Lifting Optimized Binaries to Canonical Compiler IR via Structure-Aware Retrieval and Iterative Verification

Anonymous ACL submission

Abstract

Lifting stripped and highly optimized binaries to the canonical compiler intermediate representation (IR) enables program analysis when source code is unavailable. However, compiler optimizations severely distort control-flow and data-flow structure, making existing rule-based and LLM-based decompilation approaches brittle. We present BRIDGE, a system that reliably lifts optimized binaries to analysis-friendly compiler IR. BRIDGE combines control-flow-aware retrieval-augmented generation with feedback-driven verification. It uses pseudo-probe instrumentation to align optimized binary fragments with normalized IR semantics, and then employs an iterative refinement loop guided by static analysis and runtime feedback to improve executability and semantic consistency. We evaluate BRIDGE on HumanEval-Decompile and MBPP, lifting x86-64 and ARM64 binaries to LLVM IR. BRIDGE outperforms seven baselines, achieving an average of over 30% higher re-executability than the strongest general-purpose LLM baseline.

1 Introduction

Binary decompilation is central to software security analysis and legacy software modernization. Most existing techniques focus on recovering high-level, human-readable source code (Hosseini and Dolan-Gavitt, 2022). However, source-level recovery is often neither necessary nor desirable, as the recovered source code is ambiguous and potentially misleading (Yadavalli and Smith, 2019a; Dramko et al., 2024).

Lifting binaries to a canonical compiler intermediate representation (IR) provides a faithful and stable abstraction of program semantics. Compiler IRs make control and data dependencies explicit and preserve low-level semantic invariants, while remaining independent of source-level syntax. This abstraction is sufficient, and often preferable, for system-level tasks such as automated reasoning, program analysis and transformation, optimization inspection, and retargeting across heterogeneous architectures (Lattner et al., 2020). In these settings, semantic fidelity and analysis reliability are more important than source-level readability.

Despite these benefits, accurately lifting optimized binaries to canonical, pre-optimized compiler IR remains challenging. Aggressive compiler optimizations like -O3 restructure control flow, inline abstractions, and reorder computations, introducing a substantial semantic gap between the binary and its pre-optimization representation (Yadavalli and Smith, 2019b; Zhang and Leach, 2025).

Most prior binary-to-IR lifting approaches rely on deterministic translation rules and heuristic-driven analysis. Rule-based systems such as LLVM MCTOLL (Yadavalli and Smith, 2019b) and RetDec (Software, 2017) employ static analysis to map machine instructions to IR constructs, using fixed heuristics to reconstruct control-flow graphs (CFGs) and recover variables. While effective for simple or lightly optimized binaries, these methods degrade significantly in the presence of aggressive compiler optimizations (e.g., -O3), function inlining, and instruction reordering. As a result, although they may produce functionally executable IR, the recovered code often lacks canonical structure, yielding bloated and fragmented IR that is difficult to analyze, transform, or reuse in downstream system workflows.

Large language models (LLMs) have opened new possibilities for binary analysis, with recent work demonstrating their potential in decompiling binaries into C (Armengol-Estapé et al., 2024; Tan et al., 2024) on x86 platforms. Yet, general-purpose LLMs still struggle to generate compiler IR code effectively. Unlike decompilation to C, where human readability is the primary goal, IR generation requires precise typing, structural fidelity, and strict adherence to semantics.

Translating heavily optimized binaries back to a

pre-optimized compiler IR is difficult because compiler optimizations often destroy high-level structures that align with a developer's mental model, creating a semantic gap that challenges standard LLMs in three fundamental ways. **First, underexplored IR semantics** (Jiang et al., 2025a). General-purpose LLMs are primarily trained on high-level languages, such as Python and C, whereas compiler IRs are underrepresented in their training data. As a result, these models have a limited understanding of strict IR syntax and invariants, which frequently leads to structural errors and invalid IR generation. **Second, control-flow entropy**. Aggressive optimizations flatten structured constructs into unstructured control flow, for example, through inlining and control-flow rewriting. LLMs therefore struggle to recover an analysis-friendly IR, such as canonical nested loops commonly seen in source programs. Reconstructing the original modular structure becomes a high-entropy inverse problem, where the model must infer boundaries that no longer exist in the linearized binary. **Third, misalignment between probabilistic generation and formal semantics**. LLMs generate code by predicting likely next tokens, whereas compiler IR requires strict compliance with deterministic rules, including static single assignment (SSA) form and type constraints. Even with sufficient context, LLMs often hallucinate plausible but invalid constructs(e.g., undefined variables or broken dependency chains) because their objective optimizes statistical likelihood rather than formal correctness (see also Appendix A).

This paper introduces BRIDGE, a framework designed to lift optimized binaries into canonical, pre-optimized compiler IR. BRIDGE integrates a control-flow-aware Retrieval-Augmented Generation (RAG) pipeline with a rigorous iterative verification scheme. First, the framework builds a fine-grained, structure-aware retrieval database that encapsulates key IR constructs and preserves assembly-to-IR alignment via pseudo-probe instrumentation. Second, during inference, it retrieves control-flow-matched exemplars to guide the LLM in generating an initial IR candidate. This generation is then fed into a feedback-directed refinement loop that performs up to five rounds of correction, using both static analysis and runtime execution information to enforce structural and functional correctness. By integrating pseudo-probe alignment with iterative verification, BRIDGE bridges the semantic gap caused by compiler optimizations to enable robust IR reconstruction.

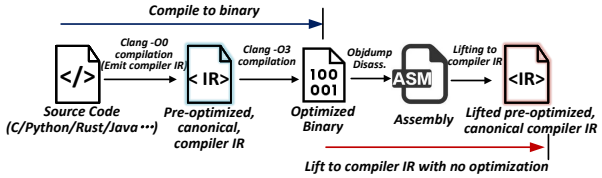


Figure 1: The end-to-end compilation pipeline from source code to optimized binary code, as well as the reverse process in which the binary is lifted back into pre-optimized compiler IR.

We compare BRIDGE against seven baselines: two industry-strength rule-based decompilers- RetDec (Křoustek et al., 2017) and MCTOLL (Yadavalli and Smith, 2019b), and five LLM-based methods: ChatGPT-4o (OpenAI, 2025), DeepSeek-V3 (DeepSeek, 2025), Claude sonnet 4.5 (Anthropic, 2025), Qwen-plus (Tongyi-Lab, 2025), and Gemini-2 (Google, 2025). We evaluate all methods on the HumanEval-Decompile (adapted from HumanEval (Chen et al., 2021)) and MBPP (Austin et al., 2021) benchmarks using x86-64 and ARM64 binaries compiled with four LLVM/Clang optimization levels `-O(0-3)`. BRIDGE outperforms all baselines, generating LLVM IR code that exhibits the best functional and semantic accuracy. This paper makes the following contributions:

- We introduce pseudo-probe instrumentation to build a structure-aware RAG database, aligning optimized binary structure with canonical compiler IR semantics.
- We propose a verification-guided refinement loop that integrates static analysis and runtime feedback to correct hallucinations and enforce SSA validity and functional correctness.
- We evaluate BRIDGE on HumanEval-Decompile and MBPP across x86-64 and ARM64, achieving state-of-the-art re-executability compared to existing baselines.

2 Overview

Figure 1 illustrates the compilation and binary lifting pipeline for the LLVM/Clang compiler. Figure 2 provides an overview of BRIDGE for lifting optimized binaries to pre-optimized, canonical compiler IR through structure-aware retrieval and iterative refinement. This two-stage process involves constructing a `RAG Database`, and `Inference and Refinement`. We first build a fine-grained retrieval database using a set of training programs. This produces a fine-grained retrieval

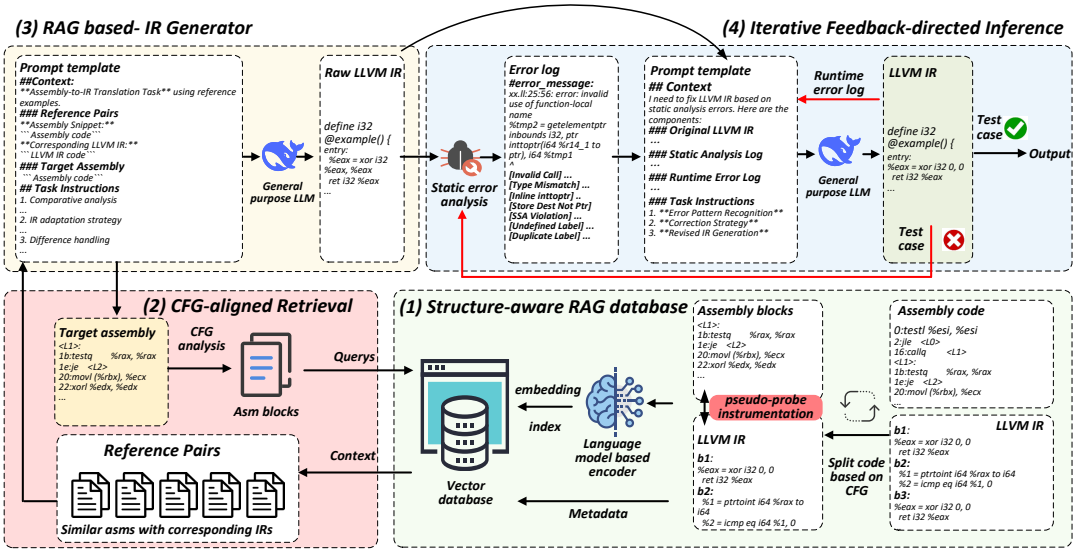


Figure 2: Overview of BRIDGE

database by partitioning pre-optimized, canonical IR into semantic constructs (e.g., loops) and aligning them with optimized assembly (obtained from binary code generated through compiling the training programs) segments using pseudo-probe instrumentation (He et al., 2024). This method maps distorted control flow back to pre-optimized, canonical source structures. Specifically, We use Nova-S-1.3B (an assembly encoder) (Jiang et al., 2025b) to generate input embeddings and FAISS (Johnson et al., 2019) for similarity search. During inference, BRIDGE employs `angr` (Shoshitaishvili et al., 2016) to decompose the input binary into a CFG, We then leverage a lightweight, rule-based segmentation method to identify coarse-grained boundaries for structure-aware retrieval. We retrieve the top-k semantically aligned exemplars to compose a context-rich prompt for LLM. The generated output then enters an iterative feedback loop that uses both static analysis and runtime feedback to correct structural and functional deviations iteratively.

