Self-Refining Decompilation with Execution Feedback

Anonymous ACL submission

Abstract

Decompilation—recovering source code from compiled binaries—is essential for program analysis, reverse engineering, and legacy software maintenance. However, existing decompilers often produce code that fails to *re*compile or execute correctly, limiting its practical utility. In this paper, we present AGENT4DECOMPILE, a multi-agent framework that transforms decompiled code into re-executable source through multi-level constraint-guided refinement. Our approach uses three validation stages: syntax validation ensures parseable C code, compilation validation catches type errors and missing declarations, and execution validation checks test-driven behavioral equivalence against the original binary. When validation fails at any level, specialized LLM agents refine the code using structured error feedback. We evaluate AGENT4DECOMPILE on 1,641 binaries from ExeBench across three decompilers (Ghidra, Angr, RetDec). AGENT4DECOMPILE achieves 40–46% re-executability on this large-scale benchmark, improving baseline output by 18–28 percentage points. Compared to state-of-the-art methods, AGENT4DECOMPILE (50.3%) outperforms single-pass LLM refinement (35.2%) and fine-tuned LLM4Decompile (12.1%) by 15–38 percentage points. Our ablation study shows that execution-based validation is critical: compile-only approaches achieve only 32–42% re-executability despite 99–100% compilation rates.

1 Introduction

Decompilation is the process of recovering high-level source code from compiled binary executables. This challenging task is a linchpin for many software security and reverse engineering tasks. Security analysts rely on decompilation for malware analysis (Yakdan et al., 2015), vulnerability discovery (Pewny et al., 2015), and understanding proprietary or legacy software (Cifuentes, 1994). An ideal decompiler would produce source code that is not only readable but also *re-compilable* and *re-executable* — that is, capable of being compiled and executed to produce the same behavior as the original binary. Such re-executable decompilation would enable powerful downstream applications including binary rewriting, automated patching, and fuzz testing of source code recovered via decompilation.

Unfortunately, existing decompilers fall far short of this ideal. State-of-the-art tools like Ghidra (National Security Agency, 2019), IDA Pro (Hex-Rays, 2018), and RetDec (Křoustek et al., 2017) employ sophisticated program analysis techniques, but produce output that frequently contains syntactic errors, type mismatches, and semantic inconsistencies. Our empirical analysis reveals that raw decompiler output achieves only 22–26% re-executability across different tools (Table 1). This gap between decompiler output and usable source code represents a substantial barrier to practical binary analysis workflows.

Recent advances in Large Language Models (LLMs) have sparked interest in leveraging neural approaches for decompilation (Tan et al., 2024, 2025). These methods treat decompilation as a translation task, training or prompting LLMs to convert assembly or decompiled pseudocode into compilable C code. While promising, existing LLM-based approaches face fundamental limitations: they operate in a single-shot manner without feedback, lack mechanisms to verify behavioral correctness. More fundamentally, these limitations suggest that decompilation cannot be solved as a single-pass translation problem.

In this paper, we present AGENT4DECOMPILE, a multi-agent framework that transforms decompiled code into re-executable source through multi-level constraint-guided refinement. Our key insight is that decompilation is inherently a multi-level problem arising from the lossy nature of compilation. During compilation, code is lowered to assem-

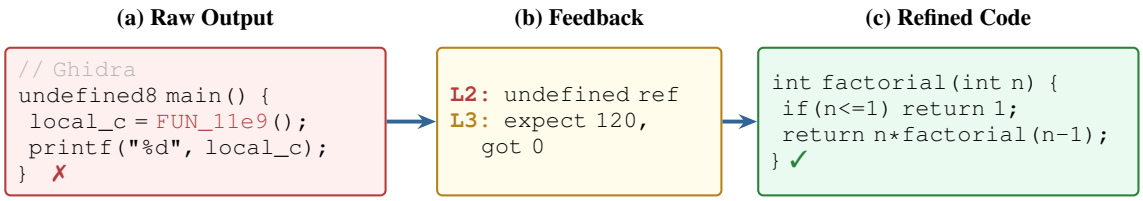

Figure 1: Motivating example: (a) Raw Ghidra output contains undefined functions and type errors. (b) Multi-level constraints provide structured feedback: L2 catches missing function references, L3 catches behavioral errors. (c) AGENT4DECOMPILE produces correct, re-executable code through iterative refinement.

bly and machine code, leading to a loss different types of information at different stages. Consequently, no single validation signal can recover all lost information. Variable and struct names are discarded early, causing syntax-level inconsistencies in decompiled output; type information is erased, leading to compilation failures; and control flow semantics are obscured by optimization, resulting in behavioral mismatches. Crucially, these error classes are *not jointly optimizable*: execution feedback cannot fix syntax errors, and compilation diagnostics cannot detect semantic mismatches. This necessitates a hierarchical validation strategy that integrates complementary signals across levels.

Therefore, we formulate decompilation refinement as a *progressive search space reduction*: given the vast space of possible C programs, each constraint level defines a successively smaller feasible subspace—L1 (syntax) restricts to parseable programs, L2 (compilation) further narrows to type-correct programs, and L3 (execution) identifies behaviorally equivalent programs. Figure 1 illustrates this progression. By decomposing the search into hierarchical stages, AGENT4DECOMPILE makes the refinement tractable: rather than searching the entire program space with a single weak signal, we progressively filter through increasingly precise constraints, each providing targeted feedback for its corresponding error class.

Our ablation study reveals a critical finding: *compilation success does not imply behavioral correctness*. A compile-only validation approach (Levels 1+2) achieves 99–100% compilation rates but only 32–42% re-executability; the code compiles but produces incorrect outputs. Incorporating execution-based validation (Level 3) improves re-executability to 43–50%, closing part of this gap. This result underscores the necessity of end-to-end validation for re-executable decompilation.

