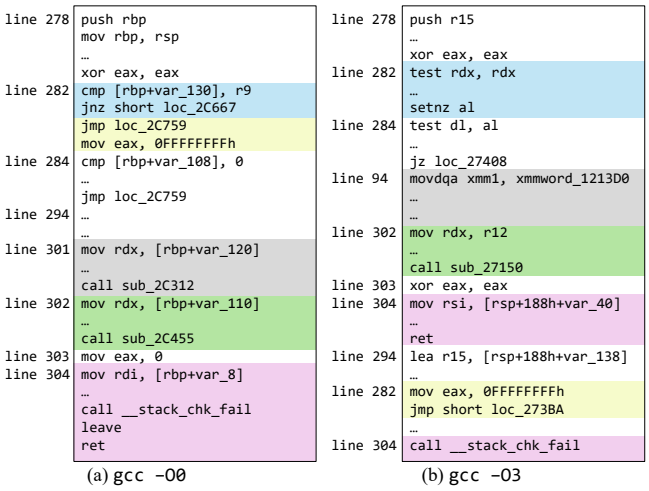


Figure 7: Simplified assembly functions of blake2b. The line numbers denote their corresponding source lines.

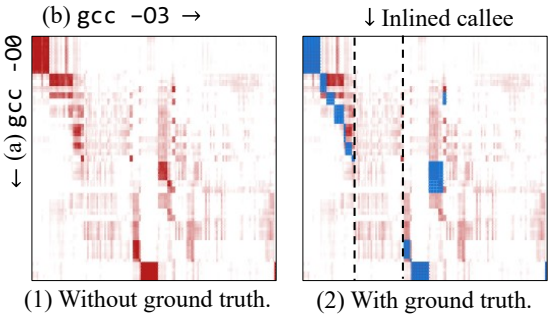


Figure 8: Instruction alignment similarity heatmap of Fig. 7. A red cell (i, j) of (1) denotes a high similarity ($S_{i,j}$) defined in Eq. 7; a blue cell (i, j) of (2) denotes $M_{i,j} = 1$ (§4.2). The columns between two black dotted vertical lines of (2) have no blue cells, indicating that those instructions belong to inlined callees (i.e., line 94 of Fig. 7(b)).

to code fragmentation. Similarly, line 304 is split in Fig. 7(b), with a compiler-inserted call interleaved between them.

Robustness to Function Inlining. Function inlining poses a prominent challenge for BCSA [20, 21], as the inlined callee's semantics are absent in the non-inlined variant. In Fig. 7, the function called at line 301 in (a) is inlined into (b) (line 94). As shown in Fig. 8, the model correctly assigns low similarity to instructions without ground-truth correspondences (nearly white cells), while the code marked by green in (a) achieves high similarity (around 0.7) with its counterparts in (b). Comparing the predicted heatmap (1) with the ground truth matrix (2), the high-similarity regions closely match the true correspondences, confirming that the model identifies instructions from the same source line even in the presence of inlining.

Alignment Failures. To understand the limitations of instruction alignment, we manually analyze all 216 BCSA failures of RQ5's best

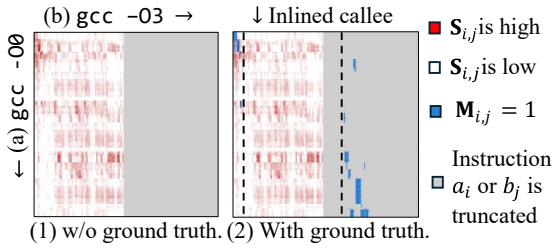


Figure 9: Instruction alignment similarity heatmap of `cut_file`. (a)↓ × (b)→ denotes 102×465 instruction alignment similarities ($S_{i,j}$).

model on the validation set, i.e., query functions (`gcc -O0`) whose top-1 `-O3` match is incorrect, and group them into four causes.