3 Structure-Aware RAG Databases

3.1 Offline RAG Database Preparation

We construct an offline RAG database as the retrieval source for lifting. The database is built from the ExeBench dataset (Armengol-Estapé et al., 2022), which offers a diverse collection of code. To ensure diversity, we filter out benchmarks with high similarity (edit distance similarity > 0.95) and *remove any samples that overlap with our evaluation set*. For each retained benchmark, we compile it into its pre-optimized, canonical LLVM IR (using `clang -O0 -emit-llvm -S`). This IR serves as the ground-truth lifting target. We then compile each IR program with the LLVM/Clang compiler at four optimization levels (`clang -O{0-3}`). Each resulting binary is then disassembled using `llvm-objdump` to produce the corresponding assembly code. Finally, we remove the benchmarks that fail to compile. The final dataset contains 69,253 unique, function-level LLVM IR targets. The complete RAG database contains 277,012 (69,253 $\times$ 4 levels) assembly-IR pairs.

3.2 Control-flow-aware Data Structuring

To transform our flat collection of code pairs into a control-flow-aware database, we employ a three-step structuring process to establish precise mappings between high-level semantic constructs in LLVM IR and their complete assembly-code counterparts. The workflow is illustrated in Figure 3.

Identifying semantic boundaries. We first establish the boundaries of high-level control structures within the source code. Utilizing an AST-based instrumentation script, we inject opaque marker intrinsics (`PROBE(x)`) enclosing constructs such as `if` conditionals, and `for` loops. These anchors are preserved during the translation to pre-optimized, canonical LLVM IR (Figure 3a-b), partitioning the CFG into semantically cohesive subgraphs.

Fine-grained IR-to-assembly mapping. To propagate these semantic partitions into the final machine code, we leverage LLVM native pseudo-probe instrumentation (He et al., 2024). These lightweight probes are injected at the entry of each IR basic block and persist through backend optimizations. This mechanism guarantees a deterministic mapping between each IR block (and its enclosing semantic marker) and the corresponding sequence of disjoint assembly instructions (Figure 3c). Con-

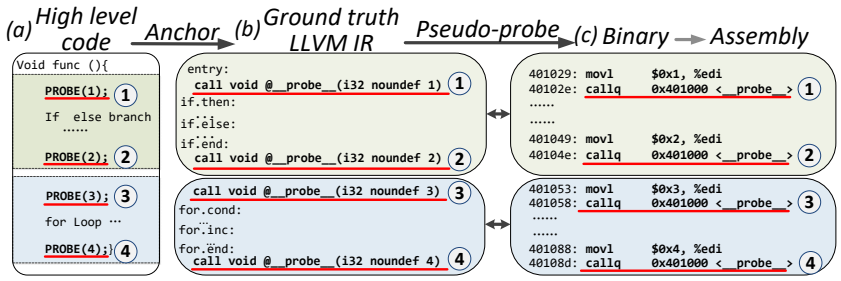


Figure 3: Workflow for constructing control-flow-aware LLVM IR and assembly code pairs.

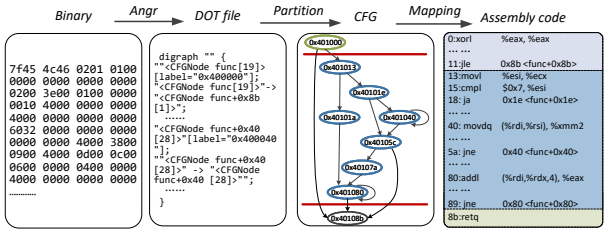


Figure 4: CFG-Guided binary segmentation.

sequently, we can precisely locate the high-level semantic boundaries in the optimized binary even in the presence of instruction scheduling or block reordering.

Constructing structured database entries. Finally, we combine these inputs to construct the retrieval index. The semantic anchors (Step 1) define the boundaries of logical constructs, while the pseudo-probes (Step 2) link the internal IR blocks to specific machine instructions. By intersecting these mappings, we extract the precise assembly sequence for each semantic unit, which is subsequently vectorized to populate the database.

4 CFG-aligned, Iterative Feedback-directed Inference

4.1 Control-Flow-Aware Retrieval

At inference time, we segment the input machine code to facilitate structure-aware retrieval. Specifically, Figure 4 illustrates our binary code segmentation process. We employ `Angr` (Shoshitaishvili et al., 2016) to extract function-level CFGs, pruning external call nodes (e.g., addresses not starting with `0x4`). The graph is partitioned by designating divergence points (out-degree ≥ 2, such as `0x401000`, `0x401013`) as headers for control structures, aggregating linear successors (where out-degree ≤ 1) until a subsequent divergence. We refine these segments based on three rules, (1) Loop Detection, which coalesces cyclic nodes via back-edges, (2) Unrolling Recovery, which merges contiguous self-looping fragments (e.g., `0x401040` and `0x401080`) to restore singular loop semantics disrupted by SIMD optimization, and (3) Conditional Merging, where divergent paths converging at a common join node (e.g., `0x40105c` and `0x401080`) are unified to reconstruct `if-else` logic. This process yields a representation that faithfully preserves high-level program flow.

Next, we execute retrieval using FAISS (Johnson et al., 2019) over a control-flow-aware RAG corpus. To ensure corpus diversity and minimize redundancy, we apply MinHash-based near-duplicate filtering (Hassanian-esfahani and javad Kargar, 2018) prior to indexing. Each candidate fragment is tokenized and projected into a 128-dimensional vector. Fragments exhibiting high similarity are treated as duplicates, and only a single representative instance is retained. For the retrieval index, we embed each CFG-segmented fragment using Nova-S-1.3B (Jiang et al., 2025b), a specialized assembly encoder that effectively captures the semantics, and utilize an `IndexFlatL2` structure wrapped with `IndexIDMap` to support precise 1:1 vector-to-fragment mapping. In detail, we perform a nearest neighbor search to identify the top-k structurally similar entries. Empirical evaluation with $k \in \{1, 3, 5\}$ indicates that k=3 provides the optimal trade-off between retrieval accuracy (Top-1: 95.92%, Top-3: 98.61%, Top-5: 98.83%) and noise robustness. Each retrieved entry consists of an aligned ⟨assembly, LLVM IR⟩ pair. To construct the prompt (details in Appendix F), we append the top-k exemplars to the query fragment ordered by retrieval rank. The resulting context conditions the general-purpose LLM for IR reconstruction, the output of which is subsequently passed to the verification loop.

4.2 Verification-Guided Iterative Refinement

To bridge the semantic gap between probabilistic generation and formal correctness, we implement an iterative, feedback-directed inference scheme. This process leverages both static and dynamic error information to guide the model from an initial, potentially flawed prediction to a functionally valid result (Appendix F lists the prompts used). First, we feed the initial inferred LLVM IR to static veri-

fication, utilizing the native LLVM verifier and attempting compilation to detect structural anomalies such as multiple definitions, SSA violations, and invalid constant expressions (Appendix D provides the details). Second, we validate semantic correctness by running the program with a suite of unit tests. Upon detecting a failure, we aggregate static diagnostic logs and runtime error traces to construct a targeted feedback prompt. This feedback guides the model in a subsequent refinement step, iterating up to a maximum of five times to progressively align the generated IR with strict compiler constraints and functional specifications.

5 Experimental Setup

5.1 Evaluation Datasets

We evaluate BRIDGE on HumanEval-Decompile (Tan et al., 2024) and MBPP (Austin et al., 2021) datasets on X86-64 and ARM64 instruction sets under the Linux platform, adapting both benchmarks to the binary-to-IR lifting task. To establish ground truth, we compile the original source code into pre-optimized, canonical LLVM IR with `-O0`. These programs are then compiled into binaries using LLVM/Clang v19 across four optimization levels (`-O0` to `-O3`), and the corresponding assembly is extracted via `llvm-objdump`. The resulting inputs vary significantly in complexity, ranging from 2 to 628 assembly instructions (approximately 21 to 10,647 tokens for the DeepSeek-V3 tokenizer). All evaluation benchmarks are excluded from our retrieval database.