On a 157-binary benchmark, AGENT4DECOMPILE improves re-executability from 22–26% (baseline) to 43–50% across three decompilers (Ghidra, Angr, and RetDec), representing improvements of 18–28 percentage points. Compared to state-of-the-art refinement methods, AGENT4DECOMPILE (50.3%) outperforms single-pass LLM refinement (35.2%) and fine-tuned LLM4Decompile-Ref (12.1%) by 15–38 percentage points. The system scales effectively: on 1,641 binaries from ExeBench, AGENT4DECOMPILE maintains 40–46% re-executability with an average of 1.5–2.3 iterations per binary.

This paper makes the following contributions:

1. **Insight:** We identify that compilation success does not imply behavioral correctness: a 57–68 pp gap exists between compile rate (99%) and re-executability (32–42%). This motivates multi-level validation with execution feedback (§2).

2. **System:** We present AGENT4DECOMPILE, a multi-agent framework that iteratively refines decompiled code through syntax, compilation, and execution constraints, improving re-executability from 22–26% to 43–50% across three decompilers (§2).

3. **Evaluation:** We demonstrate 15–38 pp improvement over prior refinement methods on 1,641 binaries, with ablations showing L3 execution feedback is essential for behavioral correctness (§3). We release all benchmarks and artifacts at: `https://anonymous.4open.science/r/agent4decompile-artifacts-0F69`.

2 Approach

We introduce AGENT4DECOMPILE, a system that transforms decompiled C code into re-executable source through constraint-guided LLM refinement. Given a binary, an off-the-shelf decompiler first produces initial code C_0. AGENT4DECOMPILE then validates and repairs this code through three

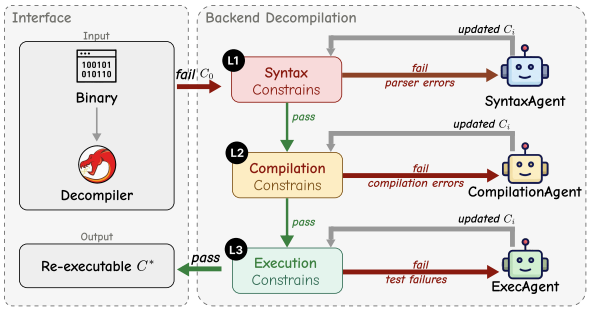

Figure 2: Overview of AGENT4DECOMPILE. A binary is first processed by a traditional decompiler to produce C_0. The code then passes through L1 syntax validation, L2 compilation validation, and L3 execution validation. Failures trigger specialized LLM repair agents; passing all levels yields re-executable source C^*.

increasingly strict levels: syntax, compilation, and execution. Figure 2 summarizes the workflow.

2.1 Constraint-Guided Validation

AGENT4DECOMPILE is based on the observation that decompilation errors appear at different levels, and each level provides a different diagnostic signal. L1 checks whether the decompiled output is parseable C code. L2 checks whether the parsed code compiles successfully, catching type mismatches, missing declarations, and linkage errors. L3 checks behavioral equivalence by compiling the repaired source, executing it on test inputs, and comparing its outputs against the original binary.

These constraints form a hierarchy: code must pass syntax before compilation, and compilation before execution. This hierarchy makes repair more targeted. Syntax errors provide localized parser feedback; compilation errors expose type and symbol inconsistencies; execution failures expose semantic divergence that is invisible to the compiler. Algorithm 1 in Appendix A formalizes the loop, and Appendix B gives the full constraint definitions.

2.2 LLM Repair Agents

When validation fails, AGENT4DECOMPILE invokes a repair agent specialized to the failing level. The L1 agent receives parser errors and repairs malformed C syntax. The L2 agent receives compiler diagnostics and repairs type, declaration, and symbol errors. The L3 agent receives failing input-output examples, including expected output from the original binary and actual output from the recompiled source, and repairs behavioral errors.

Table 1: 157-binary results. L1/L2/L3 denote syntax, compilation, and re-execution; Δ is percentage-point improvement over the baseline.

	Base			+A4D			Δ		
Dec.	**L1**	**L2**	**L3**	**L1**	**L2**	**L3**	**L1**	**L2**	**L3**
Ghidra	86.0	86.0	22.3	100.0	100.0	**50.3**	+14.0	+62.4	+28.0
Angr	79.0	65.0	24.8	100.0	100.0	45.2	+21.0	+35.0	+20.4
RetDec	98.7	98.7	25.5	99.4	99.4	43.3	+0.7	+0.7	+17.8
Avg.	**87.9**	**67.1**	**24.2**	**99.8**	**99.8**	**46.3**	**+11.9**	**+32.7**	**+22.1**

All agents receive the current code, the failing constraint level, and structured diagnostic feedback. They output the full corrected C program directly, without explanations or markdown. The loop is stateless: each repair step uses the current code and current feedback rather than the full history of prior attempts. This keeps prompts compact while still allowing the code to progress through increasingly strict validation levels. Appendix B describes the prompt structure and feedback formatting.

2.3 Implementation Summary

We implement AGENT4DECOMPILE as a Python framework with four components: decompiler wrappers, constraint validators, LLM repair agents, and a refinement-loop orchestrator. The system integrates Ghidra (National Security Agency, 2019), Angr (Shoshitaishvili et al., 2016), and RetDec (Křoustek et al., 2017) behind a unified interface, so each backend returns decompiled C source for the same validation-and-repair pipeline.

Validators use GCC for syntax and compilation checks, and use sandboxed execution for behavioral validation. Expected outputs are obtained by running the original binary on the same test inputs. We use DeepSeek V3.2 (Liu et al., 2025) as the default repair model with deterministic decoding. Additional implementation details, including decompiler integration, validator settings, timeouts, caching, and logging, are provided in Appendix C.

3 Evaluation

We evaluate AGENT4DECOMPILE on two benchmarks across three decompiler architectures, comparing against state-of-the-art methods and analyzing the contribution of each design component. Our evaluation addresses three research questions: (1) **RQ1 (Effectiveness):** How effective is AGENT4DECOMPILE at improving decompilation

quality, and does it scale to larger benchmarks? (2) **RQ2 (Ablation):** What is the contribution of each constraint level (L1, L2, L3) to the overall system performance? (3) **RQ3 (Comparison):** How does AGENT4DECOMPILE compare to existing decompilation refinement approaches?