Cause 1: Semantics-Deprived Wrapper Queries (162/216). The dominant cause is over-simple query functions. At `-O0`, these functions are thin wrappers whose observable code merely prepares the calling context (e.g., marshalling arguments) before delegating the real work to a callee. The callee's semantics are invisible in the query, so the argument-preparation instructions carry little discriminative signal. Consequently, the query aligns to many similar wrappers: its similarity matrix is dominated by high-similarity cells over shared boilerplate. For instance, the following assembly code for `xcharalloc` is a wrapper that forwards its argument to `xmalloc`:

```
push  rbp
mov   rbp, rsp
mov   [rbp+var_8], rdi  ; marshal argument
mov   rdi, [rbp+var_8]
call  sub_106436        ; invisible semantics (i.e., xmalloc)
leave
retn
```

Cause 2: Context-Window Truncation (14/216). When the optimized counterpart inlines many callees, its instruction sequence exceeds the model's input length (1,024 tokens) and is truncated, so the discriminative instructions never receive embeddings. Fig. 9 shows this for `cut_file`, aligning its 102 `-O0` instructions against 465 `-O3` instructions. The columns between the two dashed lines in (2) are inlined callees, which have no counterpart in the `-O0` query (no blue cells). By consuming the context window, they push `cut_file`'s own body past the token limit, truncating it (gray columns). Many `-O0` instructions whose true counterparts fall in this truncated region (blue cells over gray) then find no high-similarity match, yielding near-zero row maxima that drag down the MAS. This cause reflects an intrinsic input-length limit.

Cause 3: Near-Twin Siblings (20/216). The query and its true match share an almost identical body and differ only in a single branch or one callee, yet the top-1 result is the sibling twin. Because the differing callee's semantics are invisible, almost every query instruction aligns with equally high similarity to both the twin and the true match, so their similarity matrices are nearly identical; An example is `xvprintf` vs. `xvfprintf`, which are identical except for the delegated callee:

```
; xvprintf                    ; xvfprintf
call _vprintf                 call _vfprintf
... identical error handling (ferror, gettext, error) ...
```

```
1  cwrite(n_out == 0,hold,n_hold);
2  n_out += n_hold;
3  if (n_hold > bufsize) // Deleted in patch
4    hold = xirealloc(hold,bufsize); // Deleted in patch
5  n_hold = 0;
6  hold_size = bufsize; // Deleted in patch
```

Figure 10: The code snippet of vulnerable function `line_bytes_split`.

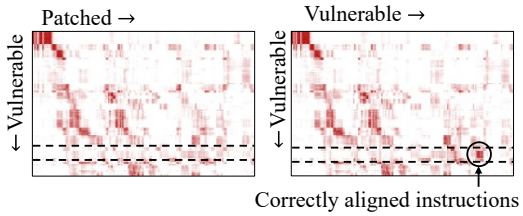


Figure 11: Instruction alignment similarity heatmap for patch presence detection. The vulnerable function compiled with `gcc -O0`↓ vs. patched and vulnerable functions compiled with `gcc -O3`→. The rows between two dotted horizontal lines correspond to assembly compiled from deleted lines of Fig. 10.

Cause 4: Size, Constant, or Global-Only Differences (20/216). These memory-manipulation-dominated functions share the similar structure and differ only in memory sizes, magic constants, or the referenced global. We find that corresponding instructions still earn high per-instruction similarity, saturating the similarity matrix and driving MAS close to one for both candidates; distinguishing them requires operand-level reasoning that the embeddings do not capture. For instance, `md5_init_ctx` and `sha224_init_ctx` write the same store pattern and differ only in the initialization constants and structure size; under `-O3` both collapse to a vectorized load of an initialization-vector constant:

```
; md5_init_ctx (-O3)          ; sha224_init_ctx (-O3)
movdqa xmm0, <ptr_A>          movdqa xmm0, <ptr_B>
movups [rdi], xmm0            movups [rdi], xmm0
mov    [rdi+14h], 0           movdqa xmm0, <ptr_C>
mov    [rdi+18h], 0           movups [rdi+10h], xmm0
```

Implication. Causes 1 and 2 stem from information absent from the query (invisible callee semantics or dropped code), and causes 3 and 4 stem from the model's inability to reason about subtle differences in operands. Causes 1, 3, 4 are alignment false positives (high MAS on wrong candidates), while cause 2 often results in low MAS on the true match.

Patch Presence Detection. Beyond BCSA, instruction alignment may offer interpretable evidence for patch presence detection (PPD), which hinges on capturing subtle patch signatures [18, 69, 73]. Since a vulnerable function and its patched version typically share near-identical overall semantics, coarse-grained function-level representations struggle to tell them apart [57, 73]. We present a preliminary case to illustrate how fine-grained instruction alignment could help.

