

ATLAS: A Multi-Agent Framework for Lifting Binary to Canonical Compiler IR

Xiaobao Zhu, Ni Zhang, Jie Ren*, *Member, IEEE*, Zhiqiang Li, Jie Zheng, Zhanyong Tang *Member, IEEE*, Zheng Wang *Member, IEEE*

Abstract—Lifting stripped, heavily optimized binaries to a canonical, pre-optimization LLVM intermediate representation (IR) is essential for downstream security analysis and legacy software migration when source code is unavailable. However, aggressive compiler optimizations rewrite control and data flow, remove structural metadata, and introduce type-erased low-level idioms, causing current lifters to produce malformed or semantically divergent IR.

We present ATLAS, a multi-agent framework designed to reconstruct formally valid and canonical LLVM IR from optimized disassembly. ATLAS combines (i) Structure-aware, region-level retrieval, which utilizes pseudo-probe attribution to align optimized instructions with canonical IR exemplars, and (ii) a collaborative refinement loop that closes the gap between probabilistic generation and deterministic compiler constraints. Concretely, a LIFT agent generates an initial IR draft conditioned on retrieved exemplars, an ADVISOR agent interprets LLVM verifier/compilation diagnostics and static checks to propose localized repair actions, and a VALIDATION agent applies the edits, re-runs toolchain verification and (when available) runtime checks, and feeds the resulting diagnostics back to the ADVISOR. The agents iterate until the IR is verifier-clean and behaviorally consistent, or a budget limit is reached.

On HumanEval-Decompile and MBPP, ATLAS outperforms seven baselines on lifting LLVM IR, achieving 91.8% re-executability and 89.4% CFG similarity across ARM, x86-64, and RISC-V binaries. Moreover, our evaluation demonstrates that ATLAS also outperforms the Binary to C to IR pipelines in pointer-flow precision and downstream analyzability. By recovering a well-formed, machine-centric representation, ATLAS enables practical cross-ISA recompilation and verifiable binary-only migration workflows.

Index Terms—Binary Lifting, LLVM IR, Large Language Models, Multi-Agent Systems, Retrieval-Augmented Generation.

I. INTRODUCTION

Recovering usable program representations from binaries is important for software security analysis, legacy-system maintenance, and migration across heterogeneous computing platforms. In these settings, source code may be unavailable, while the deployed executable remains available for inspection

This work was supported in part by the National Natural Science Foundation of China under Grant 62372281 and in part by the Fundamental Research Funds for the Central Universities of China under Grant GK202501010 (Corresponding author: Jie Ren.)

Xiaobao Zhu, Ni Zhang, Jie Ren and Zhiqiang Li are with the School of Artificial Intelligence and Computer Science, Shaanxi Normal University, Xi'an, Shaanxi, China. (Xiaobao Zhu and Ni Zhang are co-first authors.)

Jie Zheng, Zhanyong Tang, with the School of Computer Science, Northwest University, Xian, China.

Zheng Wang is with the School of Computing, University of Leeds, LS2 9JT Leeds, U.K.

E-mail: renjie@snnu.edu.cn.

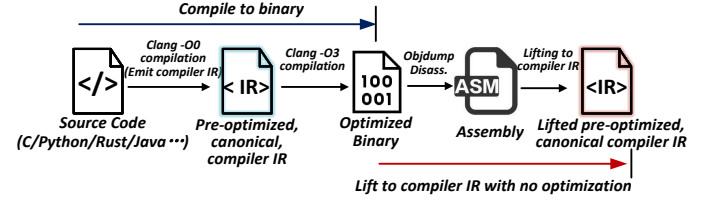


Figure 1. The compilation pipeline from source code to optimized binaries and the reverse lifting task, which targets canonical, pre-optimization compiler IR.

and testing. Binary recovery must therefore provide a representation that supports both reasoning about program behavior and subsequent transformation. Although much recent decompilation research targets human-readable source code [1], [2], downstream compiler workflows also require explicit control flow, consistent types, and well-defined memory operations.

Compiler intermediate representation (IR) provides a useful target for these workflows. LLVM IR can be checked by the LLVM verifier, consumed by program-analysis tools, and recompiled using different compiler backends. Figure 1 illustrates the lifting task considered in this paper: reconstructing canonical, pre-optimization LLVM IR from optimized binaries. We use *canonical* as an operational reference, defined by the unoptimized IR emitted under a fixed `clang -O0` toolchain configuration. This reference provides a consistent target for structural recovery; it does not imply that an optimized binary uniquely determines its original IR. The practical objective is to recover well-formed IR that preserves the binary’s behavior on the evaluated inputs and remains useful for analysis and recompilation.

This recovery task is difficult because compiler optimizations alter control and data flow, eliminate variables, and replace structured computations with architecture-specific instruction sequences. Rule-based lifters, including LLVM-MCTOLL [3] and RetDec [4], translate machine instructions and reconstruct program structure through deterministic rules and analysis. Their recovered representations can nevertheless differ substantially from compiler-generated IR. Large language models (LLMs) provide a complementary means of inferring code structure, but plausible output can still contain invalid types, inconsistent static single assignment (SSA) definitions, or incorrect computations. Successful lifting consequently requires both structural checks and behavioral evidence.

Our prior work, BRIDGE [5], combines control-flow-aware retrieval with iterative verification for binary-to-canonical-IR lifting. It uses pseudo-probe attribution to construct aligned

assembly-IR exemplars and incorporates static diagnostics and runtime feedback into refinement. ATLAS builds on this foundation and develops the repair and validation workflow further. Its focus is on localizing repairs, obtaining behavioral evidence when developer-written tests are unavailable, and assessing whether the resulting IR supports reuse across instruction set architectures (ISAs).

Three challenges motivate these extensions. **First, compiler diagnostics must be translated into appropriately scoped repairs.** A reported type error or malformed `phi` instruction can involve definitions, uses, and predecessor blocks beyond the reported line. Repair therefore requires identifying the relevant dependencies and enforcing their constraints together. Regenerating a function from an error log provides little control over which valid parts of the current candidate are changed. A more targeted workflow must connect each diagnostic to an affected IR region and a concrete repair action.

Second, behavioral feedback requires an executable interface and a usable reference. Curated benchmarks supply function signatures and unit tests, whereas binary-only settings may provide neither. Generating test inputs alone is insufficient: the system must infer how to call the function, construct a feasible harness, and obtain reference observations from the original executable. Incorrect interface assumptions can make test failures uninformative. The resulting observations must also be interpreted within their coverage, since passing a finite set of tests does not establish equivalence for all inputs.

Third, successful execution on the source architecture does not establish cross-ISA reusability. Recovered IR may retain assumptions about data layout, external interfaces, or machine-specific operations. Recompiling and testing it on another architecture therefore provides an additional assessment of its utility. This assessment complements structural similarity and same-architecture execution by examining whether recovered computations can be reused through a different compiler backend.

We present ATLAS, a multi-agent framework that organizes lifting around three explicit responsibilities: initial synthesis, repair planning, and patch validation. The LIFT AGENT uses CFG-aligned retrieval to generate an initial IR candidate. The ADVISOR AGENT interprets compiler, verifier, and runtime diagnostics, identifies the affected instructions and dependencies, and proposes localized repair actions. Its repair context includes predecessor constraints for `phi` nodes, def-use slices for inconsistent values, and available binary-interface information for external calls. The VALIDATION AGENT applies these actions and re-runs toolchain checks and available behavioral tests. Failures are returned with the updated candidate for further diagnosis, with refinement bounded by a maximum of five repair iterations. This organization makes the diagnostic evidence, proposed changes, and validation outcomes explicit at each step.

To support refinement without developer-written tests, ATLAS also constructs behavioral checks from the original binary. It combines static interface inference with observations from dynamic instrumentation, uses these inputs to propose candidate function signatures and test harnesses, and screens candidates through consistency checks and fuzzing-based exe-

cution. For accepted harnesses, it compares the original binary and the recompiled IR on the same inputs, including return values and observable memory effects. These differential observations provide concrete evidence for subsequent repairs without requiring source-level expected outputs.

We evaluate ATLAS on HumanEval-Decompile [1] and MBPP [6], covering x86-64, ARM64, and RISC-V binaries compiled at optimization levels `-O0` through `-O3`. The evaluation includes rule-based and general-purpose LLM baselines where supported, studies retrieval granularity, refinement, and test availability, and examines downstream SVF-based analysis [7] and cross-ISA migration. On HumanEval-Decompile, ATLAS achieves 94.5% re-executability for x86-64 inputs, averaged across optimization levels. In the x86-64-to-ARM64 migration experiment, it achieves 93.6% re-executability on the target architecture. These results characterize lifting quality and downstream utility under the evaluated configurations.

This paper makes the following contributions:

- We present ATLAS, a multi-agent framework that separates synthesis, repair planning, and validation for lifting optimized, stripped binaries to canonical LLVM IR.
- We support structure-aware retrieval across multiple ISAs using aligned assembly-IR exemplars and LoRA adaptation of the assembly encoder for ARM64 and RISC-V.
- We develop localized repair that maps diagnostics to affected IR slices and derives type, def-use, and control-flow constraints to guide targeted edits.
- We develop a behavioral validation workflow combining interface inference, dynamic observations, harness screening, and differential execution to guide repairs without developer-written tests.
- We evaluate ATLAS across three ISAs and four optimization levels, examining lifting quality, component effectiveness, downstream analysis, and cross-ISA migration.

II. MOTIVATION

To motivate our work, we present a task of lifting an `-O3` optimized binary back to its corresponding pre-optimized, canonical LLVM IR (`-O0`). *In this paper, canonical is an operational target: we define it as the unoptimized LLVM IR emitted by a fixed clang -O0 toolchain configuration.* Figure 2a presents the canonical, pre-optimized LLVM IR source code, 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`. Specifically, we first compile the LLVM IR with Clang (version 19) [8] using the `-O3` flag to generate an optimized binary. We then use `llvm-objdump` [8] to obtain the assembly instructions, which serve as input to different lifters for reconstructing the original LLVM IR.

Figure 2 compares the lifting results of six approaches for converting `-O3` optimized assembly code to pre-optimized, canonical LLVM IR (`-O0`). These include the LLVM-based decompiler RetDec [4], the Meta fine-tuned LLM Compiler FTD [9], and two typical general-purpose LLMs: ChatGPT-4o [10] and Gemini-3 [11]. Additionally, we explore two