3.1 Experimental Setup

Benchmarks. We use two benchmarks. First, the **157-Binary** benchmark comprises 125 ExeBench (Armengol-Estapé et al., 2022) binaries and 32 custom binaries, curated for detailed analysis and SOTA comparison. The ExeBench portion consists of 32 functions compiled at four optimization levels (O0–O3), with 3 functions missing O0 variants; these include 17 functions from real-world open-source projects (`real_test`) and 15 synthetic functions (`synth_test`). The custom portion consists of 8 textbook algorithms (e.g., `binary_search`, `bubble_sort`, `factorial`, `fibonacci`) compiled at O0–O3, yielding 32 standalone programs with `main()` entry points verified via stdout comparison. Second, the **1,641-Binary** benchmark comprises 1,561 ExeBench binaries and 80 custom binaries for scalability evaluation. The ExeBench portion consists of 396 functions from the test set—247 from `real_test` (real-world code) and 149 from `synth` (synthetic functions)—compiled at O0–O3. The custom portion adds 20 algorithm implementations (e.g., `heapsort`, `mergesort`, `quicksort`, `bst`, `gcd`) disjoint from the 157-binary benchmark.

We evaluate on ExeBench (Armengol-Estapé et al., 2022) rather than HumanEval-Decompile (Tan et al., 2024) or MBPP (Austin et al., 2021), which are commonly reported in decompilation literature. ExeBench presents a more challenging testbed: functions are larger, derived from real-world open-source projects, and compiled across multiple optimization levels (O0–O3). Additionally, ExeBench provides IO-pair test cases with concrete expected outputs, which our feedback-based iterative refiner leverages to identify specific behavioral errors. In contrast, the original HumanEval (Chen et al., 2021) benchmark was designed for Python with assertion-based testing for automated black-box verification; this design was preserved when the benchmark was converted to C for decompilation evaluation. Assertion-based test cases provide only binary pass/fail signals without actionable diagnostic information—a benchmark design choice that limits their utility for iterative refinement.

Table 2: Scalability on 1,641 binaries. Each cell reports L3 re-executability after refinement, with Δ pp over the baseline in parentheses.

Decompiler	O0	O1	O2	O3
Ghidra	**50.4** (+24.7)	**47.0** (+23.0)	**45.7** (+21.7)	**40.9** (+19.4)
Angr	43.7 (+13.1)	42.9 (+14.9)	41.7 (+14.4)	35.1 (+12.9)
RetDec	42.4 (+21.8)	40.7 (+21.0)	39.9 (+20.2)	36.4 (+23.5)
Avg.	**45.5** (+19.9)	**43.5** (+19.6)	**42.4** (+18.7)	**37.5** (+18.6)

Decompilers. We evaluate three decompilers representing different architectures: **RetDec** (Křoustek et al., 2017) (ML-based, uses LLVM IR), **Ghidra** (National Security Agency, 2019) (rule-based, industry standard), and **Angr** (Shoshitaishvili et al., 2016) (lifting-based, uses VEX IR).

Metrics. We report three metrics corresponding to increasingly strict correctness levels: **L1 Syntax Rate** (percentage of outputs that parse successfully, checked via `gcc -fsyntax-only -w file.c`), **L2 Compile Rate** (percentage that compile successfully, checked via `gcc -c file.c`), and **L3 Re-exec Rate** (percentage that execute correctly on test inputs by comparing output against ground-truth, original binary execution). Re-exec rate (L3) serves as our primary metric since it captures behavioral correctness.

Configuration. We use DeepSeek-V3.2 (Liu et al., 2025) as the base model for each agent for all experiments, with maximum 5 refinement iterations and 10-second execution timeout.

3.2 RQ1: Effectiveness and Scalability

How effective is AGENT4DECOMPILE *at improving decompilation quality, and does it scale to larger benchmarks?*

Table 1 presents results on the 157-binary benchmark across all decompilers and metrics. AGENT4DECOMPILE substantially improves re-executability for all three decompiler architectures: Ghidra improves from 22.3% to 50.3% (+28.0 pp), Angr from 24.8% to 45.2% (+20.4 pp), and RetDec from 25.5% to 43.3% (+17.8 pp). To assess scalability, we evaluate on the full 1,641-binary ExeBench benchmark (Table 2). AGENT4DECOMPILE achieves 40–46% re-executability across all decompilers, with Ghidra performing best (46.0%) followed by Angr (40.9%) and RetDec (39.9%).

Pipeline Effect. Table 1 shows improvements at each constraint level. AGENT4DECOMPILE raises syntax validity (L1) from 87.9% to 99.8% and compilation success (L2) from 67.1% to 99.8%—near-perfect. Most importantly, re-executability (L3) nearly doubles from 24.2% to 46.3% (+22.1 pp). The remaining gap between L2 (99.8%) and L3 (46.3%) reflects the fundamental difficulty of decompilation: producing code that compiles is straightforward, but achieving correct runtime behavior requires resolving semantic errors that are invisible to the compiler. This is precisely why our approach uses L3 execution feedback as the ultimate correctness check.

Optimization Level Robustness. Performance varies by 5–10 pp across optimization levels O0–O3 (Table 2), with O0 achieving the highest rates (45.5% average) and O3 the lowest (37.5%). This gap reflects the increased difficulty of decompiling aggressively optimized code: O3 applies function inlining, loop unrolling, and dead code elimination that obscure original program structure. Nevertheless, AGENT4DECOMPILE achieves consistent improvements across all optimization levels, demonstrating robustness to compiler transformations.

A concrete refinement trace in Appendix D.1 illustrates how L3 feedback turns a compile-correct but behaviorally wrong output into re-executable code.

