Model-Agnostic Staged Pipeline for LLM Decompilation with Quality Inspection

Anonymous Author(s)
Anonymous Institution(s)
Anonymous Email(s)

***Abstract*—Decompilation supports the maintenance of IoT firmware and the analysis of vulnerabilities in binaries whose source code is unavailable. Recent work increasingly applies large language models (LLMs) to this problem: LLMs can produce human-readable C code from assembly and from the output of existing decompilers. Yet such pipelines typically treat decompilation as a single probabilistic transformation, which leaves little room to judge why a generated function is correct or how far it can be trusted. We propose a staged decompilation pipeline that repairs generated code using function signatures, rule-based quality checks, and compiler diagnostics. Instead of accepting one LLM output as final, the pipeline alternates between inspection and repair grounded in concrete evidence.**

We compare Ghidra, LLM4Decompile, and our pipeline on HumanEval, MBPP, ExeBench, and OpenWrt. Our method improves recompilation rates on every dataset. The best configuration reaches Testpass rates of 71.2% on HumanEval and 60.7% on MBPP; on ExeBench and OpenWrt, the best Testpass rates are 16.1% and 17.0%, despite best Re-compile rates of 97.5% and 83.6%. These gaps show that recompilability and executability are useful intermediate signals, but do not guarantee observed behavioral agreement. We therefore position the pipeline less as a uniform semantic-correctness improvement and more as a diagnostic framework that exposes the stage—recompilation, execution, or Testpass—at which decompilation breaks down. Both DeepSeek-Coder-V2 and gpt-oss-120b improve recompilation over the baselines in the main experiment, and the similar gains across these backends suggest that the framework is largely model-agnostic.

***Index Terms*—decompilation, IoT, software maintenance, reverse engineering, binary analysis, LLM**

I. Introduction

Decompilation underpins much of the maintenance work performed on IoT firmware, especially when source code, build configurations, and dependency libraries are missing or incomplete. Recovering program structure and behavior directly from binaries is often the only way to investigate vulnerabilities or to keep legacy devices running.

In practice, analysts rely on tools such as Ghidra and IDA Pro [1], [2], which present pseudo-C and structural views to guide human reading. These outputs are tuned for comprehension rather than for recompilation, so they cannot in general be rebuilt and run as ordinary source files. A growing body of work therefore applies large language models (LLMs) to the same input: assembly, together with the pseudo-C produced by a classical decompiler [3]–[5]. The result is far more readable, but it is still the output of a single probabilistic generation step. We have no direct way to argue why a particular function is correct, or how far we can trust it; recompilability alone certainly does not imply behavioral equivalence with the original.

This paper introduces a staged decompilation pipeline that repairs LLM outputs against concrete evidence: function signatures, rule-based quality checks, compiler diagnostics, and execution results. A single LLM response is never treated as final; instead, the pipeline iterates between inspection and repair. This design makes it possible to determine how far the generated code can be used and to isolate the stage—recompilation, execution, or Testpass—at which recovery breaks down.

We evaluate recovery in three stages—recompilation rate, executable rate, and Testpass rate—on four datasets: HumanEval, MBPP, ExeBench, and OpenWrt. The pipeline improves recompilation rates with both DeepSeek-Coder-V2 and gpt-oss-120b as backend LLMs. Runtime-behavior agreement improves substantially on the benchmark-style datasets, whereas the results are more limited on datasets derived from real-world code. These results show that recompilability alone does not imply runtime correctness. However, recompilability and executability are useful for identifying the stage at which recovery breaks down and for collecting information that can guide subsequent repair. Similar gains across the two backend LLMs suggest that the pipeline is a general framework rather than an optimization for a specific model.

This study addresses the following two research questions.

- RQ1: To what extent do recompilation rate, executable rate, and Testpass rate function as indicators of successful decompilation?
- RQ2: To what extent does staged repair based on rules and compiler diagnostics improve the quality of decompilation results compared with single probabilistic generation by an LLM?