5.2 Competitive Baselines

We compare BRIDGE against seven baselines, including two rule-based decompilers RetDec (Křoustek et al., 2017) and MCTOLL (Yadavalli and Smith, 2019b) (only support x86-64), and five general-purpose LLMs, which are DeepSeek-V3 (DeepSeek, 2025), ChatGPT-4o (OpenAI, 2025), Claude sonnet 4.5 (Anthropic, 2025), Qwen-Plus (Tongyi-Lab, 2025), and Gemini-2 (Google, 2025). All LLMs are accessed via the API interfaces. We execute each evaluation five times using identical prompts and report the geometric mean for all metrics. We exclude LLM Compiler FTD (Cummins et al., 2025) and Forklift (Armengol-Estapé et al., 2024) due to a fundamental difference in problem scope. These methods focus on lifting compiler-emitted assembly text (such as `Clang -S` output) rather than the stripped binaries targeted by our work.

5.3 Metrics

We employ five metrics to evaluate the lifting performance. `Re-compilability` measures the percentage of lifted LLVM IR code that compiles successfully without errors. *Re-executability* measures the percentage of outputs that pass unit tests. *Edit Similarity* quantifies textual fidelity via normalized edit distance (Armengol-Estapé et al., 2024). For structural evaluation, we report *CFG Full Match Accuracy*, which tests for exact graph isomorphism (Hagberg et al., 2024), and *CFG Similarity*, which assesses topological alignment using the Weisfeiler-Lehman kernel (Siglidis et al., 2020).

6 Experimental Results

6.1 Overall Performance

Table 1 reports the performance of all methods.

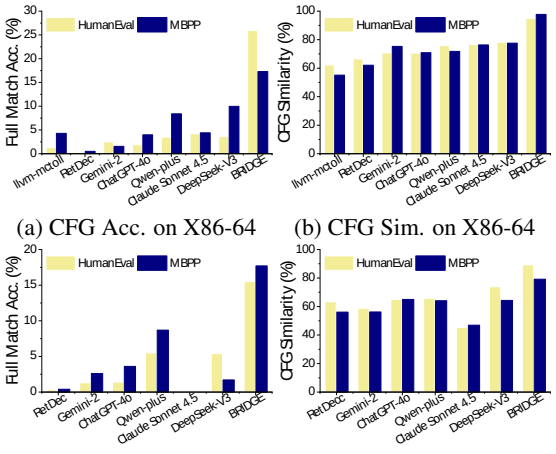
Outperforming baselines. BRIDGE outperforms both rule-based decompilers and general-purpose LLMs by a significant margin on re-executability and edit similarity. On the x86-64 HumanEval-Decompile benchmark, BRIDGE achieves an average re-executability of 65.2%. While both the rule-based decompiler MCTOLL and RetDec outperform general-purpose LLMs in re-executability, they still fall short of our approach, indicating limitations in bridging the semantic gap for complex logic. General-purpose LLMs such as Claude Sonnet 4.5 struggle to generate executable code (11.9%). Furthermore, we identify a divergence in rule-based tools, while RetDec achieves high re-compilability (92.5% on x86-64 HumanEval), nearly half of this code fails to execute, indicating it produces syntactically valid but semantically hollow code. In contrast, BRIDGE maintains a strong correlation between compilation and execution, demonstrating that our feedback loop enforces both syntactic validity and semantic integrity.

Robustness to compiler optimizations. A major challenge in lifting tasks is the structural entropy introduced by compiler optimizations, which severely disrupts existing methods. While MCTOLL performs competitively at `-O0` (77.0%) on x86-64, it degrades sharply at `-O3` (54.7%) due to the brittleness of pattern matching. General-purpose LLMs struggle even more severely. For instance, DeepSeek-V3 sees its re-executability drop to 7.4% under aggressive optimization. In contrast, BRIDGE maintains robust performance in both re-executability and edit similarity, demonstrating that structure-aware retrieval effectively mitigates topological distortion to recover high-level semantics.

X86-64																
		Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
Dataset	Opt. Level	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
HumanEval-Decompile	MCTOLL	87.2	75.0	63.5	62.8	72.1	77.0	62.1	54.7	54.7	62.1	24.6	22.0	21.8	21.6	22.5
	RetDec	**93.9**	**92.7**	**92.1**	**91.5**	**92.5**	55.5	48.2	39.6	37.8	45.3	35.6	35.5	32.0	31.7	33.7
	Qwen-plus	25.7	9.5	8.7	6.8	12.7	16.2	9.4	4.1	3.4	8.3	37.5	25.6	25.4	23.6	28.0
	Gemini-2	33.7	12.2	10.8	10.1	16.7	14.2	4.7	4.1	2.0	6.3	33.9	23.5	21.2	20.7	24.8
	ChatGPT-4o	20.3	9.5	8.8	7.4	11.5	11.5	2.0	2.0	2.0	4.4	41.8	25.0	24.9	23.5	28.8
	Claude Sonnet 4.5	25.0	27.4	22.3	19.5	18.3	18.3	14.0	8.5	6.7	11.9	39.9	29.7	29.9	28.7	32.0
	DeepSeek-v3	30.4	35.8	25.0	22.3	28.4	14.2	9.5	9.4	7.4	10.1	44.9	30.5	30.7	31.0	34.3
	BRIDGE	94.6	90.5	83.8	83.8	88.2	**85.1**	**63.5**	**56.8**	**55.4**	**65.2**	**62.8**	**49.7**	**40.6**	**40.5**	**48.4**
MBPP	MCTOLL	84.8	76.2	71.0	70.4	75.6	74.2	65.0	60.4	59.8	64.9	25.5	25.8	22.2	22.0	23.9
	RetDec	**94.6**	**95.4**	**95.8**	**95.8**	**95.4**	49.4	48.0	36.8	36.4	42.7	34.2	34.0	30.0	29.7	32.0
	Qwen-plus	36.2	22.6	18.6	18.4	24.0	21.4	8.8	7.8	7.8	11.5	33.6	24.2	23.9	22.7	26.1
	Gemini-2	14.6	11.4	11.2	11.0	12.1	6.4	4.6	4.6	3.6	4.8	33.9	23.1	22.8	22.6	25.6
	ChatGPT-4o	28.0	20.0	19.4	18.0	21.4	14.2	7.6	7.6	7.8	9.3	38.5	22.9	22.4	21.7	26.4
	Claude Sonnet 4.5	40.8	35.2	33.2	32.0	35.3	29.4	21.2	18.0	17.4	21.5	40.9	28.6	28.6	27.1	31.3
	DeepSeek-v3	41.0	35.2	32.4	31.0	34.9	25.0	17.6	14.2	15.6	18.1	40.8	26.6	26.5	25.5	29.9
	BRIDGE	93.2	88.8	86.6	82.4	87.8	**82.4**	**62.6**	**59.8**	**56.6**	**65.4**	**61.3**	**39.9**	**38.9**	**38.9**	**44.8**
ARM64																
		Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
Dataset	Opt. Level	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
HumanEval-Decompile	RetDec	**93.9**	**92.1**	**92.7**	**92.7**	**92.8**	0	0.6	0.6	0.6	0.5	33.8	34.0	32.1	31.8	32.9
	Qwen-plus	26.2	11.1	18.5	13.6	17.4	8.5	0.6	1.2	2.5	3.2	35.2	25.1	25.7	25.9	28.0
	Gemini-2	7.9	6.8	4.9	6.8	6.6	1.2	0.0	1.2	1.2	0.9	32.5	22.5	23.5	22.9	25.4
	ChatGPT-4o	20.1	11.7	9.3	10.5	12.9	8.5	3.1	1.2	1.2	3.5	31.3	23.0	23.6	23.5	25.4
	Claude Sonnet 4.5	10.4	26.2	20.1	22.0	19.7	7.9	11.0	7.9	7.9	8.7	38.5	27.5	27.9	27.5	30.4
	DeepSeek-v3	27.4	42.6	34.0	35.5	34.6	13.4	10.5	8.6	7.4	10.0	41.9	27.7	28.3	28.2	31.6
	BRIDGE	78.0	65.2	63.4	62.8	67.4	**64.0**	**42.7**	**38.4**	**37.2**	**45.6**	**59.4**	**45.8**	**43.8**	42.7	**48.0**
MBPP	RetDec	**96.0**	**93.4**	**93.4**	**93.2**	**94.0**	0.2	0.4	0.4	0.4	0.4	32.2	31.8	29.8	29.4	30.7
	Qwen-plus	36.0	25.3	23.0	22.5	26.7	19.6	9.4	8.6	8.2	11.5	33.8	24.1	24.4	25.0	26.9
	Gemini-2	19.6	16.6	14.1	13.1	15.9	6.0	4.5	5.2	5.7	5.3	34.2	22.9	23.6	23.2	26.0
	ChatGPT-4o	33.2	18.2	16.2	15.6	21.4	18.2	7.8	7.0	6.8	10.0	30.6	22.0	22.2	22.4	24.3
	Claude Sonnet 4.5	16.8	22.2	22.4	22.8	21.1	12.0	11.6	11.0	11.0	11.4	38.9	25.5	26.2	26.6	29.3
	DeepSeek-v3	39.2	46.4	44.6	42.3	43.1	17.6	15.9	15.7	15.5	16.2	40.2	24.5	25.5	25.3	29.0
	BRIDGE	89.4	75.1	67.1	69.4	75.2	**61.1**	**37.7**	**34.9**	**34.9**	**42.1**	**52.8**	**46.6**	**45.0**	**45.5**	**47.5**

Table 1: Comparison of binary-to-LLVM IR (unoptimized) lifting performance among BRIDGE, the rule-based decompiler RetDec and MCTOLL (x86-64 only), and five general-purpose LLMs. The evaluation covers the HumanEval-Decompile and MBPP benchmarks across x86-64 and ARM64 architectures under four compiler optimization flags (`-O0`–`-O3`) in recompilability, re-executability, and edit similarity.