We study CVE-2024-0684 [13], affecting `line_bytes_split` in Coreutils (Fig. 10), whose patch removes lines 3–4 (line 6 is also deleted but optimized out under `-O3`). We compile the vulnerable

ver. 9.2 with `gcc -O0` and `gcc -O3`, and the patched ver. 9.5 with `gcc -O3`. Queried with the `-O0` binary, $\texttt{InsnAlign}_{\texttt{jtrans}}$ ranks both the vulnerable and patched `-O3` variants at the top among over 2,700 Coreutils functions, and their function-level embeddings are too close to differentiate. The instruction alignment heatmaps (Fig. 11), however, expose the difference: instructions from lines 3–4 of Fig. 10 align to high-similarity counterparts in the vulnerable `-O3` variant but not in the patched one, where those source lines are absent.

This example illustrates the potential of instruction alignment for PPD, not a complete solution. Realizing it must still overcome the intrinsic limitations discussed above, such as invisible callee semantics and limited input length. For instance, the patch-relevant code may be truncated. We thus leave PPD application to future work.

6 Related Work

ML-Based Binary Code Similarity Analysis. ML-based BCSA methods learn semantic embeddings from assembly code for efficient similarity retrieval. Representative approaches include graph neural networks [14, 67], recurrent networks [40, 41], PV-DM with random walks [9, 27], BERT-based pre-training [8, 28, 44], and Transformer architectures with jump-aware encodings [60] or natural language supervision [53]. Other studies explored customized graph semantics [1, 17, 58] and comparison models [39, 54]. These methods universally operate at the function level with symbol-based matching as the sole supervision signal [16, 39] and contrastive learning as the dominant training paradigm. Our work introduces instruction-level alignment as an auxiliary objective using multi-positive InfoNCE [25], complementing any existing embedding-based BCSA model. Non-ML approaches [6, 7, 12, 36, 37, 42, 55, 59, 62, 68, 70] are orthogonal to our method.

Fine-Grained Representations and Explainability. While most BCSA methods produce a single embedding per function, some work has explored finer granularities. PalmTree [28] learns instruction embeddings via pre-training tasks, but these are not designed for cross-function instruction alignment; we thus omit it in our evaluation. DeepBinDiff [10] generates basic-block-level embeddings for binary diffing, but often relies on semantically-irrelevant features like strings [55] and requires expensive pairwise computation. In contrast, `InsnAlign` learns instruction embeddings from instruction correspondences, which can be directly and efficiently aligned, providing fine-grained evidence of the semantic correspondence.

LLM-Based Binary Code Analysis. Recent large-language-model-based (LLM-based) research has substantially expanded the scope of binary analysis. Some studies directly decompile binary code [24, 51] or refine conventional decompiler outputs [45, 51, 52, 63, 66], aiming to improve readability, recompilability, or semantic correctness. Another line evaluates or adapts LLMs for binary comprehension tasks, such as function-name recovery [23], binary summarization [50], similarity analysis [24], and algorithm classification or multi-task binary reasoning [33, 49]. These studies demonstrate the potential of LLMs for reverse engineering. However, the explanations provided by them are often expressed as human-readable artifacts, such as decompiled code or summaries. Such artifacts are cognitively useful, but readability alone does not guarantee that an explanation reflects a model's internal similarity decision or helps distinguish hard positive and negative pairs.

In contrast, `InsnAlign` focuses on binary representation learning for similarity retrieval. Its instruction-level alignments expose the fine-grained instruction correspondences between two binary functions, providing inspectable evidence for a similarity judgment rather than an external, post-hoc artifact, as the alignment is derived from the same instruction embeddings that form the function representation. We evaluate this signal not only qualitatively, but also through hard-negative discriminability and accuracy. `InsnAlign` is also related to explainable retrieval methods such as XSearch [30], which reformulates natural-language-to-code search as concept-to-code alignment. However, XSearch aligns query concepts with source-code statements, whereas `InsnAlign` aligns assembly instructions across binary variants.

7 Conclusion

We propose instruction alignment as a fine-grained supervision signal for binary code representation learning. By deriving instruction-level correspondences from compiler debug information, we augment function-level contrastive training with an auxiliary alignment objective that improves both instruction-level and function-level embedding quality across diverse compiler configurations; the resulting instruction-level alignment provides a more discriminative signal and inspectable evidence for similarity judgments.

Acknowledgements

This paper was supported in part by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China HKUST (No. C6004-25G) and an ITF grant under the contract ITS/161/24FP.

Data Availability Statement

The code and data are available at https://doi.org/10.5281/zenodo.19343892, and we promise to maintain a public repository https://github.com/whj0401/InsnAlign for future research.