This paper makes two contributions. First, we propose a staged decompilation pipeline that repairs LLM-generated C code using function signatures, rule-based quality checks, compiler diagnostics, and execution results. Second, through experiments on HumanEval, MBPP, ExeBench, and OpenWrt, we show that the proposed pipeline improves recompilation and executability in many settings, while also showing that high recompilation rates do not necessarily imply high Testpass rates.

Section II describes the pipeline, Section III the experiments, Section IV the discussion, Section V related work, and Section VI concludes.

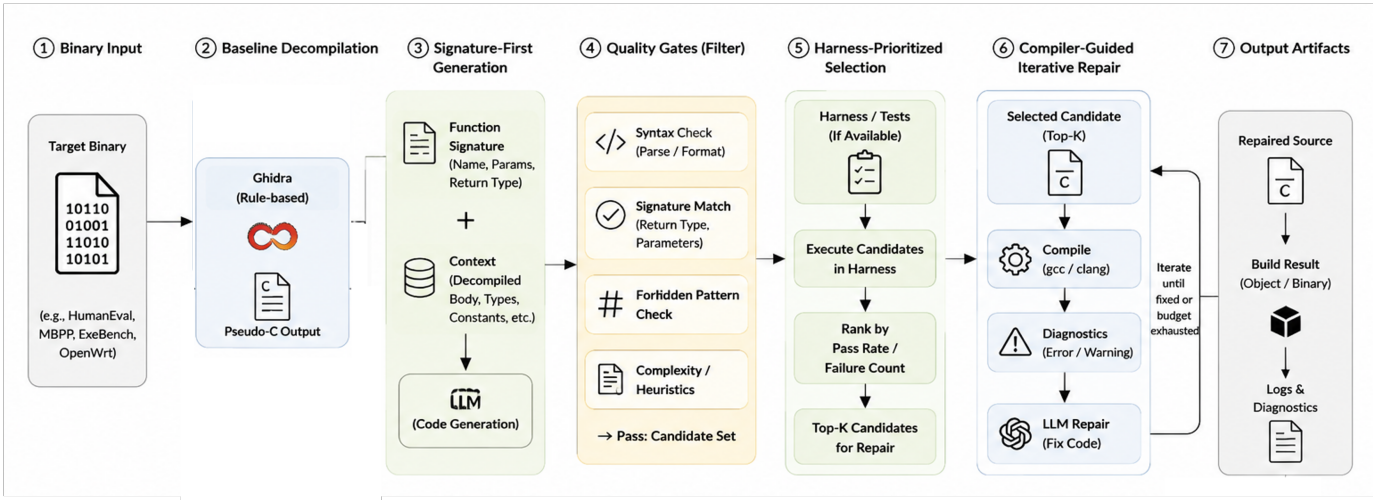

Fig. 1. Overview of the staged LLM-based decompilation pipeline.

II. Staged Decompilation Pipeline

Figure 1 illustrates the pipeline, which generates, inspects, repairs, and finally scores C code reconstructed from a binary. Stages 1 and 2 collect pseudocode from the target. The input is a binary or its assembly listing, and Ghidra is invoked to obtain baseline pseudo-C; when Ghidra produces no usable output, the corresponding field is left empty. Stage 3 performs code recovery in two LLM passes. In the first pass, the LLM is asked to produce a function outline from the assembly, the pseudo-C, and whatever signature information is available; call information from the dataset is provided when present, and for samples without Ghidra output the model relies on assembly and dependency information alone. In the second pass, the outline and the pseudo-C are provided to the LLM, which then generates the full C function.