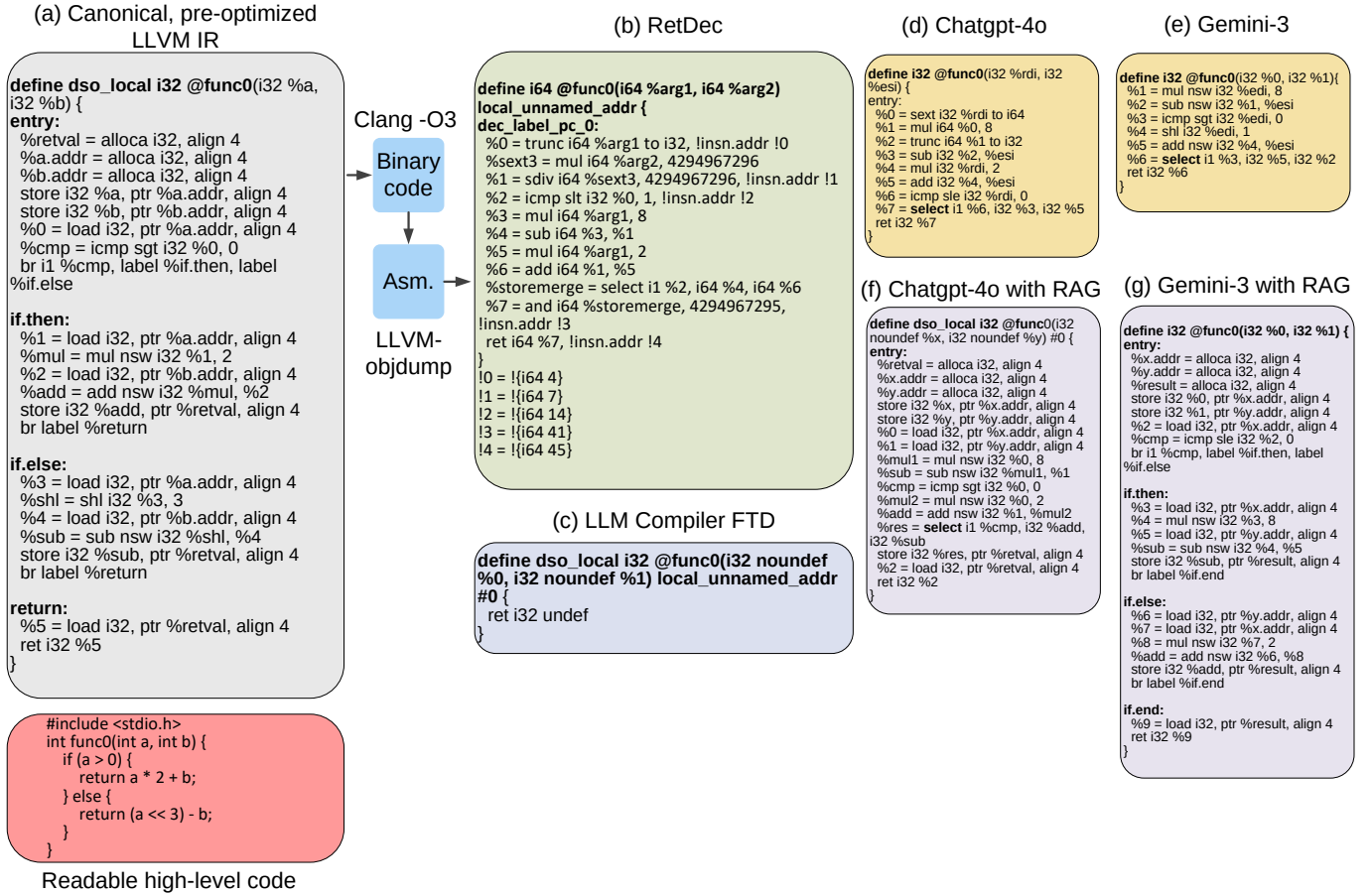


Figure 2. Lifting results of different approaches (b–g) compared with the ground-truth canonical, pre-optimized LLVM IR (a). The rule-based lifter RetDec distorts the original `if-else` control-flow structure. The general-purpose LLMs, especially Gemini-3 with RAG, better preserve the CFG and produce IR closer to the reference.

variants that integrate RAG with two general-purpose LLMs. RAG retrieves similar code segments from the ExeBench dataset [12] to construct contextual prompts. For the two general-purpose LLMs, we use the prompt: *{Convert the following llvm-objdump assembly (compiled with -O3) into canonical LLVM IR that matches the unoptimized clang -O0 IR style and passes llvm-as/opt -verify. Do not apply optimizations}*. Table I reports the lifting performance using four metrics:

- **Re-executable** evaluates whether the reconstructed LLVM IR executes all test cases correctly, ensuring functional equivalence to the original program.
- **Edit Similarity** [13] measures the similarity between the decompiled 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 Control Flow Graph from the assembly’s control flow.
- **PCI (Pointer Coverage Index)** measures how much pointer-related structure in the lifted IR can be recovered by SVF. Higher is better, indicating clearer pointer types/flows and more complete downstream analysis.

RetDec [4] produces compilable and test-passing IR, but the

Table I
COMPARISON OF DIFFERENT DECOMPILED METHODS IN EDIT SIMILARITY, TEST CASE PASS, AND BRANCH RECOVERY ON X86-64.

Method	Re-executable	Edit Sim. (%)	CFG Match.	PCI
RetDec	✓	38.2	✗	25
LLM Compiler FTD	✗	11.0	✗	0
ChatGPT-4o	✓	32.0	✗	18
ChatGPT-4o+RAG	✓	65.0	✗	32
Gemini-3	✓	26.0	✗	21
Gemini3+RAG	✓	82.5	✓	39

result can remain far from the canonical pre-optimization LLVM IR targeted in this work. In Figure 2, RetDec passes the available tests, yet achieves only 38.2% edit similarity, does not recover the original CFG exactly, and attains a low Pointer Coverage Index (PCI = 25). Concretely, the lifted IR replaces the original branch-based structure with a `select`-based merged computation and introduces low-level arithmetic normalization and masking, which obscures the original control-flow and data-flow organization. As a result, although the output is executable, it is structurally misaligned with the reference IR and substantially less suitable for downstream static analysis.

LLM Compiler FTD [9] is a fine-tuned model for reconstructing LLVM IR from assembly text. However, under the raw dis-

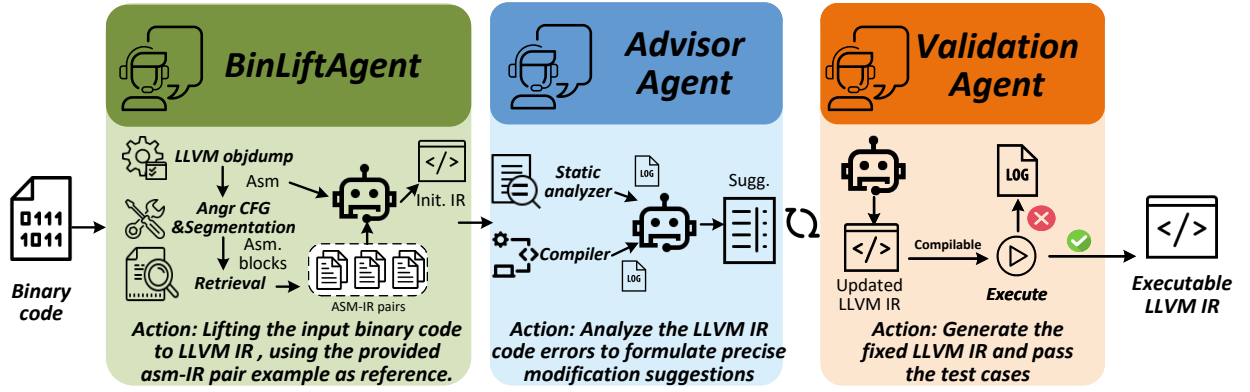


Figure 3. Overview of ATLAS

assembly setting studied in this paper, its outputs can exhibit substantial behavioral errors. In our motivating example, the model introduces `undef` and fails to recover key arithmetic operations and the conditional (`if-else`) structure implied by the assembly, leading to incorrect execution. A primary reason is *a mismatch in the input domain*. LLM Compiler FTD, as well as related approaches such as Forklift [14], is trained on compiler-emitted assembly (e.g., `clang -S`), which retains structured labels, symbol names, and other high-level cues. In contrast, raw-binary disassembly from `llvm-objdump` is typically stripped of symbols and exposes lower-level artifacts, including indirect control flow and ABI-specific details, which makes faithful lifting to canonical `-O0` IR significantly more challenging.

General-Purpose LLMs. We observe that both ChatGPT-4o (Figure 2d) and Gemini-3 (Figure 2e) have limitations when lifting raw disassembly to canonical, pre-optimized LLVM IR. First, even when the underlying arithmetic is correct and the outputs pass the available test cases, the generated IR often diverges from the target pre-optimization form, with edit similarity scores of 32.0% and 26.0%, respectively. In particular, ChatGPT-4o replaces the original branch-based `if-else` structure with a `select`-based formulation, collapsing the CFG into a single block. Although this if-converted IR remains executable, it no longer matches the canonical `-O0` control-flow structure and is therefore less suitable for downstream CFG-sensitive analyses. This loss of structural fidelity is also reflected in weak downstream analyzability, with low PCI values of 18 for ChatGPT-4o and 21 for Gemini-3.

General-Purpose LLMs with Our RAG Scheme. To evaluate the impact of retrieval augmentation on LLVM IR reconstruction, we integrate our control-flow-aware RAG module with both general-purpose LLMs. The retrieved context provides semantically related assembly-IR exemplars, allowing the models to ground generation in canonical compiler patterns rather than relying solely on open-ended sequence generation. As shown in Figure 2(f)–(g), RAG substantially improves both structural fidelity and downstream analyzability. For ChatGPT-4o, RAG raises the edit similarity from 32.0% to 65.0% and improves the PCI from 18 to 32. Although the generated IR still uses a `select`-based formulation rather than fully recovering the original branch-based CFG, the retrieved exemplars guide the model toward a more canonical stack-

based `-O0`-style structure, with explicit local allocations and clearer memory accesses. This results in IR that is closer to the reference and more suitable for downstream analysis. The improvement is even more pronounced for Gemini-3. After integrating RAG, the model produces compilable and re-executable LLVM IR that recovers the original `if-else` control-flow structure exactly. This increases edit similarity from 26.0% to 82.5%, enables exact CFG matching, and raises PCI from 21 to 39. These results show that retrieval not only reduces structural hallucinations, but also provides the missing control-flow and memory-structure cues needed for reconstructing canonical, analysis-friendly IR from optimized binary disassembly.

Insight. Lifting optimized binary code back into canonical, pre-optimized LLVM 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. The motivation reveals that binary-to-LLVM-IR lifting cannot be evaluated solely by compilability or test passing. Rule-based lifters and general-purpose LLMs may produce executable outputs, yet the recovered IR can still deviate substantially from the canonical pre-optimization form, fail to preserve the original control-flow organization, and expose weak pointer/value-flow structure to downstream analyzers. This gap is especially visible in PCI and CFG-based metrics, which show that behaviorally plausible IR may still be structurally unsuitable for compiler and security analyses. At the same time, retrieval-augmented generation substantially improves both structural fidelity and analyzability, suggesting that the missing information is not only lexical but also control- and memory-structural. These observations motivate ATLAS: a lifting framework that combines control-flow-aware retrieval with verification-guided refinement to recover LLVM IR that is not only executable, but also canonical and analysis-friendly.

III. OVERVIEW

Figure 3 overviews ATLAS, a multi-agent framework that lifts optimized binaries to LLVM IR in a canonical pre-optimization form (i.e., the unoptimized IR produced by a fixed `clang -O0` toolchain configuration). The pipeline comprises three agents: LIFT, ADVISOR, and VALIDATION, each with a distinct and non-overlapping responsibility.

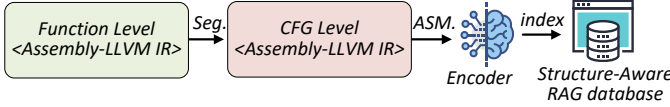


Figure 4. The process of building our Structure-Aware RAG database.

Given a binary, the LIFT AGENT first disassembles it with `llvm-objdump` and recovers a coarse function-level CFG using `angr` [15]. It applies a CFG-aligned segmentation procedure to partition the disassembly into control-structure regions (Section V-A). Each region is encoded as a retrieval query against a structure-aware RAG database of aligned (*assembly*, *LLVM IR*) exemplars (Section IV). Conditioned on the retrieved exemplars, the agent invokes an LLM to produce an initial LLVM IR draft, which is passed to the refinement loop.

The ADVISOR AGENT checks the draft with the LLVM toolchain (`llvm-as, opt -verify`) and a dedicated static analyzer implemented as an LLVM pass. It converts diagnostics into concrete repair actions, including type corrections, control-flow consistency fixes, and merge/value-definition repairs.

The VALIDATION AGENT applies the proposed edits and re-runs compilation and validation. When test inputs are available, it also executes the recompiled artifact against assertion-based tests. Any new compiler errors or runtime failures are fed back to the ADVISOR AGENT. This loop continues until the IR passes LLVM verification and functional checks, or the iteration budget is exhausted.

IV. STRUCTURE-AWARE RAG DATABASE

Figure 4 overviews the construction of a structure-aware RAG database used to lift optimized binaries to *canonical* LLVM IR.

A. Offline RAG Database Preparation

We build an offline retrieval database from ExeBench [12] to provide aligned exemplars for lifting machine code into pre-optimized, canonical LLVM IR.

Deduplication and leakage control. To improve diversity and avoid leakage, we (i) remove near-duplicate functions by computing normalized token-level edit similarity on canonical $-O0$ LLVM IR and filtering pairs with similarity > 0.95 , and (ii) remove any samples that overlap with our evaluation set.

Generating aligned assembly-IR pairs. For each retained program, we compile it to canonical LLVM IR using `clang -O0 -emit-llvm -S` as the ground-truth target. We then compile the same source with Clang at four optimization levels ($-O0$ to $-O3$) to obtain binaries, and disassemble each binary with `llvm-objdump` to produce the corresponding assembly. After removing samples that fail to compile or disassemble, the final corpus contains 69,253 unique function-level canonical IR targets. Pairing each target with assembly from four optimization levels yields 277,012 function-level (*(assembly, LLVM IR)*) pairs.