Finding 1: AGENT4DECOMPILE improves re-executability by 18–28 percentage points across all decompiler types and scales to 1,641 binaries with 40–46% success rate.

3.3 RQ2: Ablation Study

What is the contribution of each constraint level (L1, L2, L3) to the overall system performance?

To understand the contribution of each constraint level, we compare four configurations: Baseline (no refinement), L1 only (syntax checking), L1+L2 (syntax + compile), and Full AGENT4DECOMPILE (L1+L2+L3 with execution). Table 3 shows the results.

Baseline Analysis. Raw decompiler outputs exhibit distinct error profiles. Ghidra achieves 86.0% syntax validity but only 37.6% compilation success due to undefined type annotations (`undefined4`, `bool`), resulting in 22.3% re-executability. RetDec produces highly compilable code (98.7% L1/L2) due to aggressive type normalization, but achieves only 25.5% re-executability—the code compiles but produces incorrect behavior. Angr achieves 79.0% syntax validity and 65.0% compilation, with 24.8% re-executability, reflecting VEX IR artifacts that reduce code quality. All three baselines cluster around 22–25% re-executability, motivating the need for iterative refinement.

Iteration Budget Analysis. Most gains occur within the first two refinement iterations, with diminishing returns thereafter; Appendix D.2 provides the full convergence curve.

The Compile-Correct Gap. L1+L2 achieves 99–100% compile rate but only 32–42% re-executability. This 57–68 pp gap between compilation success and behavioral correctness represents semantic errors invisible to the compiler: code that parses, type-checks, and links, yet produces wrong outputs at runtime. These errors arise from decompiler artifacts (incorrect type casts, wrong operator precedence, misinterpreted control flow) that are syntactically valid but semantically incorrect.

L3 Execution Feedback. Adding L3 constraints improves re-executability by 8.0–10.4 pp (21–33% relative improvement). L3 feedback provides concrete failing test cases—specific inputs where expected and actual outputs diverge—enabling the agent to localize and fix behavioral errors that compilation cannot detect. The remaining gap between L2 (99–100%) and Full AGENT4DECOMPILE (42–50%) reflects the fundamental difficulty of decompilation: some semantic information is irreversibly lost during compilation.

Decompiler-Specific Error Patterns. The three decompilers produce distinct error patterns, yet AGENT4DECOMPILE addresses each through targeted feedback. **Ghidra** preserves internal type annotations (`undefined4`, `bool`) and stack canary artifacts causing L1 failures; L1 feedback guides the agent to replace these with standard C types. **RetDec** achieves high compile rates (L2=98.7%) but collapses all types to `int64_t`, causing subtle L3 failures; execution feedback reveals type mismatches through incorrect outputs, enabling the agent to infer lost types from function names and operation patterns. **Angr** produces VEX IR artifacts (`uint128_t`, `[D]` markers) causing both compilation and behavioral errors; L3 feedback enables iterative correction of these artifacts.

Table 3: Ablation study: contribution of each constraint level. Each cell reports syntax / compile / re-exec rates (%).

Configuration	Ghidra S / C / R	RetDec S / C / R	Angr S / C / R
Baseline	86.0 / 86.0 / 22.3	98.7 / 98.7 / 25.5	79.0 / 65.0 / 24.8
L1 only	100 / 100 / 36.0	100 / 100 / 23.4	100 / 100 / 28.8
L1+L2	100 / 100 / 41.6	100 / 100 / 35.2	100 / 100 / 32.0
AGENT4DECOMPILE	**100 / 100 / 50.4**	**99.4 / 99.4 / 43.2**	**100 / 100 / 42.4**

Table 4: Comparison with refinement baselines on 157-binary benchmark (Ghidra decompiler). All methods use identical decompiler output as input. Re-exec rate (%) reported.

Method	O0	O1	O2	O3	Avg
Ghidra Baseline (No Refinement)	18.9%	17.5%	25.0%	27.5%	22.3%
DeepSeek Single Pass	43.2%	40.0%	35.0%	22.5%	35.2%
LLM4Decompile-6.7B-v2	10.8%	10.0%	12.5%	15.0%	12.1%
SK2Decompile (Simulated)	32.4%	17.5%	20.0%	15.0%	21.0%
AGENT4DECOMPILE (Ours)	**54.1%**	**50.0%**	**52.5%**	**45.0%**	**50.3%**

Finding 2: Compilation success does not imply behavioral correctness: a 58–68 pp gap exists between compile rate (99–100%) and re-executability (32–42%). L3 execution feedback closes part of this gap, adding 8.0–10.4 pp (21–33% relative improvement).

3.4 RQ3: Comparison with Prior Work

How does AGENT4DECOMPILE *compare to existing decompilation refinement approaches?*

Our work refines decompiler output rather than performing end-to-end binary-to-source translation. We compare against three refinement approaches: DeepSeek Single Pass (general-purpose LLM without iteration), LLM4Decompile-6.7B-v2 (Tan et al., 2024) (fine-tuned refinement model), and SK2Decompile (Tan et al., 2025) (two-phase readability-focused refinement). Since the original SK2 model was trained on IDA Pro pseudocode, we simulate SK2Decompile by applying its two-phase prompting strategy with DeepSeek on identical Ghidra output: Phase 1 (Skeleton Extraction) recovers control flow structure, and Phase 2 (Skin Recovery) renames variables for readability. Table 4 shows that AGENT4DECOMPILE achieves 50.3% re-executability, outperforming DeepSeek Single Pass (35.2%) by 15.1 pp, SK2Decompile (21.0%) by 29.3 pp, and LLM4Decompile (12.1%) by 38.2 pp.