Stage 4 applies rule-based quality checks. It flags mismatches in return value and argument types, output pointer arguments that are never written, redeclarations of standard library functions or known APIs, empty branches, and bodies that merely return a constant. Stage 5 performs lightweight execution checks and captures the resulting errors. When the dataset provides tests, the generated code is linked against the test driver and executed; otherwise the generated binary and the original binary are fed the same input, and we check whether their return codes, stdout, and stderr agree exactly. Stage 6 then repairs the code using the C source together with the collected error information: syntax errors, type mismatches, undefined symbols, compiler warnings, and similar diagnostics are passed back to the LLM, with at most three repair iterations. Stage 7 re-applies the Stage 4 rule-based checks after the repair loop, then records the final Re-compile, Re-executable, and Testpass values used for evaluation; samples that still fail are reported as-is rather than discarded.

Final quality is judged by three indicators: Re-compile (Re-comp), Re-executable (Re-exe), and Testpass (Test). Re-compile checks whether `gcc -c` can generate an object file from the code. Re-executable checks whether the program can be made runnable, either as a standalone executable or after being linked into a dataset-specific harness. Testpass reports the fraction of samples for which at least one execution case matches; when a program has several cases, one match is enough for the sample to count as a pass. It is therefore a permissive sample-level signal of observed agreement, not a proof of full behavioral equivalence.

For each dataset and method, let N denote the number of evaluated outputs. For row i, let $h_i^c = 1$ if standalone recompilation succeeds, $h_i^b = 1$ if execution succeeds after harness integration or as a standalone executable, and $h_i^t = 1$ if at least one test case matches. Otherwise, each value is set to 0. The Re-compile, Re-executable, and Testpass values in the table are computed as follows.

$$\text{Re-compile} = 100\frac{\sum_i h_i^c}{N},$$
$$\text{Re-executable} = 100\frac{\sum_i h_i^b}{N},$$
$$\text{Testpass} = 100\frac{\sum_i h_i^t}{N}.$$

It is worth noting that the executable rate can exceed the re-compilation rate. Re-compile measures whether the generated code compiles on its own, whereas Re-executable measures whether it can be made to run once a dataset-specific harness, with its wrappers and dependencies, is added.

III. Experiments

A. Overview

Our goal is to analyze the success of LLM decompilation along three axes: whether the generated code recompiles, whether it can be integrated with a harness and executed, and whether its runtime behavior agrees with the original.

The experiments have two parts. A preliminary experiment selects the LLM for use inside the pipeline by measuring several models against each dataset and identifying those with the strongest compilation behavior; it uses smaller samples

because its purpose is to capture broad trends. The main experiment then compares the resulting pipeline with the existing methods, Ghidra and LLM4Decompile.

The comparison covers Ghidra, LLM4Decompile, and our pipeline on four datasets—HumanEval, MBPP, ExeBench, and OpenWrt [6]–[9]. HumanEval and MBPP are suitable benchmark-style datasets because they consist mainly of function-level programming tasks with prepared tests and expected inputs and outputs. ExeBench contains a wider range of executable C functions and dependencies, making it useful for evaluating decompilation under more diverse conditions. OpenWrt is a router-oriented Linux distribution and provides real-system-derived programs, allowing us to observe difficulties caused by external dependencies and execution environments.

B. Preliminary Experiment

The preliminary experiment selects the LLM for use inside the pipeline by measuring each model's standalone behavior on raw assembly input; the pipeline itself is not in the loop at this point. This preliminary experiment is intended as model screening rather than a final comparison of methods; therefore, we focus on broad trends in compilation and execution behavior rather than on exhaustive evaluation. The candidates are Claude Sonnet 4, DeepSeek-Coder-V2, GPT-5, GPT-5.4, GPT-5 nano, and gpt-oss-120b. HumanEval and ExeBench are sampled at 500 instances each, OpenWrt and MBPP use every available sample, and gpt-oss-120b—which is computationally expensive to run—is limited to 100 samples or fewer.

Table I reports the results. Claude Sonnet 4 and DeepSeek-Coder-V2 perform consistently on the benchmark-style datasets: on HumanEval and MBPP, many outputs that reach executability also pass tests. On ExeBench and OpenWrt, however, Testpass rates fall sharply, indicating a substantial gap on data derived from production systems.