B. Control-Flow-Aware Data Structuring

Function-level pairs provide broad semantic coverage but bundle multiple control constructs into a single retrieval unit, making it difficult to isolate the structural patterns most relevant to a specific query region. We therefore construct *region-level* entries that pair assembly snippets with structurally coherent IR slices at the granularity of individual control constructs (e.g., branch regions, loop bodies), providing a tighter alignment between retrieved exemplars and the target region at inference time. As illustrated in Figure 5. Each region-level entry stores a triple $\langle asm^{-Ox}, ir^{-O0}, id \rangle$ where ir^{-O0} is a canonical $-O0$ IR slice and asm^{-Ox} is the corresponding assembly snippet at optimization level $-Ox$.

(1) Identifying Structure region boundaries. We first identify boundaries of high-level control constructs in the source program. Using an AST-based instrumentation script, we insert a lightweight marker `PROBE(x)` around constructs such as `if` conditionals and `for` loops, and insert lightweight anchors `PROBE(x)` to mark region start/end points. These anchors are preserved when generating the canonical $-O0$ LLVM IR (Figure 5a-b) and define structurally coherent regions for retrieval.

(2) Fine-grained IR-to-assembly attribution with pseudo-probes. To propagate these semantic partitions to the final machine code, we leverage LLVM pseudo-probe instrumentation [16]. We inject lightweight pseudo-probes at IR basic-block entries so that probe identifiers are carried through optimization and emitted into the optimized binary. Pseudo-probes enable robust attribution of machine-instruction ranges back to their originating IR blocks, even when backend optimizations reorder instructions or rearrange basic blocks (Figure 5c). Consequently, we can precisely locate the high-level semantic boundaries in the optimized binary even in the presence of instruction scheduling or block reordering.

(3) Constructing structure-aware database entries. Finally, for each anchored region, we extract (i) the canonical $-O0$ IR slice delimited by `PROBE` anchors and (ii) the corresponding region-complete assembly snippet obtained from pseudo-probe attribution. These region-level pairs form the retrieval units used by ATLAS at inference time. Across the corpus, this process yields 285K region-level entries.

C. Embedding and Indexing

We encode each segmented assembly region into a d -dimensional embedding using Nova-1.3B [17]. Embeddings are stored as `float32` and indexed with FAISS using an exact `IndexFlatL2` wrapped by `IndexIDMap` [18], enabling direct lookup of the paired $\langle assembly, LLVM IR \rangle$ entry via a unique region identifier. Nova is chosen because it is trained to represent assembly patterns across optimization settings and can map diverse low-level realizations to similar semantics. While the base encoder is strongest on x86-64, we adapt it to ARM64 and RISC-V with LoRA [19] (rank $r=8$) using aligned region-level assembly pairs from our corpus, which improves cross-ISA retrieval robustness.

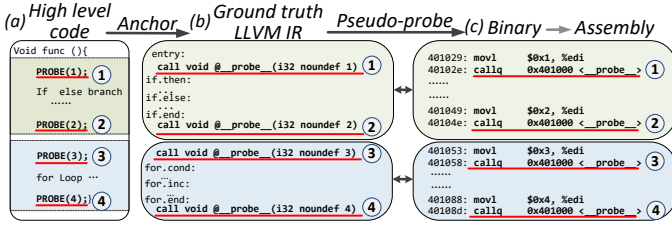


Figure 5. Workflow for constructing control-flow-aware LLVM IR and assembly code pairs.

V. LIFT AGENT

The LIFT AGENT produces an initial LLVM IR draft from an input binary. It first disassembles the binary using `llvm-objdump`. It then performs coarse control-flow recovery with `angr` [15] and applies a CFG-aligned segmentation procedure to partition the disassembly into control-structure regions (Section V-A). Each region is used as a retrieval query to the previously mentioned structure-aware RAG database (Section IV), which returns the top- k aligned $\langle \text{assembly}, \text{LLVM IR} \rangle$ exemplars. Conditioned on the query region and the retrieved exemplars, the agent constructs an in-context prompt and invokes the LLM to generate an initial LLVM IR candidate, which is then passed to the ADVISOR AGENT for tool-guided refinement.

A. CFG-aligned Segmentation

Figure 6 illustrates the segmentation procedure implemented as a tool that can be called by the LIFT AGENT. Given a binary, we use `angr` [15] to recover a function-level CFG and export it in DOT format, where nodes correspond to basic blocks labeled by their start addresses. We filter external stubs and unresolved callees (e.g., PLT entries) and segment only the internal control flow of the target function.

We identify candidate region headers at control-flow divergence points, defined as blocks with out-degree ≥ 2 . Starting from each header, we follow successor edges along paths with out-degree ≤ 1 and aggregate visited blocks until reaching another divergence point or a function exit. To handle loops, we detect back-edges during traversal and group the blocks on the cycle into a loop region. For aggressively optimized code where unrolling yields multiple adjacent loop-like fragments, we merge contiguous loop fragments when they share a common header and rejoin pattern.

Finally, we merge regions corresponding to alternative branches when they converge at a common join block that is not a function exit. The resulting regions provide stable retrieval units that capture local control structure (e.g., branch and loop regions) while remaining robust to CFG noise. When CFG recovery is incomplete, we conservatively fall back to linear address-range regions derived from recovered basic blocks.

B. Control-flow-aware Retrieval

Given a CFG-segmented assembly fragment, the LIFT AGENT computes a dense embedding using Nova-1.3B [17] and performs nearest-neighbor search over the FAISS index

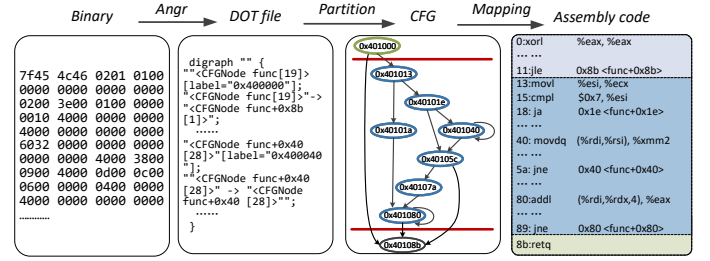


Figure 6. CFG-Guided binary segmentation.

Table II
ERROR TYPES DETECTED BY ATLAS 'S STATIC ANALYSIS DURING LLVM IR VERIFICATION.

Error Type	Description
AssertError	This indicates that the function has failed its test cases within the expected time budget, reflecting a functional or behavioral error.
FuncNameError	Arises when a function attempts to call an external function that is undeclared, or when a mismatch or conflict occurs in an external function's declaration.
PhiError	A common SSA violation, typically caused by incorrect phi instruction usage. This includes referencing values not from immediate predecessor blocks or using operands with inconsistent types.
Definition Conflict	Represents conflicts in variable definitions. This category encompasses errors such as TypeMismatch, UndefinedValueError, and MultiDefineError.

to retrieve the top- k most similar exemplars. Each retrieved exemplar provides an aligned $\langle \text{assembly}, \text{LLVM IR} \rangle$ pair, which is used as in-context evidence for IR synthesis. To quantify retrieval quality, we define retrieval accuracy as the fraction of queries for which at least one of the top- k retrieved exemplars shares the same canonical region label as the query in the database. Under this metric, Nova-1.3B achieves Top-1/Top-3/Top-5 accuracy of 95.92%, 98.61%, and 98.83%, respectively.

C. Initial IR Synthesis

To synthesize an initial IR draft, the LIFT AGENT constructs an in-context prompt consisting of (i) the query assembly region and (ii) the top- k retrieved exemplar pairs $\langle \text{assembly}, \text{LLVM IR} \rangle$, ordered by retrieval rank. The prompt is subject to a fixed context budget. To fit within the budget while preserving the target evidence, the agent may drop the lowest-ranked exemplar, while keeping the full query region intact. The final prompt instructs the LLM to emit a single-function LLVM IR draft in a canonical, pre-optimized style consistent with the target toolchain configuration. This draft is then passed to the ADVISOR and VALIDATION agents for verification-guided refinement.

VI. ADVISOR AGENT

The ADVISOR AGENT is responsible for error localization and repair planning given an initial lifted LLVM IR candidate. It aggregates diagnostics from the LLVM toolchain (e.g., `llvm-as` and `opt -verify`) and, when available, failures reported by the runtime test suite. It first assigns each failure to an error category and then localizes the failure to a concrete

IR scope, such as a function, a basic block, or a small set of instructions. Concretely, it parses toolchain logs to extract the failing condition and referenced IR entities (SSA values, block labels, and type signatures), maps them to the corresponding IR lines, and constructs a minimal surrounding slice for context. We use failure categories to select specialized repair prompts and constraints, which reduces prompt noise and restricts edits to the minimal scope needed for the observed error, improving convergence and stability of the refinement loop. Based on the category and localized slice, the ADVISOR AGENT produces a structured set of repair actions, including type adjustments, CFG consistency updates, insertion of missing declarations, or targeted rewrites of incorrect computations. These actions are passed to the VALIDATION AGENT, which applies them and re-runs compilation and runtime checks.

- **AssertError (behavioral mismatch).** The IR passes LLVM validation but fails the assertion-based tests, indicating a semantic mismatch on the available inputs. We derive a lightweight semantic summary from the disassembly (control-flow skeleton and key data dependencies) and provide the failing test evidence as an explicit behavioral signal. Conditioned on this structural summary and the observed test failure, the ADVISOR AGENT prompts the LLM to propose targeted logic edits to the relevant IR slice.
- **FuncNameError (linkage/ABI mismatch).** This category covers unresolved external symbols, incorrect prototypes, and calling-convention mismatches. The agent extracts authoritative interface information from the binary (e.g., symbol table and relocation entries via `readelf`) and uses it to repair external declarations and call sites, ensuring that names, signatures, and calling conventions match the binary interface.
- **PhiError (merge inconsistency at control-flow joins).** These failures arise from invalid value merges at join points, such as missing incoming mappings, incompatible incoming types, or references to values not available along a predecessor path. The agent localizes the conflicting `phi` instructions, reconstructs the predecessor set for the join block, and provides explicit merge constraints (required incoming blocks and type consistency). The LLM is then prompted to repair the incoming mappings and types within the localized slice.
- **DefinitionConflict (type and def-use inconsistency).** These errors arise from conflicting definitions, type mismatches across uses, or inconsistent value propagation. The agent performs def-use slicing around the implicated values to collect local data-flow context from definitions to uses, and prompts the LLM to reconcile conflicting definitions and restore type consistency while preserving the surrounding computation.

VII. VALIDATION AGENT

The VALIDATION AGENT applies the ADVISOR AGENT’s repair actions to patch the initial LLVM IR draft and verifies the updated IR in an iterative loop with the ADVISOR AGENT.

Given a set of suggested edits, it first updates the IR and performs *compilation verification* using the LLVM toolchain (e.g., `llvm-as` and `opt -verify`). If verification or compilation fails, the agent records the diagnostics together with the patched IR and returns them to the ADVISOR AGENT for further localization and repair planning.

If compilation succeeds, the VALIDATION AGENT performs *functional verification* by executing the compiled artifact under an assertion-based test suite. In practice, the dominant remaining failures are behavioral mismatches rather than toolchain errors. For a failing test, the agent extracts the relevant assembly slice and its recovered CFG, and prompts an LLM to summarize the function logic in natural language based on the CFG and the original disassembly. This summary serves as a semantic reference for subsequent repairs. The agent then packages the failure evidence, including the failing input, assertion message, crash trace when applicable, and the generated logic description, and returns it with the current IR to guide the next repair step.

The ADVISOR and VALIDATION agents iterate until the IR passes toolchain verification and all available tests. To bound cost and ensure termination, we stop early on success and cap the loop at five repair iterations.

VIII. EXPERIMENTAL SETUP

A. Competitive Baselines

We compare ATLAS with seven baselines, including two rule-based lifters, RetDec [4] and MCTOLL [3], and five general-purpose LLMs: DeepSeek-V3.2 [20], ChatGPT-4o [10], Claude Sonnet 4.5 [21], Qwen-Plus [22], and Gemini-3 [11]. MCTOLL supports only x86-64. All LLM baselines are accessed through their API interfaces. We evaluate each API-based baseline five times with the same prompt and report the average across runs.

We do not include LLM Compiler FTD [9] or Forklift [14], because they target a different input setting: compiler-emitted assembly text (e.g., `clang -S`) rather than raw binary disassembly. Our task starts from `llvm-objdump` output extracted from optimized binaries, which differs substantially from compiler-emitted assembly due to stripped symbols, indirect control flow, and relocation-related artifacts.

