Bridging the Semantic Gap in Decompilation

Abhishek Manhas, *Department of Mathematics and Computer Science*
Advised by Dr. John Musgrave

Abstract

Decompilation converts executable binaries into readable source-like code, but recovered programs often lose semantic clarity because compiler optimizations remove important information, including structure, type intent, and high-level abstractions. This study explores whether large language models can improve decompiled output through semantic-aware optimization, simplifying control flow and reconstructing higher-level representations while maintaining correctness. A pipeline is developed that combines static decompiler signals with constrained LLM rewriting and lightweight verification checks. Experiments show improvements in readability, abstraction quality, and structural similarity to reference implementations, also highlighting limitations such as hallucinated logic and context sensitivity, motivating a guarded workflow for reliable LLM-assisted reverse engineering.

Development of Decompilers

1970
Allen formalizes control-flow analysis, giving later decompilers a precise graph view of branch structure rather than ad-hoc jump tracing [2].

1973
Kildall introduces global data-flow equations, which became core machinery for optimization and later binary analysis passes used in decompilers [17].

1991
Cytron et al. publish practical SSA construction, improving variable/value reasoning and enabling cleaner lifting and simplification pipelines [9].

1994
Cifuentes publishes *Reverse Compilation Techniques*, a milestone that frames decompilation as a systematic reverse-compilation pipeline [7].

2015
DREAM (*No More Gotos*) shows pattern-independent control-flow structuring, reducing brittle compiler-pattern assumptions in reconstructed code [32].

2019
NSA publicly releases Ghidra at RSA (March 5, 2019), dramatically broadening access to industrial-grade reverse-engineering capabilities [24, 23].

2024
DeGPT and LLM4Decompile demonstrate that LLM-based post-processing can materially improve decompiler output quality when grounded in classical analysis outputs [5, 29].

Constructing a Decompiler

The decompiler was constructed as a multi-stage pipeline that prioritizes structural correctness before improving readability. It combines classical static analysis with controlled post-processing to retain program behavior while reducing low-level artifacts. The approach focuses on separating faithful reconstructions.

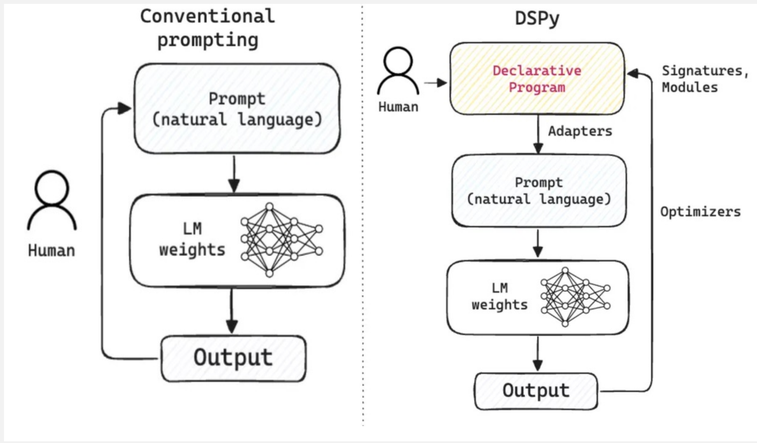
1. A staged decompilation process reconstructs control flow and normalizes syntax from low-level output, ensuring the resulting code is compilable and structurally aligned.

2. A DSPy-guided LLM refinement layer selectively simplifies code, renames variables, and appends comments under strict constraints, with verification checks to preserve semantic correctness.

Decompilation Pipeline:

1. Executable → 2. Disassembly → 3. Control Flow Recovery → 4. IR Reconstruction → 5. High Level Code

DSPys



A DSPy is used to structure the refinement process as a controlled, modular pipeline rather than a single-step transformation. The approach begins with decompiler output C_d, applies an initial refinement to produce C_0, and validates it through compilation. A referee module then generates a decision vector that determines which transformations such as simplification, renaming, or commenting should be applied. Each selected transformation is executed and re-validated, ensuring semantic correctness at every step. This iterative and verified pipeline allows refinement to remain constrained, interpretable, and aligned with the original program behavior.

ML assisted Decompilers

ML-assisted decompilers use large language models to improve the readability and semantic quality of decompiled code by recovering higher-level abstractions and intent [2]. These approaches typically operate as direct transformations over decompiler output, generating more human-readable code but with limited control over intermediate steps [3]. In contrast, the proposed pipeline introduces staged refinement with validation after each transformation, reducing hallucination and preserving structural fidelity. This makes the approach more reliable than prior methods that rely on single-pass generation without explicit verification.

Conclusion

This study investigated a central challenge in reverse engineering: how to recover code that is not only executable, but also understandable, when the original source is unavailable. In many practical settings— malware analysis, firmware inspection, legacy maintenance, and security auditing—analysts receive binaries, not clean source files. One of the main foundations of this work is that decompilation cannot be perfect. Compilation results in a loss of information that makes reverse compilation inherently underdetermined, i.e. multiple high-level programs can correspond to the same low-level representation. For that reason, the goal of this study was not exact source recovery. The goal was to come up with a concept that produces outputs that are structurally faithful and much easier for humans to read.

Referenced Studies

[1] Cristina Cifuentes. “Reverse Compilation Techniques”. PhD thesis. Queensland University of Technology, 1994

[2] Yulong Chai et al. “DeGPT: Optimizing Decompiler Output with Large Language Models”. In: Network and Distributed System Security Symposium (NDSS). 2024. doi: 10.14722/ndss.2024.24401. url:https://www.ndss-symposium.org/ndss-paper/degpt-optimizing-decompiler-output-with-large-language-models/

[3] Hao Tan et al. “LLM4Decompile: Decompiling Binary Code with Large Language Models”. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2024, pp. 4585–4601. doi: 10.18653/v1/2024.emnlp-main.278. url: https://aclanthology.org/2024.emnlp-main.278/