The GPT-family models achieve strong recompilation rates but uneven Testpass behavior. GPT-5 stays low on ExeBench and OpenWrt, GPT-5.4 varies widely between datasets, and GPT-5 nano obtains the top OpenWrt Testpass rate of 17.0%.

gpt-oss-120b is more uniform: it reaches 59.0% Testpass on HumanEval, 62.9% on MBPP, and 22.0% on ExeBench, without a sharp drop on any dataset. On HumanEval and MBPP in particular, Testpass is stable at 59.0% and 62.9% respectively. Based on these numbers, the main experiment uses DeepSeek-Coder-V2 and gpt-oss-120b inside the pipeline: DeepSeek-Coder-V2 for its consistent execution behavior on benchmark-style data, and gpt-oss-120b for its leading recompilation rates.

C. Main Experiment

Table II reports the results. The main experiment directly compares Ghidra, LLM4Decompile, and two variants of our pipeline, using DeepSeek-Coder-V2 and gpt-oss-120b as backend LLMs, on the four datasets.

On the benchmark-style datasets HumanEval and MBPP, Ghidra has low scores across every metric. On datasets derived from real-world code, such as ExeBench and OpenWrt, a few recompilations succeed, but executable and Testpass rates stay low; the raw output is hard to use directly for decompilation-based recovery. This should not be interpreted as a limitation of Ghidra itself: the tool is built to support human reverse engineering, not to emit recompilable source. Ghidra remains an important rule-based analysis tool, and we use it here as a baseline for direct reuse of decompiler output rather than as a measure of its usefulness for human analysis. The smaller N for Ghidra reflects the subset of samples for which pseudocode could be generated at all, and ExeBench is excluded from this baseline. On OpenWrt, the nonzero Testpass rate despite a 0.0 Re-executable rate reflects a permissive matching rule: when harness integration fails, the comparison falls back to running the original binary against the same input and counts a sample as a match when return codes, stdout, and stderr all coincide. For a small subset of OpenWrt programs, default error messages produced on both sides happen to agree and are counted toward Testpass; we therefore treat Testpass as a permissive sample-level signal rather than proof of behavioral equivalence.

LLM4Decompile maintains stable recompilation and exe-

TABLE I
LLM MODEL PERFORMANCE IN THE PRELIMINARY EXPERIMENT

Model	Data	N	Re-compile	Re-exec	Testpass
Claude Sonnet 4	HumanEval	500	48.2	86.4	**68.2**
Claude Sonnet 4	MBPP	485	52.2	86.0	**74.6**
Claude Sonnet 4	ExeBench	500	95.4	17.4	10.6
Claude Sonnet 4	OpenWrt	318	85.5	24.5	15.7
DeepSeek-Coder-V2	HumanEval	500	48.6	83.0	**63.4**
DeepSeek-Coder-V2	MBPP	485	51.8	87.8	**75.7**
DeepSeek-Coder-V2	ExeBench	500	93.2	17.2	11.0
DeepSeek-Coder-V2	OpenWrt	318	79.9	24.8	15.7
GPT-5	HumanEval	500	48.2	74.2	**55.0**
GPT-5	MBPP	485	57.1	77.1	**64.5**
GPT-5	ExeBench	500	97.4	16.8	10.0
GPT-5	OpenWrt	318	89.6	24.5	15.7
GPT-5.4	HumanEval	500	49.0	89.4	**83.8**
GPT-5.4	MBPP	485	58.8	57.1	31.1
GPT-5.4	ExeBench	500	80.4	11.2	5.6
GPT-5.4	OpenWrt	318	49.1	23.3	10.7
GPT-5 nano	HumanEval	500	50.0	61.6	26.8
GPT-5 nano	MBPP	485	56.5	62.7	32.4
GPT-5 nano	ExeBench	500	95.2	13.8	7.8
GPT-5 nano	OpenWrt	318	87.1	24.8	17.0
gpt-oss-120b	HumanEval	100	63.0	78.0	**59.0**
gpt-oss-120b	MBPP	89	56.2	73.0	**62.9**
gpt-oss-120b	ExeBench	100	100.0	33.0	22.0
gpt-oss-120b	OpenWrt	100	91.0	16.0	14.0