Cross-architecture generalization. The ARM64 results validate the architecture-agnostic nature of our approach. Traditional tools like MCTOLL lack support for ARM64, and while RetDec supports the architecture, it fails completely in semantic recovery, yielding negligible re-executability (about 0.5% across benchmarks). Detailed analysis reveals that RetDec struggles to resolve callee addresses and prototypes on ARM64, generating non-executable placeholders like `_decompiler_undefined_function_`. However, the edit similarity remains high because the tool correctly lifts the remaining standard instructions, preserving textual structure despite the functional failure. Similarly, general-purpose LLMs also struggle with the ARM64 context. For instance, on the ARM64 HumanEval-Decompile benchmark, Gemini-2 achieves only 0.9% average re-executability. BRIDGE significantly outperforms these baselines, averaging 42.1%. The cross-architectural performance consistency of BRIDGE confirms that combining control flow aware retrieval with iterative feedback refinement establishes a robust structural invariant, enabling effective generalization across diverse architectures.



(a) CFG Acc. on X86-64 (b) CFG Sim. on X86-64

(c) CFG Acc. on ARM64 (d) CFG Sim. on ARM64

Figure 5: LLVM IR CFG reconstruction performance on the HumanEval-Decompile and MBPP datasets across x86-64 and ARM64 architectures.

6.2 Structural Reconstruction Analysis

Figure 5 presents the recovered CFG accuracy of the lifted LLVM IR. BRIDGE performs best in recovering the correct program CFG among seven

	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
Opt. Level	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Basic Block level	67.5	50.6	44.5	36.4	49.8	56.1	23.6	20.9	20.2	30.2	62.7	41.2	40.2	39.8	46.0
Control flow level	70.9	**60.8**	**54.1**	**54.1**	**60.0**	56.1	**28.4**	**27.0**	**25.7**	**34.3**	**65.0**	**49.9**	**42.8**	**41.5**	**49.8**
Function level	**77.7**	53.4	53.4	47.3	58.0	**63.5**	22.3	17.6	16.2	29.9	64.9	41.2	40.9	39.3	46.6

Table 2: BRIDGE with RAG at different retrieval granularities on the HumanEval-Decompile dataset (x86-64).

	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
Opt. Level	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Without error feedback	70.9	60.8	54.1	54.1	60.0	56.1	28.4	27.0	25.7	34.3	**65.0**	**49.9**	**42.8**	**41.5**	**49.8**
With error feedback	**94.6**	**90.5**	**83.8**	**83.8**	**88.2**	**85.1**	**63.5**	**56.8**	**55.4**	**65.2**	62.8	49.7	40.6	40.5	48.4

Table 3: BRIDGE with and without error feedback on the HumanEval-Decompile dataset (x86-64).

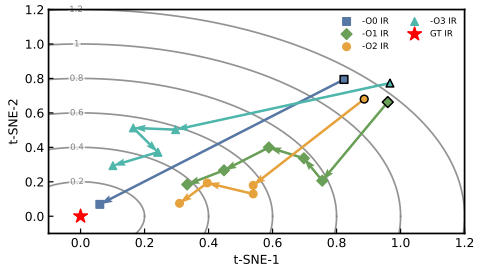


Figure 6: t-SNE visualization of the semantic refinement trajectory for one test case (ID:132) in the HumanEval-Decompile dataset. We project the latent embeddings of the lifted LLVM IR using `IR2vector` (VenkataKeerthy et al., 2020). The paths trace the evolution from the initial input through five iterative refinement cycles toward the Ground Truth (Red Star).

baselines. On x86-64 HumanEval-Decompile, it achieves a CFG Full Match Accuracy of 25.9%, compared to just 3.6% for DeepSeek-V3. This advantage persists on ARM64, where BRIDGE achieves 17.7% exact matches on MBPP. The high CFG similarity scores (>94% on x86-64) indicate that our retrieval mechanism effectively imposes structural constraints, preventing the control-flow hallucinations common in standard LLMs.

6.3 Lifting Direction Correction

Figure 6 shows the semantic convergence of the lifted LLVM IR. Trajectories initiate from the rule-based baseline (MCTOLL) and trace the refinement steps. We observe that initial embeddings for all optimization levels cluster in a region distant from the Ground Truth, indicating that rule-based lifting consistently delivers representations that are structurally distinct from the ground truth. For input compiled with the `-O0` flag, the BRIDGE bridges this semantic gap in a single step (blue trajectory), confirming that low-entropy input facilitates straightforward recovery. In contrast, high-level optimized binaries (`-O1`–`-O3`) necessitate a multi-step iterative refinement process. These trajectories exhibit gradual, stepwise progression, demonstrating the BRIDGE's effort to incrementally disentangle the complex structural distortions introduced by optimization before converging to the target semantic neighborhood.

6.4 Ablation Studies

Multi-level RAG for IR reconstruction. Table 2 evaluates the lifting performance using basic-block, control-flow, and function-level retrieval granularities. The results show that control-flow level retrieval achieves the highest average re-compilability(60.0%), re-executability (34.3%) and edit similarity (49.8%). While function-level retrieval excels at `-O0` (63.5% re-executability) by leveraging intact high-level semantics, it suffers severe degradation at `-O3` (dropping to 16.2%) as compiler transformations distort the global function topology. Conversely, basic-block retrieval lacks sufficient structural context, resulting in the worst overall performance. The results demonstrate that control-flow-aware segmentation captures structural context to preserve semantics while remaining fine-grained, thereby remaining resilient to global structural distortions introduced by high optimization levels.

With and without error verification scheme. Table 3 demonstrates the impact of our iterative error feedback mechanism on lifting performance. The data shows that feedback is essential for functional correctness, as it nearly doubles the average re-executability from 34.3% to 65.2%. This gain is especially visible in highly optimized code, where `-O3` performance increases from 25.7% to 55.4%. These results confirm that the feedback loop resolves subtle semantic errors, such as type mismatches, that the initial generation often misses. Interestingly, this improvement in executability happens without a corresponding increase in edit similarity, which remains effectively flat (49.8% vs. 48.4%). This decoupling suggests that the feedback process prioritizes logical validity over strict textual adherence to the reference code, modifying the output to satisfy the compiler even if it diverges slightly from the exact ground truth structure.

Iteration	Re-executability (%)	Edit Similarity (%)
0	34.3	49.8
1	44.1	44.3
3	58.4	48.3
5	65.2	48.4
7	65.8	47.3
10	66.9	47.7

Table 4: Impact of the iterative refinement count on BRIDGE performance on the HumanEval-Decompile dataset (x86-64). $k = 0$ represents the initial inference without feedback.

Impact of iterative refinement. Table 4 analyzes the efficacy of our feedback-guided refinement loop by varying the number of iterations (k). We observe that re-executability improves significantly in the early stages, increasing by 21.1% from $k = 0$ to $k = 5$. In edit similarity, BRIDGE achieves 49.8% at $k = 0$, as it learns from retrieved templates; however, re-executability is low (34.3%) due to semantic hallucinations. At $k = 1$, similarity drops sharply to 44.3% as the model executes aggressive syntactic restructuring to satisfy the verifier, effectively sacrificing textual fidelity for structural validity. However, as the logic converges ($k = 3$ to 5), similarity rebounds to 48%. This recovery confirms that the feedback loop actively aligns the code with the canonical ground truth, rather than merely applying superficial fixes. Beyond $k = 5$, we observe diminishing returns in re-executability. Furthermore, excessive iterations ($k = 10$) prove detrimental to structural stability. Consequently, we adopt $k = 5$ as the threshold to balance functional correctness with computational efficiency.

7 Related Work

Symbolic and rule-based IR recovery. Recovering compiler IR from binaries is foundational for analysis tasks like malware inspection. Traditional symbolic approaches, such as McSema (Bits, 2021), LLVM-MCTOLL (Yadavalli and Smith, 2019b), RetDec (Software, 2017), and RevGen (S2E Team, 2020), rely on deterministic transformation rules to map instructions to IR. While effective for unoptimized code, these methods struggle with the stochastic nature of aggressive optimizations (e.g., `-O3`), frequently failing to reconstruct SSA form or resolve indirect control flow. Although recent work (Wodiany et al., 2024) introduces debloating and structuring techniques, they often lack generalizability or prioritize code size over the semantic precision required for high-fidelity analysis.

Neural sequence generation for decompilation. Recent works like LLM4Decompile (Tan et al., 2024) demonstrating success in inferring high-level C semantics from binaries. However, lifting to LLVM IR presents a distinct constrained generation challenge, requiring strict adherence to rigid compiler semantics that standard LLMs often violate. Recent studies highlight that aligning LLMs with LLVM IR semantics remains a significant challenge in representation learning (Zhang and Leach, 2025). Furthermore, approaches like Meta's LLM Compiler FTD (Cummins et al., 2025) and Forklift (Armengol-Estapé et al., 2024) typically target compiler-emitted assembly text rather than stripped binaries, relying on metadata absent in true decompilation scenarios. BRIDGE addresses these limitations by leveraging structure-aware retrieval to enforce strict semantic compliance within a general-purpose generation framework.