References

[1] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/17c3433fecc21b57000debdf7ad5c930-Paper.pdf

[2] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30, 7 (1997), 1145–1159. doi:10.1016/S0031-3203(96)00142-2

[3] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 678–689. doi:10.1145/2950290.2950350

[4] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. 2018. GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 794–803. https://proceedings.mlr.press/v80/chen18a.html

[5] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences.* routledge. doi:10.4324/9780203771587

[6] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. New York, NY, USA, 79–94. doi:10.1145/3062341.3062387

[7] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. New York, NY, USA, 349–360. doi:10.1145/2594291.2594343

[8] Jacob Devlin. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy*. IEEE, 472–489. doi:10.1109/SP.2019.00003

[10] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Network and Distributed Systems Security (NDSS) Symposium*. doi:10.14722/ndss.2020.24311

[11] DWARF Debugging Information Format Committee. 2017. *DWARF Debugging Information Format*. https://dwarfstd.org/doc/DWARF5.pdf

[12] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, 303–317.

[13] Paul Eggert. 2026. *Coreutils fix for CVE-2024-0684*. https://github.com/coreutils/coreutils/commit/c4c5ed8f4e9cd55a12966d4f520e3a13101637d9

[14] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 480–491. doi:10.1145/2976749.2978370

[15] David J Hand and Christoforos Anagnostopoulos. 2023. Notes on the H-measure of classifier performance. *Advances in Data Analysis and Classification* 17, 1 (2023), 109–124. doi:10.1007/s11634-021-00490-3

[16] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (April 2021), 38 pages. doi:10.1145/3446371

[17] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1759–1776.

[18] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. {RapidPatch}: firmware hotpatching for {Real-Time} embedded devices. In *31st USENIX Security Symposium (USENIX Security 22)*. 2225–2242.

[19] SA Hex-Rays. 2026. IDA Pro: Powerful Disassembler, Decompiler & Debugger. https://hex-rays.com/ida-pro.

[20] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 2023. 1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.

[21] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. 2024. Cross-inlining binary function similarity detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[22] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Article 224, 13 pages. doi:10.1145/3597503.3639100

[23] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 2025. Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs. In *Network and Distributed System Security (NDSS) Symposium*. doi:10.14722/ndss.2025.240797

[24] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2025. Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning. In *International Conference on Learning Representations*. 95905–95926. https://proceedings.iclr.cc/paper_files/paper/2025/file/ef283d62b4bce30854a8d4827f331229-Paper-Conference.pdf

[25] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. (2020), 18661–18673. https://proceedings.neurips.cc/paper_files/paper/2020/file/d89a66c7c80a29b1bdbab0f2a1a94af8-Paper.pdf

[26] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2023. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1661–1682. doi:10.1109/TSE.2022.3187689

[27] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32* (Beijing, China) *(ICML'14)*. JMLR.org, II–1188–II–1196.

[28] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251. doi:10.1145/3460120.3484587

[29] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*. 2253–2265. doi:10.1145/3510003.3510217

[30] Yiming Liu, Ruofan Liu, Yun Lin, Zicong Zhang, Weiyu Kong, Pengnian Qi, Xiao Cheng, Weinan Zhang, Qianxiang Wang, and Linpeng Huang. 2026. XSearch: Explainable Code Search via Concept-to-Code Alignment. arXiv:2605.16046 [cs.SE] https://arxiv.org/abs/2605.16046

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[32] Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, and Yutian Tang. 2025. KEENHash: Hashing programs into function-aware embeddings for large-scale binary code similarity analysis. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 801–824. doi:10.1145/3728911

[33] Zhibo Liu, Huaijin Wang, Wai Kin Wong, Daoyuan Wu, and Shuai Wang. 2026. No More Translation at Runtime: LLM-Empowered Static Binary Translation. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 1023–1040. doi:10.1145/3767295.3803600

[34] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1100–1119. doi:10.1109/SP46214.2022.9833799

[35] Hongyi Lu, Zhibo Liu, Shuai Wang, and Fengwei Zhang. 2024. Dtd: Comprehensive and scalable testing for debuggers. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1172–1193. doi:10.1145/3643779

[36] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. 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*. 389–400. doi:10.1145/2635868.2635900

[37] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. doi:10.1109/TSE.2017.2655046

[38] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *Network and Distributed Systems Security (NDSS) Symposium*. doi:10.14722/ndss.2023.24415

[39] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.

[40] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329. doi:10.1007/978-3-030-22038-9_15

[41] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

[42] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*.