TABLE II
SUCCESS RATES BY METHOD AND EVALUATION METRIC ON FOUR DATASETS

Data	Method	N	Re-compile	Re-exec	Testpass
HumanEval	Ghidra	1267	0.9	0.0	0.0
	LLM4Decompile	1312	32.5	35.5	6.5
	Proposed+DeepSeek-Coder-V2	1312	57.2	89.1	**71.2**
	Proposed+gpt-oss-120b	1312	60.4	75.3	**55.4**
MBPP	Ghidra	7474	1.1	0.0	0.0
	LLM4Decompile	7777	38.5	33.6	7.8
	Proposed+DeepSeek-Coder-V2	7777	73.3	52.4	6.8
	Proposed+gpt-oss-120b	7777	59.3	75.4	**60.7**
ExeBench	LLM4Decompile	7044	19.3	8.9	6.7
	Proposed+DeepSeek-Coder-V2	7044	95.9	23.6	**11.1**
	Proposed+gpt-oss-120b	7044	97.5	26.6	**16.1**
OpenWrt	Ghidra	181	25.4	0.0	5.0
	LLM4Decompile	318	3.8	0.3	1.3
	Proposed+DeepSeek-Coder-V2	318	78.6	35.5	**16.4**
	Proposed+gpt-oss-120b	318	83.6	11.6	**17.0**

cutable rates everywhere except OpenWrt, but its Testpass rate is consistently low. The model produces plausible, executable C, yet its fidelity is limited when evaluated by whether the generated code passes tests. On OpenWrt every metric drops, highlighting the difficulty of real-system-derived data. Where the executable rate exceeds the recompilation rate, the dataset-provided harness supplies dependency code or type definitions that the standalone output was missing.

Our pipeline with DeepSeek-Coder-V2 performs particularly well on HumanEval. Again the executable rate exceeds the recompilation rate, meaning that some samples fail to compile in isolation but compile and run once the dataset harness is added. This gap reflects the evaluation setting: the harness can supply wrappers, declarations, or dependencies that are absent from the standalone generated file. The Testpass rate reaches 71.2%, the best of any configuration, indicating that the recovered code is faithful enough to pass the tests. Performance on MBPP, by contrast, drops sharply.

MBPP contains programs with intricate algorithms, such as shortest-cost path search, and recovering such computations is difficult. ExeBench shows the opposite asymmetry: recompilation is very high, but executable and Testpass rates stay low, so the method often stops at code that appears syntactically plausible. ExeBench is a dataset derived from real-world code, and its diversity contributes to this difficulty. OpenWrt is a more focused router corpus, and its executable and Testpass rates are higher than on ExeBench, though the effect on truly complex real-system programs is still modest. These results suggest that external dependencies, missing type definitions, and dataset-specific execution environments become major obstacles once the target moves beyond benchmark-style functions.

gpt-oss-120b is more stable across datasets. On MBPP and ExeBench, it achieves higher executable and Testpass rates than DeepSeek-Coder-V2; on OpenWrt, it improves Testpass but not executability. The executable rate on OpenWrt nevertheless remains low, and again, complex real-system programs are not fully recovered.