B. Evaluation Datasets

We evaluate ATLAS and all baselines on HumanEval-Decompile and MBPP [6]. HumanEval-Decompile [1] is a benchmark originally designed for binary-to-C decompilation and contains 164 programs with executable test cases. We adapt it to our setting by compiling the original C implementations into canonical LLVM IR.

We also use MBPP, a standard benchmark for code generation, and select 500 programs (IDs 11–510) that can be consistently compiled and executed in our pipeline. For both HumanEval-Decompile and MBPP, we first generate ground-truth canonical LLVM IR using a fixed Clang 19 -O0 configuration, and then compile the same programs into binaries at four optimization levels (-O0 to -O3). We extract assembly using `llvm-objdump` and use it as input to the lifting task.

Table III

COMPARISON OF BINARY-TO-CANONICAL LLVM IR LIFTING PERFORMANCE FOR ATLAS, MCTOLL, RETDEC, AND FIVE GENERAL-PURPOSE LLMs ON HUMANEVAL-DECOMPILE AND MBPP, ACROSS THREE ISAs AND FOUR OPTIMIZATION LEVELS (−O0 TO −O3), MEASURED BY RE-COMPILABILITY, RE-EXECUTABILITY, AND EDIT SIMILARITY. MCTOLL SUPPORTS ONLY X86-64, AND RETDEC SUPPORTS X86-64 AND ARM64. GREEN CELLS INDICATE THE BEST RESULT, AND BLUE CELLS INDICATE THE SECOND-BEST WITHIN EACH ROW AND PLATFORM GROUP.

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	23.6	25.6	25.4	28.0
	Gemini-3	42.7	46.3	45.1	48.8	45.7	33.5	30.5	30.5	26.2	30.2	30.7	28.0	26.7	27.8	28.3
	ChatGPT-4o	20.3	7.4	9.5	8.8	11.5	11.5	2.0	2.0	2.0	4.4	41.8	23.5	25.0	24.9	28.8
	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
	DeepSeek-v3.2	30.4	35.8	25.0	22.3	28.4	14.2	9.4	7.4	9.5	10.1	44.9	30.5	30.7	31.0	34.3
	ATLAS	100.0	99.4	98.8	98.8	99.2	96.3	95.1	92.7	93.9	94.5	64.3	47.0	46.6	47.7	51.4
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	22.7	23.9	24.2	26.1
	Gemini-3	39.6	43.4	41.2	41	41.3	36.0	25.8	27.8	26.8	29.1	34.1	28.6	26.8	24.1	28.4
	ChatGPT-4o	28.0	20.0	18.0	19.4	21.4	14.2	7.6	7.6	7.8	9.3	38.5	21.7	22.4	22.9	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	27.1	28.6	28.6	31.3
	DeepSeek-v3.2	41.0	35.2	32.4	31.0	34.9	25.0	17.6	14.2	15.6	18.1	40.8	25.5	26.5	26.6	29.9
	ATLAS	98.8	98.8	98.2	98.8	98.7	96.2	95.0	94.6	95.8	95.4	61.7	46.8	46.3	46.0	50.2
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-3	29.5	43.9	45.1	46.3	41.1	23.2	35.4	34.1	23.2	29.0	35.0	32.7	31.3	30.4	32.4
	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-v3.2	27.4	42.6	34.0	35.5	34.6	13.4	10.5	7.4	8.6	10.0	41.9	27.7	28.3	28.2	31.6
	ATLAS	98.2	99.4	98.1	99.4	98.8	94.5	95.1	92.6	94.4	94.2	60.8	39.0	38.6	38.5	44.3
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-3	31.4	40.0	39.2	36.6	36.8	25.0	30.6	29.8	28.6	28.5	39.0	29.5	29.8	29.4	31.9
	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-v3.2	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
	ATLAS	96.4	97.6	96.1	97.3	96.6	92.6	93.7	90.4	91.4	92.0	58.1	37.4	37.5	37.7	42.8
RISC-V		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	Qwen-plus	42.1	25.9	29.0	32.1	32.3	5.5	0.6	1.9	1.9	2.4	34.1	20.8	20.0	20.7	23.9
	Gemini-3	52.4	55.6	66.1	71.6	61.4	36.0	22.2	25.3	26.5	27.5	42.0	35.2	35.2	35.1	36.9
	ChatGPT-4o	20.1	10.5	7.4	8.0	11.5	6.1	1.9	1.9	1.9	2.9	28.1	22.3	22.6	22.6	23.9
	Claude Sonnet 4.5	17.7	25.9	26.5	24.7	23.7	8.5	9.9	11.7	8.6	9.7	28.8	22.9	22.7	22.6	24.3
	DeepSeek-v3.2	56.7	48.8	43.8	53.7	50.8	12.8	9.9	8.0	12.3	10.8	40.6	28.5	28.3	28.4	31.5
	ATLAS	99.4	97.5	96.9	97.5	97.8	88.4	79.0	77.8	77.2	80.6	63.9	48.1	48.0	47.2	51.8
MBPP	Qwen-plus	52.8	32.2	33.5	31.4	37.6	9.4	2.7	3.3	3.3	4.7	29.8	19.8	19.6	19.6	22.2
	Gemini-3	64.6	69.8	72.7	71.8	69.7	41.8	33.1	30.1	29.2	33.8	39.2	33.2	33.2	33.2	34.7
	ChatGPT-4o	28.8	17.1	20.8	18.0	21.2	9.8	6.1	7.6	6.1	7.4	28.0	20.9	20.6	21.2	22.6
	Claude Sonnet 4.5	37.6	43.1	39.6	41.8	40.5	21.2	18.0	16.7	17.1	18.3	28.6	20.4	20.9	20.8	22.7
	DeepSeek-v3.2	65.6	59.8	58.4	57.6	60.4	17.0	13.7	14.9	14.9	15.2	38.1	26.4	26.0	26.4	29.2
	ATLAS	99.2	98.8	98.2	99.4	98.9	91.8	88.8	89.8	89.8	90.0	46.3	40.0	39.7	39.0	41.3

C. Metrics

We use five metrics to evaluate lifting quality: re-compilability, re-executability, edit similarity, CFG full-match accuracy, and CFG similarity. These five metrics serve as the primary evaluation measures for lifting performance [23], [1], [2], capturing toolchain validity, behavioral correctness, and structural fidelity to the canonical reference IR. We also evaluate *analysis-friendliness* and *migration ability* on lifted IR separately in Section X and Section XI.

Re-compilability measures the percentage of lifted LLVM IR functions that can be successfully assembled and compiled without errors.

Re-executability measures the percentage of lifted functions that not only compile successfully but also pass the associated test cases, reflecting behavioral consistency with

the reference program under the available inputs.

Edit similarity [13] measures the similarity between the lifted LLVM IR and the ground-truth canonical IR using normalized edit distance. Higher values indicate closer correspondence to the reference IR.

CFG full-match accuracy measures the percentage of functions whose reconstructed control-flow graphs are exactly identical to the reference CFG. We compute this metric using `networkx.isomorphic(G1, G2)` [24], which tests whether the two graphs are isomorphic under an edge-preserving one-to-one node mapping.

CFG similarity quantifies structural similarity between the lifted and reference CFGs using the Weisfeiler–Lehman graph kernel [25]. Higher values indicate closer agreement in graph structure even when the CFGs are not exactly identical.

IX. EXPERIMENTAL RESULTS

A. Overall Performance

Table III summarizes the overall lifting performance of ATLAS and seven baselines across two benchmarks, three ISAs, and four compiler optimization levels.

We can observe that ATLAS achieves the highest re-executability in nearly all reported settings. On HumanEval-Decompile, it reaches 94.5% on x86-64, 94.2% on ARM64, and 80.6% on RISC-V (averaged over $-O0-O3$). The strongest non-ATLAS baselines remain below 30% on ARM64 and RISC-V, indicating that direct lifting with retrieval and refinement is substantially more effective at preserving executable semantics across architectures than open-loop generation.

Across both benchmarks, we observe a clear separation between re-compilability and re-executability. Rule-based lifters, especially RetDec, often achieve high re-compilability (e.g., 92.5% on x86-64 and 92.8% on ARM64 for HumanEval-Decompile), yet their re-executability is much lower (45.3% on x86-64 and 0.5% on ARM64). A similar gap appears on MBPP, where RetDec remains highly compilable (95.4% on x86-64 and 94.0% on ARM64) but achieves limited re-executability (42.7% and 0.4%). This pattern shows that producing verifier-accepted IR is comparatively easy, while preserving behavior from optimized binaries is the dominant challenge.

ATLAS also produces IR that is closer to the fixed canonical $-O0$ reference. On HumanEval-Decompile, its average edit similarity is 51.0% (x86-64), 44.3% (ARM64), and 51.8% (RISC-V), exceeding all baselines in the corresponding settings. The same trend holds on MBPP, where ATLAS improves edit similarity while achieving substantially higher re-executability. Taken together, these results suggest that improved executability is not achieved by unconstrained rewriting, but by converging toward a representation closer to the canonical target.

In detail, on x86-64, ATLAS achieves the best performance on all three metrics across both HumanEval-Decompile and MBPP. On ARM64, it again outperforms RetDec on all metrics for both benchmarks. This shows that retrieval-augmented lifting with refinement improves both semantic preservation and syntactic correctness. On RISC-V, where no rule-based baselines are available, ATLAS delivers the largest margin over all general-purpose LLMs, reaching 97.8% re-compilability and 80.6% re-executability on HumanEval-Decompile. These gains indicate that the framework generalizes well beyond x86-style control-flow and instruction patterns.

The optimization-level breakdown shows that ATLAS is also robust under aggressive compiler optimization. On HumanEval-Decompile, the re-compilability of ATLAS remains above 98% both on x86-64 and on ARM64 from $-O0$ to $-O3$, while re-executability remains comparatively stable. On MBPP, ATLAS also maintains stable re-executability across four optimization levels, consistently outperforming all baselines.

Figure 7 evaluates CFG reconstruction using two metrics: full-match CFG accuracy (graph isomorphism) and CFG sim-

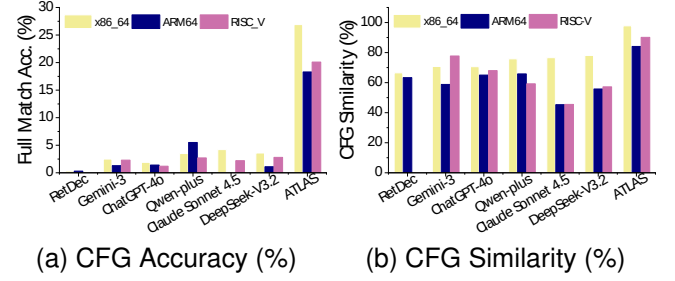


Figure 7. LLVM IR CFG reconstruction performance on the HumanEval-Decompile datasets across x86-64, ARM64 and RISC-V.

ilarity (WL-kernel similarity, normalized to $[0, 100]$). Exact CFG recovery is challenging under optimization because transformations such as if-conversion, block merging/splitting, and indirect control transfers can substantially reshape the graph, and accordingly all baselines achieve low full-match accuracy (below 6% on x86-64, ARM64, and RISC-V). In contrast, ATLAS attains 26.9%, 18.3%, and 20.1% full-match accuracy on the three ISAs, indicating a markedly stronger ability to recover complete control-flow structure rather than only partial fragments. CFG similarity shows the same pattern: ATLAS reaches 97.8, 84.1, and 90.2, consistently outperforming all baselines and demonstrating that its reconstructed CFGs remain close to the reference even when exact isomorphism is not achieved. The RISC-V results are particularly informative: although full-match accuracy is lower than on x86-64/ARM64, CFG similarity remains high, suggesting that ATLAS recovers most high-level control-flow organization while some fine-grained details remain difficult under aggressive optimization. Overall, these results support the effectiveness of ATLAS's region-level retrieval and verification-guided refinement in recovering compiler-meaningful control-flow structure across architectures.

B. Multi-Level RAG in Lift Agent

To study how retrieval granularity affects LLVM IR reconstruction, we compare three RAG configurations that retrieve exemplars at different code granularities: basic-block level, control-flow (region) level, and function level. Basic-block retrieval emphasizes local instruction patterns, control-flow retrieval captures branch and loop structure, and function-level retrieval provides the broadest semantic context. To isolate the effect of retrieval granularity, we disable the iterative refinement loop and perform a single inference pass for each setting.