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

[44] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2023. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2776–2790. doi:10.1109/TSE.2022.3231621

[45] Hu Peiwei, Liang Ruigang, and Chen Kai. 2024. DeGPT: Optimizing Decompiler Output with LLM. In *NDSS*. doi:10.14722/ndss.2024.24401

[46] Joshua Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2021. Contrastive Learning with Hard Negative Samples. arXiv:2010.04592 [cs.LG] https://arxiv.org/abs/2010.04592

[47] Takaya Saito and Marc Rehmsmeier. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS one* 10, 3 (2015), e0118432.

[48] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823. doi:10.1109/CVPR.2015.7298682

[49] Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2025. BinMetric: a comprehensive binary code analysis benchmark for large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence* (Montreal, Canada) *(IJCAI '25)*. Article 858, 9 pages. doi:10.24963/ijcai.2025/858

[50] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2024. How Far Have We Gone in Binary Code Understanding Using Large Language Models . In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. doi:10.1109/ICSME58944.2024.00012

[51] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[52] Wong Wai Kin, Wu Daoyuan, Liu Zhibo, Wang Huaijin, Li Zongjie, and Wang Shuai. 2026. BinRAG: An RAG-Based Decompilation Framework Fusing Name Prediction and Calling Context. In *Proceedings of the 2026 International Symposium on Software Testing and Analysis (ISSTA '26)*. doi:10.1145/3832245

[53] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 503–515. doi:10.1145/3650212.3652145

[54] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 149–161. doi:10.1145/3650212.3652117

[55] Huaijin Wang and Zhiqiang Lin. 2026. vSim: Semantics-aware value extraction for efficient binary code similarity analysis. In *Network and Distributed Systems Security (NDSS) Symposium*. doi:10.14722/ndss.2026.240213

[56] Huaijin Wang, Zhibo Liu, Yanbo Dai, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. Preserving Privacy in Software Composition Analysis: A Study of Technical Solutions and Enhancements. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2329–2341. doi:10.1109/ICSE55347.2025.00055

[57] Huaijin Wang, Zhibo Liu, Shuai Wang, Ying Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2024. Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis. In *2024 IEEE 9th European Symposium on Security and Privacy*. 506–523. doi:10.1109/EuroSP60621.2024.00034

[58] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. sem2vec: Semantics-aware Assembly Tracelet Embedding. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 90 (May 2023), 34 pages. doi:10.1145/3569933

[59] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Enhancing DNN-based binary code function search with low-cost equivalence checking. *IEEE Transactions on Software Engineering* 49, 1 (2022), 226–250. doi:10.1109/TSE.2022.3149240

[60] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1–13. doi:10.1145/3533767.3534367

[61] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2022. Generating Effective Software Obfuscation Sequences With Reinforcement Learning. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2022), 1900–1917. doi:10.1109/TDSC.2020.3041655

[62] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 319–330. doi:10.5555/3155562.3155606

[63] Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530* (2023).

[64] Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. BinAug: Enhancing Binary Similarity Analysis with Low-Cost Input Repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (Lisbon, Portugal). Article 7, 13 pages. doi:10.1145/3597503.3623328

[65] Wai Kin Wong, Huaijin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving Deep Neural Networks-Based Binary Code Matching with Adversarial Programs. In *2022 IEEE International Conference on Software Maintenance and Evolution*. 117–128. doi:10.1109/ICSME55016.2022.00019

[66] Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA081 (June 2025), 24 pages. doi:10.1145/3728958

[67] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. 363–376. doi:10.1145/3133956.3134018

[68] Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 401–412. doi:10.1145/3611643.3616301

[69] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387. doi:10.1145/3395363.3397361

[70] Chengfeng Ye, Anshunkang Zhou, and Charles Zhang. 2026. Enhancing Semantic-Aware Binary Diffing with High-Confidence Dynamic Instruction Alignment. In *Network and Distributed Systems Security (NDSS) Symposium*. doi:10.14722/ndss.2026.240663

[71] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152. doi:10.1609/aaai.v34i01.5466

[72] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 3872–3883.

[73] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. 887–902.

[74] Liu Zhibo, Wang Huaijin, and Wang Shuai. 2026. The Unseen Delta: Characterizing the Compiler Optimization Landscape via Top-Down Differential Analysis. In *Proceedings of the 2026 International Symposium on Software Testing and Analysis (ISSTA '26)*. doi:10.1145/3832164

[75] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Network and Distributed Systems Security (NDSS) Symposium*. doi:10.14722/ndss.2019.23492