Retrieval-augmented generation for code RAG (Lewis et al., 2020) effectively enhances code synthesis by conditioning generation on external context. However, its application to low-level binary analysis remains underutilized. Existing methods (Wang et al., 2025) focus on high-level source code, lacking mechanisms to model the non-linear topology of assembly. While token-based retrieval for binaries exists (Cao et al., 2024), it does not account for complex IR dependencies. BRIDGE advances this domain by pioneering a control-flow-aware RAG system specifically for binary-to-IR lifting, utilizing fine-grained alignment to outperform generative baselines in structural fidelity.

8 Conclusion

We have presented BRIDGE, a control-flow-aware RAG framework to lift stripped binaries to canonical and analysis-friendly compiler IR. By leveraging control-flow-aware retrieval as context and utilizing iterative compiler feedback, our approach effectively recovers the structural patterns obscured by aggressive optimization flags. BRIDGE mitigates the hallucination issues inherent in general-purpose LLMs and overcomes the rigid fragility of traditional rule-based lifting, offering a robust solution for automated binary analysis. Extensive evaluation on x86-64 and ARM64 instruction sets demonstrates that BRIDGE achieves state-of-the-art performance.

Online materials The evaluation datasets and additional experimental results are available in our anonymous repository at: `https://anonymous.4open.science/r/BRIDGE-5E6A`.

Limitations

We have demonstrated that our approach generalizes across x86-64 and ARM64. Applying this method to other architectures (e.g., MIPS or RISC-V) requires reconstructing the structure-aware RAG database for the target instruction set. This is not a methodological limitation, as our fully automated pipeline allows for straightforward adaptation by re-running data generation with the appropriate compiler backend. Moreover, our evaluation benchmarks (HumanEval-Decompile and MBPP) provide ready-made test cases. However, in practical reverse engineering scenarios, such as analyzing malware or legacy firmware, such ground truth oracles are rarely available. To address this, BRIDGE can integrate automated test generation techniques. Existing literature offers robust solutions for synthesizing input-output pairs directly from binaries, such as symbolic execution engines or binary fuzzing frameworks. By incorporating these tools to generate a proxy oracle, BRIDGE can support feedback-directed refinement in blind decompilation without requiring developer-written tests.

Ethical considerations

Our research relies exclusively on publicly available, open-source datasets (ExeBench, HumanEval, MBPP). We ensured that all code and models integrated into our research adhere to open-access policies. The methodology ensures full compliance with copyright and intellectual property laws, thereby eliminating any potential for infringement or unauthorized use of protected materials.

References

Anthropic. 2025. Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5. Accessed: 2025-11-24.

Jordi Armengol-Estapé, Rodrigo C. O. Rocha, Jackson Woodruff, Pasquale Minervini, and Michael O'Boyle. 2024. Forklift: An extensible neural lifter. In *First Conference on Language Modeling (COLM)*.

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. 2022. Exebench: An ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, page 50–59.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O'Boyle. 2024. Slade: A portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 67–80.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Lifting Bits. 2021. Mcsema: Lifting x86 and x86-64 binaries to llvm bitcode. https://github.com/lifting-bits/mcsema. Accessed: 2025-01-24.

Qingqing Cao, Sewon Min, Yizhong Wang, and Hannaneh Hajishirzi. 2024. Btr: Binary token representations for efficient retrieval augmented language models. In *International Conference on Representation Learning*, pages 44229–44246.

Mark Chen, Jerry Tworek, Heewoo Jun, and so on. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, page 141–153.

DeepSeek. 2025. Deepseek chat. https://chat.deepseek.com/. Accessed: 2025-01-24.

Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A taxonomy of c decompiler fidelity issues. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 379–396.

Google. 2025. Gemini: Google's AI Model. https://gemini.google.com/. Accessed: 2025-01-24.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2024. Networkx: Graph isomorphism — networkx documentation. https://networkx.org/documentation/stable/reference/algorithms/isomorphism.html. Accessed: 2025-01-24.

Roya Hassanian-esfahani and Mohammad javad Kargar. 2018. Sectional minhash for near-duplicate detection. *Expert Systems with Applications*, 99:203–212.

Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. 2024. Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 322–333.

Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the c: Retargetable decompilation using neural machine translation. In *Workshop on Binary Analysis Research at NDSS 2022*.

Hailong Jiang, Jianfeng Zhu, Yao Wan, Bo Fang, Hongyu Zhang, Ruoming Jin, and Qiang Guan. 2025a. Can large language models understand intermediate representations in compilers? In *Forty-second International Conference on Machine Learning*.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2025b. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In *The Thirteenth International Conference on Learning Representations (ICLR) 2025*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Jakub Křoustek, Peter Matula, and Petr Zemek. 2017. Retdec: An open-source machine-code decompiler. *July 2018*.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. Mlir: A compiler infrastructure for the end of moore's law. *Preprint*, arXiv:2002.11054.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474.

LLVM Project. 2025. The llvm compiler infrastructure project. https://github.com/llvm/llvm-project. Accessed: 2025-01-24.

OpenAI. 2025. ChatGPT. https://chatgpt.com/. Accessed: 2025-01-24.

S2E Team. 2020. Translating binaries to llvm with revgen. https://s2e.systems/docs/Tutorials/Revgen/Revgen.html. Accessed: 2025-01-24.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157.

Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2020. Grakel: A graph kernel library in python. *Journal of Machine Learning Research*, 21(54):1–5.

Avast Software. 2017. Retdec: Open-source machine-code decompiler. https://github.com/avast/retdec.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3473–3487.

Tongyi-Lab. 2025. Qwen. https://chat.qwen.ai/. Accessed: 2025-01-24.

S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.*, 17(4).

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214.

Igor Wodiany, Antoniu Pop, and Mikel Luján. 2024. Leanbin: Harnessing lifting and recompilation to debloat binaries. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, page 1434–1446.

S Bharadwaj Yadavalli and Aaron Smith. 2019a. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 213–218.

S. Bharadwaj Yadavalli and Aaron Smith. 2019b. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, page 213–218.

Yifan Zhang and Kevin Leach. 2025. Training large language models to comprehend llvm ir via feedback-driven optimization. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1477–1478.

A Motivating Case Study: Lifting Optimized Binaries

To illustrate the structural challenges addressed in this work, we detail the specific task of lifting an aggressively optimized (`-O3`) binary back to its canonical LLVM IR (`-O0`). Figure 7 visualizes the complete transformation pipeline used in our evaluation. The process begins with the reference ground truth LLVM IR (Figure 7a), which uses conditional `if-else` control flow to perform arithmetic operations, either doubling or bit-shifting variable `a` and combining the result with variable `b`.

To create the optimized binary, we compile this LLVM IR using Clang (version 19) (LLVM Project, 2025) with the **-O3** flag. The decompilation task begins by disassembling the optimized binary into assembly code with **llvm-objdump** (LLVM Project, 2025). This assembly code is then used as input for various decompilers, which attempt to reconstruct the LLVM IR.

Figure 7 compares the lifting results of six approaches for converting `-O3` optimized assembly code to canonical LLVM IR (`-O0`). These include the LLVM-based decompiler MCTOLL (Yadavalli and Smith, 2019b), the Meta fine-tuned LLM Compiler FTD (Cummins et al., 2025), and two general-purpose large language models: ChatGPT-4o (OpenAI, 2025) and DeepSeek-V3 (DeepSeek, 2025). Additionally, we explore two variants that integrate RAG with two general-purpose LLMs. RAG retrieves similar code segments from the ExeBench dataset (Armengol-Estapé et al., 2022) to construct contextual prompts, which enhances the decompilation process. For the two general-purpose LLMs, we use the prompt: `Convert the following assembly code to LLVM IR (no optimization)`. Table 5 reports the lifting performance using three metrics:

- **Re-executable** assesses whether the reconstructed LLVM IR executes all test cases correctly, ensuring functional equivalence to the original program.
- **Edit Similarity** (Armengol-Estapé et al., 2024) measures the similarity between the lifted LLVM IR and the ground-truth LLVM IR. It yields a score between 0 and 1, where higher values indicate better syntactic and semantic preservation.
- **CFG Match** evaluates whether the reconstructed LLVM IR recovers the original if−else branching logic from the machine code.

LLVM MCTOLL (Yadavalli and Smith, 2019b) is a tool within the LLVM ecosystem designed for binary lifting by applying manually designed transformation rules. While it can lift optimized binaries into functionally correct LLVM IR, the resulting code structurally diverges from the original. It replaces explicit control flow like (e.g., `if-else`) statements with `select` instructions and low-level arithmetic. This process obscures high-level semantics, which complicates further analysis of the code.

LLM Compiler FTD (Cummins et al., 2025) is fine-tuned to reconstruct LLVM-IR from assembly code. The model's inferred results, however, exhibit significant semantic errors. It incorrectly generates a return of an undefined value (`undef`). This failure to reconstruct essential arithmetic operations and conditional (`if-else`) control-flow structures from the assembly input highlights its limitations. A key distinction is that LLM Compiler FTD targets the lifting of compiler-emitted assembly text (such as output from `Clang -S`), not the disassembly produced from raw binaries using tools like `llvm-objdump`. The latter form of assembly, which is the focus of this paper, differs significantly by including stripped symbols and indirect control flow.

