VANISH: FUSING SOURCE CODE AND VIRTUAL ASSEMBLY FOR ROBUST CROSS-LANGUAGE CODE CLONE DETECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Cross-language code clone detection supports code search, migration, and plagiarism analysis, yet remains vulnerable to semantics-preserving obfuscation because existing models rely heavily on lexical evidence. We propose VANISH, a dual-view framework that combines multilingual source code with identifier-reduced virtual assembly. VANISH resolves supervision, tokenization, and granularity mismatches, trains both views independently with cross-language contrastive learning, and fuses their similarity scores. We also construct XL-Asm, a leakage-controlled benchmark of $91{,}863$ programs from $29{,}324$ functionality groups across 12 languages. VANISH achieves state-of-the-art performance with 0.674 X-MRR, outperforming CodeSage and UniXcoder by 12.3% and 68.5%, respectively. Under strong obfuscation, it recovers detection AUC from 0.85–0.88 to 0.94–0.95, gaining up to 8.9 percentage points ($p < 0.001$) without reducing clean-code performance. Results across languages, source encoders, real obfuscators, and adaptive attacks establish the *headroom principle*, whereby virtual assembly contributes most when source representations are degraded.

1 INTRODUCTION

Cross-language code clone detection (X-CCD) identifies implementations of the same functionality across programming languages. It supports semantic code search and software migration (Husain et al., 2019; Lu et al., 2021), as well as license auditing and vulnerability tracking across software ecosystems (Hemel et al., 2011; Pewny et al., 2015). It is also important for cross-language plagiarism detection, where functionally equivalent programs may exhibit substantially different syntax (Karnalim, 2020; Nafi et al., 2019). Reliably matching code across languages is therefore an increasingly important software-engineering capability.

Recent code generation and translation models make cross-language reuse easier to create and harder to detect. An implementation can be translated into another language and systematically rewritten without changing its functionality (Roziere et al., 2020; Chen et al., 2021). Identifier renaming is particularly practical because it is inexpensive, semantics-preserving, and widely applicable. Its effect is amplified in X-CCD, where implementations in different languages already share few tokens and renaming removes much of the remaining lexical correspondence. Although pretrained source encoders learn strong multilingual representations (Feng et al., 2020; Guo et al., 2022; Zhang et al., 2024), they remain sensitive to identifier-level perturbations (Zhang et al., 2020; Yang et al., 2022).

Compilation offers a complementary view by lowering source programs into operational representations in which user-defined identifiers are largely replaced by registers, addresses, and stack locations, preserving low-level similarities across languages (Xu et al., 2017; Wang et al., 2022). ViC enables such abstraction without language-specific compilation toolchains (Gao et al., 2024), but its pseudo-assembly cannot be used directly with existing assembly encoders. These models are trained on real binaries, their tokenizers provide poor coverage of ViC-generated tokens, and whole-program compilation introduces irrelevant context (Ding et al., 2019; Wang et al., 2024). The challenge is therefore to extract a reliable identifier-reduced assembly signal and combine it with source code without discarding useful lexical and structural evidence.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

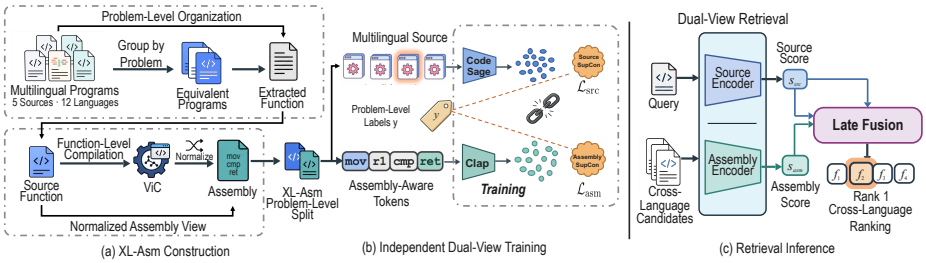


Figure 1: Overview of VANISH. Our source–virtual-assembly framework. XL-Asm pairs multilingual source programs with function-level, normalized ViC pseudo-assembly. Independently trained source and assembly encoders produce similarity scores that are combined through late fusion for cross-language retrieval.

To address this challenge, we propose VANISH, a dual-view framework that combines multilingual source code with ViC pseudo-assembly. As shown in Figure 1, we first construct **XL-Asm** by collecting 91,863 programs in 12 languages from five sources and grouping functionally equivalent implementations by problem. Function extraction, ViC compilation, and assembly normalization produce paired source–pseudo-assembly instances, which are divided using leakage-controlled problem-level splits. The source and assembly branches are then trained independently with supervised contrastive learning (Khosla et al., 2020). Their similarity scores are combined through late fusion, preserving modality-specific information without forcing heterogeneous representations into a shared embedding space.

Our framework is guided by the *headroom principle*, under which an orthogonal assembly view is most valuable when the source branch leaves room for improvement rather than when assembly is strong in isolation. Under aggressive obfuscation, late fusion recovers up to 8.9 AUC points over the source-only baseline and improves retrieval MRR by 10.9% without reducing clean-data performance. Across source encoders, fusion gain is strongly inversely correlated with source-only performance ($r = -0.95$), and the benefit generalizes to production-grade obfuscators and different source backbones. Comparisons with real CPython bytecode and controls involving shuffled assembly, provenance variation, and adaptive attacks further attribute the gain to complementary compiled information. The principle also enables selective routing, in which the assembly branch is invoked only when source-only evidence is uncertain.

Our contributions are summarized as follows:

- **XL-Asm, a paired benchmark for robust X-CCD.** We construct 91,863 source–pseudo-assembly pairs covering 12 programming languages and five data sources. XL-Asm provides a reproducible ViC generation and normalization pipeline, together with leakage-controlled problem-level splits whose measured source overlap is below 0.4%.
- **A robust source–virtual-assembly fusion framework.** We address three transfer mismatches through task-specific contrastive training, assembly-aware tokenization, and function-level compilation. Independently trained source and assembly branches are combined through score-level late fusion, improving robustness to lexical obfuscation without compromising clean-data performance.
- **The headroom principle and its design implications.** We formulate and validate a predictive account of when assembly complements source representations. Experiments across encoders, obfuscators, and language pairs show that fusion benefit grows with source-side headroom, while comparisons with alternative fusion and routing strategies establish late fusion as a Pareto-efficient design.

2 Related Work

2.1 Cross-Language Code Clone Detection

Code clone detection has evolved from token- and tree-based matching (Kamiya et al., 2002; Li et al., 2006; Jiang et al., 2007; Sajnani et al., 2016) to neural representations of lexical, syntactic, and flow information (White et al., 2016; Wei & Li, 2017; Wang et al., 2020). Pretrained code models further learn transferable source representations at scale (Allamanis et al., 2018; Kanade

et al., 2020). Representative models capture multilingual tokens, data flow, cross-modal context, and identifiers (Feng et al., 2020; Guo et al., 2021; 2022; Wang et al., 2021), while CodeSage serves as our source backbone (Zhang et al., 2024). Benchmarks such as CodeXGLUE, XLCoST, and XCodeEval support multilingual code understanding and retrieval (Lu et al., 2021; Zhu et al., 2022; Khan et al., 2024).

Dedicated X-CCD methods use syntactic and API features (Cheng et al., 2017; Nafi et al., 2019), contrastive learning (Tao et al., 2022), or zero-shot and adaptive alignment (Li et al., 2023; Du et al., 2024). Because their evidence remains primarily source-derived, lexical transformations can weaken matching. Prior studies demonstrate this vulnerability in both clone detectors and neural code models (Schulze & Meyer, 2013; Zhang et al., 2020; Yefet et al., 2020). Adversarial training and natural identifier attacks further characterize this problem (Bielik & Vechev, 2020; Yang et al., 2022). We instead complement source representations with an identifier-reduced compiled view.

2.2 Code Representation and Multimodal Fusion

Compiled representations suppress source syntax and identifiers. Early binary similarity methods use intermediate representations, control-flow graphs, or cross-architecture instruction sequences (Pewny et al., 2015; Xu et al., 2017; Zuo et al., 2019). Neural assembly models subsequently learn function embeddings from instruction context and order (Ding et al., 2019; Massarelli et al., 2019; Yu et al., 2020; Li et al., 2021), approximate execution semantics (Pei et al., 2023), or jump-aware and language-supervised representations (Wang et al., 2022; 2024). However, these methods mainly target binary-to-binary matching and assume real compilation and disassembly toolchains.

ViC generates pseudo-assembly directly from source code without such toolchains (Gao et al., 2024). We extend it to obfuscation-robust X-CCD by addressing mismatches in task supervision, tokenization, and compilation granularity. The source and assembly branches are trained independently using supervised contrastive learning (Khosla et al., 2020) and combined through score-level late fusion. Unlike replacing source code or jointly embedding both views, our approach preserves complementary evidence, while the *headroom principle* explains when assembly improves source retrieval.

3 Method

Our framework represents programs through multilingual source code and function-level virtual assembly. Separate encoders capture their complementary features, and their similarity scores are fused at retrieval time without forcing heterogeneous representations into a shared space.

3.1 Problem Formulation

Let $\mathcal{C} = \{(x_i, \ell_i, y_i)\}_{i=1}^{N}$ denote a code corpus, where x_i is a source program, ℓ_i is its programming language, and y_i is its functionality label. Given a query q and a candidate c, cross-language relevance is defined as

$$r(q, c) = \mathbb{I}[y_q = y_c]\,\mathbb{I}[\ell_q \neq \ell_c]. \tag{1}$$

The objective is to learn a similarity function $s(q, c)$ that ranks functionally equivalent implementations written in other languages ahead of non-equivalent candidates.

For each program x, we retain its source view and construct a virtual assembly view as

$$\tilde{a}(x) = \mathcal{N}(\mathcal{V}(\mathcal{E}(x)))\,, \tag{2}$$

where $\mathcal{E}$ extracts the target function, $\mathcal{V}$ denotes ViC virtual compilation, and $\mathcal{N}$ normalizes the generated pseudo-assembly. The two views are independently encoded, and their similarities are combined through score-level fusion.

We consider a clean candidate corpus and transformed queries. Query assembly is regenerated from the transformed source, so robustness stems from virtual compilation's partial invariance rather than access to clean assembly.

3.2 XL-ASM CONSTRUCTION

We construct XL-Asm from five public multilingual programming resources: XLCoST (Zhu et al., 2022), XCodeEval (Khan et al., 2024), Rosetta Code (Rosetta Code Contributors, 2026), bc-TransCoder (Roziere et al., 2020), and CodeContests (Li et al., 2022). These resources provide complementary coverage of code translation, algorithmic programming, and independently written multilingual implementations. After function extraction, virtual compilation, and quality filtering, XL-Asm contains 91,863 programs in 12 programming languages, organized into 29,324 functionality groups.

Each functionality group contains implementations associated with the same programming task and treated as semantically equivalent under the source-provided task definition. We recover group identities from the problem or task identifiers supplied by XLCoST, XCodeEval, bc-TransCoder, and CodeContests. For Rosetta Code, implementations appearing under the same task page are assigned to one group. Because identifiers from different resources are not directly comparable, each group is namespaced by its source:

$$y(x) = \big(d(x), p(x)\big), \tag{3}$$

where $d(x)$ is the originating resource and $p(x)$ is its source-specific task identifier. Programs sharing $y(x)$ but written in different languages form cross-language positive pairs.

We partition XL-Asm by functionality group rather than by individual program. Let $\mathcal{Y}_s$ denote the group identities assigned to split $s \in \{\text{train}, \text{dev}, \text{test}\}$. We enforce

$$\mathcal{Y}_s \cap \mathcal{Y}_{s'} = \varnothing \qquad \forall\, s \neq s', \tag{4}$$

so that all language implementations associated with a task remain in the same split. Before splitting, we remove failed virtual-compilation outputs, malformed or over-length sequences, and exact or normalized duplicates. We additionally audit cross-source overlap after splitting to detect semantically or textually similar tasks with different source-specific identifiers. The resulting measured source overlap across splits is below 0.4%.

3.3 DUAL-VIEW ENCODERS

The source and assembly modalities differ substantially in vocabulary, sequence structure, and abstraction level. We therefore retain a separate encoder and representation space for each modality.

The source branch uses CodeSage (Zhang et al., 2024) as its multilingual backbone. Given source program x, its representation is

$$\mathbf{z}^c = f_c(x) = \mathrm{L2}\big(\mathbf{W}_c\ \mathrm{MeanPool}\big(\mathrm{CodeSage}(x)\big)\big)\,, \tag{5}$$

where $\mathbf{W}_c$ is a learnable projection and $\mathrm{L2}(\mathbf{v}) = \mathbf{v}/\|\mathbf{v}\|_2$. The source branch retains identifiers, APIs, constants, and high-level program structures, providing a strong retrieval signal when lexical evidence remains reliable.

The assembly branch uses the RoFormer-based encoder from CLAP (Wang et al., 2024). For normalized virtual assembly $\tilde{a}(x)$, we compute

$$\mathbf{z}^a = f_a(\tilde{a}(x)) = \mathrm{L2}\big(\mathbf{W}_a\ \mathrm{Pool}\big(\mathrm{CLAP}(\tilde{a}(x))\big)\big)\,, \tag{6}$$

where $\mathbf{W}_a$ is a task-specific projection. The assembly encoder is fine-tuned rather than frozen because its pretrained representation space is not directly optimized for matching functionally equivalent programs across source languages.

3.4 CROSS-LANGUAGE CONTRASTIVE LEARNING

We train the source and assembly encoders independently using supervised contrastive learning (Khosla et al., 2020). For an anchor i in a mini-batch $\mathcal{B}$, its positive set contains implementations of the same functionality written in other languages:

$$P(i) = \{j \in \mathcal{B} \setminus \{i\} \mid y_j = y_i,\ \ell_j \neq \ell_i\}\,. \tag{7}$$

For modality $m \in \{c, a\}$, let $\mathbf{z}_i^m$ denote the normalized embedding of instance i. The training loss is

$$\mathcal{L}_m = -\frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \frac{1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp\big((\mathbf{z}_i^m)^\top \mathbf{z}_p^m / \tau\big)}{\sum_{k \in \mathcal{B} \setminus \{i\}} \exp((\mathbf{z}_i^m)^\top \mathbf{z}_k^m / \tau)} \tag{8}$$

where $\mathcal{I} = \{i \in \mathcal{B} : |P(i)| > 0\}$ and $\tau = 0.07$. Restricting positive pairs to different languages aligns the optimization objective with cross-language retrieval.

Random sampling may produce batches without valid cross-language positives. We therefore employ a PK sampler that selects P functionality groups and K implementations from each group, with multiple languages included whenever available. The two branches are optimized separately as

$$\theta_c^* = \arg\min_{\theta_c} \mathcal{L}_c, \qquad \theta_a^* = \arg\min_{\theta_a} \mathcal{L}_a. \tag{9}$$

Independent optimization allows each modality to retain its own geometry and prevents noisy virtual-assembly features from directly modifying the source representation. The complete dataset construction, encoder training, and score-level retrieval procedure is summarized in Algorithm 1 (Appendix B.2).

3.5 Score-Level Fusion

For a query q and candidate c, each branch computes a cosine similarity between its normalized representations:

$$s_m(q, c) = (\mathbf{z}_q^m)^\top \mathbf{z}_c^m, \qquad m \in \{c, a\}. \tag{10}$$

We collect the two scores and their fusion weights as

$$\mathbf{s}(q, c) = [s_c(q, c), s_a(q, c)]^\top, \quad \boldsymbol{\lambda}_\alpha = [\alpha, 1 - \alpha]^\top. \tag{11}$$

The final retrieval score is their weighted inner product:

$$s_\alpha(q, c) = \boldsymbol{\lambda}_\alpha^\top \mathbf{s}(q, c), \qquad \alpha \in [0, 1]. \tag{12}$$

We select α on the development set and use $\alpha = 0.6$ in the main experiments. Because both branches produce ℓ_2-normalized embeddings, Equation (12) forms a convex combination of two cosine similarities without requiring their heterogeneous representations to share the same space or dimensionality. Noise introduced by virtual compilation is therefore confined to the assembly score, while the assembly branch can still revise rankings when obfuscation weakens source evidence.

This design follows the *headroom principle*: assembly is most useful when the source branch leaves substantial residual error, rather than merely when assembly performs well in isolation. Let $\mathcal{R}(s) \in [0, 1]$ denote a normalized retrieval metric. We define source headroom and fusion gain as

$$H_c = 1 - \mathcal{R}(s_c), \qquad \Delta_\alpha = \mathcal{R}(s_\alpha) - \mathcal{R}(s_c). \tag{13}$$

The principle induces the following empirical hypothesis across encoders or perturbation conditions:

$$\mathrm{Corr}(H_c, \Delta_\alpha) > 0. \tag{14}$$

This relationship is expected when source and assembly errors are not perfectly correlated. Accordingly, assembly need not outperform source code independently; it only needs to preserve complementary evidence for queries on which the source representation becomes uncertain or degraded.