Table IV shows that **control-flow-level retrieval provides the strongest overall balance** across all three metrics. It achieves the highest average re-compilability (87.8%), the highest average re-executability (56.9%), and the highest average edit similarity (51.0%). Compared with basic-block retrieval, control-flow retrieval improves average re-compilability by 19.4 points (68.4%→87.8%) and average re-executability by 7.7 points (49.2%→56.9%), indicating that conditioning on region-level structure is important for generating verifier-clean IR and preserving executable semantics. The advantage is most visible under heavier optimizations, where

Table IV
INITIAL LLVM IR QUALITY PRODUCED BY THE LIFT AGENT WITH RAG UNDER DIFFERENT RETRIEVAL GRANULARITIES ON HUMAN-EVAL-DECOMPILE (x86-64).

Opt. Level	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Basic Block level	85.5	71.6	63.4	53.2	68.4	66.2	48.5	41.9	40.2	49.2	58.8	40.5	40.2	39.8	44.8
Control flow level	90.2	86.6	84.1	90.2	87.8	67.7	50.6	53.7	55.5	56.9	63.8	46.6	46.2	47.3	51.0
Function level	91.2	83.6	82.4	79.4	84.1	70.6	52.9	52.9	41.1	54.4	65.0	40.0	42.5	40.0	46.9

Table V
EFFECT OF THE ADVISOR-VALIDATION REFINEMENT LOOP ON HUMAN-EVAL-DECOMPILE (x86-64). WITHOUT REFINEMENT LOOP: LIFT-ONLY. WITH REFINEMENT LOOP: FULL ATLAS.

Opt. Level	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Without refinement loop	90.2	86.6	84.1	90.2	87.8	67.7	50.6	53.7	55.5	56.9	63.8	46.6	46.2	47.3	51.0
With refinement loop	100.0	99.4	98.8	98.8	99.2	96.3	95.1	92.7	93.9	94.5	64.3	47.0	46.6	47.7	51.4

basic-block retrieval degrades substantially at $-O3$ (53.2% re-compilability and 40.2% re-executability), while control-flow retrieval remains stable (90.2% re-compilability and 55.5% re-executability). This suggests that region-level exemplars provide structural cues that remain informative even when local instruction sequences are heavily rewritten by optimization.

Function-level retrieval improves on basic-block retrieval but does not outperform control-flow retrieval overall. While function-level retrieval achieves the best re-compilability at $-O0$ (91.2%) and the best re-executability at $-O0$ (70.6%), its performance degrades more at higher optimization levels, especially for re-executability at $-O3$ (41.1%). A key reason is that aggressive optimizations (e.g., inlining, if-conversion, and code motion) substantially alter whole-function structure and instruction composition, making function-level similarity less discriminative for the specific region that determines behavior. As a result, whole-function retrieval is more likely to return exemplars that are globally similar but locally mismatched, which can bias generation toward an incorrect computation while still producing compilable IR. In contrast, control-flow-level retrieval matches exemplars at the same granularity used during inference and repair, providing region-aligned structural cues that remain more stable under optimization and improving semantic fidelity.

C. Ablation: Impact of the Advisor-Validation Refinement Loop

Table V isolates the contribution of the iterative ADVISOR-VALIDATION loop. The **Lift-only** setting reports the initial IR produced by the LIFT AGENT (single-pass synthesis with the same segmentation and retrieval), without invoking the ADVISOR or VALIDATION agents. The **full** setting runs ATLAS with the complete refinement loop.

Enabling the refinement loop improves both toolchain validity and functional correctness. Re-compilability increases from 87.8% to 99.2% (+11.4 points), while re-executability rises from 56.9% to 94.5% (+37.6 points). The gains are consistent across optimization levels, with particularly large improvements at $-O1$ – $-O3$, where optimized binaries introduce more structural and semantic ambiguity that is difficult to resolve in a single pass.

Edit similarity remains nearly unchanged (51.0% vs. 51.4%), suggesting that refinement primarily performs localized, diagnostic-driven repairs rather than unconstrained rewriting. Overall, this ablation shows that retrieval-grounded initial synthesis provides a strong starting point, but the tool-guided, closed-loop refinement is the main driver of end-to-end correctness.

X. DOWNSTREAM ANALYZABILITY

A key motivation for lifting binaries to canonical LLVM IR is to support downstream compiler and security analyses when source code is unavailable. In many real-world settings, such as third-party software auditing, legacy-system maintenance, and binary-only migration, the deployed binary is the only trustworthy artifact. Lifting such binaries to LLVM IR enables the reuse of mature compiler and static-analysis infrastructures while remaining closer to deployed behavior than analyses performed on reconstructed high-level code. However, compilability alone does not imply that the lifted IR is suitable for downstream analysis. Rule-based lifters may produce executable IR that is structurally fragile or overly type-erased, while decompilation-to-C pipelines primarily optimize for readability and executability rather than analysis-oriented IR quality. Our goal is therefore not only to recover executable code, but to reconstruct pre-optimized, canonical LLVM IR that is verifier-valid, analyzable, and reusable for downstream workflows.

We evaluate whether the lifted IR can support downstream pointer and value-flow analysis using SVF. We compare ATLAS with a Binary $\rightarrow$ C $\rightarrow$ IR baseline, **LLM4Decompile+clang**, on HumanEval-Decompile. We report three metrics. **SVF completion rate** measures the fraction of functions on which SVF finishes successfully under a fixed configuration and resource budget. **Average alias set size** measures the mean size of SVF-computed alias sets, where smaller values indicate more precise pointer reasoning. **Pointer Coverage Index (PCI)** measures how much pointer-related structure in the lifted IR can be successfully recovered and tracked by SVF. Higher PCI indicates that the IR preserves clearer pointer types and memory flows, enabling SVF to construct richer pointer/value-flow graphs.

Table VI
SVF-BASED DOWNSTREAM ANALYZABILITY ON HUMANEVAL-DECOMPILE (x86-64).

Metric	LLM4Decompile+clang	ATLAS	RetDec	Description
SVF completion (%) $\uparrow$	81.7	85.5	82.5	Fraction of functions for which SVF finishes successfully under a fixed configuration and resource budget. Higher values indicate that the lifted IR is more analysis-ready.
Avg. alias set size $\downarrow$	1.05	1.00	1.03	Mean size of SVF-computed alias sets. Smaller values indicate more precise pointer reasoning.
Pointer Coverage Index (PCI) $\uparrow$	80.4	132.9	107.1	Normalized measure of how much pointer and value-flow structure in the lifted IR can be recovered by SVF. Higher values indicate richer downstream analyzability.

Table VII
CROSS-ISA MIGRATION ON HUMANEVAL-DECOMPILE (x86-64 $\rightarrow$ ARM64). EACH METHOD LIFTS AN x86-64 BINARY TO LLVM IR, THEN RECOMPILES THE LIFTED IR FOR ARM64. RE-COMPILABILITY AND RE-EXECUTABILITY ARE MEASURED ON THE ARM64 TARGET.

Method	Re-compilability	Re-executability
RetDec	90.6%	44.1%
LLM4Decompile	82.5%	40.7%
ATLAS	98.3%	93.6%

Table VI shows that ATLAS produces LLVM IR that is more amenable to downstream analysis than both the Binary $\rightarrow$ C $\rightarrow$ IR baseline and the rule-based lifter RetDec. SVF completes on 85.5% of functions for ATLAS, compared with 81.7% for LLM4Decompile+clang and 82.5% for RetDec, indicating that ATLAS yields IR that is more consistently analyzable under the same tool configuration. For functions that SVF can process, ATLAS also achieves the smallest average alias set size (1.00), suggesting slightly better pointer precision than the baselines. The largest gain is observed in PCI, where ATLAS reaches 132.9, substantially higher than LLM4Decompile+clang (80.4) and RetDec (107.1). The results indicate that direct lifting to canonical LLVM IR better preserves analysis-relevant typing and memory-flow information, improving both SVF robustness and the informativeness of downstream value-flow graphs.

XI. DOWNSTREAM CROSS-ISA MIGRATION

To evaluate whether ATLAS recovers architecture-independent program semantics rather than ISA-specific surface patterns, we study a downstream *cross-ISA migration* task. The key idea is that canonical, pre-optimization LLVM IR decouples high-level program logic from the source instruction set, making it possible to recompile lifted binaries for a different target architecture. This provides a direct test of whether the recovered IR can serve as a reusable intermediate representation for software migration.

We first compare ATLAS with representative baselines for cross-ISA migration from x86-64 to ARM64. All methods start from an x86-64 binary and transform it into high-level code, which is then recompiled for ARM64 and evaluated on the ARM64 target. **RetDec** follows a rule-based lifting pipeline that translates the x86-64 binary into LLVM IR using static recovery heuristics. **LLM4Decompile** follows

Table VIII
CROSS-TARGET **RE-COMPILABILITY** OF ATLAS ON HUMANEVAL-DECOMPILE. ROWS DENOTE THE SOURCE ISA OF THE LIFTED BINARY, AND COLUMNS DENOTE THE TARGET ISA USED FOR RECOMPILATION.

Source ISA	Target x86-64	Target ARM64	Target RISC-V
x86-64	—	98.3%	98.3%
ARM64	98.8%	—	98.8%
RISC-V	97.8%	97.8%	—

Table IX
CROSS-TARGET **RE-EXECUTABILITY** OF ATLAS ON HUMANEVAL-DECOMPILE. ROWS DENOTE THE SOURCE ISA OF THE LIFTED BINARY, AND COLUMNS DENOTE THE TARGET ISA USED FOR RECOMPILATION AND EXECUTION.

Source ISA	Target x86-64	Target ARM64	Target RISC-V
x86-64	—	93.6%	93.6%
ARM64	94.2%	—	94.2%
RISC-V	80.0%	80.5%	—

a Binary to C pipeline, where the x86-64 binary is first decompiled to C and then recompiled for ARM64 platform. ATLAS directly lifts the x86-64 binary to canonical LLVM IR and refines it with tool-guided feedback. As shown in Table VII, ATLAS achieves the best migration performance, reaching 98.3% re-compilability and 93.6% re-executability after recompilation to ARM64. In contrast, RetDec attains 90.6% re-compilability but only 44.1% re-executability, while LLM4Decompile yields 82.5% re-compilability and 40.7% re-executability. The large gap in re-executability indicates that ATLAS more reliably preserves architecture-independent semantics required for cross-target reuse, whereas the baselines often produce artifacts that recompile but fail to preserve behavior under migration.

We further report full source-target transfer matrices in Tables VIII and IX. Table VIII shows that ATLAS maintains high recompilation success across all source-target pairs, with all results exceeding 97%. Table IX demonstrates that these recompilation gains translate to functional success, with the best cross-ISA migration results again obtained when lifting from x86-64 and recompiling to ARM64 or RISC-V (both 93.6%). Similarly, transfers from ARM64 and RISC-V also exhibit strong performance. Overall, these results indicate that ATLAS recovers a stable canonical IR that supports cross-architecture recompilation and execution. The consistently

Table X
PORTABILITY ACROSS BACKBONE LLMs. PERFORMANCE ON HUMAN-EVAL-DECOMPILE WHEN USING A GENERAL-PURPOSE LLM ALONE (**BASE**) VERSUS AUGMENTING IT WITH OUR FRAMEWORK (**+ATLAS**: STRUCTURE-AWARE RAG + ADVISOR-VALIDATION REFINEMENT).

Model / Setting	Re-compilability (%)					Re-executability (%)					Edit Similarity (%)				
	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.	-O0	-O1	-O2	-O3	Avg.
Qwen-Plus (Base)	25.7	9.5	8.7	6.8	12.7	16.2	9.4	4.1	3.4	8.3	37.5	23.6	25.6	25.4	28.0
Qwen-Plus (+ATLAS)	69.6	58.1	54.1	50.0	58.0	51.4	40.5	38.5	37.8	42.1	54.4	40.6	39.8	40.9	43.9
Claude Sonnet 4.5 (Base)	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 (+ATLAS)	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
ChatGPT-4o (Base)	20.3	7.4	9.5	8.8	11.5	11.5	2.0	2.0	2.0	4.4	41.8	23.5	25.0	24.9	28.8
ChatGPT-4o (+ATLAS)	57.4	42.6	45.3	43.2	48.6	42.6	37.2	35.8	35.8	37.9	50.4	41.1	41.8	41.3	43.7
DeepSeek-V3.2 (Base)	30.4	35.8	25.0	22.3	28.4	14.2	9.4	7.4	9.5	10.1	44.9	30.5	30.7	31.0	34.3
DeepSeek-V3.2 (+ATLAS)	94.6	90.5	83.8	88.2	88.2	86.6	67.1	61.0	57.9	68.2	63.6	41.3	43.5	43.4	47.9
Gemini-3 (Base)	42.7	46.3	45.1	48.8	45.7	33.5	30.5	30.5	26.2	30.2	30.7	28.0	26.7	27.8	28.3
ATLAS (Gemini-3 backbone)	100.0	99.4	98.8	98.8	99.2	96.3	95.1	92.7	93.9	94.5	64.3	47.0	46.6	47.7	51.4

strong transfer performance further suggests that ATLAS is not merely translating local instruction patterns, but reconstructing an architecture-independent intermediate representation that preserves program logic across targets, which is essential for migration workflows in binary-only settings.