The baselines exhibit distinct failure modes. Single-pass refinement lacks error feedback—the model cannot identify which parts of its output are incorrect, so semantic errors that compile but fail at runtime go undetected. LLM4Decompile was trained on isolated pseudo-code without compiler-generated helper functions, causing it to ignore target functions when presented with full Ghidra output; this out-of-distribution sensitivity explains why it performs worse than raw Ghidra (12.1% vs. 22.3%). Although SK2Decompile is also a multi-stage refinement approach, its design objective differs fundamentally from ours: SK2 navigates the trade-off between correctness preservation and readability improvement, rather than directly optimizing for re-executability. Phase 1 recovers program skeleton, and Phase 2 improves human readability by renaming variables—neither phase addresses type errors or semantic bugs. Variable renaming does not improve re-executability, and each LLM call risks introducing code deformations; two sequential calls compound this noise. The result is performance indistinguishable from the unrefined baseline (21.0% vs. 22.3%), demonstrating that readability-oriented refinement provides no benefit for functional correctness.

In contrast, AGENT4DECOMPILE's feedback-based iterative refiner targets re-executability as its optimization goal. Rather than balancing correctness against readability, our agent workflow uses L3 execution feedback as the guiding signal: concrete failing test cases with expected outputs enable the agent to identify *which* parts of the code produce incorrect behavior. Iterative refinement allows incremental correction—the agent does not need to produce a perfect output in one pass, but can progressively fix type mismatches, off-by-one errors, and semantic bugs across iterations. The 29.3 pp gap between AGENT4DECOMPILE (50.3%) and SK2Decompile (21.0%) demonstrates that the choice of optimization objective is decisive: execution feedback drives meaningful semantic corrections, while readability transformations merely introduce noise without improving functional correctness.

Finding 3: Execution-guided refinement outperforms single-pass, fine-tuned, and readability-focused approaches by 15–38 pp. Optimizing for the correct objective (re-executability via test feedback) is essential; readability-oriented transformations provide no benefit for functional correctness.

4 Discussion

Scope of interpretation. Our results should be read in the context of the evaluation setting. Prompts, iteration limits, and timeouts affect refinement behavior, so we use fixed settings and report ablations rather than tuning per benchmark. ExeBench includes real open-source functions, but production binaries may be larger, obfuscated, or inter-procedural. Finally, re-executability is a practical test-based proxy for semantic correctness: passing available tests does not prove equivalence on unseen inputs, although our tests cover common boundary cases and achieve 66% average line coverage.

Remaining failures. Most unresolved cases arise from failures that the refinement loop cannot fully observe or repair. Function inlining can remove the target body before refinement; logic errors can survive compilation and only appear under specific executions; signature mismatches can make otherwise plausible code incompatible with the harness. Residual syntax errors are rare, indicating that L1 repair is largely saturated, while the harder open problem is semantic recovery under incomplete behavioral evidence.

Future directions. Stronger validation signals could improve correctness beyond test-based feedback, including recovered type signatures, dynamic invariants, symbolic constraints, and whole-program context. Extending AGENT4DECOMPILE beyond independent functions would require modeling inter-procedural dependencies and shared state, while deobfuscation preprocessing would broaden applicability to protected binaries. The refinement traces also provide supervised data for future fine-tuning, potentially reducing the number of inference-time repair iterations.

5 Related Work

5.1 Binary Decompilation and Analysis

Binary decompilation has a long history, beginning with foundational work by Cifuentes (Cifuentes, 1994). Modern tools—both commercial (IDA Pro (Hex-Rays, 2018)) and open-source (Ghidra (National Security Agency, 2019), Angr (Shoshitaishvili et al., 2016), RetDec (Křoustek et al., 2017))—employ sophisticated techniques for control flow recovery (Kinder and Veith, 2008), type inference (Lee et al., 2011), and variable recovery (Balakrishnan and Reps, 2007). Recent AI-based approaches frame decompilation as machine translation, using RNNs (Katz et al., 2018), Transformers (Liang et al., 2021), or hybrid methods (Fu et al., 2019). Contrastive pre-training on binary code has also been explored to learn general-purpose binary representations (Zhang et al., 2022). More recent LLM-based systems achieve stronger results: LLM4Decompile (Tan et al., 2024) fine-tunes on decompilation pairs, while SALT4Decompile (Wang et al., 2025b) and SK2Decompile (Tan et al., 2025) use two-phase abstraction. Evaluating decompilation quality has evolved from syntactic similarity (Lacomis et al., 2019) to re-executability, as adopted by recent benchmarks (Tan et al., 2024; Armengol-Estapé et al., 2022).

5.2 LLMs for Code

Code LLMs such as Codex (Chen et al., 2021), CodeLlama (Rozière et al., 2023), and StarCoder (Li et al., 2023) excel at generation and completion, while also demonstrating strong code understanding capabilities including summarization (Ahmed and Devanbu, 2022), documentation (Khan and Uddin, 2022), and bug detection (Pearce et al., 2022). Recent work has further explored training LLMs to comprehend low-level representations such as LLVM IR (Zhang and Leach, 2025), narrowing the gap between source and compiled code. Context-aware and retrieval-augmented techniques have also been applied to optimize code runtime performance (Acharya et al., 2025) and improve decompilation re-executability (Wang et al., 2025a), demonstrating LLMs' potential beyond conventional code generation.

The LLM agent paradigm extends these capabilities through tool use and iterative reasoning. ReAct (Yao et al., 2023) combines reasoning and acting; Reflexion (Shinn et al., 2023) enables learning from mistakes; MetaGPT (Hong et al., 2023) coordinates multi-agent collaboration.

5.3 Program Repair

Automated program repair (APR) techniques fix bugs given failing tests (Le et al., 2016; Gazzola et al., 2019), using generate-and-validate search (Le Goues et al., 2012) or symbolic constraints (Nguyen et al., 2013). LLM-based approaches use conversational feedback (Xia et al., 2023), fine-tuning (Silva et al., 2023), or self-debugging (Chen et al., 2024). Iterative refinement with execution feedback also appears in code gener-

ation: CodeRL (Le et al., 2022) and AlphaCode (Li et al., 2022) filter candidates by test results, while self-repair (Olausson et al., 2023) studies LLM self-correction capabilities.