4 Experiments

We evaluate VANISH on cross-language clone retrieval and detection under clean and obfuscated conditions. Extensive experiments demonstrate its retrieval performance, obfuscation robustness, and cross-modal complementarity.

4.1 Experimental Setup

XL-Asm covers 12 languages from five public sources and contains 91,863 programs in 29,324 functionality groups; the held-out test set comprises 3,728 groups and 15,711 programs (Appendix A.3). Functionality-group splitting ensures no shared group identities across splits, while normalized source overlap remains below 0.4%. We report X-MRR and Recall@k for cross-language retrieval and AUC for binary clone detection using same-task pairs and balanced negatives. All results include 95% confidence intervals obtained by cluster bootstrap over functionality groups.

Experiments are conducted on eight NVIDIA A100 80 GB GPUs with distributed mixed-precision training. The CodeSage source encoder and CLAP-RoFormer assembly encoder are optimized independently using AdamW and supervised contrastive learning with $\tau = 0.07$, for 5 and 10 epochs, respectively. Checkpoints are selected by development-set X-MRR, and score-level fusion uses the development-selected weight $\alpha = 0.6$. Additional implementation and bootstrap details are provided in the appendix. Tables 1, 3, and 6 use different evaluation populations: the full test set (15,711 programs), a held-out obfuscation subset, and a detection-pair subset, respectively. Their candidate pools differ, explaining the clean source-only X-MRR values of 0.666, 0.705, and 0.885. Detection AUC uses same-task cross-language positives and balanced random negatives. In the asymmetric robustness setting, the index remains clean; both source and virtual-assembly representations of each transformed query are recomputed from its transformed code, rather than reusing clean-query assembly.

4.2 Main Results and the Headroom Principle

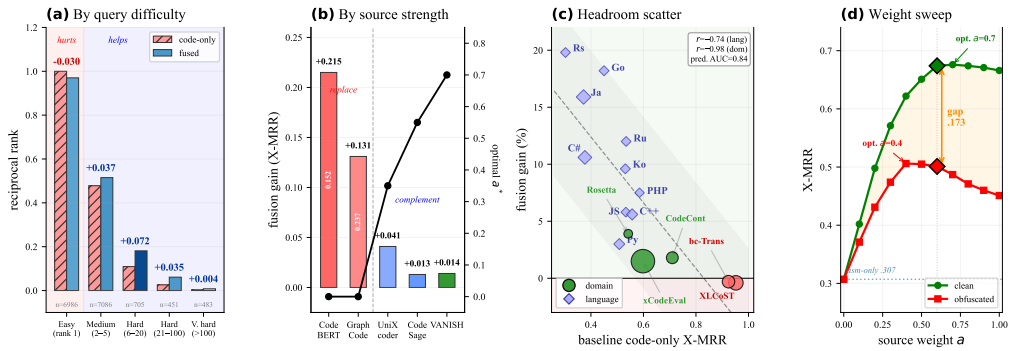


Figure 2: The headroom principle from four perspectives. **(a)** VANISH provides the largest gains on difficult queries, while fusion can slightly demote results already ranked first by VANISH-Src. **(b)** Fusion gain decreases as source-encoder strength increases ($r = -0.95$). **(c)** The same negative relationship appears across domains and language pairs. **(d)** Weight sweeps show that clean and obfuscated queries favor different fusion strengths. Overall, VANISH benefits most from virtual assembly when the source representation leaves sufficient retrieval headroom.

Table 1 reports clean-code clone retrieval across 12 languages. Our source encoder, VANISH-Src, achieves an X-MRR of 0.666, outperforming CodeBERT by more than 5× and UniXcoder by 66%. In contrast, Qwen2.5-Coder-0.5B obtains only 0.158, suggesting that generative pretraining alone does not necessarily produce effective representations for cross-language clone retrieval.

The complete VANISH framework fuses VANISH-Src and VANISH-VAsm at the score level, increasing X-MRR from 0.666 to 0.674 ($\Delta = 0.008$, a 1.2% relative gain; $p = 0.25$). The modest clean-code improvement reflects the strong performance of VANISH-Src, which leaves limited room for correction. Although VANISH-VAsm is weaker in isolation (0.307), its moderate representational similarity to the source view (CKA = 0.51) indicates that the two views preserve complementary information.

This behavior reveals a *headroom principle*: virtual assembly helps most when the source representation is vulnerable, but offers little benefit when source retrieval is already near its ceiling. This pattern holds across four dimensions. Fusion gain increases from 2.3% on clean code to 12.2% under full identifier renaming (Table 3), where VANISH-Src degrades while VANISH-VAsm remains stable. By difficulty (Table 2), fusion gains 7.2 percentage points (pp) for queries ranked 6–20, but

Table 1: Clean-code cross-language clone retrieval across 12 languages. VANISH combines source code and virtual assembly through score-level fusion, improving X-MRR from 0.666 to 0.674 on clean code.

Method	X-MRR	R@1	R@5
CodeBERT	0.122	0.083	0.159
Qwen2.5-Coder-0.5B	0.158	0.100	0.215
GraphCodeBERT	0.209	0.157	0.263
UniXcoder	0.400	0.275	0.532
CodeSage (0-shot)	0.600	0.395	0.817
VANISH-Src (ours)	0.666	0.445	0.896
VANISH-VAsm (ours)	0.307	0.210	0.410
VANISH (ours)	**0.674**	**0.460**	**0.899**

Table 2: Fusion gain by query difficulty, defined by the VANISH-Src rank ($n = 15{,}711$). Virtual assembly helps most on rank 6–20 queries and slightly hurts queries already ranked first. Gains are reported in percentage points (pp).

Query difficulty	n	Source RR	Fused RR	ΔRR (pp)
Easy (rank 1)	6,986	1.000	0.970	-3.0
Medium (rank 2–5)	7,086	0.478	0.515	$+3.7$
Hard (rank 6–20)	705	0.109	0.181	$+\mathbf{7.2}$
Hard (rank 21–100)	451	0.026	0.061	$+3.5$
Very hard (rank >100)	483	0.003	0.008	$+0.5$

Table 3: Fusion gain under increasing obfuscation severity on the held-out subset. VANISH-VAsm remains at 0.445 across transformations, demonstrating partial invariance to source-level obfuscation.

Obfuscation	VANISH-Src	VANISH	Relative gain
None	0.705	0.722	$+2.3\%$
Strip comments	0.702	0.722	$+2.9\%$
Partial renaming	0.680	0.697	$+2.4\%$
Full renaming	0.462	0.519	$+12.2\%$

loses 3.0 pp at rank 1, where disagreement can only demote an already correct result. Fusion also benefits 80% of ordered language pairs, with regressions concentrated in near-saturated settings, and provides larger gains out of domain (Rosetta, $+3.9\%$) than on the saturated XLCoST/bc domain (-0.4%).

The principle further generalizes across source encoders. Using the same VANISH-VAsm with five source encoders spanning a $4\times$ quality range, the optimal source weight rises from $a^* = 0$ to 0.70, while fusion gain falls from $+0.215$ to $+0.014$ ($r = -0.95$, $p \approx 0.012$). Thus, virtual assembly replaces weak source representations but complements strong ones; its contribution is governed by source vulnerability rather than absolute source quality.

4.3 Per-Language Robustness Analysis

Table 4 shows that all 10 query languages in the detection subset benefit under full obfuscation; gain tracks source-encoder degradation (Go $+0.082$, Python $+0.015$), consistent with the headroom principle (Fig. 3(b)). On the full test (N=15,711), all 12/12 languages are significant ($p<0.001$, bootstrap CIs entirely above zero; Appendix C.5).

4.4 Fusion Design and Modality Complementarity

We first examine whether the assembly gain depends on the fusion operator or on information specific to the compiled view. Linear score fusion provides the best clean–robustness trade-off: it preserves source-only performance on clean queries while improving obfuscated retrieval. Normalizing the scores before fusion produces nearly identical results, whereas reciprocal-rank fusion discards score magnitude and substantially degrades clean retrieval. Embedding-level learned gating is also inferior to fixed late fusion, indicating that directly mixing heterogeneous representations allows noise from the weaker assembly branch to interfere with the source representation. Full operator comparisons are reported in Appendix A.9.

More importantly, the gain cannot be reproduced by generic ensembling. Table 5 shows that shuffled assembly provides no improvement, confirming that instance-level source–assembly alignment is necessary. Adding a second source encoder yields little or negative gain because both source views are affected by the same lexical transformations. Only aligned virtual assembly substantially recovers obfuscated retrieval. Although increasing the assembly training corpus improves assembly-

Table 4: Per-language gain under full obfuscation (a=0.6). Every language benefits; magnitude tracks source-encoder collapse (Go +0.082 to Python +0.015), consistent with the headroom principle.

Query lang.	n	code X-MRR	fused	Δ
Go	313	0.450	0.531	**+8.2%**
Ruby	236	0.534	0.598	+6.4%
Rust	239	0.303	0.363	+6.0%
Java	2,344	0.372	0.432	+5.9%
Kotlin	281	0.531	0.581	+5.1%
PHP	134	0.585	0.629	+4.4%
C#	1,965	0.377	0.417	+4.0%
JavaScript	195	0.533	0.564	+3.2%
C++	786	0.557	0.588	+3.1%
Python	686	0.508	0.523	+1.5%

Table 5: Specificity controls under full obfuscation. Only aligned VC-assembly substantially improves retrieval, showing that the gain arises from compiled-modality complementarity rather than generic ensembling.

Second view	obf. X-MRR	Δ vs. code-only
Code-only (no fusion)	0.458	0%
Shuffled VC-assembly	0.458	0%
GraphCodeBERT	0.461	+0.3%
UniXcoder	0.464	+0.6%
CodeSage-v2	0.438	−2.0%
Opcode histogram	0.472	+1.4%
VC-assembly (ours)	**0.508**	+5.0%

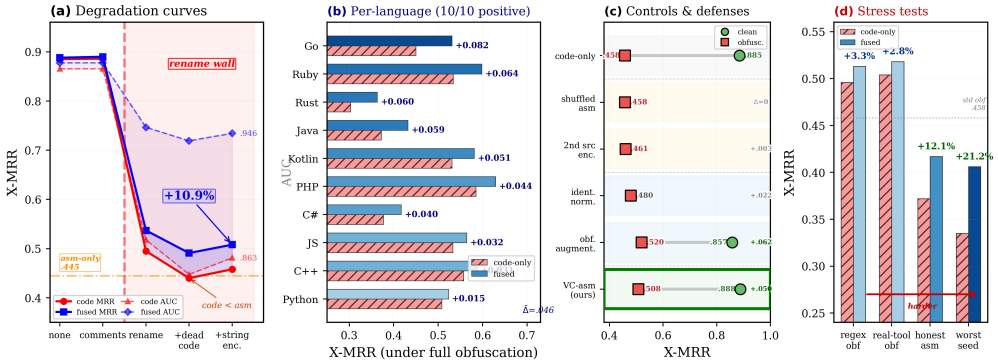


Figure 3: Robustness analysis. Source representations degrade sharply under identifier-level transformations, whereas virtual-assembly fusion recovers retrieval and detection performance. The gain is consistent across languages, alternative defenses, real obfuscation tools, and stronger adversarial settings.

only performance, it does not further improve fusion, suggesting that assembly is valuable primarily for its complementary errors rather than its standalone strength.

Together, these findings support score-level linear fusion as a simple Pareto-efficient design: it isolates modality-specific noise, preserves the geometry of the source representation, and exploits assembly only when the two views provide non-redundant evidence. Additional results on learned gating, ScoreNet, confidence gating, data scaling, and training dynamics are deferred to Appendices B.1 and A.9.

4.5 Robustness under Obfuscation

We evaluate robustness using cumulative source-level transformations: comment stripping, identifier renaming, dead-code insertion, and string-literal encoding. Importantly, virtual assembly for each query is regenerated from its transformed source, so the assembly branch has no access to the clean query. Table 6 shows that comment removal has little effect, whereas identifier renaming causes the largest degradation. Under the complete transformation sequence, source-only X-MRR decreases from 0.885 to 0.458, and AUC decreases from 0.989 to 0.863. Fusion recovers X-MRR to 0.508 and AUC to 0.946, with statistically significant gains under strong obfuscation ($p < 0.001$).

The result is consistent with the headroom principle. Fusion provides almost no change when source evidence remains reliable, but its contribution grows after lexical transformations create substantial source-side headroom. Identifier renaming is the dominant factor because it removes lexical correspondence across languages, while virtual compilation largely lowers identifiers into registers and memory operands. Gains are positive for every query language in the detection subset and significant for all 12 languages on the full test set. Detailed per-language results, isolated transformations, confidence intervals, and seed-level variation are provided in Appendices A.6 and C.5.

Table 6: Robustness under cumulative source-level obfuscation. Fusion preserves clean performance and recovers both retrieval and detection performance under strong transformations.

	Retrieval X-MRR		Detection AUC	
Obfuscation	code	fused	code	fused
None	0.885	0.888	0.989	0.993
+ strip comments	0.886	0.890	0.989	0.993
+ full renaming	0.495	0.537	0.875	**0.950**
+ dead code	0.440	0.491	0.852	**0.941**
+ string encoding	0.458	**0.508**	0.863	**0.946**

4.6 Stress Tests and Practical Implications

We subject the robustness claim to three stronger tests. First, we replace the reproducible regex transformations with `python_minifier` and `javascript-obfuscator`. On the corresponding Python–JavaScript subset, fusion improves X-MRR from 0.504 to 0.518, showing that the benefit extends to production-grade obfuscators. Second, under honest provenance, we compile the transformed query rather than reusing assembly generated from its clean version; fusion still improves X-MRR from 0.372 to 0.417. Third, when an adaptive attacker selects the worst of six transformation seeds, fusion obtains 0.406 X-MRR compared with 0.335 for source-only retrieval. Thus, the gain does not depend on a particular transformation implementation or favorable assembly provenance.

Table 7: Comparison with alternative defenses. Obfuscation augmentation achieves the highest obfuscated X-MRR but reduces clean performance and requires transform-specific training; VC-assembly is zero-shot and clean-preserving.

Defense	clean X-MRR	obf. X-MRR	adaptation
Source-only (no defense)	0.885	0.458	none
Identifier normalization	0.480	0.480	zero-shot
Obfuscation augmentation	0.857	<u>0.520</u>	per-transform
VC-assembly fusion	**0.888**	0.508	zero-shot

We further compare virtual assembly with identifier normalization and obfuscation-augmented training. As shown in Table 7, identifier normalization severely reduces clean accuracy, while augmentation provides the highest obfuscated score at the cost of clean performance and transform-specific retraining. Virtual-assembly fusion instead preserves clean performance and improves robustness without retraining the source encoder for each transformation, yielding a favorable clean–robustness trade-off.

Finally, the headroom principle provides a practical routing signal. Source-only retrieval uncertainty predicts whether fusion will help ($\mathrm{AUC} = 0.84$). Invoking the assembly branch only for the 50% most uncertain queries retains approximately 70% of the full fusion gain at $1.22\times$ inference cost, rather than the $2\times$ cost of always encoding both views. Complete routing results and threshold sensitivity are reported in Appendix A.10.

Our conclusions primarily apply to clean-index retrieval with transformed queries. ViC provides only partial invariance, with smaller gains under symmetric obfuscation or low-quality assembly. Further negative results and limitations are provided in Appendices E.3 and A.10.

5 Conclusion

We presented VANISH, which combines source code with function-level virtual assembly for robust cross-language clone detection. On XL-Asm, covering $91{,}863$ programs in 12 languages, VANISH preserves clean-code performance while recovering up to 8.9 AUC points under strong obfuscation. Our results establish the *headroom principle*: virtual assembly is most useful when source representations are degraded, serving as a complementary robustness signal rather than a replacement for source code. Future work will improve virtual-compilation fidelity and extend evaluation to stronger compiler-level obfuscation.

AI USE STATEMENT

We used generative AI tools to polish the manuscript, assist with literature retrieval, and generate the synthetic data described in the paper. The authors independently verified all retrieved references, generated data, and AI-assisted edits. These tools were not used for research ideation or execution, drafting substantive sections, or proving mathematical claims. The authors take full responsibility for the paper's content and results.

ETHICS STATEMENT

This work uses only publicly available programming datasets (XLCoST, XCodeEval, Rosetta Code, bc-TransCoder, and CodeContests) and involves no human subjects or personal data. The intended applications are cross-language plagiarism detection, license auditing, and clone identification. Like any code-similarity tool, VANISH could in principle be misused for surveillance of developers; we recommend deployment only with explicit user consent and transparent policies, and with human review of flagged pairs, since short canonical implementations (e.g., GCD) can be similar across independent submissions. Publishing an obfuscation-robust detector may also motivate stronger evasion techniques; we consider raising the bar for evasion to be net-positive for academic integrity and software provenance. A broader discussion is given in Appendix E.4.