XII. FURTHER ANALYSIS

A. Generalization Across Backbone LLMs

To isolate the contribution of our lifting pipeline from the choice of backbone model, we instantiate the framework with five general-purpose LLMs: Qwen-Plus, Claude Sonnet 4.5, ChatGPT-4o, DeepSeek-V3.2, and Gemini-3. Table X reports results on HumanEval-Decompile across four optimization levels ($-O0$ – $-O3$). For each backbone, we compare the base model against the same model augmented with our framework (structure-aware retrieval and the Advisor-Validation refinement loop), using identical prompts, the same retrieval database, and the same refinement settings.

Table X shows that the ATLAS consistently improves all backbone models on all three primary lifting metrics. Averaged across the five backbones, our method increases re-compilability from 24.4% to 70.7%, re-executability from 13.0% to 54.3%, and edit similarity from 30.3% to 46.4%. These gains indicate that the benefit does not come from a single backbone, but from the combination of control-flow-aware retrieval and verification-guided refinement.

The greatest improvements are observed for DeepSeek-V3.2 and Gemini-3. For DeepSeek-V3.2, our scheme improves average re-compilability from 28.4% to 88.2% and re-executability from 10.1% to 68.2%. For Gemini-3, the same scheme raises re-compilability from 45.7% to 99.2% and re-executability from 30.2% to 94.5%, while also improving edit similarity from 28.3% to 51.4%. This result suggests that stronger base models benefit most from the structured retrieval and repair loop, since they can better exploit retrieved exemplars and tool feedback.

The improvements are also consistent across optimization levels. For weaker backbones such as Qwen-Plus and ChatGPT-4o, the raw models degrade sharply as optimization increases, but the proposed scheme substantially narrows this gap. For example, Qwen-Plus improves re-compilability from 6.8% to 50.0% and re-executability from 3.4% to 37.9% at $-O3$. In contrast, ATLAS remains stable across optimization

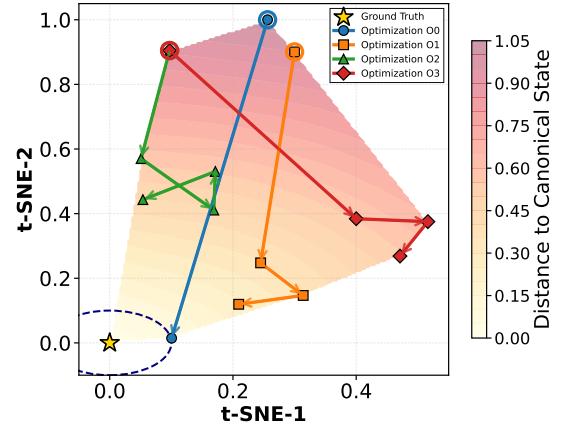


Figure 8. t-SNE visualization of the semantic refinement trajectory for one test case (ID:132) in the HumanEval-Decompile dataset. We project the latent embeddings of the lifted LLVM IR using IR2vector [26]. The paths trace the evolution from the initial input through five iterative refinement cycles toward the Ground Truth (Red Star).

levels, maintaining 98.8–100.0% re-compilability and 93.9–96.1% re-executability from $-O0$ to $-O3$. Overall, these results show that the proposed framework is backbone-agnostic and that its gains arise primarily from the structure-aware RAG and iterative refinement design, rather than from relying on a particular proprietary model.

B. Lifting Direction Correction

Figure 8 provides a qualitative view of how iterative refinement changes the lifted IR in representation space. We embed each intermediate LLVM IR produced during refinement using IR2Vec [26] and project the embeddings to two dimensions with t-SNE. Each trajectory starts from the initial draft (here initialized from the rule-based lifter MCTOLL) and follows successive refinement steps toward the ground-truth canonical IR. Across optimization levels, the initial drafts lie far from the ground-truth embedding, suggesting that rule-based lifting often produces IR that is structurally and operationally different from the canonical reference. For $-O0$ inputs, the trajectory typically reaches the neighborhood of the ground truth in one refinement step, consistent with the lower degree of optimization-induced rewriting. In contrast, $-O1$ – $-O3$ inputs typically require multiple iterations, with trajectories moving toward the reference in a stepwise manner. We can also

Table XI
IMPACT OF THE ITERATIVE REFINEMENT COUNT ON ATLAS
PERFORMANCE ON THE HUMANEval-DECOMPILE DATASET (x86-64).
 $k = 0$ REPRESENTS THE INITIAL INFERENCE WITHOUT FEEDBACK.

Iteration	Re-Comp (%)	Re-Exec (%)	Edit-Sim (%)
0	87.8	56.9	50.9
1	94.8	78.4	51.2
2	97.3	88.1	51.3
3	97.9	91.9	51.4
4	98.9	93.3	51.4
5	99.2	94.5	51.4
6	98.8	94.8	51.4
7	99.5	95.0	51.4

observe a two-phase behavior: the largest displacement toward the ground truth often occurs during the initial synthesis stage, indicating that retrieval-grounded generation provides coarse alignment to the canonical $-\text{OO}$ target even for highly optimized binaries. Subsequent refinement steps make smaller, localized movements, consistent with targeted repairs that address remaining legality and behavioral mismatches while staying within the same semantic neighborhood. Although t-SNE is qualitative and does not by itself establish semantic equivalence, this pattern is consistent with our ablations showing that retrieval substantially improves the initial draft and that the Advisor-Validation loop yields additional gains in end-to-end re-executability.

C. Impact of iterative refinement.

Table XI evaluates the effect of the refinement-loop iteration count (k). The results show that iterative refinement substantially improves both re-compilability and re-executability in the early stages. Starting from the initial draft without feedback ($k = 0$), re-compilability increases from 87.8% to 99.2% at $k = 5$, while re-executability improves from 56.9% to 94.5%, a gain of 37.6 percentage points. These results indicate that the feedback loop is effective at correcting both structural and behavioral errors in the lifted IR. In contrast, edit similarity remains essentially unchanged throughout refinement (50.9% at $k = 0$ and 51.4% from $k = 1$ onward), suggesting that the iterative process improves functional correctness and verifier validity without materially changing the overall textual proximity to the canonical reference IR. Beyond $k = 5$, the gains become marginal: re-executability increases only slightly from 94.5% to 95.0% at $k = 7$, while re-compilability no longer improves and even drops slightly at $k = 7$. Therefore, we set $k = 5$ as the default iteration budget, since it captures most of the benefit of refinement while avoiding unnecessary computational overhead.

D. Impact of Input Length on Decompile Performance

We also investigate the impact of assembly code length on the lifting performance of ATLAS. Figure 9 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 10 compares ATLAS with seven baselines across six

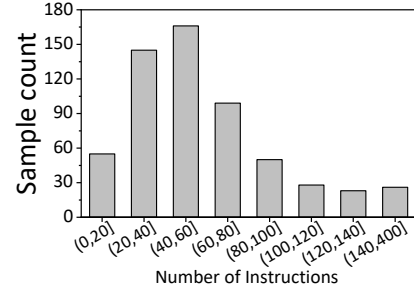


Figure 9. Distribution of input assembly lengths (measured by the number of instructions) in the HumanEval-Decompile-IR benchmarks.

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 39.7%. When the instruction count exceeds 60, re-executability drops below 9% for all models except Gemini-3, which retains a higher rate of 31.0%, making them ineffective for non-trivial functions. In contrast, ATLAS significantly extends this effective window. It maintains robust functional correctness well into medium-length sequences, achieving 100% 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. Even in the longest sequences ([140, 395] instructions), ATLAS retains 100% Re-compilability and 93.8% CFG Similarity. This significantly outperforms the best LLM baselines, proving that our framework preserves the global topology of massive functions where others produce non-compilable code. While structural recovery remains high, Re-executability drops to 73.1% for the longest functions. This divergence suggests that while ATLAS successfully recovers the “program skeleton,” the accumulation of minor logic errors in very long sequences remains a challenge for end-to-end execution.

Figure 11 presents the lifting performance of ATLAS across varying optimization levels and input instruction counts. We can observe that the re-executability remains high for all levels in short sequences, but -O1 and -O3 exhibit a decline at the extreme long-tail (140+ instructions). This trend indicates that aggressive transformations, such as inlining and loop restructuring, introduce functional complexities that challenge the ATLAS in long-context scenarios. Regarding textual fidelity, the ATLAS maintains a robust average edit similarity of 64.3% for $-\text{OO}$. However, this metric decreased in the long-tail regime (140-395 instructions, accounting for 54.2% of the HumanEval-Decompile). Moreover, it is worth noting that in the [2, 20] instruction range, we observe a decoupling of functional and structural metrics. While ATLAS achieves near-perfect *Re-executability*, both *CFG Similarity* (28.6%) and *Edit Similarity* (49.0%) remain low. This is not

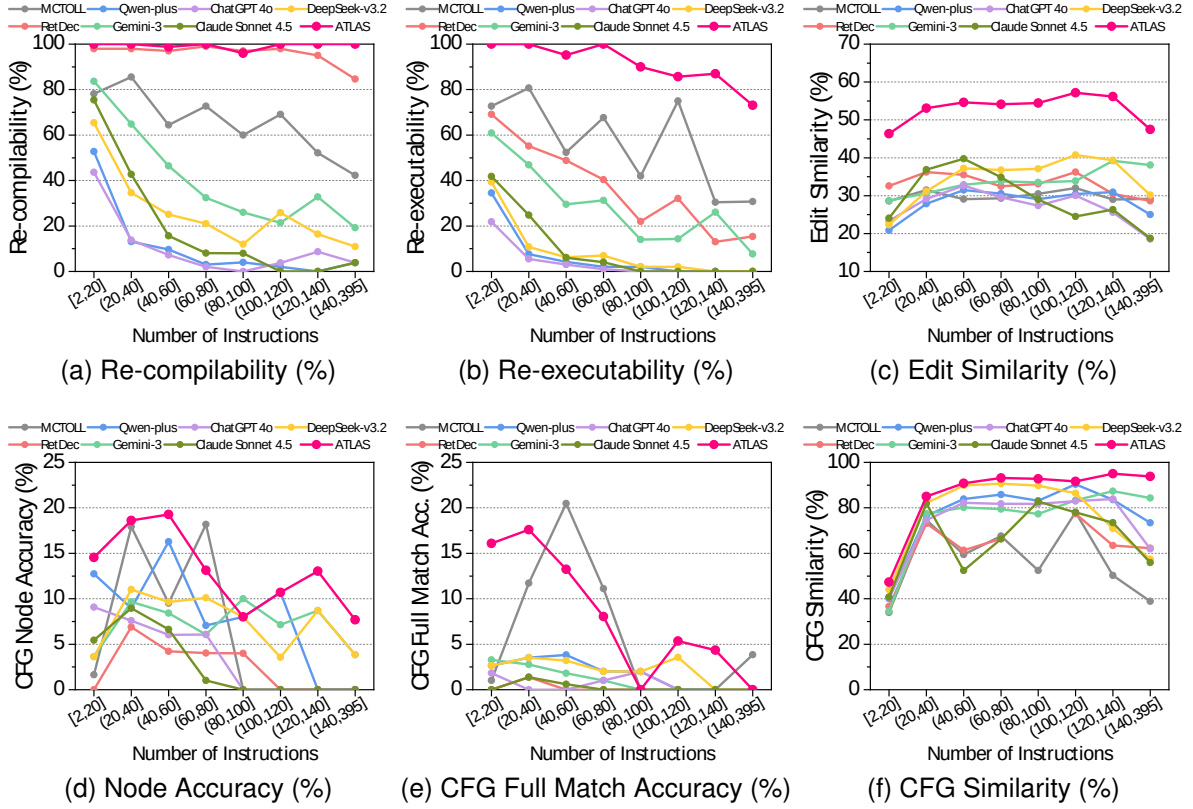


Figure 10. ATLAS lifting performance comparison across different input assembly code lengths of HumanEval-Decompile dataset (x86-64).