6 Conclusion

We present AGENT4DECOMPILE, a multi-agent system for constraint-guided decompilation. The core idea is to treat decompilation repair as a multi-level validation problem: L1 catches syntax errors, L2 catches compilation failures, and L3 checks behavioral equivalence against the original binary. Specialized LLM agents use the feedback from each level to iteratively refine decompiler output into re-executable source.

Our evaluation shows consistent gains across decompiler architectures and benchmark scales. AGENT4DECOMPILE improves re-executability by 18–28 percentage points across Ghidra, Angr, and RetDec, reaching 43–50% on the 157-binary benchmark and 40–46% on the 1,641-binary benchmark. It also outperforms prior single-pass and fine-tuned refinement methods by 15–38 percentage points.

The ablation study highlights the main lesson: compilation success is not enough. Compile-only refinement reaches 99–100% compilation but only 32–42% re-executability, while execution feedback closes part of this gap. These results suggest that future decompilation systems should use executable constraints as a central signal, enabling recovered source code to support downstream analysis, modification, and security workflows.

Limitations

Our work has three main limitations in scope, benchmark coverage, and model behavior. First, we focus on single-function decompilation rather than whole-program analysis. ExeBench functions are relatively small and self-contained, whereas real-world binaries involve multi-function interactions, stripped symbols, and complex control flow; extending to full binaries requires handling inter-procedural dependencies and shared state. Our feedback mechanism also requires IO-pair test cases, which limits direct comparison with assertion-based benchmarks (HumanEval-Decompile, MBPP) without format conversion.

Second, several targets are explicitly out of scope. We do not aim to recover exact original source code—variable names, comments, and certain structural choices are irrecoverably lost during compilation. We do not target obfuscated binaries (virtualization, control flow flattening, opaque predicates), which require specialized deobfuscation as preprocessing. Our implementation focuses on x86-64 Linux ELF binaries following recent decompilation literature (Tan et al., 2024, 2025; Lacomis et al., 2019); extension to other architectures (ARM, MIPS) and platforms (Windows PE, macOS Mach-O) is straightforward but not evaluated.

Third, LLMs may exhibit semantic drift, "fixing" code by overfitting to test cases rather than recovering true semantics. We observe this in 75% of failures where code compiles and runs but produces incorrect output. Our L3 execution constraints catch many such errors, but some false positives may remain. Additionally, our vulnerability pattern analysis (34.7% preservation rate) detects code structures rather than exploitability; true vulnerability preservation requires executing actual exploits, though we validated this direction with CGC binaries achieving 96.7% pattern preservation.

References

Manish Acharya, Yifan Zhang, Kevin Leach, and Yu Huang. 2025. Optimizing code runtime performance through context-aware retrieval-augmented generation. *arXiv preprint arXiv:2501.16692*.

Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

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

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

Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–28.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278.

Cristina Cifuentes. 1994. Reverse compilation techniques. In *PhD Thesis, Queensland University of Technology*.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. CODA: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67.

Hex-Rays. 2018. IDA Pro: The interactive disassembler. https://hex-rays.com/ida-pro/.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356.

Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using GPT-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Johannes Kinder and Helmut Veith. 2008. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 423–427.

Jakub Křoustek, Peter Matula, and Avast Software. 2017. RetDec: A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec. Presented at Botconf 2017.

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*.

Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. *IEEE Transactions on Software Engineering*, 42(4):318–339.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. In *IEEE Transactions on Software Engineering*, volume 38, pages 54–72.

JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*.

Jiliang Li, Yifan Zhang, Yu Huang, and Kevin Leach. 2025. Malmixer: Few-shot malware classification with retrieval-augmented semi-supervised learning. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, pages 268–288. IEEE.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Jo ao Monteiro, Oleh Shliazhko, and 48 others. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. 2021. Neutron: An attention-based neural decompiler. In *Proceedings of the 2021 IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

National Security Agency. 2019. Ghidra. https://ghidra-sre.org/. Software Reverse Engineering Framework.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 772–781.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT self-repair for code generation. *arXiv preprint arXiv:2306.09896*.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768.

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

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

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

André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698*.

Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. 2025. Sk2decompile: Llm-based two-phase binary decompilation from skeleton to skin. *arXiv preprint arXiv:2509.22114*.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling binary code with large language models. *Preprint*, arXiv:2403.05286.

Xiaohan Wang, Yuxin Hu, and Kevin Leach. 2025a. Context-guided decompilation: A step towards re-executability. *arXiv preprint arXiv:2511.01763*.

Yongpan Wang, Xin Xu, Xiaojie Zhu, Xiaodong Gu, and Beijun Shen. 2025b. Salt4decompile: Inferring source-level abstract logic tree for llm-based binary decompilation. *arXiv preprint arXiv:2509.14646*.

Cerdic Wei Kit Wong. 2022. American fuzzy lop (afl) fuzzer.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE.

Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 363–376.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*. Internet Society.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Yifan Zhang, Chen Huang, Kevin Cao, Yueke Zhang, Scott Thomas Andersen, Huajie Shao, Kevin Leach, and Yu Huang. 2022. Pre-training representations of binary code using contrastive learning. *arXiv preprint arXiv:2210.05102*.

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

A Refinement Algorithm

B Additional Approach Details

B.1 Scope and Assumptions

We target stripped ELF binaries compiled from C with standard optimization levels (O0–O3). We assume test inputs are available for behavioral validation, either from domain knowledge, fuzzing,

Algorithm 1 Constraint-Guided Refinement

Require: Binary B, test suite T, decompiler D, max iterations N
Ensure: Refined code C^* or FAILURE

```
C ← D(B)                          ▷ Initial decompilation C_0
for k = 1 to N do
(ℓ, E) ← VALIDATE(C, T)           ▷ Returns (level, errors)
if ℓ = PASS then
return C                          ▷ All constraints satisfied
C ← AGENT_ℓ(C, E)                 ▷ LLM repair for level ℓ
end for
return FAILURE

function VALIDATE(C, T)
if ¬φ_1(C) then return (L1, ParseErrors(C))
if ¬φ_2(C) then return (L2, CompileErrors(C))
if ¬φ_3(C, T) then return (L3, FailedTests(C, T))
return (PASS, ∅)
end function
```

or LLM-generated cases. We do not address obfuscated binaries or architectures beyond x86-64; these extensions are left to future work.