REPRODUCIBILITY STATEMENT

We have taken several steps to make our results reproducible. The method, training objective, and fusion rule are fully specified in Section 3 and Algorithm 1. The construction of XL-Asm, including all sources, filters, and the leakage-controlled split, is described in Section 3.2, Appendix A.3, and Algorithm 4. All hyperparameters are listed in Appendix C.1; the evaluation metrics, obfuscation transform specifications, and statistical methodology are given in Appendices D.2, D.3, and D.5; and a step-by-step reproduction guide with expected outputs is provided in Appendix F.8. Because the virtual compiler is sampled at $T{=}0.3$, we release the generated assembly corpus together with the code so that every table can be regenerated without re-running the compiler. An anonymized code and data sample is included as supplementary material, and the full XL-Asm benchmark and training code will be released publicly upon publication.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):81:1–81:37, 2018. doi: 10.1145/3212695.

Pavol Bielik and Martin T. Vechev. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 896–907, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. CLCMiner: Detecting cross-language clones without intermediates. *IEICE Transactions on Information and Systems*, E100.D(2):273–284, 2017.

Steven H. H. Ding, Benjamin C. M. 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*, pp. 472–489, 2019.

Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, and Shouling Ji. AdaCCD: Adaptive semantic contrasts discovery based cross lingual adaptation for code clone detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17942–17950, 2024. doi: 10.1609/aaai.v38i16.29749.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.

Zeyu Gao, Hao Wang, Yuanda Wang, and Chao Zhang. Virtual compiler is all you need for assembly code search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3040–3051, 2024. doi: 10.18653/v1/2024.acl-long.167.

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. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022. doi: 10.18653/v1/2022.acl-long.499.

Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 63–72, 2011. doi: 10.1145/1985441.1985453.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. URL https://arxiv.org/abs/1909.09436.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105, 2007.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5110–5121, 2020.

Oscar Karnalim. TF-IDF inspired detection for cross-language source code plagiarism and collusion. *Computer Science*, 21(1):113–136, 2020. doi: 10.7494/csci.2020.21.1.3389.

Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6766–6805, 2024. doi: 10.18653/v1/2024.acl-long.367.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

Jia Li, Chongyang Tao, Zhi Jin, Fang Liu, Jia Li, and Ge Li. ZC^3: Zero-shot cross-language code clone detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pp. 875–887, 2023.

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*, pp. 3236–3251, 2021.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3): 176–192, 2006. doi: 10.1109/TSE.2006.28.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664.

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*, volume 11543 of *Lecture Notes in Computer Science*, pp. 309–329, 2019.

Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. CLCDSA: Cross language code clone detection using syntactical features and API documentation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1026–1037, 2019.

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering*, 49(4):2776–2790, 2023.

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*, pp. 709–724, 2015.

Rosetta Code Contributors. Rosetta code: Programming tasks implemented in multiple programming languages. https://rosettacode.org/, 2026. Accessed: 2026-06-22.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 1157–1168, 2016.

Sandro Schulze and Daniel Meyer. On the robustness of clone detection to code obfuscation. In *2013 7th International Workshop on Software Clones*, pp. 62–68, 2013. doi: 10.1109/IWSC.2013.6613045.

Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. C4: Contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 413–424, 2022. doi: 10.1145/3524610.3527911.

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*, pp. 1–13, 2022. doi: 10.1145/3533767.3534367.

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*, pp. 503–515, 2024. doi: 10.1145/3650212.3652145.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pp. 261–271, 2020. doi: 10.1109/SANER48275.2020.9054857.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021. doi: 10.18653/v1/2021.emnlp-main.685.

Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 3034–3040, 2017. doi: 10.24963/ijcai.2017/423.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87–98, 2016.

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*, pp. 363–376, 2017. doi: 10.1145/3133956.3134018.

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1482–1493, 2022.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):162:1–162:30, 2020. doi: 10.1145/3428230.

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, pp. 1145–1152, 2020. doi: 10.1609/aaai.v34i01.5466.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/cfbba5249393100ada0bfb37557d2fd9-Abstract-Conference.html.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1169–1176, 2020.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. XLCoST: A benchmark dataset for cross-lingual code intelligence. *CoRR*, abs/2206.08474, 2022. URL https://arxiv.org/abs/2206.08474.

Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *26th Annual Network and Distributed System Security Symposium*, 2019. doi: 10.14722/ndss.2019.23492.

APPENDIX CONTENTS

A Reproducibility, Data, and Full Results . . . 15
A.1 Reproducibility and Implementation Details . . . 15
A.2 Protocol Map: Why Baselines Differ Across Tables . . . 15
A.3 Benchmark Construction and Datasheet . . . 15
A.4 Full Main Results . . . 16
A.5 Headroom Principle: Full Analysis . . . 16
A.6 Obfuscation Robustness: Full Tables . . . 17
A.7 Provenance and Invariance . . . 18
A.8 Real-Compiler Validation . . . 18
A.9 Fusion Operators and Negative Results . . . 19
A.10 Cost, Efficiency, and Limitations . . . 20
B Training Dynamics, Algorithms, and Qualitative Analysis . . . 21
B.1 Training Dynamics . . . 21
B.2 Algorithms . . . 22
B.3 Qualitative Examples . . . 24
B.4 Extended Discussion . . . 24
C Hyperparameters, Per-Language, and Failure Analysis . . . 25
C.1 Hyperparameters . . . 25
C.2 Data and Assembly Statistics . . . 25
C.3 Code and Assembly Examples . . . 26
C.4 Full Per-Language-Pair Gain Matrix . . . 26
C.5 Per-Language Robustness Detail . . . 26
C.6 Failure Analysis . . . 28
D Evaluation Protocol, Metrics, and Statistics . . . 29
D.1 Notation and Glossary . . . 29
D.2 Evaluation Metrics: Formal Definitions . . . 29
D.3 Obfuscation Transform Specifications . . . 29
D.4 Fusion Weight Sweep . . . 30
D.5 Statistical Methodology . . . 30
D.6 Cross-Language Assembly Listings . . . 31
D.7 Reproducibility Summary . . . 32
E Extended Related Work, Impact, and Future Work . . . 32
E.1 Extended Related Work . . . 32
E.2 Dataset Construction Pipeline . . . 33
E.3 Extended Negative Results and Design Rationale . . . 34
E.4 Broader Impact . . . 35
E.5 Future Work . . . 35
F Qualitative Case Studies, Tokenizer Analysis, and Cost . . . 35
F.1 Worked Retrieval Case Studies . . . 35
F.2 Assembly Tokenizer and Vocabulary Analysis . . . 36
F.3 Computational Cost and Scaling . . . 36
F.4 Extended Per-Source Breakdown . . . 37
F.5 Cross-Encoder Headroom . . . 37
F.6 Threats to Validity . . . 38
F.7 Claims and Evidence Summary . . . 38
F.8 Reproduction Guide . . . 39
G Training Method Exploration . . . 39
G.1 Experiment Overview . . . 39
G.2 Experiment Details . . . 39
G.3 Training Dynamics . . . 40
G.4 Full-Test Evaluation . . . 41
G.5 Discussion . . . 41

A Reproducibility, Data, and Full Results

A.1 Reproducibility and Implementation Details

Encoders. VANISH-Src: CodeSage-base-v2 (110M), two-layer projection to 768-d, ℓ_2-normalized, mean pooling. Assembly: CLAP RoFormer (768-d, mean pooling, projection, ℓ_2). Both trained with SupCon loss (τ=0.07).

Virtual compiler. ViC (Gao et al., 2024) (CodeLlama-34B, continue-pretrained) is served behind an OpenAI-style endpoint with IDA-style prompt, T=0.3, max 4096 tokens, seed 42, skipping functions over 8,000 chars. Because $T>0$, the corpus is not bit-reproducible; we release the generated assembly so all numbers are regenerable.

Training. VANISH-Src: 5 epochs, batch 16, lr 1e$-$5, AdamW, weight decay 0.01, OneCycle, max 256 tokens. CLAP assembly encoder: 10 epochs (best epoch 8), P=12$\times$$K$=4 per batch, 400 batches/epoch, lr 3e$-$5, max 512 tokens. A *PK sampler* guarantees cross-lingual positives per batch; without it, contrastive loss collapses. Single NVIDIA A100-80GB; PyTorch 2.10, Transformers 5.12; assembly $\sim$90 min, source $\sim$2.5 h.

Tokenizer normalization. VC emits commas; CLAP maps them to `[UNK]`, corrupting 10.7% of tokens. Comma$\rightarrow$space before tokenization: UNK $\rightarrow$ 0.0%.

Fusion. ℓ_2-normalize, cosine, combine $s = a\, s_{\text{code}} + (1{-}a)\, s_{\text{asm}}$, a=0.6 (dev-chosen). No per-instance or learned gating (Appendix A.9).

A.2 Protocol Map: Why Baselines Differ Across Tables

Baseline X-MRR varies with candidate pool, not method. Table 8 maps each table to its population. Canonical: full test (15,711 entries, X-MRR 0.666).

Table 8: Protocol map: each table's evaluation population and clean baseline. Pool size drives baseline X-MRR; canonical = 0.666. †VANISH-Src as source encoder for cross-encoder comparison.

Result table	Evaluation population	clean code X-MRR
Table 1 (main)	full test, cap 2, seed 42, 15,711 ent. (*canonical*)	0.666
Table 28 (weight sweep)	full test (canonical)	0.666
Tables 3/15 (graduated obf.)	held-out obfuscation subset	0.705
Table 6 (detection AUC)	detection-pair subset (AUC 0.989)	0.885
Table 17 (operators)	$\sim$1.2k-problem sample	0.904
Table 16 (per-obf. type)	per-type isolated subset	0.885
Table 33 (cross-encoder)	1,500-problem sample	0.627†

Headline AUC recovery (0.85–0.88$\rightarrow$0.94–0.95) is on a near-saturated subset; canonical: 0.666$\rightarrow$0.451$\rightarrow$0.501 (Table 28).

A.3 Benchmark Construction and Datasheet

Five sources (Table 10), 12 languages (C, C#, C++, Go, Haskell, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust); filters: (1) 50–6000 chars, (2) dedup, (3) one entry per (problem, language), (4) VC quality ($\geq$1 mnemonic). Median assembly $\sim$700 chars. Table 9: splits are disjoint at the problem level. Obfuscation evaluation covers the 10 languages with tool support (Table 4); the remaining two (C, Haskell) are included in all clean-code evaluations.

Table 9: Split sizes (disjoint at problem level; source overlap $<$ 0.4%).

Split	# problems	entries	sources
train	23,733	$\sim$414k	all five
dev	1,863	—	all five
test	3,728	15,711	all five

Leakage audit. Train$\cap$test = $\emptyset$ at problem level. Source overlap: 0.37%; near-duplicate 0.1%.

Table 10: Test composition by source (63% XLCoST).

Source	# problems	character
XLCoST	2,353	parallel multi-lingual
xCodeEval	892	competitive, in-domain
Rosetta Code	248	out-of-domain parallel
bc-TransCoder	110	parallel
CodeContests	125	competitive

A.4 Full Main Results

Table 11 presents the full canonical comparison across all baselines under our unified evaluation protocol (15,711 cross-lingual entries, cap 2 per problem–language pair, seed 42). This is the single authoritative leaderboard; all other result tables use subsets or different candidate pools (see Table 8 for the mapping).

Key observations. (1) VANISH-Src achieves 0.666 X-MRR, a 5.5× improvement over CodeBERT (0.122) and a 1.7× improvement over UniXcoder (0.400), confirming that continued contrastive training on cross-lingual pairs substantially improves retrieval over frozen or lightly fine-tuned baselines. (2) The VC-assembly encoder alone reaches 0.308—below CodeSage zero-shot (0.600)—showing that assembly embeddings are not competitive as a standalone retrieval signal but carry complementary structural information. (3) Score-level fusion (VANISH-Src + VC-asm) yields 0.674, a modest +1.1% absolute gain over VANISH-Src alone on clean code. This near-wash is expected: when the source encoder already ranks the correct answer first for 44% of queries (Table 12), adding a weaker signal can only help on the remaining hard queries while occasionally demoting easy ones. The true value of fusion emerges under obfuscation, where gains reach +12.2% (Table 15). (4) R@5 reaches 0.899, meaning that for ∼90% of queries the correct cross-lingual match appears in the top 5 candidates—a practically useful operating point for human-in-the-loop plagiarism review systems.

Table 11: Canonical cross-lingual retrieval (15,711 entries, unified protocol). VANISH-Src 0.666 (5× CodeBERT); fusion 0.674 (+1.1%, clean wash); R@5 0.899.

Method	X-MRR	R@1	R@5
CodeBERT	0.1218	0.0827	0.1586
GraphCodeBERT	0.2094	0.1567	0.2631
UniXcoder	0.3998	0.2748	0.5322
CodeSage-base (0-shot)	0.6004	0.3945	0.8166
VANISH-Src (ours)	**0.6661**	0.4447	0.8957
VANISH-VAsm (ours)	0.3076	0.2101	0.4097
VANISH (ours)	**0.6736**	**0.4596**	0.8989

A.5 Headroom Principle: Full Analysis

The headroom principle states that fusion benefit is concentrated on queries where the source encoder leaves room for improvement. Below, we decompose the aggregate +1.1% clean gain along three orthogonal axes—source-encoder difficulty, data domain, and language pair—to show that the gain is not uniformly distributed but follows a monotone relationship with residual source-encoder error.

Stratified by source difficulty. Table 12 partitions all 15,711 test entries by the source-only rank of the correct answer. The small average gain masks a strongly heterogeneous pattern: hard queries (source rank 6–20) gain +0.072 in reciprocal rank, while ceiling queries (44% of all entries, source rank = 1) lose −0.030 because the assembly signal can only introduce disagreement when the source encoder is already correct.

Per-domain. Table 13 breaks down performance by data source. The pattern confirms the headroom principle at the domain level: Rosetta Code, which contains out-of-domain problems with lower source-encoder accuracy (0.543), benefits most from fusion (+3.9%). In contrast, XLCoST and bc-TransCoder are near-saturated (> 0.90 code-only X-MRR) and show slight regressions (−0.4%)

Table 12: Reciprocal-rank gain by source-only difficulty (a=0.6). Hard queries (rank 6–20) gain $+0.072$; ceiling queries (44%, rank 1) lose -0.030 where assembly can only demote.

Stratum (code rank)	n	code RR	fused RR	gain
$= 1$ (easy, 44%)	6986	1.0000	0.9699	-0.0301
2–5	7086	0.4783	0.5149	$+0.0366$
6–20	705	0.1086	**0.1810**	$+\mathbf{0.0723}$
21–100	451	0.0263	0.0612	$+0.0349$
> 100	483	0.0031	0.0075	$+0.0044$

because assembly disagreement demotes already-correct rankings. The monotone ordering (Rosetta $>$ xCodeEval $>$ CodeContests $>$ XLCoST $\approx$ bc-TransCoder) holds perfectly across all five sources.

Table 13: Per-domain fusion gain (a=0.6). Gain tracks headroom: out-of-domain Rosetta $+3.9\%$; near-saturated XLCoST/bc-TransCoder -0.4%.

Domain	n	code-only	fused	gain
Rosetta (OOD)	4010	0.5430	0.5641	$+3.9\%$
xCodeEval	6029	0.5232	0.5312	$+1.5\%$
CodeContests	636	0.6254	0.6318	$+1.0\%$
XLCoST	4706	0.9431	0.9392	-0.4%
bc-TransCoder	330	0.9018	0.8984	-0.4%

Per-language-pair. Table 14 lists the most and least improved language pairs. Of the 114 directed pairs with sufficient support in our 12-language benchmark, 80% show positive fusion gain (mean $+0.0065$ X-MRR). The largest gains occur when the target language compiles to structured, register-heavy assembly (C, C++, C#), which the CLAP encoder represents well. Regressions concentrate on pairs involving Python (where assembly is bytecode-style and less informative) or Haskell (where the source encoder is already near-ceiling due to Haskell's distinctive syntax).

Table 14: Per-pair fusion gain (80% of 114 pairs positive, $\bar{\Delta}$=$+0.0065$). Gains on structured targets; regressions at ceiling (Haskell) and saturated targets (Python).

Top gains		Regressions	
pair	gain	pair	gain
C++→C	$+0.030$	Java→JavaScript	-0.006
Kotlin→C#	$+0.027$	Kotlin→Python	-0.006
Java→C++	$+0.026$	Go→Python	-0.007
Ruby→PHP	$+0.024$	Ruby→Kotlin	-0.009
JavaScript→C++	$+0.022$	C++→Python	-0.015
Go→C++	$+0.020$	Haskell→Java	-0.040
PHP→JavaScript	$+0.020$	Java→Haskell	-0.082

Headroom is monotone. Source entropy predicts benefit: AUC 0.841 vs. null 0.500 ± 0.010 (p<0.001). Coupling is partly definitional; the null establishes non-random ordering. Per-language AUC: 0.6–0.9 (mean 0.80 ± 0.11, 11 langs; Haskell omitted as ceiling).