General-Purpose LLMs. `ChatGPT-4o` successfully infers the arithmetic logic, but it fails to recover the canonical control flow graph. Instead of reconstructing the original if-else hierarchy, it generates a flattened, linearized representation using select instructions, effectively acting as a translator for the optimized assembly rather than a true lifter. `DeepSeek-V3` produces a highly concise and semantically correct implementation, but it fundamentally fails the "De-optimization" objective. In fact, it generates code that resembles optimized LLVM IR (e.g., after mem2reg and simplifycfg passes) rather than the target unoptimized (-O0) form. *Despite capturing core computations like shift and add, both General-Purpose LLMs fail to recover the original* `if-else` *control flow, limiting structural fidelity.*

General-Purpose LLMs with Our RAG Scheme. To evaluate the impact of RAG on LLVM IR reconstruction, we integrate RAG modules that supply semantically similar assembly-IR pairs to the

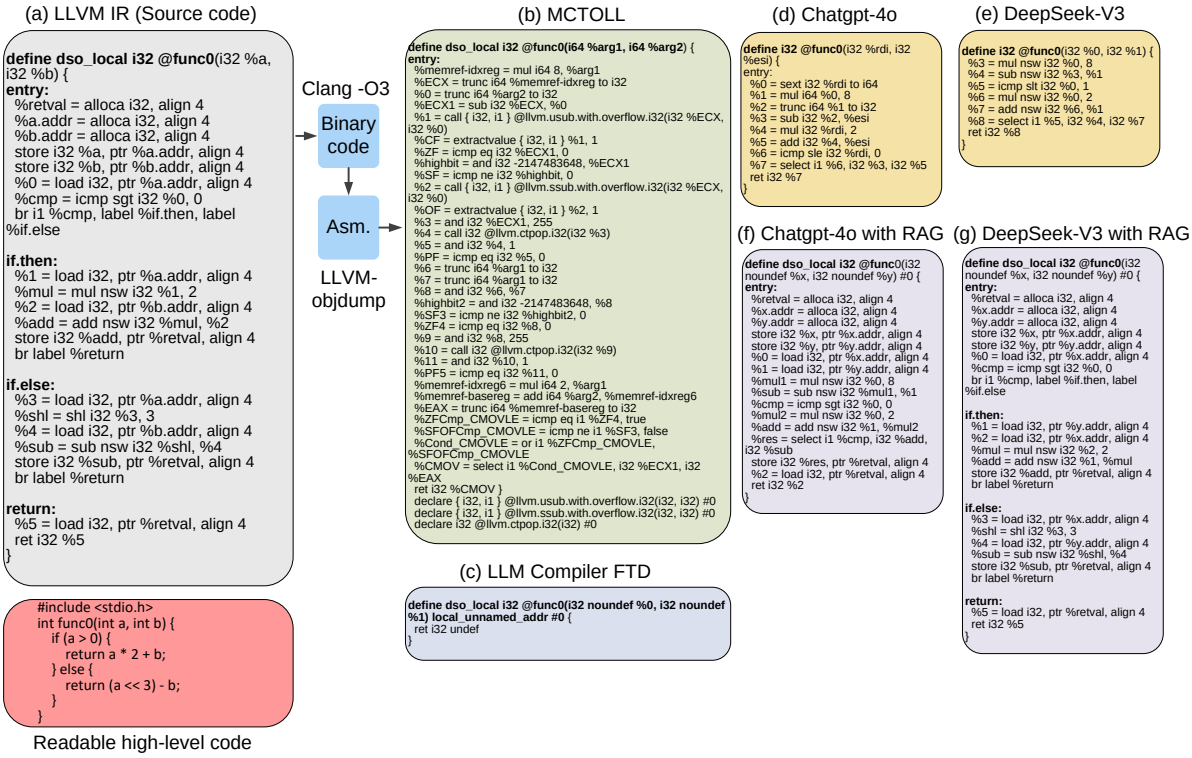


Figure 7: Decompilation results of various approaches (b - g) to the ground truth (a).

Method	Re-executable	Edit Sim. (%)	CFG Match.
MCTOLL	✓	31.6	✗
LLM Compiler FTD	✗	11.0	✗
ChatGPT-4o	✓	32.0	✗
ChatGPT-4o+RAG	✓	65.0	✗
DeepSeek-v3	✓	30.0	✗
DeepSeek+RAG	✓	**85.9**	✓

Table 5: Comparison of different decompilation methods in edit similarity, test case pass, and branch recovery on x86-64.

two LLMs. The outputs generated by ChatGPT-4o and DeepSeek-v3, when guided by our control-flow-aware RAG scheme, demonstrate a significant behavioral shift compared to their unguided counterparts. The retrieved exemplars serve as effective constraints, enforcing alignment with the target unoptimized IR's memory model and control flow topology. `ChatGPT-4o with RAG` (Figure 7f) successfully incorporates key structural elements that are absent in its unguided generation. It correctly adopts the canonical unoptimized memory model, deploying `alloca` instructions for local variables (`%x.addr`, `%y.addr`) alongside the requisite `load`/`store` operations. Furthermore, the arithmetic logic is correctly inferred (multiplication/addition and shift/subtraction). However, a critical structural deviation persists: the model fails to reconstruct the explicit conditional branching of the source. Instead, it retains the flattened, linear topology inferred from the optimized assembly and implements the logic via a `select` instruction. This result indicates that while RAG successfully enforces memory model conventions, it does not fully override ChatGPT-4o's strong bias toward the optimized, flat structure. Consequently, the refined output improves Edit Similarity to 65%, however, it still fails functional correctness and does not match the original control-flow graph. In contrast, `DeepSeek-v3 with RAG` (Figure 7g) demonstrates a successful lift to the canonical target form, satisfying both functional and structural requirements. Similar to ChatGPT-4o, it correctly synthesizes the `alloca`/`load`/`store` memory model required for the `-O0` target. Crucially, however, the model successfully reconstructs the explicit CFG. It generates distinct basic blocks for the conditional branches (`if.then`, `if.else`) connected by conditional `br` instructions, achieving a topology that aligns with the Ground Truth.

Insight. Lifting optimized binary code back into canonical Compiler IR is challenging, primarily due to aggressive compiler optimizations (e.g., `-O3`) that obscure high-level control structures by replacing them with low-level, performance-oriented instructions. This transformation complicates accurate reconstruction, a task that general-purpose LLMs struggle with due to their limited understand-

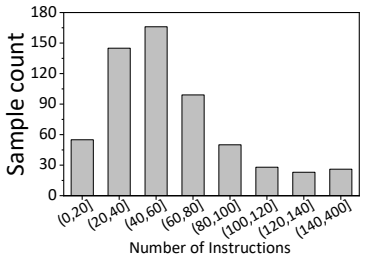


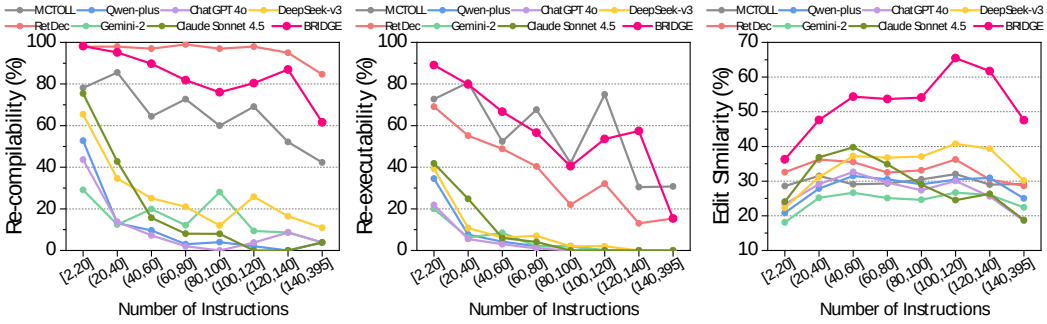
Figure 8: Distribution of input assembly lengths (measured by the number of instructions) in the HumanEval-Decompile-IR benchmarks.

ing of LLVM IR's specific semantics, including strict SSA rules and type correctness. To overcome these limitations, we provide LLMs with contextually relevant code examples to assist in LLVM IR reconstruction. Our results show that *general-purpose LLMs, when equipped with a suitable RAG module that retrieves relevant LLVM IR content, their reconstruction accuracy improves significantly. This enhancement allows them to recover both arithmetic logic and control-flow structures, even from highly optimized binaries.*

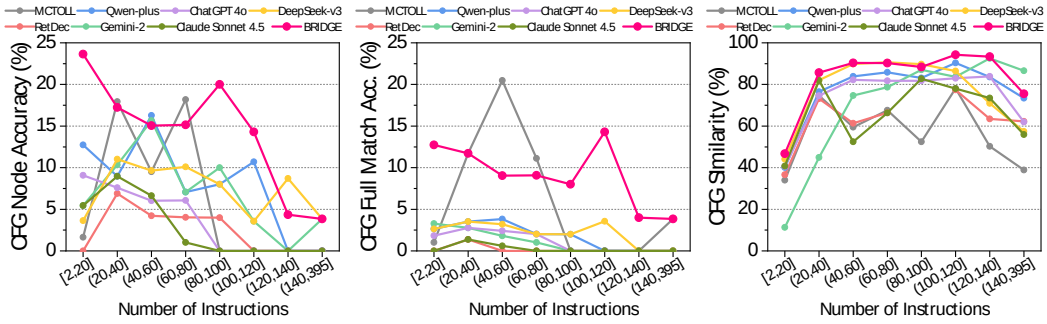
B Impact of Input Length on Decompilation Performance