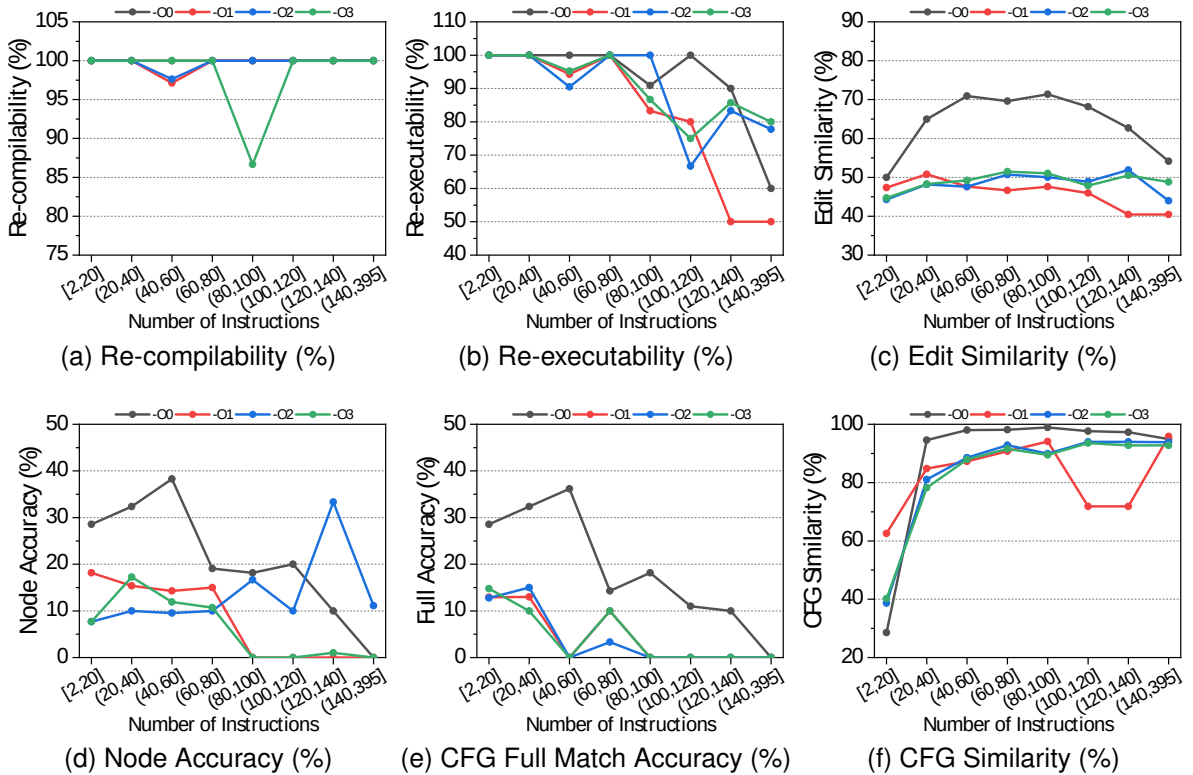


Figure 11. ATLAS's performance across varying input assembly code lengths and optimization levels on the HumanEval-Decompile dataset (x86-64).

an indication of lifting failure, but rather a reflection of metric sensitivity to sparse data. In small functions, minor variations in register naming, function attributes, or the inclusion of

entry/exit blocks represent a significant percentage of the total code or graph structure, leading to disproportionately low similarity scores despite functional equivalence.

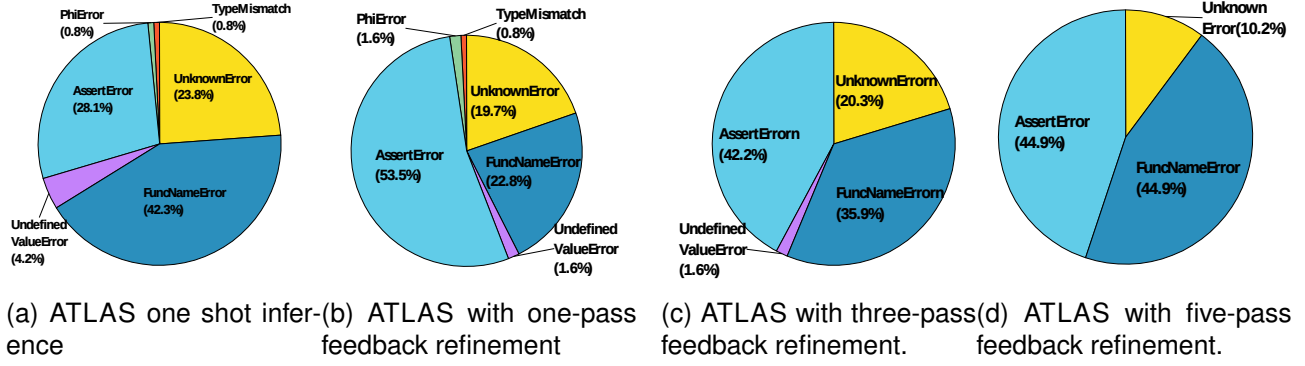


Figure 12. Distribution of error types in ATLAS under one-shot inference compared to one, three, and five iterations of feedback-directed refinement on the HumanEval-Decompile dataset (x86-64).

Table XII
PERFORMANCE ON LONG FUNCTIONS WITH MORE THAN 200 INSTRUCTIONS.

Method	Re-compilability (%)	Re-executability (%)	Edit-sim. (%)
RetDec	91.1	10.7	23.9
ATLAS	100	76.9	44.4

Table XIII
PERFORMANCE UNDER DIFFERENT TEST CASE SOURCES ON HUMAN-EVAL-DECOMPILE (X86-64).

Test Case Setting	Re-Comp (%)	Re-Exec (%)	Edit-Sim (%)
Structural Only (No Tests)	99.1	88.0	51.2
LLM-Generated Tests	98.6	89.5	51.4
Human-written Tests	99.2	94.5	51.4

Table XII lists the performance of ATLAS on a harder subset of long functions, including 56 cases with more than 200 instructions from HumanEval and MBPP. We compare ATLAS with the rule-based lifter RetDec on this subset. RetDec achieves relatively high re-compilability (91.1%), yet its re-executability is only 10.7% and edit similarity is 23.9%, indicating that the generated code often compiles but fails to preserve the ground-truth program behavior. In contrast, ATLAS achieves 100% re-compilability on these long functions, while substantially improving re-executability to 76.9% and increasing edit similarity to 44.4%. This demonstrates that ATLAS not only preserves structural correctness but also significantly enhances functional fidelity, even for structurally complex functions. These results highlight that ATLAS scales effectively to long functions, maintaining both syntactic validity and semantic consistency, whereas rule-based approaches tend to degrade in behavioral correctness under increased structural complexity.

E. Error Types Considered in ATLAS

To evaluate the effectiveness of the ATLAS iterative refinement scheme, we categorize the errors encountered during the lifting process into four primary types: *FuncNameError*, *AssertError*, *Phi conflict*, and *Definition Conflict* (Table II lists the details). The Advisor-Validation loop progressively reduces structural failures and shifts the remaining failures toward behavioral mismatches. Early iterations prioritize

compiler- and verifier-reported issues, which enables more candidates to pass IR validation and reach execution. For example, *FuncNameError* decreases from 110 under one-shot inference to 22 by the third refinement pass, and *PhiError* and *DefinitionConflict* exhibit similar reductions. By the fifth iteration, the total failure count drops by 81.2%, and the dominant remaining failures are *AssertError*. This shift indicates that refinement moves the pipeline from failing at compilation/verification to reaching runtime checking, where the remaining discrepancies are primarily semantic.

F. Impact of Test-Case-Guided Feedback

Table XIII quantifies how functional test cases affect our iterative refinement. We compare two settings: (i) *structural-only* refinement, where the loop relies on verifier/compilation checks and structural feedback without runtime tests, (ii) LLM-generated tests, and (iii) refinement with *human-written* unit tests.

For the *LLM-generated tests* setting, we follow an automated pipeline that first recovers a plausible calling interface for each target function and then synthesizes executable test drivers. In detail, *anqr* performs control-flow analysis, calling-convention inference, and candidate prototype generation for the target function, producing structured static facts. We then use *DynamoRIO drwrap* [27] for function-level dynamic instrumentation, executing the function under a small set of heuristic inputs to collect parameter access patterns, return-type signals, and pointer/string behaviors, which form a dynamic observation report. Next, we package the disassembly, static facts, and dynamic observations into a structured prompt for DeepSeek-V3.2, which outputs multiple candidate signatures with confidence scores. These candidates are filtered and validated in two stages: we first discard signatures that contradict strong dynamic observations, and then run *libFuzzer*-based [28] feasibility checks by compiling each candidate into a harness that invokes the original binary and accepting it only if it remains crash-free under large volumes of randomized inputs. If all candidates fail, we feed the failure evidence back to the model for regeneration. Finally, for verified signatures, we perform differential checking by executing the original binary function and the lifted IR on the same inputs and comparing return values and observable

Table XIV
AVERAGE TOKEN USAGE AND MONETARY COST ACROSS INFERENCE
SESSIONS IN ATLAS ON HUMANEval (x86-64).

Inference session	Input tokens	Output tokens	Cost (\$)
Session 1 (Initial)	2832.49	1710.90	0.006548933
Session 2 (IR 1)	3404.53	2060.09	0.007882541
Session 3 (IR 2)	3418.04	2016.53	0.007758606
Session 4 (IR 3)	3378.10	1987.15	0.007650497
Session 5 (IR 4)	3281.11	1926.72	0.007420707
Session 6 (Final)	3258.95	1908.56	0.007355145
Total	19573.21	11609.94	0.044616429

memory side effects, and the discrepancies are recorded as runtime evidence to guide subsequent repairs.

We can observe that the re-compilability is high in all cases (98.6–99.2%), indicating that toolchain checks and structural constraints are sufficient to produce verifier-clean IR regardless of the test source. In contrast, re-executability varies substantially with the availability and quality of tests. Using no tests yields 88.0% re-executability, while adding LLM-generated tests improves it to 89.5% (+1.5 points), and human-written tests further raise it to 94.5% (+6.5 points over structural-only). This pattern suggests that tests primarily contribute behavioral constraints that resolve residual semantic ambiguity that is not visible to compilation and CFG-level checks. Edit similarity remains essentially unchanged (51.2–51.4%), implying that the gains in re-executability are not driven by broader rewrites toward the reference text, but by localized corrections that improve behavior. Overall, the results indicate that ATLAS remains effective even without tests, but stronger functional oracles yield materially better semantic fidelity, while LLM-generated tests provide a lightweight proxy when developer-written tests are unavailable.

G. Cost Analysis of ATLAS

Table XIV reports the average token usage and cost of ATLAS instantiated with the Gemini-3 backbone on HumanEval (x86-64), broken down by inference sessions. The initial synthesis stage consumes 2,832 input tokens and 1,711 output tokens on average (\$0.0065). Each subsequent refinement step incurs a similar per-iteration cost, with inputs around 3.26K–3.42K tokens and outputs around 1.91K–2.06K tokens (\$0.0074–\$0.0079 per step). Over the full six-session run (initial draft plus five refinement rounds), the total averages 19,573 input tokens and 11,610 output tokens, corresponding to \$0.0446 per function. In practice, the full five-round refinement is primarily needed for harder cases, many functions converge in fewer iterations. Consequently, the average cost reported here should be interpreted as an upper-bound configuration under a fixed iteration budget rather than the typical cost for all inputs.