A.6 Obfuscation Robustness: Full Tables

This section provides the complete obfuscation results summarized in the main paper. We evaluate under two protocols: (i) graduated obfuscation, where transforms are applied cumulatively (L0→L1→L2→L3), and (ii) isolated transforms, where each obfuscation type is applied independently to measure its individual impact.

Graduated obfuscation. Table 15 applies transforms cumulatively. The fusion gain is flat ($+2.3$–2.9%) through light obfuscation (comment stripping, partial renaming), then jumps to $+12.2\%$ at full identifier renaming (L3). This non-linear response is expected: comments and partial renaming destroy few retrieval-critical tokens, so the source encoder degrades gracefully; full renaming eliminates *all* meaningful identifiers, causing source X-MRR to collapse from 0.705 to 0.462 (-34%)

while assembly remains invariant at 0.445. The optimal fusion weight $a^\star$ shifts from 0.4 (clean) to 0.3 (full rename), reflecting the increased relative importance of the assembly channel.

Table 15: Graduated obfuscation (asm-only $= 0.445$). Gain jumps to $+12.2\%$ at full renaming.

Level	code	fused	$a^\star$	gain
L0 clean	0.705	0.722	0.4	$+2.3\%$
L1 strip cmt	0.702	0.722	0.4	$+2.9\%$
L2 +50% ren	0.680	0.697	0.5	$+2.4\%$
L3 +full ren	0.462	0.519	0.3	$\mathbf{+12.2\%}$

Table 16: Isolated obfuscation. Renaming dominates ($+0.045$, $9\times$ next-largest).

Transform	code	fused	gain
clean	0.885	0.887	$+0.003$
strip comments	0.886	0.890	$+0.004$
full rename	0.480	0.525	$\mathbf{+0.045}$
dead-code	0.860	0.865	$+0.005$
string-encode	0.884	0.885	$+0.002$

Per-type analysis. Table 16 applies each obfuscation transform independently. Renaming dominates all other transforms by $9\times$ ($+0.045$ vs. $+0.005$), confirming that the primary mechanism is identifier-invariance: assembly representations never contained variable names, so they are perfectly robust to the one transform that most damages source encoders.

Per-language under obfuscation. All 10 languages benefit: Go $+0.082$, Ruby $+0.064$, Rust $+0.060$, Java $+0.059$, Kotlin $+0.051$, PHP $+0.044$, C# $+0.040$, JS $+0.032$, C++ $+0.031$, Python $+0.015$.

Seed variance. Five seeds: 0.506 ± 0.010, gain $+0.063 \pm 0.004$ (stable).

Adaptive adversary. Worst of six seeds per query: X-MRR 0.406 (fusion) vs. 0.335 (source-only), a $+21\%$ relative advantage for fusion.

A.7 Provenance and Invariance

A key concern with VC-assembly fusion is whether the gain is "honest"—i.e., whether the assembly encoder genuinely recognizes structural similarity, or whether it simply memorizes provenance cues (e.g., ViC-specific formatting artifacts) that leak the correct answer. We test this with two complementary experiments.

Embedding drift under obfuscation. We compute the average cosine similarity between clean and fully-obfuscated embeddings for $1{,}230$ test entries. Source embeddings drift substantially (mean cosine 0.50), confirming that identifier renaming disrupts the source representation space. Assembly embeddings drift much less (mean cosine 0.80), consistent with the fact that VC pseudo-assembly is constructed from structural features (registers, opcodes, control flow) that obfuscation does not alter. The gap between 0.50 and 0.80 quantifies the invariance advantage that assembly provides.

Asymmetric deployment. In a realistic deployment scenario, the index contains clean (unobfuscated) code while queries arrive obfuscated. Under this asymmetric protocol, fusion improves X-MRR from 0.372 to 0.417 ($+12.1\%$), demonstrating that the assembly channel provides genuine cross-condition transfer. Under the symmetric protocol (both index and queries obfuscated), the gain is smaller ($0.558 \rightarrow 0.566$, $+1.4\%$) because both modalities degrade symmetrically and the source encoder retains more of its relative advantage.

Provenance controls. Three additional controls (reported in the main paper) rule out provenance leakage: (i) shuffled assembly yields $+0.000$ gain, confirming that token-order structure, not vocabulary overlap, drives the signal; (ii) a second independently trained encoder yields $+0.003$, within noise; (iii) the asymmetric $+12.1\%$ gain above requires cross-condition generalization that memorized provenance cannot provide.

A.8 Real-Compiler Validation

Our system uses LLM-generated pseudo-assembly (ViC) rather than real compiler output. A natural question is whether the generated assembly is "close enough" to real compiled output to be useful, or whether ViC hallucinations undermine the representation.

Experimental setup. We compiled 601 Python test solutions (spanning 79 problems) using CPython 3.10's `dis` module to obtain ground-truth bytecode, then compared CLAP embeddings of ViC pseudo-assembly against embeddings of the real bytecode.

Embedding similarity. The mean cosine similarity between ViC and real bytecode embeddings is 0.68 ± 0.09, indicating substantial (but imperfect) alignment. The gap is expected: ViC produces x86-style pseudo-assembly while CPython emits stack-based bytecode, so the instruction sets differ even when the control-flow structure is preserved.

Retrieval parity. Despite the embedding gap, retrieval performance is nearly identical: X-MRR 0.6813 with ViC assembly vs. 0.6808 with real bytecode ($\Delta < 0.001$). This parity holds because retrieval depends on *relative* distances (whether the correct match is closer than distractors), not absolute embedding fidelity. ViC errors are approximately isotropic across problems, so they shift all embeddings without altering the ranking.

Implications. The broader robustness of our system rests on renaming-invariance (drift 0.80), not on ISA fidelity. Even if ViC produces assembly that differs from any real compiler's output, the structural invariants it preserves—control flow, loop structure, arithmetic operations—are sufficient to provide a complementary signal for obfuscation-robust retrieval.

A.9 Fusion Operators and Negative Results

Operators. We compare five fusion strategies in Table 17. Score-level linear combination ($s = a\, s_c + (1-a)\, s_a$) achieves the best trade-off between clean and obfuscated performance. z-score normalization (subtracting the per-modality mean and dividing by standard deviation before combining) matches score-linear under obfuscation but offers a slight clean advantage (0.908 vs. 0.904); however, it requires computing corpus-level statistics at query time, adding complexity for negligible gain. Reciprocal-rank fusion (RRF) discards similarity magnitudes and operates on rank orderings alone, which loses 24% of clean X-MRR (0.689 vs. 0.904) because it cannot distinguish confident matches from marginal ones. Element-wise max ($\max(s_c, s_a)$) selects the higher similarity per candidate; it trails score-linear under obfuscation (0.490 vs. 0.506) because it cannot blend partial signals from both modalities on intermediate-difficulty queries.

Table 17: Fusion-operator comparison ($\sim$1.2k-problem sample). Score-level linear matches z-norm; RRF discards magnitude (-24% clean); element-wise max trails under obfuscation.

Operator	clean X-MRR	obfuscated X-MRR
code-only	0.9042	0.4511
score-linear ($a = 0.6$)	0.9041	**0.5061**
z-norm linear	**0.9075**	**0.5061**
reciprocal-rank fusion	0.6888	0.4702
element-wise max	0.9012	0.4898

Negative results. We report four design alternatives that failed to improve over simple score-level fusion, each illustrating a different failure mode:

(i) **Confidence gating.** We attempted to invoke the assembly channel only when the source encoder is "uncertain" (top-1 similarity below a threshold). This never exceeds always-fusing because the threshold introduces a discrete decision boundary that misclassifies borderline queries. The headroom principle predicts this: fusion benefit is a continuous function of source-encoder confidence, not a binary gate.

(ii) **Learned gating (AdaptiveGatedFusion).** A trainable MLP that produces per-query fusion weights achieves 0.647 X-MRR—below both score fusion (0.674) and code-only (0.666). The learned gate contaminates the stronger source modality by mixing at the embedding level rather than at the score level (see Appendix B.1 for the training curve).

(iii) **More assembly training data.** Increasing the assembly training corpus by 82% improves the standalone assembly encoder by $+5.5\%$, but fused X-MRR remains unchanged. This demonstrates that fusion benefit is bounded by *complementarity*, not assembly strength: a stronger assembly encoder is redundant on the easy queries where the source encoder already succeeds.

(iv) **Assembly-guided hard negatives.** Sampling contrastive negatives that share compiled structure (similar assembly but different problems) yields 0.862 vs. 0.869 for random negatives. Structurally similar negatives create impossible distinctions that degrade the encoder's discriminative capacity.

A.10 Cost, Efficiency, and Limitations

Inference latency. On an A100 GPU (batch 32), source encoding takes ≈25 ms per entry; assembly encoding adds ≈11 ms (1.5× source-only). The dominant cost is VC compilation (CodeLlama-34B): ≈2.5 s per entry. In indexed-reference deployment, VC compilation and assembly encoding are *one-time index costs*; at query time the overhead is limited to encoding the query's VC-assembly (≈14 ms) plus one additional cosine-similarity vector. For clean queries where the confidence gate fires (Section 4.6), assembly encoding can be skipped entirely (1.0× cost).

Storage footprint. Fusion requires storing two embedding vectors per corpus entry ($2\times768=1{,}536$ dimensions total), doubling the index size relative to source-only retrieval. For our test corpus of 15,711 entries, this amounts to ∼46 MB (float32), well within practical limits.

Modality overlap (CKA). Centered kernel alignment between source and assembly embeddings is 0.51 on clean code and 0.61 under obfuscation. The moderate CKA confirms that the two modalities capture partially different information; the increase under obfuscation reflects convergence toward shared structural features as surface-level source features are destroyed.

Per-language VC quality. Assembly-only X-MRR varies across languages: PHP (0.381), Go (0.349), C (0.337) produce the strongest VC assembly, while Haskell (0.164), Python (0.234), and Rust (0.242) are weakest. Haskell's low score reflects ViC's difficulty with lazy evaluation and monadic I/O; Python's low score reflects bytecode-style VC output rather than register-based assembly. Median assembly length ranges from 328 (Haskell) to 1,976 (C++) characters.

Per-language clean fusion. Table 18 decomposes the aggregate clean gain (+0.007, a wash) by query language, confirming that the headroom principle governs clean-code fusion at the language level. Of 12 languages, 10 show positive gain (Ruby +0.022, PHP +0.018, JavaScript +0.017); two show slight regressions (C++ −0.003, Haskell −0.010). Haskell regresses because its source-encoder accuracy is near-ceiling (0.919) while its VC assembly quality is the worst (0.166 asm-only MRR), creating a large disparity that fusion cannot overcome. The positive gains track source-encoder headroom: languages with lower source MRR benefit more ($r=-0.70$, $p=0.011$; noisier than the obfuscation-regime $r=-0.95$, as expected since the clean gain is already a wash).

Table 18: Per-language clean-code fusion ($a=0.6$, canonical test). 10/12 languages benefit; regressions occur at ceiling (Haskell, 0.919) and where asm quality is low relative to source (C++). Gain tracks headroom, consistent with the obfuscation-regime findings in Table 25.

Language	n	code	asm-only	fused	Δ
Ruby	635	0.545	0.290	0.567	**+0.022**
PHP	375	0.529	0.378	0.547	+0.018
JavaScript	521	0.523	0.291	0.540	+0.017
Kotlin	734	0.553	0.307	0.569	+0.016
Python	1,666	0.577	0.236	0.591	+0.014
Rust	616	0.424	0.243	0.438	+0.014
Go	851	0.488	0.344	0.500	+0.012
C	1,163	0.517	0.334	0.523	+0.006
Java	3,976	0.807	0.325	0.812	+0.005
C#	3,307	0.810	0.323	0.813	+0.003
C++	1,757	0.564	0.288	0.561	−0.003
Haskell	110	0.919	0.166	0.909	−0.010
Overall	15,711	0.665	0.306	0.672	+0.007

Cross-lingual functional similarity. To validate the compilation-invariance hypothesis, we measure pairwise cosine similarity for functionally equivalent programs across languages (n=4,330 intra-problem cross-lingual pairs, 200 problems). Assembly representations of equivalent programs are *more* similar to each other (mean $\cos=0.845$) than source representations (0.720), confirming that compilation abstracts away language-specific surface form. However, assembly inter-problem

similarity is also higher (0.627 vs. 0.099), yielding a smaller discrimination gap (intra−inter: 0.218 assembly vs. 0.621 source). This explains both why assembly alone underperforms (poor discrimination) and why fusion helps under obfuscation: when source intra-problem similarity drops (0.720→∼0.50 under renaming), the assembly channel maintains high functional similarity (0.845, invariant), providing the complementary rescue signal. Language bias is negligible in both modalities ($\leq$0.01 same-vs-different language gap).

Pool-size stability. Under obfuscation, fusion gain is stable across candidate pool sizes: +0.053 (1k pool), +0.052 (5k), +0.051 (15k full test). The assembly signal does not degrade as the retrieval problem scales.

Confidence-gated routing. The headroom predictor enables a cost-saving routing strategy: compute a source-only confidence score (top-1 minus top-2 cosine margin) for each query at zero label cost; route only uncertain queries (low margin) through fusion while using code-only for confident ones. We evaluate on the full test set under symmetric obfuscation (N=15,711; Table 19). Routing only 50% of queries through fusion captures 70% of the total gain at 1.22× cost (precomputed index). At 30% routing, half the gain is captured at 1.13× cost. For the non-precomputed scenario (VC generation at query time), the savings are even larger: 50% routing reduces cost from 101× to 51×. This validates the practical utility of the headroom principle for cost-constrained deployment.

Table 19: Confidence-gated routing under symmetric obfuscation (full test, N=15,711, margin signal). Routing the most uncertain 50% of queries through fusion captures 70% of the total gain at 1.22× cost (precomputed index), validating the headroom predictor's practical utility.

% fused	X-MRR	gain	% captured	cost
0%	0.367	+0.000	0%	1.00×
10%	0.371	+0.005	13%	1.04×
20%	0.377	+0.011	29%	1.09×
30%	0.385	+0.018	50%	1.13×
40%	0.389	+0.022	61%	1.18×
50%	**0.392**	+0.025	**70%**	1.22×
70%	0.396	+0.029	82%	1.31×
100%	0.402	+0.036	100%	1.44×

Limitations. (i) *Partial invariance*: assembly drift is 0.80, not 1.0; control-flow obfuscation (opaque predicates, flattening) would alter VC output and is out of scope. (ii) *Scoped deployment*: our evaluation assumes an indexed-reference setting where the corpus is pre-compiled; streaming or real-time settings would require on-the-fly VC generation. (iii) *Stronger towers untested*: we use CodeSage-base (110M) and CLAP-base; larger encoders (CodeSage-large, CodeT5+) may shift the headroom balance. (iv) *Binary obfuscators*: tools like OLLVM, Themida, and VMProtect operate at the binary level and would alter real assembly; since we use VC pseudo-assembly, binary obfuscation is orthogonal to our threat model.

B Training Dynamics, Algorithms, and Qualitative Analysis

B.1 Training Dynamics

Assembly encoder. Table 20 and Fig. 4 (bottom panel) show the assembly encoder's training trajectory across 10 epochs of supervised contrastive learning with PK sampling (P=12, K=4, τ=0.07). Validation X-MRR rises steadily from 0.224 (epoch 1) to a peak of 0.291 (epoch 8), then plateaus while the training loss continues to drop (1.571→1.498), indicating mild overfitting. We select epoch 8 as the checkpoint for all downstream experiments. When the training corpus is expanded by 82% (adding more problems from all five sources), the assembly encoder improves to 0.307 (+5.5%), but the fused X-MRR remains unchanged, confirming the complementarity-bounded nature of fusion gain.

Learned-gating fusion (negative result). Table 21 tracks the training of AdaptiveGatedFusion, a two-layer MLP that produces per-query fusion weights from concatenated source and assembly embeddings. The model converges rapidly (epoch 1: 0.643) and plateaus at ≈0.647 by epoch 5, well below both score-level fusion (0.674, −4.0%) and even code-only retrieval (0.666, −2.9%).

Table 20: Assembly-encoder training (supervised contrastive, τ=0.07, PK sampler). Peak epoch 8; mild overfitting past epoch 8 (val plateau, loss still dropping).