We also investigate the impact of assembly code length on the lifting performance of BRIDGE. Figure 8 illustrates the distribution of function lengths in the HumanEval-Decompile dataset, where the majority concentrate in the 40–60 (166 functions) and 20–40 (145 functions) instruction ranges. Figure 9 compares BRIDGE with seven baselines across six evaluation metrics: re-compilability, re-executability, edit similarity, node accuracy (which measures whether the node count matches the ground truth), CFG full-match accuracy, and CFG similarity, covering input lengths from 2 to 394 instructions (approximately 42 to 8,325 tokens). We can observe that general-purpose LLMs exhibit a sharp performance degradation as function complexity increases. In the range of [2, 20] instructions, general-purpose LLMs achieve an average re-executability of 31.5%. When the instruction count exceeds 60, re-executability drops below 5% for all models except DeepSeek-V3, which retains a marginally higher rate of 7.0%, making them ineffective for non-trivial functions. In contrast, BRIDGE significantly extends this effective window. It maintains robust functional correctness well into medium-length sequences, achieving 56.6% re-executability at [60, 80] instructions, whereas ChatGPT-4o exhibits the lowest re-executability among all general-purpose LLMs at 1.0%, demonstrating that structure-aware retrieval effectively mitigates the context erosion that typically hampers standard LLMs. For the longest sequences ([140, 395] instructions), we observe a divergence between structural recovery and functional precision. While BRIDGE maintains high *Re-compilability* (61.5%) and *CFG Similarity* (75.5%), far outperforming the best LLM baselines (4.2% and 62.0%, respectively), its *Re-executability* drops to 15.4%. This indicates that while our approach successfully preserves the global topology and syntactic validity of massive functions, the accumulation of minor probabilistic errors in specific logic statements eventually disrupts end-to-end execution. Notably, in the extreme long-tail ([140, 395] instructions), the rule-based MCTOLL surpasses BRIDGE in re-executability (30.8% vs. 15.4%), despite yielding significantly lower edit similarity (29.2% vs. 47.6%). The results show that for monolithic functions, the fixed logic of rule-based systems avoids the 'hallucination drift' that can affect LLMs. Although these complex functions are rare (about 4% of the HumanEval-Decompile), we can address this in future work by using a rule-based lifter like MCTOLL as a reference. By combining our approach with deterministic instruction translation, we can improve recovery ability for these long inputs.

Figure 10 presents the lifting performance of BRIDGE across varying optimization levels and input instruction counts. We can observe a decline in re-executability at higher optimization levels(from `-O1` to `-O3`), as input length increases. This trend indicates that aggressive transformations, such as inlining and loop restructuring, introduce functional complexities that challenge the BRIDGE in long-context scenarios. Augmenting the retrieval corpus with targeted `-O3` patterns can help mitigate this degradation. Regarding textual fidelity, the BRIDGE maintains a robust average edit similarity of 48.4%, peaking at 80.9% for `-O0`. However, this metric decreased in the long-tail regime (140-395 instructions, accounting for 4% of the HumanEval-Decompile). This indicates that while BRIDGE can reconstruct high-level semantics for standard-complexity functions, it remains a challenge for extremely long, highly optimized function bodies. Conversely, CFG similarity remains consistent across all optimization levels, demonstrating that our structure-aware retrieval mechanism successfully preserves the global control topology even under significant local instruction distortion.

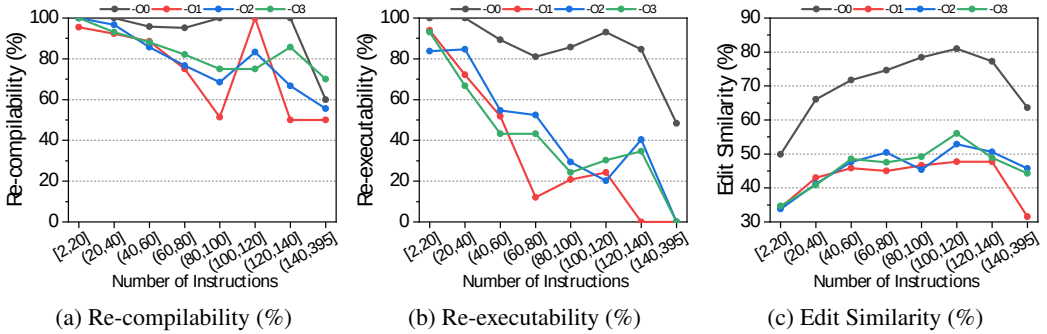


(a) Re-compilability (%) (b) Re-executability (%) (c) Edit Similarity (%)

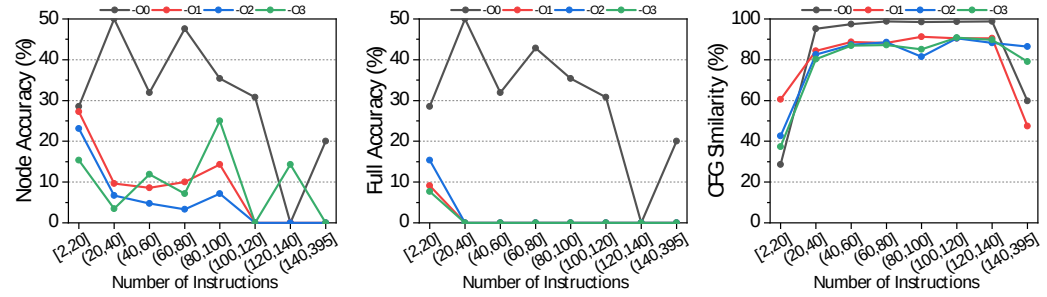


(d) Node Accuracy (%) (e) CFG Full Match Accuracy (%) (f) CFG Similarity (%)

Figure 9: BRIDGE lifting performance comparison across different input assembly code lengths of HumanEval-Decompile dataset (x86-64).



(a) Re-compilability (%) (b) Re-executability (%) (c) Edit Similarity (%)



(d) Node Accuracy (%) (e) CFG Full Match Accuracy (%) (f) CFG Similarity (%)

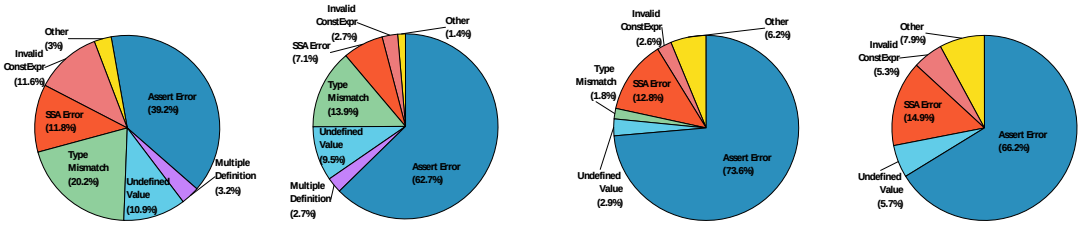
Figure 10: BRIDGE's performance across varying input assembly code lengths and optimization levels on the HumanEval-Decompile dataset (x86-64).

C General-purpose LLMs with Our RAG-enhanced Framework

Table 6 evaluates the portability of our approach by applying the structure-aware RAG and iterative feedback (IF) mechanisms to four commercial Large Language Models. The results demonstrate that our framework serves as a universal performance multiplier, yielding dramatic gains across all baselines. Overall, our framework consistently improves baseline performance, yielding an aver-

	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
Opt. Level	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Qwen-plus	25.7	9.5	8.7	6.8	12.7	16.2	9.4	4.1	3.4	6.8	37.5	23.6	25.6	25.4	28.0
Qwen-plus (w/ RAG and IF)	69.6	58.1	54.1	50	57.9	51.4	40.5	38.5	37.8	42.1	54.4	40.6	39.8	40.9	43.9
Gemini-2	33.7	10.8	12.2	10.1	16.7	14.2	4.1	4.7	2.0	6.3	33.9	23.5	21.2	20.7	24.8
Gemini-2 (w/ RAG and IF)	72.9	59.5	58.8	55.4	61.7	56.8	43.2	39.9	39.2	44.8	59.7	43.3	42.1	42.6	46.9
ChatGPT 4o	20.3	7.4	9.5	8.8	11.8	11.5	2.0	2.0	2.0	4.4	41.8	23.5	25.0	24.9	28.8
ChatGPT 4o (w/ RAG and IF)	57.4	48.6	45.3	43.2	48.6	42.6	37.2	35.8	35.8	38.2	50.4	41.1	41.8	41.3	43.6
Claude Sonnet 4.5	25.0	27.4	19.5	22.3	18.3	18.3	14.0	8.5	6.7	11.9	39.9	28.7	29.7	29.9	32.0
Claude Sonnet 4.5 (w/ RAG and IF)	70.1	60.4	54.3	53.0	59.5	50.6	47.6	37.8	35.4	42.8	49.7	40.5	41.5	49.5	45.3
BRIDGE	97.2	86.4	86.5	83.8	88.5	83.1	63.4	58.4	54.5	64.7	70.6	48.3	45.6	41.1	51.4

Table 6: Performance of four general-purpose LLMs augmented with our structure-aware RAG database and iterative feedback (denoted as w/ RAG and IF) on the HumanEval-Decompile dataset (x86-64).