B.2 Formal Constraint Definitions

Let $\mathcal{S}$ denote the set of all strings, $\mathcal{C} \subset \mathcal{S}$ the set of syntactically valid C programs, and $\mathcal{B}$ the set of executable binaries. We define the following operators:

- parse : $\mathcal{S} \to \mathcal{C} \cup \{\bot\}$ attempts to parse a string as C code.
- compile : $\mathcal{C} \to \mathcal{B} \cup \{\bot\}$ compiles valid C code to a binary.
- exec : $\mathcal{B} \times \mathcal{I} \to \mathcal{O}$ executes a binary on input $i \in \mathcal{I}$ and returns output $o \in \mathcal{O}$.

Given decompiled code $C \in \mathcal{S}$, the syntax constraint holds if C is parseable:

$$\phi_1(C) \triangleq \text{parse}(C) \neq \bot.$$

The compilation constraint holds if parsed code compiles successfully:

$$\phi_2(C) \triangleq \phi_1(C) \wedge \text{compile}(\text{parse}(C)) \neq \bot.$$

Given a test suite $T = \{(i_1, o_1), \ldots, (i_n, o_n)\}$ derived from the original binary B, the execution constraint holds if the recompiled code produces matching outputs:

$$\phi_3(C, T) \triangleq \phi_2(C) \wedge \forall (i, o) \in T : \text{exec}(\text{compile}(C), i) = o.$$

B.3 Agent Feedback and Iteration

All agents receive prompts with four components: a fixed system instruction, the current constraint status, diagnostic feedback from the failing level, and the complete current code. Each agent returns only corrected C code, without explanations or markdown formatting.

When $\phi_1(C)$ fails, the syntax agent receives GCC parser errors truncated to 500 characters. When $\phi_2(C)$ fails, the compilation agent receives GCC compiler errors truncated to 500 characters. When $\phi_3(C, T)$ fails, the execution agent receives up to two failing test cases, each with input, expected output from the original binary, actual output from the recompiled code, and a short unified diff. A diagnostic checklist prompts the model to consider common behavioral error sources such as incorrect initialization, control-flow divergence, signedness, arithmetic mistakes, and off-by-one errors.

Each agent operates statelessly: it receives only the current error feedback, not a history of past repair attempts. The iteration limit N ensures termination even for difficult cases. We use $N = 7$ as a conservative upper bound; in practice, cases that do not converge within five iterations rarely succeed with additional attempts.

C Implementation Details

C.1 Decompiler Integration

AGENT4DECOMPILE integrates three decompilers to generalize across different architectural approaches. Ghidra 11.2 (National Security Agency, 2019), developed by the NSA, employs rule-based analysis and is invoked through its `analyzeHeadless` command-line interface. Angr 9.2 (Shoshitaishvili et al., 2016) lifts binaries to the VEX intermediate representation and provides a Python API for programmatic access. RetDec 5.0 (Křoustek et al., 2017) combines LLVM-based decompilation with machine learning for type recovery and is accessed via command-line interface.

We abstract these differences behind a unified interface: each decompiler wrapper accepts a binary path and returns decompiled C source code as a

string. This design allows AGENT4DECOMPILE to work with any decompiler that produces C output, enables straightforward addition of new backends, and supports multi-decompiler consensus when comparing outputs from multiple tools.

C.2 Constraint Validators

Each constraint level is implemented as a standalone validator that returns either success or structured error feedback suitable for LLM consumption. For syntax validation (L1), we invoke GCC with the `-fsyntax-only` flag, which performs parsing without full compilation. We capture parser errors from stderr and truncate them to 500 characters before passing them to the repair agent.

For compilation validation (L2), we invoke GCC with strict warning flags (`-Wall -Wextra -Werror`) to catch semantic errors including type mismatches, undeclared identifiers, and linkage failures. For execution validation (L3), we compile the decompiled code, execute it with test inputs, and compare outputs against the original binary. Execution is sandboxed with a 10-second timeout and 256MB memory limit to prevent runaway processes. Expected outputs are pre-computed by running the original binary on the same inputs, establishing ground-truth behavioral oracles.

C.3 LLM Configuration and Infrastructure

While our approach is model-agnostic, we use DeepSeek V3.2 (Liu et al., 2025) as the default LLM for all repair agents because of its strong code understanding capabilities and API availability. We use temperature 0.0 for deterministic, reproducible outputs across runs. We set a maximum output length of 4,096 tokens, increasing the cap to 8,192 tokens for larger functions. API failures due to rate limits or timeouts trigger exponential backoff with up to three retries; persistent failures are logged and the binary is marked as failed for manual inspection.

We process multiple binaries in parallel using Python's Process-PoolExecutor, with each binary running in an independent process to enable linear speedup on multi-core machines. Intermediate results are cached to avoid redundant computation: decompiler outputs are keyed by binary hash, test oracles are stored alongside the binary, and successful refinement results are cached for incremental re-runs. All LLM interactions are logged with full request/response pairs, enabling post-hoc analysis and debugging of refinement failures. For reproducibility, we pin all tool versions (GCC 11.4, Ghidra 11.2, Angr 9.2, RetDec 5.0) and use deterministic LLM decoding.