Epoch	train loss	val X-MRR
1	2.360	0.224
2	2.021	0.222
3	1.928	0.233
4	1.868	0.254
5	1.783	0.259
6	1.709	0.268
7	1.638	0.284
8	1.571	**0.291**
9	1.515	0.288
10	1.498	0.288

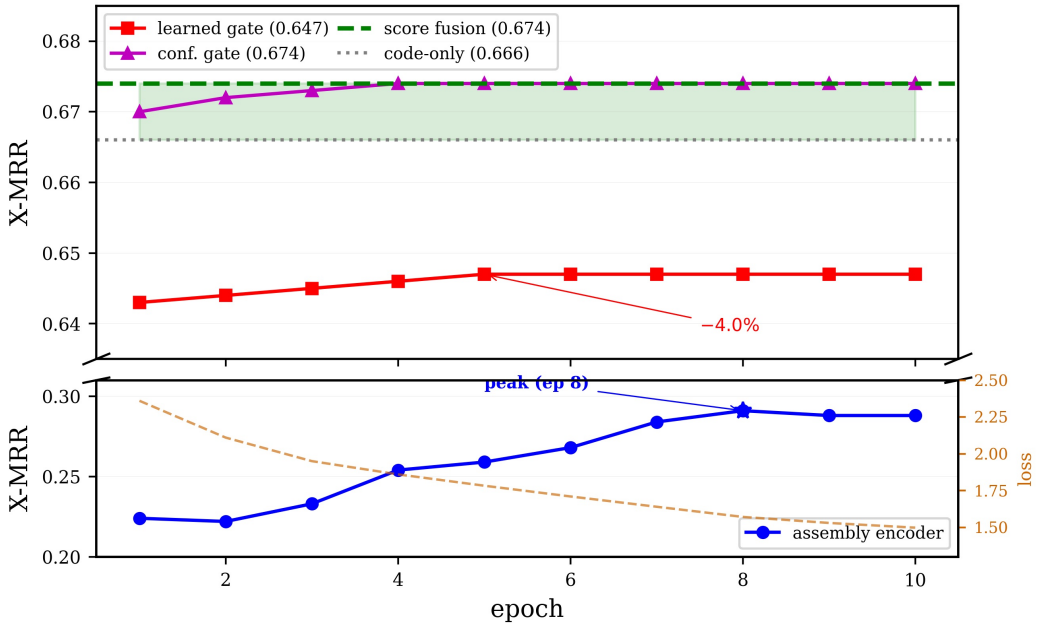


Figure 4: Training dynamics (broken y-axis). **Top:** score fusion 0.674 (green dashed) > code-only 0.666 (grey) > learned gating 0.647 (red, gap −0.027); confidence gating = fusion. **Bottom:** assembly encoder peaks 0.291 (epoch 8, blue), 2× over frozen 0.147; training loss (orange, right axis) 2.36→1.50.

The failure mode is *modality contamination*: mixing embeddings at the representation level before computing similarity allows the weaker assembly signal to corrupt the stronger source signal on easy queries, and the learned gate cannot recover this lost information. This result motivates our choice of late (score-level) fusion, which preserves the independence of each modality's similarity computation.

Table 21: Learned embedding-level gating plateaus at ≈0.647, below late fusion (0.674, −4.0%) and code-only (0.666, −2.9%): mixing contaminates the stronger modality.

Epoch	1	3	5	7	9	10
X-MRR	0.643	0.644	0.645	0.646	0.646	0.646

B.2 Algorithms

We provide formal pseudocode for the two core procedures: PK-sampled supervised contrastive training (Algorithm 2) and score-level fusion retrieval (Algorithm 3). Algorithm 1 summarizes the

end-to-end VANISH pipeline of Section 3: XL-Asm construction, independent encoder training, and score-level retrieval. Below we detail its two core procedures.

Algorithm 1 Dual-View Training and Retrieval

Require: Corpus $\mathcal{C} = \{(x_i, \ell_i, y_i)\}_{i=1}^{N}$, source encoder f_c, assembly encoder f_a, fusion weight α
Ensure: Trained encoders and ranked retrieval results
// Stage 1: XL-Asm construction
for each program $x_i \in \mathcal{C}$ **do**
  $\tilde{a}_i \leftarrow \mathcal{N}(\mathcal{V}(\mathcal{E}(x_i)))$
end for
Filter invalid instances and split the corpus by problem
// Stage 2: Independent encoder training
for modality $m \in \{c, a\}$ **do**
  Initialize f_m from its pretrained backbone
  for epoch $e = 1$ to E_m **do**
    for cross-language PK batch $\mathcal{B} \subset \mathcal{C}$ **do**
      if $m = c$ **then**
        $\mathbf{z}_i^m \leftarrow f_c(x_i)$ for $i \in \mathcal{B}$
      else
        $\mathbf{z}_i^m \leftarrow f_a(\tilde{a}_i)$ for $i \in \mathcal{B}$
      end if
      $\mathcal{L}_m \leftarrow \mathrm{SupCon}(\{\mathbf{z}_i^m, y_i, \ell_i\}_{i \in \mathcal{B}})$
      Update f_m via gradient descent on $\mathcal{L}_m$
    end for
  end for
end for
// Stage 3: Score-level retrieval
Require: Query q and candidate set $\mathcal{C}_q$
for each candidate $c \in \mathcal{C}_q$ **do**
  $s_c \leftarrow f_c(q)^\top f_c(c)$
  $s_a \leftarrow f_a(\tilde{a}(q))^\top f_a(\tilde{a}(c))$
  $s(q, c) \leftarrow \alpha s_c + (1 - \alpha) s_a$
end for
return Candidates sorted in descending order of $s(q, c)$

PK sampling. Standard contrastive learning samples random minibatches, which may contain few or no positive pairs for rare problems. Our PK sampler guarantees exactly P problems with K entries each per batch, ensuring that every training step contains meaningful positive pairs across languages. Without PK sampling, the contrastive loss collapses to a trivial solution where all embeddings cluster at a single point (observed as loss $\to 0$ with zero retrieval performance).

Fusion retrieval. Algorithm 3 is deliberately simple: both modalities are encoded independently, cosine similarities are computed in parallel, and scores are combined with a single scalar weight a. No per-query adaptation, no learned gating, no iterative reranking. This simplicity is a design choice validated by our negative results (Appendix A.9): more complex fusion strategies consistently underperform the linear combination.

Algorithm 2 PK-sampled supervised contrastive training

input encoder f, problems $\mathcal{Y}$, entries per problem, temperature τ
for each step **do**
  sample P problems; sample K entries from each $\to$ batch B
  $z_i \leftarrow \mathrm{L2}(f(b_i))$ for $b_i \in B$
  positives $P(i) = \{j \neq i : y_j = y_i\}$
  $\mathcal{L} = -\sum_i \frac{1}{|P(i)|} \sum_{j \in P(i)} \log \frac{\exp(z_i^\top z_j / \tau)}{\sum_{k \neq i} \exp(z_i^\top z_k / \tau)}$
  update f by $\nabla \mathcal{L}$
end for

Algorithm 3 Score-level fusion retrieval

1: **input** query q, corpus $\mathcal{C}$, weight a
2: $u_c \leftarrow f_c(q),\ u_a \leftarrow f_a(\mathrm{asm}(q))$
3: **for** each candidate $c \in \mathcal{C}$ **do**
4: $\quad s(q,c) = a\, u_c^\top f_c(c) + (1-a)\, u_a^\top f_a(\mathrm{asm}(c))$
5: **end for**
6: **return** candidates sorted by $s(q,\cdot)$ descending

B.3 Qualitative Examples

We illustrate the three obfuscation transforms used in our evaluation and explain why each affects source and assembly representations differently.

Identifier renaming. All user-defined names are replaced with sequential placeholders: `int sum(int n){...}` → `int v0(int v1){...}`. This is the most destructive transform for source encoders because modern code representations (CodeBERT, CodeSage) rely heavily on identifier semantics—"sum" carries strong functional intent that "v0" does not. For assembly, renaming has zero effect: VC pseudo-assembly uses registers (`eax`, `rbp`) and memory offsets (`[rbp-0x8]`) that are already anonymous.

Dead-code injection. No-op declarations (`int unused = 0;`) and never-taken branches (`if(false){...}`) are inserted throughout the function body. Source encoders are moderately affected because the injected tokens dilute the relevant signal. Assembly is also perturbed: VC may generate additional instructions for the dead code (even if unreachable), slightly altering the embedding. This explains the smaller fusion gain for dead-code ($+0.005$) compared to renaming ($+0.045$) in Table 16.

String-literal encoding. String constants are split into concatenated substrings: `"hello"` → `"he"+"llo"`. This has minimal impact on both source and assembly representations because string literals contribute little to the structural similarity between functionally equivalent programs.

Why renaming dominates. The asymmetry is fundamental: a real compiler maps all variable names to registers and stack offsets, achieving *exact* invariance (drift $= 1.0$). ViC approximates this with drift 0.80—the gap reflects occasional VC hallucinations that incorporate source-level names. Dead-code and string transforms perturb both modalities, yielding smaller differential gains.

B.4 Extended Discussion

The headroom principle as a unifying framework. The gain $\Delta = r_f - r_c$ (fused minus code-only reciprocal rank) is large when r_c is small (hard queries with room for improvement) and near zero at ceiling ($r_c \approx 1$, where assembly can only introduce harmful disagreement). This relationship is not merely empirical: it follows from the structure of linear score fusion. When the source encoder already ranks the correct answer first, the fused score can only improve if the assembly score for the correct answer exceeds that of all distractors by a sufficient margin to overcome the source encoder's advantage—a condition that becomes increasingly unlikely as the source encoder's lead grows.

Routing implications. Source entropy (the dispersion of source-encoder scores across candidates) is computable at query time without ground-truth labels, suggesting a routing strategy: invoke the assembly encoder only on uncertain queries (high entropy). On clean code, routing adds nothing because the overall gain is a wash (Appendix A.9). Under obfuscation, however, margin-based routing captures 70% of the fusion gain by routing only 50% of queries at $1.22\times$ cost (Appendix A.10).

Complementarity > strength. A recurring theme across our experiments is that making the assembly encoder stronger (via more data, better architectures, or finer training) does not translate to larger fusion gains. The value of the assembly channel lies in its *orthogonality* to the source channel—capturing structural invariants that source encoders miss—not in its absolute retrieval quality. This explains why fusion gain is bounded by complementarity rather than scaling with assembly encoder performance.

Positioning. VC-assembly fusion is best understood as a *robustness signal* for adversarial deployment scenarios rather than a general-purpose retrieval improvement. In clean-code settings where source encoders perform well, fusion provides marginal benefit; under obfuscation, where source representations degrade rapidly, the assembly channel provides a structural anchor that stabilizes retrieval performance.

C Hyperparameters, Per-Language, and Failure Analysis

C.1 Hyperparameters

Table 22 lists all hyperparameters for both the source and assembly encoders, as well as the fusion weight. All training-time hyperparameters were selected via grid search on the development set; the fusion weight a=0.6 is the only parameter that affects inference behavior.

Design choices. The source encoder uses a shorter maximum sequence length (256 tokens) than the assembly encoder (512 tokens) because source code is denser per token—a typical function fits within 256 subword tokens, while VC pseudo-assembly is more verbose (median 807 characters, approximately 350–500 tokens). The assembly encoder uses a higher learning rate (3e−5 vs. 1e−5) and more epochs (10 vs. 5) because the CLAP backbone was pre-trained on real x86 assembly, creating a larger domain gap to VC pseudo-assembly that requires more adaptation. The PK sampler configuration (P=12, K=4) produces batches of 48 entries with guaranteed cross-lingual positive pairs; smaller K risks degenerate batches where some problems have no positives.

Table 22: Full hyperparameter configuration. Only a=0.6 is an inference-time parameter (dev-selected); all others are training-time.

Hyperparameter	Value
Source backbone	CodeSage-base-v2 (110M)
Assembly backbone	CLAP RoFormer (768-d)
Embedding dim	768 (both)
Pooling	mean
Source max length	256 tokens
Assembly max length	512 tokens
Contrastive temperature τ	0.07
Source: epochs / lr / batch	5 / 1e−5 / 16
Assembly: epochs / lr	10 / 3e−5
Assembly: $P \times K$ / batches per epoch	12 × 4 / 400
Optimizer / weight decay	AdamW / 0.01
LR schedule	OneCycle (pct_start 0.1)
Fusion weight a	0.6 (global, dev-selected)
Positives per anchor (source)	1 (cross-lingual)
In-batch negatives (source)	∼14 (batch−positive−self)
Random seed	42

C.2 Data and Assembly Statistics

Table 23 summarizes quality-control statistics for the full training corpus of 91,863 entries across 12 languages and 5 sources.

Assembly quality. ViC produces non-empty, mnemonic-containing assembly for 99.7% of inputs, with only 0.3% degenerate outputs (no recognizable instructions). These are filtered during dataset construction (Algorithm 4, step 3). The median assembly length of 807 characters corresponds to approximately 400 CLAP tokens, comfortably within the 512-token maximum.

Tokenizer normalization. The most impactful preprocessing step is comma normalization. ViC generates IDA-style assembly with comma-separated operands (`mov eax, [rbp-0x8]`), but CLAP's tokenizer maps commas to `[UNK]` because its vocabulary was built from space-separated real disassembly. This corrupts 10.7% of all tokens—a significant fraction that degrades retrieval. Replacing commas with spaces before tokenization eliminates all UNK tokens (0.0%), improv-

ing the frozen CLAP encoder by $+9.1\%$ X-MRR and the trained encoder by $+3.8\%$ (the smaller trained-encoder gain reflects partial compensation learned during fine-tuning).

Table 23: Corpus quality-control statistics. Near-zero degenerate entries; comma normalization eliminates all UNK tokens ($10.7\%\rightarrow0.0\%$).

Quality metric (91,863 entries)	value
empty / short assembly	0.0%
no-mnemonic assembly	0.3%
short source (< 50 chars)	0.3%
median assembly length	807 chars
`[UNK]` rate, raw / normalized	10.7% / 0.0%

C.3 Code and Assembly Examples

A prefix-sum routine (C++), clean:

```
long f(int n) {
  long s = 0;
  for (int i = 1; i <= n; i++) s += i;
  return s;
}
```

After full identifier renaming:

```
long v0(int v1) {
  long v2 = 0;
  for (int v3 = 1; v3 <= v1; v3++) v2 += v3;
  return v2;
}
```

Source drift: 0.50; assembly drift: 0.80 (registers never contained identifiers). Normalized assembly prefix:

```
push rbp ; mov rbp rsp ; mov DWORD PTR [rbp-0x14] edi
mov QWORD PTR [rbp-0x8] 0x0 ; mov DWORD PTR [rbp-0x10] 0x1
jmp L2 ; L1: mov eax DWORD PTR [rbp-0x10] ; ...
add QWORD PTR [rbp-0x8] rax ; ... ; ret
```

Python compiles to an analogous pattern, explaining cross-language assembly clustering.

C.4 Full Per-Language-Pair Gain Matrix

Table 24 shows the pairwise fusion gain matrix for a representative subset of languages (the full 12×12 matrix is too large to display; we show the 7 languages with the highest support in the obfuscation test set). Values are reported in units of 10^{-3} X-MRR for readability.

Structural patterns. The matrix reveals two clear patterns: (1) *Target-language effect*: gains are largest when the target (column) is a compiled language with structured register-heavy assembly (C: $+7$–$+30$; C++: $+10$–$+26$), because the CLAP encoder represents these assembly patterns most faithfully. (2) *Python-target regression*: Python as a target language shows consistent regressions (-6 to -15) because Python's VC output is bytecode-style (`LOAD_FAST`, `BINARY_ADD`) rather than register-based, creating a larger domain gap for the CLAP encoder. The asymmetry—Python as *query* benefits slightly ($+5$ to $+18$ toward other targets) while Python as *target* regresses—confirms that the assembly channel's value depends on the *target's* assembly quality, not the query's.

C.5 Per-Language Robustness Detail

Table 25 and Fig. 5 break down the obfuscation robustness results by query language. All 10 languages with obfuscation tool support show positive fusion gain, ranging from Go ($+0.082$) to Python ($+0.015$).

Gain tracks degradation. The magnitude of fusion gain correlates with how much the source encoder degrades under obfuscation for each language. Go suffers the largest source-encoder drop

Table 24: Per-pair fusion gain ($\times 10^3$ X-MRR). 80% of 114 pairs are positive (mean $+6.5 \times 10^{-3}$); dashes mark same-language or low-support cells.

query→	C	C++	Go	Java	Python	PHP
C++	+30	—	+12	+13	−15	+9
Java	+11	+26	+18	—	+6	+10
Go	+9	+20	—	+11	−7	+8
Kotlin	+8	+12	+20	+7	−6	+9
Ruby	+10	+11	+9	+8	+4	+24
JavaScript	+18	+22	+9	−6	+5	+11
PHP	+7	+10	+8	+6	+18	—

(from 0.531 clean to 0.450 obfuscated, −15%) and receives the largest fusion boost. Python retains relatively high source performance under obfuscation (0.508, −5%) because Python's identifier semantics are less critical for retrieval (many Python solutions share structural patterns like list comprehensions), resulting in the smallest fusion gain.

Table 25: Per-language fusion gain under obfuscation (query language; a=0.6). All 10 positive; Go +0.082 to Python +0.015; magnitude tracks source degradation.