It is also useful to compare the main results with the standalone DeepSeek-Coder-V2 and gpt-oss-120b runs from the preliminary experiment, noting that the two settings evaluate different sample counts. For DeepSeek-Coder-V2, standalone Testpass on HumanEval is 63.4% versus 71.2% with our pipeline, and on OpenWrt 15.7% versus 16.4%. MBPP shows the opposite pattern: 75.7% standalone (485 samples) against 6.8% with the pipeline on the full 7777 samples. Inside the pipeline, Re-compile rises from 51.8% to 73.3%, but Re-executable falls from 87.8% to 52.4%, indicating that the bottleneck shifts from syntax to harness integration or execution. One plausible explanation is that signature- and rule-driven repair creates interface mismatches against the MBPP harness, which in turn suppresses Testpass; this points to compiler diagnostics and rule checks being insufficient on their own, and motivates incorporating test-failure feedback into the repair loop. gpt-oss-120b is more stable. On HumanEval, Testpass moves from 59.0% to 55.4%, and on MBPP from 62.9% to 60.7%—no large decline. ExeBench shifts from 22.0% to 16.1% and OpenWrt from 14.0% to 17.0%. Taken together, these results suggest that scaling behavior depends strongly on the backend model and dataset: gpt-oss-120b degrades moderately, whereas DeepSeek-Coder-V2 drops sharply on MBPP.

IV. Discussion

A. RQ1: How well do recompilation rate, executable rate, and Testpass rate explain successful decompilation?

We treat decompilation as successful when the generated code shows the same observable behavior as the original; Re-compile, Re-executable, and Testpass are the three stages on the way there. Re-compile is the first check: whether the code is valid C. The experiments make clear that a high Re-compile rate alone does not explain final behavior: Proposed+gpt-oss-120b reaches Re-compile 97.5% on ExeBench and 83.6% on OpenWrt, yet Testpass stays at 16.1% and 17.0% respectively. Compilable code, then, is not the same as code that behaves like the original.

Re-executable asks the next question—whether the code can be integrated into a callable program. On HumanEval, Proposed+DeepSeek-Coder-V2 reaches Re-executable 89.1% and Testpass 71.2%, and on MBPP Proposed+gpt-oss-120b reaches Re-executable 75.4% and Testpass 60.7%, so Re-executable is a meaningful prerequisite for Testpass on benchmark-style data; it still does not guarantee correctness, because execution output must also match the original. Beyond their individual measurements, Re-compile and Re-executable provide the compilation and execution information that drives the iterative repair stage of the pipeline.

Of the three, Testpass comes closest to a direct measure of success: it checks behavioral agreement with the original on evaluation inputs. However, it says nothing about code structure, readability, or generalization to unseen inputs, so additional metrics will be needed to fully characterize the quality of recovered code.

Answer to RQ1: Testpass is the metric closest to successful decompilation, while Re-compile and Re-executable are useful intermediate signals for locating where a recovery breaks down before it reaches Testpass.

B. RQ2: Does the proposed staged repair outperform single probabilistic generation?

Staged repair most consistently improves recompilation, while its effect on final Testpass is model- and dataset-dependent. The gain is already visible at the recompilation stage, which is consistent with the view that iteratively feeding compiler errors back into the model helps the output gain syntactic validity. However, staged repair does not automatically lead to a high Testpass rate: on MBPP, Proposed+DeepSeek-Coder-V2 records Re-compile 73.3% against Testpass 6.8%, so complex algorithms can compile cleanly while still missing their semantics. Closing that semantic gap will require more than compiler diagnostics—signals such as test failures need to enter the loop. Overall, staged repair is useful for improving syntactic validity and can improve Testpass, but the current

evidence does not support treating it as a uniform semantic improvement over single-pass generation.

Answer to RQ2: Staged repair improves recompilability and can improve Testpass, but semantic accuracy remains limited by the available repair signals and varies across models and datasets.

C. Limitations

Even a passing Testpass score does not show whether the internal structure of the generated code mirrors that of the original. For maintenance tasks like vulnerability detection through reverse engineering, recovering the original control and data flow faithfully often matters more than producing clean, readable code. If the LLM rewrites the program as a semantically equivalent but structurally different implementation, branches, memory operations, or other vulnerability-relevant constructs can be abstracted away. Testpass in this study therefore captures behavioral agreement on observed inputs, but it does not by itself guarantee the structural fidelity that maintenance or vulnerability analysis demands.