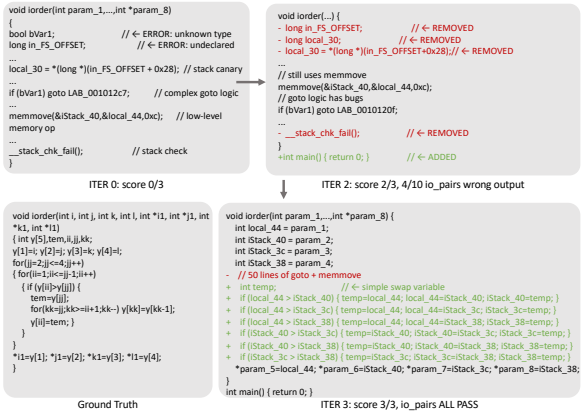

Figure 3: Iterative refinement of `iorder` (4-integer sorting, Ghidra O3). Iter 0 (raw output) fails all tests due to type errors and `goto`-based control flow. Iter 2 compiles but produces wrong output on 4/10 test cases. Iter 3, guided by L3 execution feedback, rewrites the sorting logic and passes all tests.

D Additional Evaluation Details

D.1 Iterative Refinement Example

Figure 3 illustrates the pipeline effect with a concrete example. The function `iorder` sorts four integers, but Ghidra's O3 output contains type errors and convoluted control flow. L1+L2 refinement produces code that compiles (Iter 2), but execution reveals wrong output on 4/10 test cases. Only with L3 feedback—concrete failing I/O pairs—can the agent identify the flawed sorting logic and rewrite it correctly (Iter 3). This demonstrates that execution feedback is essential for catching behavioral errors invisible to compilation.

D.2 Convergence Analysis

Figure 4 shows how re-executability improves with each refinement iteration. All three decompilers exhibit rapid initial improvement: Ghidra jumps from 25% to 40% after a single iteration, while RetDec rises from near-zero baseline to 25%. By iteration 2, performance reaches 40–45% across all decompilers. Subsequent iterations yield diminishing returns, adding only 3–5 pp. This pattern validates our bounded iteration strategy (max 5 iterations): most successful refinements converge quickly, while binaries that fail after 2–3 iterations rarely benefit from additional attempts. The convergence behavior is consistent across decompiler

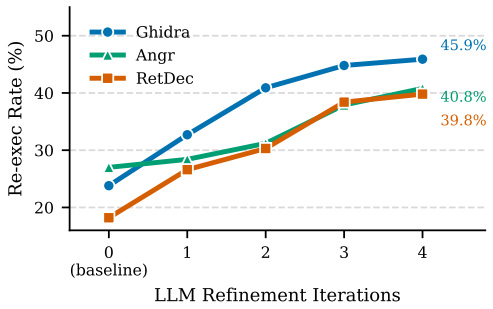

Figure 4: Convergence analysis: re-executability improves with refinement iterations across all decompilers. Most gains occur within the first 2 iterations, with diminishing returns thereafter. Ghidra reaches 45.9%, Angr 40.8%, and RetDec 39.8% after 4 iterations.

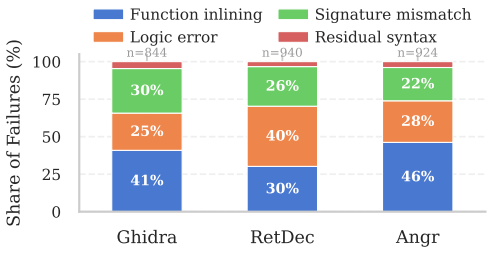

Figure 5: Failure root cause distribution on the 1,641-binary benchmark.

architectures, suggesting that the multi-level constraint hierarchy effectively guides the LLM toward correct solutions within a small number of iterations.

E Extended Discussion

E.1 Evaluation Scope

Our evaluation choices define the scope in which the results should be interpreted. Prompt templates, iteration limits, and timeouts affect refinement behavior; we therefore use fixed settings across experiments and report ablations rather than tuning these choices per benchmark. Benchmark coverage also matters. ExeBench contains functions from real open-source projects, including 247 functions from the `real_test` partition, but production binaries may be larger, obfuscated, or inter-procedural. Finally, re-executability is a practical test-based proxy for semantic correctness: code that passes the available tests may still differ semantically on untested inputs. We mitigate this by generating tests that cover common edge cases such as zero, negative, and boundary values, reaching 66% average line coverage.

E.2 Failure Analysis

Figure 5 decomposes the failures that remain after refinement into four root causes across the three decompilers. The dominant failure mode is function inlining (39%), where compiler optimizations merge the target function into `main()`, leaving no recoverable function body after preprocessing. This represents a fundamental limitation of binary-level analysis that no refinement can overcome. The second category, logic errors (31%), represents cases where refined code compiles and links correctly but produces incorrect output on test inputs, reflecting the limits of LLM-based semantic recovery. Notably, RetDec exhibits a disproportionately high logic error rate (40%) compared to Ghidra (25%) and Angr (28%), likely due to its conservative type strategy that collapses parameter types to `int64_t`; the resulting code compiles trivially but carries systematic type errors that the agent cannot always repair within five iterations. Signature mismatch (26%) occurs when the refined function's prototype differs from the ground-truth header expected by the test harness, causing linker failures despite correct internal logic. Finally, residual syntax errors account for only 4% of failures, confirming that syntax repair is effectively saturated.

E.3 Future Work

Several directions could extend this work. Beyond test-based constraints, future work could incorporate stronger specifications—formal constraints derived from binary analysis, type information from recovered signatures, and program invariants discovered through dynamic analysis—to provide stronger correctness guarantees. Supporting assertion-based benchmarks (HumanEval-Decompile, MBPP) through assertion-to-IO conversion would also enable broader comparison with prior decompilation work.

On the systems side, our current implementation processes functions independently; extending to whole-program decompilation requires handling inter-procedural dependencies and shared state. Integrating deobfuscation techniques as a preprocessing step would extend applicability to protected software. Using open-source models (CodeLlama, DeepSeek-Coder) would eliminate API costs and enable deployment in air-gapped environments, with initial experiments suggesting comparable performance.

Finally, the refinement loop generates valuable training data in the form of binary-to-working-C-code pairs. Fine-tuning on this data could improve single-pass performance and reduce iteration counts, creating a virtuous cycle between inference-time refinement and model improvement.