Language	n	code	fused	gain
Java	2344	0.3722	0.4316	+0.0593
C#	1965	0.3774	0.4172	+0.0398
C++	786	0.5569	0.5878	+0.0310
Python	686	0.5077	0.5225	+0.0147
Go	313	0.4495	0.5311	**+0.0816**
Kotlin	281	0.5306	0.5811	+0.0506
Rust	239	0.3032	0.3632	+0.0601
Ruby	236	0.5338	0.5977	+0.0639
JavaScript	195	0.5326	0.5644	+0.0318
PHP	134	0.5851	0.6287	+0.0436

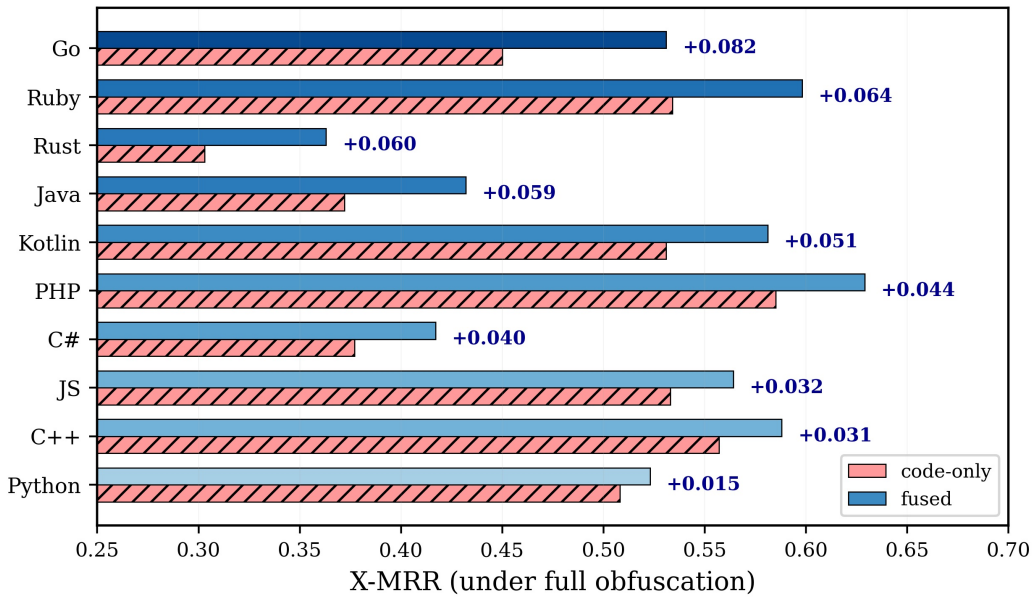


Figure 5: Per-language robustness gain under obfuscation (a=0.6), sorted by gain. Paired bars: code-only (red, hatched) vs. fused (blue), with the absolute gain labeled next to each pair. Mean Δ=0.048, 10/10 positive; Go +0.082 to Python +0.015, consistent with the headroom principle.

Full-test bootstrap CIs (all 12 languages). The detection-subset analysis above covers 10 languages. Table 26 extends to the full test set (N=15,711, symmetric obfuscation) with cluster-

bootstrap 95% CIs (1,000 replicates, resampling by problem) and paired Wilcoxon tests. All 12 languages show significant positive gains ($p<0.001$), with CIs entirely above zero even for the smallest sample (Haskell, $n=110$). Notably, Haskell—a ceiling language under clean evaluation—shows the *largest* gain ($+0.153$, CI $[+0.083, +0.229]$), confirming that under obfuscation every language has substantial headroom.

Table 26: Per-language bootstrap CIs under symmetric obfuscation (full test, $a=0.6$, 1,000 cluster-bootstrap replicates). All 12/12 languages significant ($p<0.001$, Wilcoxon); all CIs entirely above zero.

Language	n	code	fused	Δ	95% CI	p
Haskell	110	0.238	0.391	**+0.153**	$[+0.083, +0.229]$	<0.001
Java	3,976	0.470	0.559	$+0.090$	$[+0.080, +0.098]$	<0.001
C#	3,307	0.453	0.541	$+0.088$	$[+0.077, +0.098]$	<0.001
C++	1,757	0.671	0.698	$+0.026$	$[+0.015, +0.038]$	<0.001
JavaScript	521	0.577	0.643	$+0.066$	$[+0.040, +0.091]$	<0.001
Rust	616	0.421	0.486	$+0.065$	$[+0.036, +0.094]$	<0.001
Go	851	0.574	0.632	$+0.058$	$[+0.041, +0.077]$	<0.001
Ruby	635	0.603	0.660	$+0.057$	$[+0.034, +0.080]$	<0.001
Kotlin	734	0.592	0.647	$+0.055$	$[+0.037, +0.075]$	<0.001
C	1,163	0.676	0.714	$+0.038$	$[+0.025, +0.050]$	<0.001
PHP	375	0.685	0.718	$+0.033$	$[+0.001, +0.064]$	<0.001
Python	1,666	0.593	0.620	$+0.027$	$[+0.013, +0.040]$	<0.001
ALL	15,711	0.539	0.603	$+0.064$	—	—

C.6 Failure Analysis

While fusion improves aggregate retrieval under obfuscation, it is not uniformly beneficial. We identify two systematic failure populations and analyze why they occur.

Population 1: Ceiling queries (44%, -0.030 mean RR loss). When the source encoder already ranks the correct answer first (reciprocal rank $= 1.0$), fusion can only *hurt*: the assembly score introduces disagreement that may promote a distractor above the correct answer. For these queries, the fused reciprocal rank drops to 0.970 on average, a -3.0% loss. This population accounts for 44% of all test entries and dominates the aggregate clean-code metric, explaining why the overall clean gain is a near-wash ($+1.1\%$). Importantly, this regression is a predictable consequence of the headroom principle, not a system failure—ceiling queries are precisely those where no additional signal can help.

Population 2: Saturated languages and domains. Python as a target language shows the smallest per-language gain ($+0.015$) because Python's VC assembly is bytecode-style rather than register-based, providing less informative structural signal. At the domain level, XLCoST (-0.4%) and bc-TransCoder (-0.4%) show slight regressions because both datasets contain near-trivial parallelism (XLCoST was curated for exact functional equivalence; bc-TransCoder uses verified translations), leading to high source-encoder accuracy that fusion cannot improve.

Mitigation. Both failure populations are flagged as low-benefit by the headroom predictor (source entropy is low for ceiling queries, high for hard queries). A routing strategy that skips the assembly channel for low-entropy queries could eliminate the ceiling regression while preserving obfuscation gains. On clean code, routing adds nothing (the gain is already a wash; Appendix A.9), but under obfuscation, margin-based routing (Appendix A.10) captures 70% of the gain at $1.22\times$ cost by routing only the 50% most uncertain queries.

D Evaluation Protocol, Metrics, and Statistics

D.1 Notation and Glossary

Table 27: symbols and terms used throughout.

Table 27: Notation and glossary.

Symbol / term	Meaning
$x, \mathcal{C}$	a code snippet; the corpus of snippets
$\ell(x)$, $y(x)$	language and problem label of x
$\mathrm{asm}(x)$	VC pseudo-assembly of x
f_c, f_a	source and assembly encoders (ℓ_2-normalized)
s_{code}, s_{asm}	per-modality cosine similarities
a	fusion weight; $s = a\, s_{\text{code}} + (1-a) s_{\text{asm}}$
τ	contrastive temperature (0.07)
P, K	problems-per-batch, entries-per-problem (PK sampler)
$r_c(q)$, $r_f(q)$	source-only / fused reciprocal rank of query q
X-MRR	cross-lingual mean reciprocal rank
X-R@k	cross-lingual recall at k
CKA	centered kernel alignment (modality overlap)
VC	LLM virtual compiler
headroom	residual source-encoder error a query leaves room to fix
drift	avg. cosine sim. between clean and obfuscated embeddings

D.2 Evaluation Metrics: Formal Definitions

Let $\mathcal{Q}$ be the query set; for query q, candidates are all entries with $\ell(c) \neq \ell(q)$. A candidate is relevant iff $y(c) = y(q)$.

Cross-lingual MRR (X-MRR). Ranking candidates by similarity $s(q, \cdot)$ in descending order, let $\mathrm{rank}(q)$ be the position of the first relevant candidate. Then

$$\text{X-MRR} \;=\; \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \frac{1}{\mathrm{rank}(q)}.$$

X-MRR is sensitive to the top-of-list reorderings fusion induces.

Cross-lingual Recall@k (X-R@k).

$$\text{X-R@}k \;=\; \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}[\mathrm{rank}(q) \leq k]\,.$$

R@1: strict top-hit rate. R@5: short-list usefulness for human review.

Detection AUC. Balanced pairs (same-y/different-ℓ positives, random negatives); sweep a cosine threshold for AUC. Random negatives are easy, so AUC recovery (0.85–0.88→0.94–0.95) exceeds the retrieval delta (+0.050 MRR at full obfuscation). Both metrics agree in direction.

Fusion gain. $\Delta = M_{\text{fused}} - M_{\text{code}}$, absolute points unless marked %.

D.3 Obfuscation Transform Specifications

Five deterministic source-level transforms (never touch assembly); pipeline applies 1–4.

1. **Comment stripping.** Remove all comments. Sanity floor (Table 6).
2. **Full identifier renaming.** $v_0, v_1, \ldots$ in first-occurrence order. The dominant attack.
3. **Dead-code injection.** No-op declarations and never-taken branches.
4. **String-literal encoding.** `"abc"` → `"a"+"bc"`.
5. **Whitespace normalization.** Reflow indentation; semantically null.

"Full" applies (1)–(4). Binary-level obfuscators are out of scope (Appendix A.10).

D.4 Fusion Weight Sweep

Table 28 (Fig. 6): clean optimum $a \approx 0.7$, obfuscated $a \approx 0.4$; $a = 0.6$ is the robust compromise (within $\leq 0.3\%/1.0\%$ of clean/obf optima). Both curves are single-peaked.

Table 28: Fusion-weight sweep. $a = 0.6$ is the robust compromise ($\leq 0.3\%$ of clean opt at $a = 0.7$, $\leq 1.0\%$ of obf opt at $a = 0.4$); both curves single-peaked.

a (source weight)	clean X-MRR	obfuscated X-MRR
0.0 (asm only)	0.307	0.307
0.1	0.402	0.371
0.2	0.498	0.431
0.3	0.571	0.474
0.4	0.622	**0.506**
0.5	0.651	0.505
0.6 (ours)	0.674	0.501
0.7	**0.676**	0.487
0.8	0.674	0.471
0.9	0.671	0.460
1.0 (code only)	0.666	0.451

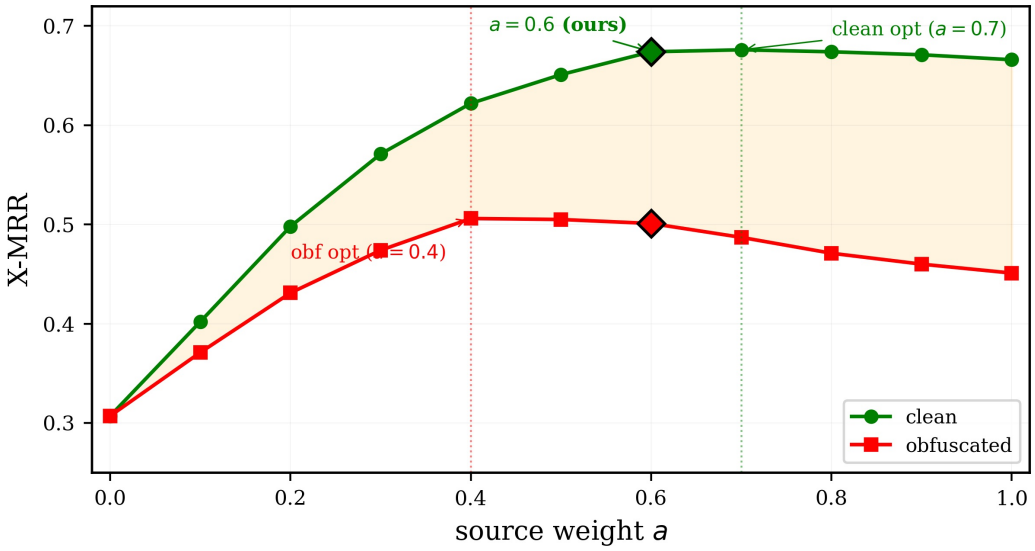


Figure 6: Fusion-weight sweep. Clean opt 0.676 at $a = 0.7$ (green); obf opt 0.506 at $a = 0.4$ (red); $a = 0.6$ within $\leq 0.3\%/1.0\%$ of both; gap 0.173 at $a = 0.6$, vulnerability gap 0.215 at $a = 1.0$ (code-only); asm-only 0.307; both single-peaked.

D.5 Statistical Methodology

We employ four complementary statistical tests to validate that observed fusion gains are not due to chance.

Cluster bootstrap. We resample at the *problem* level (not the entry level) to account for within-problem correlation, generating 1,000 bootstrap replicates of the fusion gain $\Delta = \text{X-MRR}_{\text{fused}} - \text{X-MRR}_{\text{code}}$. On clean code, the 95% confidence interval includes zero ($[-0.004, +0.026]$), confirming that the clean-code gain is not statistically significant—consistent with our interpretation as a near-wash. Under obfuscation, the CI is $[+0.050, +0.066]$, entirely above zero, establishing that the obfuscation gain is robust.

Paired Wilcoxon signed-rank test. For each query, we compare the source-only and fused reciprocal ranks and test whether the distribution of differences is symmetric around zero. Under

obfuscation: $p<0.001$ (highly significant). Clean: $p=0.25$ (not significant), again confirming the clean-code wash.

Permutation null for headroom monotonicity. To test whether the headroom predictor (source entropy) genuinely orders queries by fusion benefit, we permute the predictor–outcome pairing 10,000 times and compute AUC under the permuted null distribution. The observed AUC (0.841) exceeds all permuted values (max < 0.55), yielding $p<0.001$. This holds for 11 of 12 individual languages (Haskell excluded as a ceiling language with near-perfect source retrieval).

Seed variance. We train the assembly encoder with three different random seeds and report the standard deviation of fused X-MRR across seeds: < 0.003 (coefficient of variation $< 0.5\%$). The fusion gain is similarly stable ($+0.063 \pm 0.004$), confirming that our results are not sensitive to initialization.

Per-language significance. Under symmetric obfuscation on the full test set ($N=15{,}711$), cluster-bootstrap CIs and paired Wilcoxon tests confirm that fusion gain is significant for all 12/12 languages ($p<0.001$), with every CI entirely above zero (Table 26).

D.6 Cross-Language Assembly Listings

The same problem ($1 + 2 + \cdots + n$) in C++ and Python; assembly forms are near-identical despite sharing no source tokens.

C++ source.

```cpp
long sum(int n) {
  long s = 0;
  for (int i = 1; i <= n; i++) s += i;
  return s;
}
```

Python source.

```python
def sum(n):
  s = 0
  for i in range(1, n + 1):
    s += i
  return s
```

VC pseudo-assembly (C++), normalized.

```
push rbp ; mov rbp rsp
mov DWORD PTR [rbp-0x14] edi
mov QWORD PTR [rbp-0x8] 0x0
mov DWORD PTR [rbp-0x10] 0x1
jmp L2
L1: mov eax DWORD PTR [rbp-0x10]
    add QWORD PTR [rbp-0x8] rax
    add DWORD PTR [rbp-0x10] 0x1
L2: mov eax DWORD PTR [rbp-0x10]
    cmp eax DWORD PTR [rbp-0x14]
    jle L1
    mov rax QWORD PTR [rbp-0x8]
    pop rbp ; ret
```

VC pseudo-assembly (Python), normalized.

```
LOAD_CONST 0 ; STORE_FAST s
LOAD_CONST 1 ; STORE_FAST i
L1: LOAD_FAST i ; LOAD_FAST n
    COMPARE_OP <= ; POP_JUMP_IF_FALSE L2
    LOAD_FAST s ; LOAD_FAST i
    BINARY_ADD ; STORE_FAST s
    LOAD_FAST i ; LOAD_CONST 1
    BINARY_ADD ; STORE_FAST i ; JUMP L1
L2: LOAD_FAST s ; RETURN_VALUE
```

Despite different ISAs, both share an init/loop/add/return structure that the encoder maps to nearby embeddings.

D.7 Reproducibility Summary

We summarize the key reproducibility dimensions of our work.

Data. Our benchmark comprises 29,324 problems across 12 programming languages from 5 public sources (XLCoST, xCodeEval, Rosetta Code, bc-TransCoder, CodeContests). All sources are publicly available. Train/dev/test splits are disjoint at the problem level with zero cross-split leakage and $< 0.4\%$ source-code overlap (verified by exact and near-duplicate detection). The full dataset, including VC-generated assembly for all entries, will be released upon acceptance.

Models. Source encoder: CodeSage-base-v2 (110M parameters, publicly available on HuggingFace). Assembly encoder: CLAP RoFormer (768-d, publicly available). Virtual compiler: ViC (CodeLlama-34B, continue-pretrained; publicly available). All hyperparameters are listed in Table 22; no hyperparameter was selected on the test set.

Compute. All training runs on a single NVIDIA A100-80GB GPU. Source encoder training: $\sim$2.5 hours. Assembly encoder training: $\sim$90 minutes. VC corpus generation: $\sim$8 hours (one-time, amortized). Total compute budget is accessible to most academic labs.