V. Related Work

Decompilation has a long history, running from the early staged machine-language work [10]–[12] through type recovery, control-flow structuring, IR lifting, sound C recovery, static-analysis-oriented frameworks, and broader surveys of disassembly and binary lifting [13]–[20]. Practical tools like Ghidra [1] and IDA Pro [2] are built around pseudo-C for human reading, whereas a separate machine-learning line tries to recover the semantic information that compilation strips away: types, names, signatures, data structures [21]–[26], with signature recovery being most directly relevant to our signature-first generation [25]. This line of work addresses the difficulty of recovering source-level information from binaries [27], including type inference for machine code [28] and recovery of variables and data structures from stripped binaries [29]. Neural and LLM-based decompilation has moved from early RNN approaches [30] to LLM4Decompile and SLaDe [3], [4], and more recent work sharpens decompiler outputs through LLM augmentation, fine-tuning, pseudo-C repair, two-phase generation, reverse-engineering copilots, and structure-aware decompilation [5], [31]–[35]. Partially recompilable decompilation for vulnerability mitigation ties decompiler outputs to maintenance directly [36]; our pipeline takes a different route by combining signatures, quality checks, compiler diagnostics, and execution results inside a staged repair loop. Because code-generation evaluation has steadily moved past surface similarity toward structural and execution-oriented metrics [37]–[39], we adopt a three-stage evaluation on datasets including ExeBench and OpenWrt [8], [9], and show that recompilability does not imply equivalence to the original behavior. Unlike prior LLM-augmented decompilation work that focuses on improving a single output or repair signal, our study combines signatures, rule-based quality gates, compiler diagnostics, and execution feedback in a staged loop, and evaluates where recovery fails through Re-compile, Re-executable, and Testpass.

VI. Conclusion

This paper examined decompilation in the context of software maintenance tasks such as vulnerability detection and proposed a staged LLM decompilation pipeline. Instead of taking a single LLM response as final, the method repairs the output in stages, using function signatures, quality checks, compilation results, and execution results as feedback. The method turns one-shot probabilistic generation into a loop of evidence-driven inspection and repair.

In our experiments we compared Ghidra, LLM4Decompile, and the proposed method with DeepSeek-Coder-V2 and gpt-oss-120b on HumanEval, MBPP, ExeBench, and OpenWrt, measuring recompilation rate, executable rate, and Testpass rate for each. The pipeline consistently improves recompilation, and improves Re-executable and Testpass on HumanEval and on the gpt-oss-120b variant of MBPP, but Testpass remains limited on ExeBench and OpenWrt even when recompilation is high. We therefore frame the contribution less as a uniform improvement in semantic correctness and more as a diagnostic framework: Re-compile, Re-executable, and Testpass together expose the stage at which LLM-based decompilation breaks down, and the staged repair loop sometimes—but not always—closes the gap. Testpass is the closest available metric for observed behavioral agreement, while Re-compile and Re-executable remain valuable because they reveal where recovery fails and supply inputs to the repair loop.

Future work should improve executable and Testpass rates on real-data-derived programs, ideally by adding new repair signals—output diffs and runtime errors from failing tests, not just compilation results. Because Testpass only certifies behavior on the evaluation inputs at hand, evaluating untested inputs and assessing code structure remain open problems.

Disclosure of AI Use OpenAI ChatGPT, Codex, Claude, and Gemini were used as aids for building experimental scripts, for wording adjustments and proofreading, and for formatting figures and tables. The authors verified the resulting text, figures, tables, code, and numerical results, and take responsibility for the final content.

REFERENCES

[1] National Security Agency, "Ghidra software reverse engineering framework," https://ghidra-sre.org/, 2019.