XIII. DISCUSSION AND LIMITATIONS

We show that ATLAS generalizes across x86-64 and ARM64 in our evaluation. Extending to additional architectures (e.g., MIPS or RISC-V) requires rebuilding the structure-aware RAG database for the target ISA and toolchain con-

figuration. While this adds engineering cost, the data construction pipeline is automated and can be rerun with the appropriate compiler backend and instrumentation settings. Nevertheless, retrieval quality and repair behavior may vary with ISA-specific idioms (e.g., branch forms, calling conventions, and jump-table lowering), so cross-ISA transfer is not guaranteed without reindexing and calibration. Our evaluation on HumanEval-Decompile and MBPP benefits from curated unit tests, whereas real reverse-engineering scenarios (e.g., malware, proprietary libraries, or firmware) rarely provide input–output oracles. In such settings, functional feedback can be approximated by integrating binary test generation, for example via symbolic execution, concolic testing, or fuzzing to produce inputs that exercise diverse paths. These techniques can supply partial oracles (e.g., differential consistency checks, crash/non-crash signals, or metamorphic properties) that still guide refinement, although they may be incomplete and can introduce noise. Finally, ATLAS is constrained by input length. For large functions or binaries with extensive inlining and unrolling, disassembly and IR context can exceed the backbone model’s context window. Our current design mitigates this by segmenting code into CFG-aligned regions and performing retrieval and repair at region granularity, but very large or highly entangled functions may still require additional decomposition (e.g., loop/SCC regions or dominator-subtree partitioning) and interface contracts to stitch regions back together.

XIV. RELATED WORK

A. LLVM Intermediate Representations Recovery

Binary-to-LLVM IR decompilation is crucial for program analysis and reverse engineering tasks, including malware analysis and software migration. Tools like McSema [29], LLVM-MCTOLL [3], and RevGen [30] lift binaries to LLVM IR but struggle with Static Single Assignment (SSA) reconstruction, control flow preservation, and semantic fidelity under aggressive optimizations (e.g., -O3). For instance, McSema, while robust for x86 binaries, often produces IR with low structural similarity due to imprecise control flow recovery. Similarly, LLVM-MCTOLL generates functionally correct but structurally divergent IR, limiting its utility for high-precision analysis. Academic efforts, such as Schwartz et al. [31], propose control flow structuring but lack generalizability across optimization levels. LeanBin [32] introduces mechanisms for binary lifting with a focus on debloating, addressing semantic gaps in IR transformation. However, its emphasis on code size reduction limits its applicability to precision-focused tasks. Recent work on lifting assembly to intermediate representations [14] provides foundational insights but lacks scalability for complex binaries. ATLAS overcomes these challenges using a control-flow-aware RAG database and pseudo-probe instrumentation, achieving high CFG similarity and edit similarity across diverse optimization levels.

B. Learning-Based and Neural Decompilation Methods

Neural-based decompilation has gained traction with the use of transformers and large language models (LLMs). Methods like NeuralDecompile [33], LLM4Decompile [1], and

SALDE [13] excel in binary-to-C decompilation, inferring high-level semantics. However, adapting LLMs for binary-to-LLVM IR lifting is challenging, as IR requires precise typing and structural fidelity to compiler formats, unlike human-readable C code. Meta’s LLMDecompiler FTD [9] and Forklift [14] target decompilation from Clang -S assembly to LLVM IR, leveraging metadata (e.g., symbol tables) unavailable in raw binaries, making it unsuitable for true binary decompilation. ATLAS integrates LLMs with a structure-aware RAG framework and iterative static-guided refinement, achieving robust IR reconstruction across optimization levels (-O0 to -O3). It outperforms baselines like McSema and LLM4Decompile in structural fidelity, particularly for complex binaries with system calls or inline assembly.

C. Retrieval-Augmented Generation in Code Tasks

Retrieval-Augmented Generation (RAG) [34] enhances generative models by incorporating external context, proving effective in code completion [35] and program synthesis [36]. However, its application to binary-to-LLVM IR decompilation remains underexplored. Existing RAG-based methods, such as CodeRAG [37], focus on high-level code synthesis and lack structure-aware retrieval for low-level IR. Recent advances like BTR [38] introduce efficient token-based retrieval for binary tasks, but they do not address IR-specific structural constraints. R2AG [39] and LI-RAGE [40] propose advanced retrieval-integration architectures, offering potential for improving iterative retrieval-generation synergy. Similarly, DecLLM [23] and ITER-RETGEN [41] emphasize iterative feedback loops, aligning with ATLAS’s approach. Specifically, DecLLM focuses on recompilable C code generation via LLMs and applies intermediate lifting to IR during training. ATLAS is fundamentally different in its target (LLVM IR vs. C) and setting (binary input vs. assembly from Clang -S), and architecture. ATLAS pioneers RAG for binary-to-LLVM IR lifting, using a fine-grained, control-flow-aware database to retrieve precise IR constructs, outperforming neural baselines in CFG and edit similarity, even for large binaries.

XV. CONCLUSION

We have presented ATLAS, a multi-agent framework for lifting heavily optimized and stripped binaries into *canonical, pre-optimized* LLVM IR. Rather than reconstructing source-level syntax, ATLAS targets verifier-clean, analysis-ready IR by combining control-flow-aware retrieval with verification-guided refinement. Across HumanEval-Decompile and MBPP, three ISAs (x86-64, ARM64, and RISC-V), and optimization levels from -O0 to -O3, ATLAS consistently improves recompilability, re-executability, and similarity to the reference IR over both rule-based and LLM-based baselines. Beyond matching the reference more closely, the lifted IR better supports downstream uses, including SVF-based program analysis and cross-ISA software migration. These results suggest that targeting compiler IR, rather than decompiled source code alone, provides a practical foundation for binary recovery that is both semantically useful and directly compatible with existing analysis and compilation toolchains.

REFERENCES

- [1] H. Tan, Q. Luo, J. Li, and Y. Zhang, “LLM4Decompile: Decompile binary code with large language models,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024, pp. 3473–3487.
- [2] H. Tan, W. Li, X. Tian, S. Wang, J. Liu, J. Li, and Y. Zhang, “Sk2decompile: Llm-based two-phase binary decompilation from skeleton to skin,” *arXiv preprint arXiv:2509.22114*, 2025.
- [3] S. B. Yadavalli and A. Smith, “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*, 2019, p. 213–218.
- [4] A. Software, “Retdec: Open-source machine-code decompiler,” <https://github.com/avast/retdec>, 2017.
- [5] X. Zhu, J. Ren, Z. Li, J. Zheng, Z. Tang, and Z. Wang, “Lifting optimized binaries to canonical compiler IR via structure-aware retrieval and iterative verification,” in *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. San Diego, California, United States: Association for Computational Linguistics, Jul. 2026, pp. 11498–11516. [Online]. Available: <https://aclanthology.org/2026.acl-long.527/>
- [6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [7] Y. Sui and J. Xue, “Svf: interprocedural static value-flow analysis in llvm,” in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.
- [8] LLVM Project, “The llvm compiler infrastructure project,” <https://github.com/llvm/llvm-project>, 2025, accessed: 2025-01-24.
- [9] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, “Llm compiler: Foundation language models for compiler optimization,” in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC ’25, 2025, p. 141–153.
- [10] OpenAI, “ChatGPT,” <https://chatgpt.com/>, 2025, accessed: 2025-01-24.
- [11] Google, “Gemini: Google’s AI Model,” <https://gemini.google.com/>, 2025, accessed: 2025-01-24.
- [12] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. P. O’Boyle, “Exebench: An ml-scale dataset of executable c functions,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, p. 50–59.
- [13] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O’Boyle, “Slade: A portable small language model decompiler for optimized assembly,” in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 67–80.
- [14] J. Armengol-Estapé, R. C. O. Rocha, J. Woodruff, P. Minervini, and M. O’Boyle, “Forklift: An extensible neural lifter,” in *First Conference on Language Modeling (COLM)*, 2024.
- [15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.
- [16] W. He, H. Yu, L. Wang, and T. Oh, “Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation,” in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 322–333.
- [17] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, X. Zhang, and P. Babkin, “Nova: Generative language models for assembly code with hierarchical attention and contrastive learning,” in *The Thirteenth International Conference on Learning Representations (ICLR)* 2025, 2025.
- [18] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [19] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “LoRA: Low-rank adaptation of large language models,” in *International Conference on Learning Representations*, 2022.
- [20] DeepSeek, “Deepseek chat,” <https://chat.deepseek.com/>, 2025, accessed: 2025-01-24.
- [21] Anthropic, “Claude,” <https://www.anthropic.com/claude>, 2024, accessed: 2025-01-24.
- [22] Tongyi-Lab, “Qwen,” <https://chat.qwen.ai/>, 2025, accessed: 2025-01-24.
- [23] W. K. Wong, D. Wu, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, “Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1841–1864, 2025.

- [24] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Networkx: Graph isomorphism — networkx documentation,” <https://networkx.org/documentation/stable/reference/algorithms/isomorphism.html>, 2024, accessed: 2025-01-24.
- [25] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis, “Grakel: A graph kernel library in python,” *Journal of Machine Learning Research*, vol. 21, no. 54, pp. 1–5, 2020.
- [26] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, “IR2Vec: LLVM IR Based Scalable Program Embeddings,” *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, 2020.
- [27] D. Bruening and S. Amarasinghe, “Efficient, transparent, and comprehensive runtime code manipulation,” 2004.
- [28] K. Serebryany, “Continuous fuzzing with libfuzzer and addresssanitizer,” in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 157–157.
- [29] L. Bits, “Mcsema: Lifting x86 and x86-64 binaries to llvm bitcode,” <https://github.com/lifting-bits/mcsema>, 2021, accessed: 2025-01-24.
- [30] S2E Team, “Translating binaries to llvm with revgen,” <https://s2e.systems/docs/Tutorials/Revgen/Revgen.html>, 2020, accessed: 2025-01-24.
- [31] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, “Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring,” in *Proceedings of the 22nd USENIX Conference on Security*, 2013, p. 353–368.
- [32] I. Wodiany, A. Pop, and M. Luján, “Leanbin: Harnessing lifting and recompilation to debloat binaries,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 1434–1446.
- [33] Y. David, U. Alon, and E. Yahav, “Neural reverse engineering of stripped binaries using augmented control flow graphs,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.
- [34] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.
- [35] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Retrieval augmented code generation and summarization,” in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 2719–2734.
- [36] T. B. Hashimoto, K. Guu, Y. Oren, and P. Liang, “A retrieve-and-edit framework for predicting structured outputs,” in *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018, p. 10073–10083.
- [37] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, “CodeRAG-bench: Can retrieval augment code generation?” in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 3199–3214.
- [38] Q. Cao, S. Min, Y. Wang, and H. Hajishirzi, “Btr: Binary token representations for efficient retrieval augmented language models,” in *International Conference on Representation Learning*, 2024, pp. 44 229–44 246.
- [39] F. Ye, S. Li, Y. Zhang, and L. Chen, “R²AG: Incorporating retrieval information into retrieval augmented generation,” in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 11 584–11 596.
- [40] W. Lin, R. Blloshmi, B. Byrne, A. de Gispert, and G. Iglesias, “LI-RAGE: Late interaction retrieval augmented generation with explicit signals for open-domain table question answering,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2023, pp. 1557–1566.
- [41] Z. Shao, Y. Gong, yelong shen, M. Huang, N. Duan, and W. Chen, “Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy,” in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 9248–9274.

Xiaoao Zhu received her BS degree in Computer Science from Shaanxi Normal University in 2023. Currently, she is pursuing the MS degree of software engineering at Shaanxi Normal University. Her research interest is mainly focused on LLVM optimization.

Ni Zhang received her BS degree in software engineering from Shaanxi Normal University in 2025. Currently, she is pursuing the MS degree of Computer Science at Shaanxi Normal University. Her research interest is mainly focused on binary analysis.

Jie Ren is an associate professor at the School of Computer Science at Shaanxi Normal University, Xi’an, China. He received the PhD degree in computer system architecture from Northwest University, Xi’an, China, in 2017. His research mainly focuses on mobile system optimisation and runtime scheduling.

Zhiqiang Li received the Ph.D degree in Computer Application Technology from Wuhan University, Wuhan, China, in 2018. He is currently an Associate Professor with the School of Computer Science, Shaanxi Normal University, Xi’an, China. His main research interests include intelligence software engineering and machine learning.

Jie Zheng received the BS. degree in communications engineering from Nanchang University in 2008 and the Ph.D. degree from the Department of Telecommunications Engineering, Xidian University, China, in 2014. He is currently an Associate Professor with the Information Science and Technology Institute, Northwest University, Xi’an, China. His research interests include energy-efficient transmission, wireless resource allocation, and edge intelligence.

Zhanyong Tang is currently a Professor with the School of Information Science and Technology, Northwest University. His current research interests include network and system security.

Zheng Wang is chair professor of Intelligent Software Technology at the School of Computing at the University of Leeds and a Turing Fellow at the Alan Turing Institute. His research focuses on accelerating large-scale deep learning models, program optimisation, applied machine learning and system security.