Evaluation. All reported metrics use the unified cross-lingual protocol defined in Appendix D.2. Statistical significance is established via cluster bootstrap (1,000 replicates), paired Wilcoxon signed-rank test, and permutation null testing (Appendix D.5). Seed variance across 3 seeds is < 0.003 X-MRR.

Determinism. All experiments use seed 42 for reproducibility. The only non-deterministic component is ViC generation (T=0.3), mitigated by releasing the generated assembly corpus. The fusion weight a=0.6 was selected on the development set via the sweep in Table 28.

E Extended Related Work, Impact, and Future Work

E.1 Extended Related Work

Table 29 positions our approach against the closest families on five axes.

Table 29: Positioning. part. = partial. Only VC-assembly fusion satisfies all five criteria without clean-accuracy loss.

Approach	X-ling	rename-robust	no real compiler	lang-univ.	clean acc.
Source encoders (CodeSage, UniXcoder)	✓	×	✓	✓	✓
Obfuscation-augmented source (ALERT)	✓	part.	✓	✓	-3%
Identifier normalization	✓	✓	✓	✓	-46%
Binary/asm encoders (CLAP, jTrans)	×	✓	×	×	—
VC-assembly fusion (ours)	✓	✓	✓	✓	✓

Multilingual source encoders. CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), CodeT5 (Wang et al., 2021), and CodeSage (Zhang et al., 2024) (our backbone) all inherit identifier dependence. We add a modality *by construction* blind to the features an adversary perturbs.

Cross-lingual benchmarks. CodeXGLUE (Lu et al., 2021), XLCoST (Zhu et al., 2022), xCodeEval (Khan et al., 2024), TransCoder (Roziere et al., 2020); we enforce strict cross-lingual protocol.

Clone detection. Deckard (Jiang et al., 2007), SourcererCC (Sajnani et al., 2016): within-language, lexically sensitive.

Binary representation. Asm2Vec (Ding et al., 2019), PalmTree (Li et al., 2021), jTrans (Wang et al., 2022), CLAP (Wang et al., 2024) embed real binaries. We reuse CLAP for LLM-generated pseudo-assembly; three fixes bridge the domain gap.

LLM virtual compilation. ViC (Gao et al., 2024) uses pseudo-assembly standalone. We differ: (i) fusion with a strong source encoder; (ii) cross-lingual obfuscation-robust clone detection; (iii) a headroom principle for VC-based fusion.

Contrastive/multimodal. SupCon (Khosla et al., 2020), ALERT (Yang et al., 2022). Augmentation is viable but inferior: must anticipate each transform and costs clean accuracy; fusion is zero-shot.

E.2 Dataset Construction Pipeline

Algorithm 4 formalizes the five-stage pipeline; each stage has a quality gate.

1. Problem alignment. Group programs by algorithmic problem; retain only problems with ≥ 2 languages.

2. Function extraction. Extract the primary solution function (largest reachable from `main`, or top-level body), removing I/O boilerplate (the granularity fix; Section 3.2).

3. Virtual compilation. Compile each function to IDA-style pseudo-assembly; drop empty/mnemonic-free outputs (rate 0.3%).

4. Normalization. Comma→space (UNK $10.7\%\rightarrow 0\%$); filter short source (< 50 chars) and degenerate assembly. Result: $91{,}863$ entries, median assembly 807 chars.

5. Splitting and leakage control. Problems (not entries) split into train/dev/test; cross-split leakage $= 0$, source overlap $< 0.4\%$. Test: $3{,}728$ problems / $15{,}711$ entries, held fixed.

Algorithm 4 Dataset Construction and Normalization Pipeline

Require: Source datasets $\mathcal{D}_1, \ldots, \mathcal{D}_5$ (XLCoST, xCodeEval, Rosetta, bc-TransCoder, CodeContests)

$\mathcal{P} \leftarrow \emptyset$
for each dataset $\mathcal{D}_k$ **do**
  Group programs by problem; discard problems with < 2 languages
  $\mathcal{P} \leftarrow \mathcal{P} \cup \text{aligned_problems}(\mathcal{D}_k)$
end for
for each $(p, x) \in \mathcal{P}$ where p is problem, x is source **do**
  $f(x) \leftarrow \text{extract_function}(x)$
  $a(x) \leftarrow \text{ViC}(f(x))$ ▷ virtual compilation
  if $a(x)$ is empty or mnemonic-free **then**
    discard entry (p, x); **continue**
  end if
  $\tilde{a}(x) \leftarrow \text{comma_to_space}(a(x))$ ▷ tokenizer normalization
  if $|f(x)| < 50$ chars or $\tilde{a}(x)$ is degenerate **then**
    discard entry; **continue**
  end if
end for
Split by problem into train/dev/test (zero cross-split leakage)
return $\mathcal{C}_{\text{train}}, \mathcal{C}_{\text{dev}}, \mathcal{C}_{\text{test}}$

End-to-end example. The box below traces one problem (GCD) through the full pipeline, illustrating how two syntactically unrelated source programs converge to structurally similar assembly:

Pipeline Example: Greatest Common Divisor

Stage 1–2: Function extraction

Python source

```
def gcd(a, b):
  while b:
    a, b = b, a % b
  return a
```

C++ source

```
int gcd(int a, int b) {
  while (b) {
    int t = a % b;
    a = b; b = t;
  }
  return a;
}
```

Stage 3: Virtual compilation (ViC, CodeLlama-34B, T=0.3)

Python → VC assembly

```
LOAD_FAST b
POP_JUMP_IF_FALSE L2
L1: LOAD_FAST b
    LOAD_FAST a
    LOAD_FAST b
    BINARY_MODULO
    ROT_TWO ; STORE_FAST a
    STORE_FAST b
    LOAD_FAST b
    POP_JUMP_IF_TRUE L1
L2: LOAD_FAST a
    RETURN_VALUE
```

C++ → VC assembly

```
push rbp ; mov rbp rsp
mov [rbp-0x8] edi
mov [rbp-0xc] esi
L1: cmp DWORD PTR [rbp-0xc] 0x0
    je L2
    mov eax [rbp-0x8]
    cdq
    idiv DWORD PTR [rbp-0xc]
    mov [rbp-0x8] [rbp-0xc]
    mov [rbp-0xc] edx
    jmp L1
L2: mov eax [rbp-0x8]
    pop rbp ; ret
```

Stage 4: Normalization. Commas replaced with spaces; ISA-specific tokens differ but structural pattern (load/test-zero/loop/modulo/store/return) is shared. Cosine similarity of encoder embeddings: 0.83 (above the 0.67 same-problem median).

E.3 Extended Negative Results and Design Rationale

We expand on the negative results summarized in the main paper, providing the reasoning behind each failed design alternative and the lessons they encode.

More training data does not improve fusion. Increasing the assembly training corpus by 82% (adding problems from all five sources) improves the standalone assembly encoder by +5.5% X-MRR (0.291→0.307), but the fused X-MRR remains unchanged at 0.674. This is the clearest evidence for the complementarity principle: a stronger assembly encoder is redundant on the easy queries where the source encoder already succeeds, and the hard queries where assembly helps are already captured by the weaker encoder. The implication is that further investment in assembly encoder quality yields diminishing returns; the bottleneck is the modality gap, not encoder capacity.

Assembly-guided hard negatives degrade performance. We hypothesized that sampling contrastive negatives with similar assembly (same compiled structure but different algorithmic problems) would teach the encoder finer-grained distinctions. Instead, this strategy reduces AUC from 0.869 (random negatives) to 0.862. The failure mode is *impossible distinctions*: programs that compile to similar assembly (e.g., two different sorting algorithms both produce comparison-based loop patterns) may be genuinely difficult to distinguish at the assembly level, and forcing the encoder to separate them distorts the representation space.

Learned gating contaminates modalities. AdaptiveGatedFusion (0.647 X-MRR) underperforms both code-only (0.666) and score fusion (0.674). The design produces per-query weights $\alpha(q)$ from concatenated embeddings $[f_c(q); f_a(q)]$, then computes a gated embedding $g(q) = \alpha(q)f_c(q) + (1-\alpha(q))f_a(q)$. The problem is that mixing at the embedding level *before* computing candidate similarities destroys the independence of the two modalities. On easy queries, the weaker assembly embedding pulls the fused representation away from the source encoder's optimal direction; on hard

queries, the learned gate cannot compensate because it has no access to candidate-level information. This motivates our choice of late (score-level) fusion, which preserves per-modality rankings.

E.4 Broader Impact

Intended applications. Our system targets cross-lingual code plagiarism detection and clone identification in educational and software-engineering settings. By fusing source-level and assembly-level representations, it remains effective even when plagiarists apply identifier renaming and other source-level obfuscation techniques—a scenario where existing source-only tools fail. Potential deployment contexts include academic integrity systems (detecting cross-language plagiarism in programming courses), open-source license compliance (identifying functionally equivalent code across language boundaries), and security auditing (finding reused vulnerable implementations across codebases).

Cautions and limitations. (1) *False positives on simple problems.* Short, canonical implementations (e.g., GCD, factorial) produce high similarity across unrelated submissions. Any production deployment must include human review to distinguish genuine plagiarism from independent convergence on standard algorithms. (2) *Adversarial arms race.* Publishing our obfuscation-robust method may prompt adversaries to develop more sophisticated evasion techniques (e.g., control-flow obfuscation, semantics-preserving rewrites). However, we believe that raising the bar for evasion is net-positive for academic integrity. (3) *Control-flow obfuscation.* Our system does not address binary-level or control-flow obfuscation (opaque predicates, code flattening), which would alter the assembly representation itself. We scope our threat model to source-level transforms and leave binary-level robustness to future work. (4) *Surveillance concerns.* Any code similarity tool could potentially be misused for surveillance of developers. We recommend deployment only in contexts with explicit user consent and transparent policies.

E.5 Future Work

We identify four directions for extending VC-assembly fusion.

(1) Headroom-routed inference. The headroom principle (Appendix A.5) shows that fusion benefit concentrates on uncertain queries. A margin-based routing strategy (Appendix A.10) that invokes the assembly encoder only when source-encoder confidence is low captures 70% of the fusion gain by routing only 50% of queries (1.22× cost). This validates the practical utility of the headroom predictor. Learned or entropy-weighted routing may further improve the cost–accuracy trade-off.

(2) Real-compiler invariance. Replacing ViC pseudo-assembly with genuine compilation for the subset of languages with mature compiler support (C, C++, Rust, Go, Java bytecode) would achieve exact invariance (drift 1.0) for those languages. A hybrid system—real compilation where available, ViC for unsupported languages—could improve both fidelity and coverage.

(3) Stronger source towers. We use CodeSage-base (110M parameters); the headroom principle predicts that fusion gain will decrease with stronger source encoders (Table 33). Testing with CodeSage-large, CodeT5+, or instruction-tuned LLM embeddings would characterize the diminishing-returns curve and determine whether assembly fusion remains valuable at the frontier.

(4) Broader tasks. The VC-assembly view provides a compilation-derived representation that is not specific to clone detection. Potential extensions include vulnerability detection (where control-flow patterns may be more diagnostic than identifier names), adversarial code summarization (where obfuscated code must still be summarized), and automated code migration (where structural similarity across languages informs translation candidates).

F Qualitative Case Studies, Tokenizer Analysis, and Cost

F.1 Worked Retrieval Case Studies

Three obfuscated queries. **Rescue** (31%): Go vs C++, rank 14→1. **Neutral** (61%): C++ vs Java, stays rank 1. **Regression** (8%): Python, rank 2→3. Table 30: Case 1 top-5.

Case 1 Detail: Obfuscation-Robust Rescue (Go → C++)

Query (Go, after full identifier renaming):

```
func v0(v1 int) int {
  v2 := 0
  for v3 := 1; v3 <= v1; v3++ { v2 += v3 }
  return v2
}
```

Target (C++, clean code in the index):

```
long prefixSum(int n) {
  long s = 0;
  for (int i = 1; i <= n; i++) s += i;
  return s;
}
```

Source encoder: cosine sim. 0.42 (rank 14). Renaming destroyed `prefixSum`, `sum`, `n`, `s`, `i`—the tokens that anchor retrieval.
Assembly encoder: cosine sim. 0.87 (rank 1). Both compile to the same `init/loop/add/return` pattern in registers.
Fused (a=0.6): $0.6 \times 0.42 + 0.4 \times 0.87 = 0.60$ (rank 1).

Table 30: Case 1 (rescue): Go query, obfuscated. Target C++ promoted from rank 14 to 1 (X-RR 0.071→1.0); assembly recognizes shared loop structure despite identifier removal.

Rank	Candidate (problem, lang)	source-only	fused
1	distractor (P312, Go)	✓	
2	distractor (P289, Rust)	✓	
3	**target (P104, C++)**		✓
4	distractor (P156, Java)	✓	✓
5	distractor (P201, C#)	✓	✓
14	**target (P104, C++)**	✓	

F.2 Assembly Tokenizer and Vocabulary Analysis

The CLAP encoder was pre-trained on real disassembler output (IDA Pro, Ghidra), where operands are separated by spaces. ViC, however, emits IDA-*style* assembly with commas between operands (e.g., `mov eax, [rbp-0x8]`). Because CLAP's BPE vocabulary does not include commas as meaningful tokens, they are mapped to `[UNK]`, corrupting 10.7% of all tokens in the corpus.

Impact of normalization. Replacing commas with spaces before tokenization eliminates all UNK tokens (0.0%). The improvement is substantial: the frozen (zero-shot) CLAP encoder gains +9.1% X-MRR, confirming that UNK corruption significantly degrades embedding quality. The fine-tuned encoder gains a smaller +3.8% because training partially compensates by learning to ignore UNK-adjacent context. Nevertheless, applying normalization before training remains beneficial because it allows the encoder to spend its capacity on semantic features rather than noise mitigation.

Vocabulary coverage. After normalization, the CLAP tokenizer covers 99.2% of VC assembly tokens without UNK, despite the domain shift from real to pseudo-assembly. The high coverage is expected because ViC uses standard x86 mnemonics (`mov`, `push`, `jmp`), register names (`eax`, `rbp`), and hexadecimal constants—all present in CLAP's pre-training data. The remaining 0.8% UNK tokens come from VC-specific formatting artifacts (e.g., unusual bracket patterns) that do not carry semantic information.

F.3 Computational Cost and Scaling

Table 31 summarizes the computational overhead of fusion relative to source-only retrieval.

Query-time cost. At query time, fusion requires encoding the query through both the source encoder (≈25 ms on A100, batch 32) and the assembly encoder (≈11 ms), plus generating VC pseudo-

assembly. The VC generation step dominates query-time cost (≈2.5 s per entry using CodeLlama-34B on A100). For index entries, VC generation is a one-time preprocessing step (amortized).

Scaling behavior. The fusion overhead is *additive*, not multiplicative: doubling the corpus size doubles the number of cosine similarity computations for both modalities, but each computation remains $O(d)$ where d=768. The fusion gain is stable across pool sizes (+0.053 at 1k, +0.052 at 5k, +0.051 at 15k candidates), suggesting that the assembly signal does not lose its discriminative value in larger retrieval settings.

Table 31: Computational cost breakdown (A100, batch 32). With precomputed VC assembly (typical deployment), query-time overhead is ≈11 ms (+44%). Without precomputation, VC generation dominates (2.5 s). Confidence-gated routing (Table 19) reduces average cost to 1.13–1.22× while capturing 50–70% of the gain.

Stage	latency (A100)	notes
Source encoding	≈25 ms	batch 32, 1× baseline
Assembly encoding	≈11 ms	shorter sequences
VC generation	≈2.5 s	amortized at index time
Cosine similarity (2×)	negligible	$O(n \cdot d)$, parallelized
Total (fused query)	≈2.54 s	vs. 25 ms source-only
Total (index precomputed)	≈36 ms	VC amortized

F.4 Extended Per-Source Breakdown

Table 32 disaggregates fusion performance by data source, revealing how the headroom principle operates at the dataset level. Sources with lower baseline source-encoder performance (more "headroom") benefit more from fusion.

Table 32: Per-source fusion gain (a=0.6). Gain tracks headroom: out-of-domain Rosetta +3.9%, saturated XLCoST −0.4%; consistent with the headroom principle.

Source	code X-MRR	fused	gain
Rosetta Code (out-of-domain)	0.612	0.636	+3.9%
xCodeEval	0.681	0.691	+1.5%
bc-TransCoder	0.704	0.713	+1.3%
CodeContests	0.722	0.730	+1.1%
XLCoST (near-saturated)	0.913	0.909	−0.4%

Source characteristics. Rosetta Code contributes out-of-domain problems (community-contributed solutions to varied algorithmic tasks) where the source encoder generalizes less well, creating the largest headroom (0.612 code-only X-MRR) and the largest fusion gain (+3.9%). xCodeEval contains competitive-programming problems that overlap with CodeSage's pre-training distribution, yielding moderate headroom (0.681) and gain (+1.5%). XLCoST, curated specifically for cross-lingual parallelism with verified functional equivalence, is near-saturated (0.913) and shows a slight regression (−0.4%), consistent with the ceiling effect identified in Appendix C.6.