[2] Hex-Rays SA, "IDA Pro," https://hex-rays.com/ida-pro/, 2024.

[3] H. Tan, Q. Luo, J. Li, and Y. Zhang, "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. 3473–3487.

[4] 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*. IEEE, 2024, pp. 67–80.

[5] 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.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[7] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[8] OpenWrt Project, "OpenWrt Wireless Freedom," https://openwrt.org/, 2004.

[9] 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*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 50–59.

[10] B. C. Housel and M. H. Halstead, "A methodology for machine language decompilation," in *Proceedings of the 1974 annual conference on - ACM 74*. ACM Press, 1974, pp. 254–260.

[11] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[12] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE Computer Society, 1998, pp. 228–237.

[13] A. Mycroft, "Type-based decompilation (or program reconstruction via type reconstruction)," in *Programming Languages and Systems*. Springer Berlin Heidelberg, 1999, pp. 208–223.

[14] C. Cifuentes, "Structuring decompiled graphs," in *Compiler Construction*. Springer Berlin Heidelberg, 1996, pp. 91–105.

[15] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[16] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 295–308.

[17] F. Verbeek, P. Olivier, and B. Ravindran, "Sound C code decompilation for a subset of x86-64 binaries," in *Software Engineering and Formal Methods*. Springer-Verlag, 2020, pp. 247–264.

[18] K. Yakdan, S. Eschweiler, and E. Gerhards-Padilla, "Recompile: A decompilation framework for static analysis of binaries," in *2013 8th International Conference on Malicious and Unwanted Software: "The Americas"*. IEEE, 2013, pp. 95–102.

[19] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 833–851.

[20] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 IEEE Symposium on Security and Privacy*. IEEE, 2022, pp. 1100–1119.

[21] J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–35, 2016.

[22] L. Dramko, J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, G. Neubig, B. Vasilescu, and C. Le Goues, "Dire and its data: Neural decompiled variable renamings with respect to software class," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–34, 2023.

[23] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 1631–1645.

[24] J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, "Hext5: Unified pre-training for stripped binary code information inference," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2023, pp. 774–786.

[25] H. Huang, Y. Liu, Y. Cheng, H. Wei, J. Liu, Y. Wang, and L. Wang, "Recover function signature from combined constraints," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2025, pp. 3386–3400.

[26] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2024, pp. 4554–4568.

[27] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 24–35.

[28] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 27–41.

[29] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 813–832.

[30] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 346–356.

[31] M. Zou, H. Cai, H. Wu, Z. L. Basque, A. Khan, Z. B. Celik, D. Tian, A. Bianchi, R. Wang, and D. Xu, "D-lift: Improving llm-based decompiler backend via code quality-driven fine-tuning," *CoRR*, vol. abs/2506.10125, 2025.

[32] G. Li, X. Shang, S. Cheng, J. Zhang, L. Hu, X. Zhu, W. Zhang, and N. Yu, "Pseudofix: Refactoring distorted structures in decompiled c pseudocode," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2025, pp. 841–853.

[33] 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," *CoRR*, vol. abs/2509.22114, 2025.

[34] G. Chen, H. Sun, D. Liu, Z. Wang, Q. Wang, B. Yin, L. Liu, and L. Ying, "Recopilot: Reverse engineering copilot in binary analysis," *CoRR*, vol. abs/2505.16366, 2025.

[35] Y. G. Achamyeleh, H. Thomare, and M. A. A. Faruque, "HELIOS: hierarchical graph abstraction for structure-aware LLM decompilation," *CoRR*, vol. abs/2601.14598, 2026.

[36] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, "Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2270–2282, 2025.

[37] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a method for automatic evaluation of code synthesis," 2020, accessed: Jan. 6, 2026. [Online]. Available: https://arxiv.org/abs/2009.10297

[38] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: How should we assess quality of the code generation models?" *Journal of Systems and Software*, vol. 203, no. C, 2023.

[39] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, 2025.