(a) BRIDGE one shot inference (b) BRIDGE with one-pass feedback refinement (c) BRIDGE with three-pass feedback refinement. (d) BRIDGE with five-pass feedback refinement.

Figure 11: Distribution of error types in BRIDGE under one-shot inference compared to one, three, and five iterations of feedback-directed refinement on the HumanEval-Decompile dataset (x86-64).

age gain of 42.2% in re-compilability and 34.6% in re-executability across all models, and thus acts as a strong performance multiplier. Among them, Gemini-2 observes its average re-executability increase from 6.3% to 44.8%, corresponding to the largest absolute improvement (38.5%) among all evaluated models. This resilience is particularly evident under aggressive optimization. The base ChatGPT-4o collapses to 2.0% re-executability at `-O3`, the augmented version maintains a robust 35.8%. A key advantage of BRIDGE is its model-agnostic architecture, which enables seamless integration of emerging general-purpose LLMs. While this study focuses on a limited set of models, our framework is designed to readily leverage the advanced reasoning capabilities of future foundation models without requiring structural modifications.

D Error Types Considered in BRIDGE

We perform an error analysis to understand how feedback-directed refinement influences the failure modes of BRIDGE. Figure 11 breaks down the distribution of error types across inference iterations on the HumanEval-Decompile dataset on x86-64. In the one-shot inference setting (Figure 11a), the BRIDGE struggles significantly with syntactic and type-system constraints. Low-level violations such as `Type Mismatch` (20.2%), `Invalid ConstExpr` (11.6%), and `Undefined Value` (10.9%) constitute a major portion of failures, indicating that the model initially struggles to adhere to LLVM's strict typing rules.

Error Type	Description
Multiple Definition	This includes reusing local value names or duplicating basic block labels, violating uniqueness rules and causing IR parsing or compilation failures.
Undefined Value	Occurs when instructions (e.g., `br`, `load`) reference undefined SSA values or basic block labels, violating IR well-formedness and causing parsing or verification failures.
Type Mismatch	Incompatible operand types in LLVM IR instructions (e.g., non-pointer destinations in `store`, or non-function callees in `call`), violating strict typing rules and preventing IR verification.
SSA Violation	Malformed SSA construction in LLVM IR, especially incorrect phi nodes that violate SSA–CFG consistency and prevent IR verification.
Invalid Constant Expression	Occurs when operations such as `icmp`, `zext`, or `sext` are used inline as constant expressions inside other instructions (e.g., `select`). LLVM no longer supports these forms, requiring them to be separated as standalone instructions.

Table 7: Error types considered by BRIDGE.

However, as feedback refinement is applied (Figures 11b–d), we observe a distinct shift in the error distribution. Such as, the prevalence of `Type Mismatch` errors drops precipitously from 20.2% in one-shot to just 1.8% by the third pass. Similarly, `Undefined Value` errors decrease signifi-

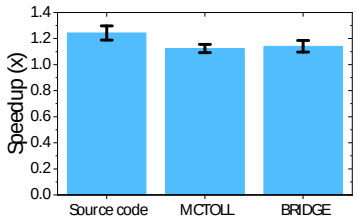


Figure 12: Performance speedup of source LLVM IR and reconstructed LLVM IR with -O3 optimization on the HumanEval-Decompile dataset (x86-64).

cantly. This confirms that the compiler feedback effectively guides the BRIDGE to resolve local syntax and type inconsistencies. As the code becomes syntactically valid (compilable), the relative proportion of `Assert Errors` indicates that while the feedback loop successfully produces *valid* IR, the remaining challenge is ensuring the *semantic correctness* of the logic against the ground truth.

E Optimization Recovery via -O3 Compilation

We also compare the performance speedup achieved by compiling the reconstructed LLVM IR code with -O3 across three sources: the ground truth LLVM IR, MCTOLL's decompilation results, and BRIDGE. The baseline for comparison is the execution time of the ground truth code compiled with -O0. We evaluate how effectively each decompilation method preserves the original code structure to enable subsequent compiler optimizations. Specifically, we select 286 benchmarks of the HumanEval-Decomple dataset for which both MCTOLL and BRIDGE produce re-executable decompiled code.

Figure 12 presents that BRIDGE achieves an average speedup of 1.20x, which is close to the ground truth (1.24x), demonstrating its ability to recover semantically and structurally accurate code that maintains the optimization potential of the original source. This performance gain is largely attributed to BRIDGE's superior reconstruction of the CFG, which preserves the program's execution flow structure (loops, branches, function calls) and enables downstream compiler optimizations like loop unrolling, dead code elimination, and register allocation to be more effectively applied. In contrast, MCTOLL delivers the poorest performance (1.11x), primarily due to its limited ability to recover precise control structures and semantic information. While MCTOLL produces functionally correct code, its structurally degraded output restricts the compiler's optimization opportunities, resulting in significantly slower execution compared to both the ground truth and BRIDGE.

F Prompts used in BRIDGE

The initial prompt for lifting machine code to canonical, pre-optimized LLVM IR.

You are an expert Compiler Engineer specializing in Binary Analysis and Reverse Engineering.
###Task. Lift the following stripped and optimized assembly code (generated via `llvm-objdump`) in x86-64/ARM64 back into *canonical, pre-optimized LLVM IR* that resembles IR produced by `clang -O0`.
###Requirements (Hard Constraints).

- **LLVM IR Syntax and SSA:** The output must be valid LLVM IR that parses with `llvm-as` and respects SSA form. Variable names should be meaningful and avoid numeric placeholders when possible (e.g., use `%i`, `%sum`, `%ptr` instead of `%0`, `%1`).
- **Preserve Explicit Control Flow:** Reconstruct branches, loops, and join points using explicit basic blocks and `br` instructions. Do not collapse control flow into `select` statements unless reflected in the assembly.
- **Preserve Variable Types:** Infer correct integer widths (`i8`, `i16`, `i32`, `i64`), pointer types, and sign/zero extension instructions according to the semantics of the input assembly.
- **No Compiler-Level Optimizations:** Do *not* introduce optimization behaviors such as loop unrolling, vectorization, instruction combining, tail calls, or SSA value reuse for unrelated semantics. Use `alloca` + `load`/`store` consistent with `-O0` style.

###Reference Examples. The following assembly–LLVM IR pairs are provided as guidance:

```
Assembly: {Asm1}
LLVM IR:  {LLVM IR1}
Assembly: {Asm2}
LLVM IR:  {LLVM IR2}
Assembly: {Asm3}
LLVM IR:  {LLVM IR3}
```

###Target Assembly.

```
{target_assembly_code}
```

###Output Format. Produce *only* the lifted LLVM IR code, including required type declarations. Do not include explanations, comments, markdown wrappers, or surrounding text.
###Output: (Place final LLVM IR for the target here.)

Iterative Refinement LLVM IR Prompt (Static Compilation Errors)

`### Task:` Refine the following LLVM IR so that it becomes *compilable and verifier-clean* by correcting all **static errors** detected during the compilation process (e.g., `llvm-as`, `opt -verify`, `llc`, or `clang`).
You are given:

- The initial inferred LLVM IR (`-O0`-style expected)
- The **static error log** produced during compilation or IR verification

Your goals:

1. Identify and correct all static compilation / verifier errors reported in the error log.
2. After applying fixes, ensure the IR remains valid SSA form and type-consistent.
3. Preserve the original semantics implied by the target assembly.
4. Output *only* the corrected LLVM IR — **no explanations, comments, or formatting outside LLVM IR**.

```
### Initial LLVM IR: {initial_llvm_ir}
### Static Compilation Error Log: {static_error_log}
### Target Assembly (Reference Only): {target_asm}
### Corrected LLVM IR Output:
```

Iterative Refinement LLVM IR Prompt (Runtime Error)

`### Task:` Refine the LLVM IR below so that it executes correctly, eliminating all **runtime errors** and passing the provided test cases.
You are given:

- The initial inferred LLVM IR (already free of static compilation errors)
- The **runtime error log**, produced by executing the compiled IR
- A set of **test cases** defining correct expected behavior
- The target assembly (reference only) to infer intended semantics

Your goals:

1. Use the runtime error log and test case failures to identify incorrect logic, missing edge conditions, incorrect control flow, or improper memory behavior in the IR.
2. Modify the IR to produce correct output for all test cases, fully eliminating the runtime failures.
3. Preserve the SSA (Static Single Assignment) structure and ensure type and pointer correctness.
4. Maintain semantic alignment with the original assembly code when inferring fixes.
5. Output **only the corrected LLVM IR code**. Do *not* include explanations, debugging notes, comments, or extra formatting.

```
### Initial LLVM IR: {initial_llvm_ir}
### Runtime Error Log: {runtime_error_log}
### Test Cases: {test_cases}
### Target Assembly (Reference Only): {target_asm}
### Corrected LLVM IR Output:
```