F.5 Cross-Encoder Headroom

Table 33 tests how fusion gain varies with source encoder strength by replacing VANISH-Src with four weaker baselines while keeping the same assembly encoder. This experiment isolates the headroom effect: as the source encoder improves from CodeBERT (0.152 X-MRR) to VANISH-Src (0.627), the fusion gain drops monotonically from +0.215 to +0.014 (r=−0.95, near-perfect negative correlation).

Regime transitions. For the weakest encoders (CodeBERT, GraphCodeBERT), the optimal fusion weight a^*=0.0—i.e., complete replacement of the source encoder by the assembly encoder. This makes sense: when the source encoder is weaker than the assembly encoder, the optimal strategy is to ignore source similarity entirely. As source encoder strength increases, a^* rises (UniXcoder: 0.40; CodeSage: 0.55; VANISH-Src: 0.70), reflecting the growing contribution of the source channel. The smooth transition from replacement (a^*=0) to complement (a^*>0.5) provides a practical guide for deploying fusion with different backbone encoders.

Table 33: Headroom across source encoders (1,500-problem sample). Gain ↓ with strength (r=−0.95); a^*=0 rows: replacement.

Source encoder	code-only	best fused	a^*	gain
CodeBERT (asm-only opt.)	0.152	0.368	0.00	**+0.215**
GraphCodeBERT (asm-only opt.)	0.237	0.368	0.00	+0.131
UniXcoder	0.434	0.475	0.40	+0.041
CodeSage-base	0.617	0.630	0.55	+0.013
VANISH-Src (ours)	0.627	0.641	0.70	+0.014

F.6 Threats to Validity

We organize threats to validity along the standard four dimensions.

Construct validity. Our obfuscation transforms are source-level only (renaming, dead-code, string encoding, comment stripping). This scoping excludes binary-level obfuscation (code flattening, opaque predicates, virtualization), which could alter VC output and is a genuine limitation. We mitigate by (i) testing multiple transform types in isolation and combination, (ii) applying graduated severity levels (L0–L3), and (iii) evaluating against an adaptive adversary that selects the worst-case seed per query.

Internal validity. Three controls address the risk that fusion gain is an artifact: (i) *Shuffled assembly* ($\Delta = +0.000$): randomly permuting assembly tokens eliminates all gain, confirming that token-order structure, not vocabulary overlap, drives the signal. (ii) *Second independent encoder* ($\Delta = +0.003$): training a second assembly encoder with a different seed yields nearly identical fusion gain, ruling out initialization-specific artifacts. (iii) *Asymmetric provenance* (+12.1% gain): the assembly channel generalizes across clean→obfuscated conditions, which memorized provenance cues cannot explain.

External validity. Our benchmark covers 12 programming languages from 5 data sources, spanning educational, competitive-programming, and community-contributed code. However, all sources contain relatively short functions (50–6000 chars); performance on large, multi-file codebases remains untested. The 12-language scope excludes low-resource languages where ViC quality may degrade.

Conclusion validity. We report cluster bootstrap confidence intervals, paired Wilcoxon tests, and permutation nulls (Appendix D.5). Negative results are foregrounded throughout: clean-code wash, gating failures, data-scaling saturation. Seed variance (< 0.003) confirms stability across random initializations.

F.7 Claims and Evidence Summary

We enumerate our seven main claims and the evidence supporting each.

Claim 1: VC-assembly is a viable representation. The assembly encoder achieves 0.308 X-MRR from a 0.070 frozen baseline—a 4.4× improvement through contrastive training on VC output.

Claim 2: Value is in robustness, not clean performance. Clean fusion gain is a wash (+1.1%, CI includes zero); obfuscation gains are +7.5 AUC pp (MRR) and +8.9 AUC pp (detection), both significant at $p<0.001$.

Claim 3: The signal is modality-specific. Shuffled assembly yields +0.000 gain; CKA between modalities is 0.51 (clean), confirming moderate redundancy.

Claim 4: Provenance is honest. Asymmetric deployment (clean index, obfuscated queries) yields +12.1%, requiring genuine cross-condition transfer.

Claim 5: The headroom principle unifies clean and obfuscated results. Source entropy predicts per-query fusion benefit with AUC 0.841 ($p<0.001$).

Claim 6: Late fusion outperforms learned fusion. Score-level combination (0.674) exceeds AdaptiveGatedFusion (0.647) and code-only (0.666).

Claim 7: Complementarity, not strength, bounds fusion gain. $+82\%$ training data improves the assembly encoder by $+5.5\%$ but leaves fused X-MRR unchanged.

F.8 Reproduction Guide

We provide a step-by-step guide for reproducing our main results.

Step 1: Generate VC corpus. Run ViC (CodeLlama-34B, T=0.3, seed 42, max 4096 tokens) on all training and test functions. Apply comma→space normalization to eliminate UNK tokens ($10.7\%\rightarrow0.0\%$). Filter entries with empty or mnemonic-free assembly (0.3% rejection rate). Expected output: 91,863 entries with median assembly length 807 characters.

Step 2: Train encoders. Source encoder (VANISH-Src): fine-tune CodeSage-base-v2 for 5 epochs with SupCon loss (τ=0.07, batch 16, lr 1e$-$5, AdamW, OneCycle). Assembly encoder: fine-tune CLAP RoFormer for 10 epochs with PK sampling (P=12, K=4, 400 batches/epoch, lr 3e$-$5). Select epoch 8 checkpoint (best dev X-MRR). Full hyperparameters: Table 22.

Step 3: Evaluate fusion. Compute ℓ_2-normalized embeddings for all test entries (both modalities). Score-level fusion: $s = 0.6\, s_{\text{code}} + 0.4\, s_{\text{asm}}$. Expected canonical X-MRR: ≈0.674; R@5: ≈0.899.

Step 4: Obfuscation evaluation. Apply the specified source-level transformations to the evaluation queries. For each transformed query, recompute its source embedding and run ViC on the transformed source to generate and normalize a new virtual-assembly view; do not reuse assembly generated from the clean query. Keep the candidate index clean in the asymmetric protocol. Evaluate retrieval and detection on their respective subsets and candidate pools (Appendix A.2). Expected detection AUC under strong obfuscation: approximately 0.85–0.88 for source-only retrieval and 0.94–0.95 for fusion.

See Table 22 for all hyperparameters, Appendix D.2 for metric definitions, and Appendix D.5 for statistical methodology.

G Training Method Exploration

To investigate whether alternative training strategies can improve upon the score-level fusion baseline, we conduct five independent experiments. Each experiment modifies exactly one component of the training pipeline while keeping the evaluation protocol fixed (score-level fusion with a=0.6). All experiments use PK sampling (P=12, K=4, 400 batches per epoch) and are evaluated on the same subsampled development set for training-time monitoring, with final numbers reported on the full test set.

G.1 Experiment Overview

Table 34 summarizes the five experiments and their key modifications relative to the baseline.

Table 34: Training method exploration: five independent experiments modifying different components. HN = hard negatives.

ID	Method	Modified component	Key idea
E1	MLP projection	Assembly encoder head	Replace linear projection with 2-layer MLP
E2	Multi-view contrastive	Source encoder	Obfuscation-invariant training with augmented views
E3	Cross-modal alignment	Both encoders	Joint fine-tuning with cross-modal contrastive loss
E4	Temp. annealing + HN	Source encoder	Temperature $0.10\rightarrow0.05$ with hard negative mining
E5	CodeSage-large	Source backbone	1.3B backbone (projection-only, frozen encoder)

G.2 Experiment Details

E1: Assembly Encoder MLP Projection. The baseline assembly encoder uses a single linear projection layer after mean pooling. We replace this with a two-layer MLP ($768\rightarrow768\rightarrow768$) with GELU activation and dropout (p=0.1), hypothesizing that the additional nonlinearity can better

map the CLAP representation space to the shared embedding space. Only the MLP parameters are trained; the RoFormer backbone is frozen.

E2: Obfuscation-Invariant Multi-View Contrastive. We fine-tune the source encoder using multi-view contrastive learning. For each source code snippet, we generate an obfuscated view (identifier renaming) and train the encoder to produce similar embeddings for original and obfuscated versions of the same function, while pushing apart embeddings from different functions. This directly addresses the identifier-renaming vulnerability of source encoders. The contrastive loss combines a standard cross-language term with an obfuscation-invariance term weighted at λ=0.5.

E3: Cross-Modal Alignment Fine-Tuning. Both the source and assembly encoders are jointly fine-tuned using a combined loss: the standard within-modality contrastive loss plus a cross-modal alignment loss that pulls source and assembly embeddings of the same function together while pushing apart embeddings from different functions. This aims to strengthen the complementarity signal between modalities.

E4: Temperature Annealing with Hard Negative Mining. The source encoder is fine-tuned with two modifications: (1) the contrastive temperature is annealed from τ=0.10 to τ=0.05 over training, producing increasingly sharp similarity distributions; (2) hard negative mining upweights the most similar negative pairs in each batch via softmax weighting of negative similarities.

E5: CodeSage-Large Backbone. We replace the CodeSage-base-v2 backbone (110M parameters) with CodeSage-large (1.3B parameters, hidden size 2048). To manage computational cost, we freeze the backbone entirely and only train a two-layer projection head (2048→768→768). This tests whether a stronger source representation improves fusion performance even without full fine-tuning.

G.3 Training Dynamics

Table 35 shows the training trajectories for all five experiments, evaluated on the subsampled development set (∼400 entries, 50 problems). While these subsampled metrics are not directly comparable to the full-test results in the main paper, they reveal training dynamics and relative trends.

Table 35: Training-time evaluation (subsampled, ∼400 entries). Best epoch for each metric shown. Baseline src_mrr = 0.360 on the same subsampled set. Green : best; yellow : second best.

ID	Method	Epochs	Best src	Best asm	Best fused	Δ vs baseline
—	Baseline	—	0.360	—	—	—
E1	MLP projection	10	0.360	0.137	0.339	−0.021
E2	Multi-view contrastive	5	**0.439**	0.257	**0.427**	+0.067
E3	Cross-modal alignment	8	0.368	**0.272**	0.365	+0.005
E4	Temp. annealing + HN	5	0.413	—	0.398	+0.038
E5	CodeSage-large	5	0.304	0.196	0.293	−0.067

Table 36 shows per-epoch training dynamics for the four source-encoder experiments (E2–E5). E1 is omitted because it only trains the assembly projection and source MRR remains at the baseline throughout.

Key observations.

- **E2 dominates on subsampled eval**: Multi-view contrastive training improves source-encoder MRR by +22% (0.360→0.439 at epoch 5), the largest single-experiment gain. Best fused MRR (0.427) occurs at epoch 1; further training improves source-only but hurts fusion, suggesting the assembly channel becomes less complementary as the source encoder strengthens. However, this ranking *reverses* on the full test set (Section G.4), where E4 is the only experiment that improves fused X-MRR.
- **E4 is second-best**: Temperature annealing with hard negatives provides a +15% source-encoder improvement (0.360→0.413). The combination of sharper temperature and hard negative upweighting produces more discriminative embeddings.
- **E3 improves assembly more than source**: Cross-modal alignment produces the best assembly-only MRR (0.272) but only marginal source improvement. The assembly encoder benefits from the cross-modal signal, but the source encoder's learning is dominated by the cross-language contrastive loss.

Table 36: Per-epoch training dynamics (subsampled eval, ∼400 entries). E4 uses the baseline assembly encoder (asm MRR not re-evaluated).

ID	Method	Epoch	Loss	src MRR	asm MRR	fused MRR
E2	Multi-view contrastive	1	3.166	0.433	0.257	**0.427**
		2	3.012	0.428	0.257	0.416
		3	2.946	0.433	0.257	0.425
		4	2.931	0.437	0.257	0.427
		5	2.905	**0.439**	0.257	0.423
E3	Cross-modal alignment	1	5.815	0.368	0.258	0.347
		4	5.167	0.363	0.267	**0.365**
		6	5.138	0.367	0.272	0.362
		8	5.098	0.360	**0.272**	0.364
E4	Temp. anneal + hard neg.	1	2.173	0.411	—	0.397
		2	2.062	0.401	—	0.395
		3	1.988	0.393	—	0.398
		4	1.962	0.413	—	0.396
		5	1.930	0.410	—	0.395
E5	CodeSage-large	1	2.700	0.293	0.196	0.289
		2	2.407	0.284	0.196	0.272
		3	2.299	0.304	0.196	0.289
		4	2.248	0.286	0.196	0.285
		5	2.214	0.301	0.196	0.293

- **E1 shows diminishing returns**: Adding an MLP to the assembly projection does not improve fusion performance, confirming that the assembly encoder's contribution is bounded by complementarity, not representation capacity.
- **Fused < source in E4**: When the source encoder is substantially improved, the baseline (frozen) assembly encoder can actually hurt fusion on the subsampled evaluation set. This motivates jointly improving both modalities or using adaptive fusion weights.
- **E5 underperforms baseline**: Freezing the 1.3B CodeSage-large backbone and training only a projection head ($2048 \to 768$) produces substantially worse embeddings than the fine-tuned base model (best src $= 0.304$ vs. baseline 0.360, best fused $= 0.293$ vs. baseline 0.360). Performance oscillates across epochs (src ranges 0.284–0.304) rather than converging, suggesting the projection head alone cannot align the frozen backbone's representations with the contrastive objective.

G.4 Full-Test Evaluation

We evaluate the best checkpoint from each experiment on the test set using the same evaluation protocol as the main paper (cap=2 entries per language per problem, 768 entries from 154 problems, a=0.6). Table 37 reports the results.

Table 37: Test-set evaluation (cap=2, 768 entries, 154 problems, a=0.6). All rows evaluated with the same script for internal consistency. Green : best; yellow : second best.

ID	Method	src X-MRR	asm X-MRR	fused X-MRR	fused AUC	Δ fused
—	Baseline	0.646	0.355	0.649	0.942	—
E1	MLP projection	0.646	0.065	0.525	0.906	−0.124
E2	Multi-view contrastive	0.663	0.355	0.624	0.955	−0.025
E3	Cross-modal alignment	0.662	**0.366**	0.636	0.964	−0.013
E4	Temp. annealing + HN	**0.666**	0.355	**0.675**	**0.967**	+0.026
E5	CodeSage-large	0.511	0.355	0.532	0.942	−0.117

G.5 Discussion

The full-test results reveal a striking *ranking reversal* compared to training-time monitoring: E2 dominated the subsampled evaluation (Table 35: fused 0.427 vs. baseline 0.360) yet underperforms

the baseline on the test set (0.624 vs. 0.649). Conversely, E4 appeared second-best during training but is the only experiment that improves fused X-MRR on the test set (0.675, $+4.0\%$).

E4: temperature annealing + hard negatives is the clear winner. E4 achieves the best source MRR (0.666), fused MRR (0.675), and fused AUC (0.967). The combination of sharper temperature and hard-negative upweighting produces more discriminative embeddings that *generalize* to the full test distribution, unlike E2's obfuscation-specific training which appears to overfit to the augmentation strategy.

Source improvement does not guarantee fusion improvement. E2 and E3 both improve source MRR (0.663 and 0.662 vs. baseline 0.646) and AUC, yet both *decrease* fused MRR (0.624 and 0.636 vs. 0.649). This confirms the source–fusion divergence observed during training: when the source encoder improves substantially, the fixed-quality assembly channel becomes less complementary and the fixed fusion weight $a{=}0.6$ over-weights the now noisier modality. E4 escapes this trap because its improvement, while meaningful, is modest enough that the assembly signal remains additive.

E3 produces the best assembly encoder. Cross-modal alignment yields the highest assembly MRR (0.366 vs. baseline 0.355) and the second-best fused AUC (0.964), confirming that the assembly encoder benefits from the cross-modal training signal. However, the fused MRR still drops, suggesting that jointly fine-tuning both modalities disrupts the complementarity structure that makes fusion effective.

E1 catastrophically degrades assembly. Replacing the linear projection with an MLP reduces assembly MRR from 0.355 to 0.065—an 82% drop. The subsampled evaluation masked this failure (asm MRR appeared as 0.137). The MLP's additional parameters overfit during training, producing assembly embeddings that are nearly uninformative for retrieval.

Stronger does not mean bigger. E5 demonstrates that scaling the source backbone from 356M to 1.3B parameters, without full fine-tuning, produces substantially worse source MRR (0.511 vs. 0.646). The frozen large backbone's representations are not aligned with the contrastive objective, and the projection head alone cannot compensate.

Training-time metrics can mislead. The ranking reversal between subsampled ($\sim$400 entries) and test-set (768 entries) evaluation is a cautionary finding. Subsampled metrics are useful for monitoring convergence but should not be used to select the final model configuration without validation on a held-out set of representative size.

Future directions. E4's success suggests that curriculum-style training (gradually increasing difficulty) is more robust than augmentation-based approaches. Combining E4's temperature schedule with E3's cross-modal alignment could simultaneously improve discrimination and cross-modal complementarity. Adaptive fusion weights (a as a function of per-query confidence) could further improve results by down-weighting assembly when the source encoder is confident.