LLMs FOR BINARY DECOMPILATION: CAN LARGE LANGUAGE MODELS RECONSTRUCT SOURCE CODE FROM COMPILED BINARIES?

Anonymous authors
Paper under double-blind review

ABSTRACT

Software decompilation in which experts reconstruct the original source code from a compiled binary file is essential for software security analysis. Given the exponential growth in coding abilities of Large Language Models (LLMs), there is growing interest in using them for decompilation. This growth has been fueled by advancements in reinforcement learning (RL) and test-time scaling. However, whether these strategies can help LLMs in reconstructing source-like code from compiled binaries remains under-investigated. To explore this question, we first present DECOMPBENCH, a dataset of 660k paired decompiled and source functions from over 2k C/C++ projects in the Gentoo package tree. We use the training split of this dataset for supervised fine-tuning, followed by RL with a reward based on control-flow graph edit distance (GED). On held-out test packages, the RL-trained model achieves better structural reconstruction quality than supervised fine-tuning. Further analysis on test-time scaling shows that sampling from SFT achieves lower oracle minimum GED at larger sampling budgets. Under a fixed candidate budget, we compare single-checkpoint and cross-checkpoint sampling using a learned selector. Cross-checkpoint sampling produces better reconstructions relative to single-checkpoint sampling, but learned selection captures only part of the improvement available in the candidate pools. We observe that better individual reconstructions do not necessarily yield better test-time scaling, and highlight the importance of evaluating candidate generation and selection separately. We release our code, data, and trained models to support future research.

1 INTRODUCTION

Decompilation is the task of recovering a high-level program representation from a compiled binary (Brumley et al., 2013; Burk et al., 2022). It is a fundamental capability for software security experts analyzing malware, proprietary binaries, vulnerable legacy systems, and patched executables without access to their source code (Burk et al., 2022; Wong et al., 2023). However, compilation is not a lossless translation: source-level names, types, and abstractions can be discarded, while programmer-intended structure itself can be modified (Chen et al., 2022). Existing decompilers translate binaries into C-like code (Shoshitaishvili et al., 2016; Hex-Rays SA, 2026; Basque et al., 2024; National Security Agency, 2023), but the resulting representation can differ substantially from the original source. Unlike generating a new program from a specification, decompilation refinement requires inferring source-level structure that is no longer explicit in the input, while remaining faithful to an existing program. At the same time, the recent progress of large language models (LLMs) in software development (Yang et al., 2024; Deng et al., 2025) presents a natural opportunity: models that can read and write code at scale may be able to recover what a decompiler fails to restore. This motivates studying whether LLMs can refine decompiler output into source-like code that more closely reflects the structure of the original program.

Much of the progress in coding abilities of these models can be attributed to reinforcement learning (RL). In particular, reinforcement learning with verifiable rewards (RLVR) enables models to learn from outcomes such as generated programs passing executable tests (Lambert et al., 2024), and methods such as Group Relative Policy Optimization (GRPO) provide a practical mechanism for optimizing this feedback (Shao et al., 2024). These methods primarily improve the quality of a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

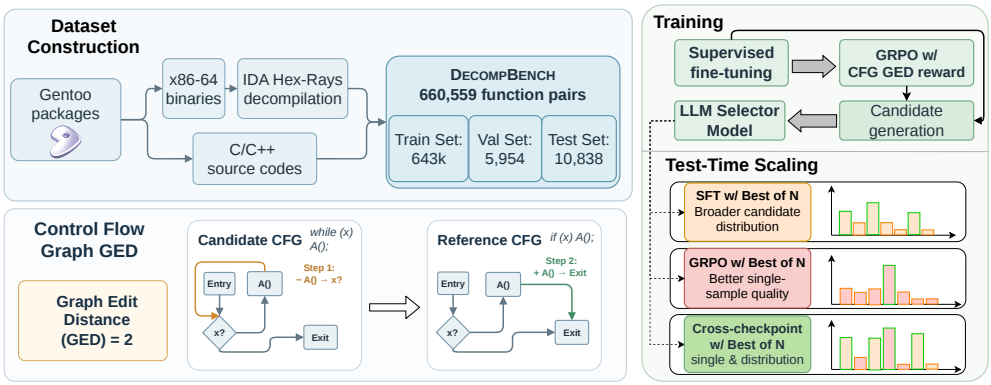


Figure 1: Overview of our decompilation refinement framework. We construct DECOMPBENCH, a dataset of 660k paired source and decompiled functions from 2,032 Gentoo packages. We supervised fine-tune LLMs to reconstruct source-like code and subsequently optimize them with GRPO using a control-flow graph edit-distance (GED) reward. At test time, we combine SFT's broader candidate diversity with GRPO's stronger single-sample quality through cross-checkpoint sampling.

single generation. Yet with repeated sampling, a model can often solve problems it fails to solve in a single attempt (Brown et al., 2024). Prior work has reported that RL can increase the probability of solving a problem in a single attempt while reducing the range of problems solved through repeated sampling (Chen et al., 2026). This motivates two questions for decompilation refinement: *how effectively can RL improve structural reconstruction*, and *do those improvements persist as the inference budget increases?*

To investigate these questions, we introduce DECOMPBENCH, a dataset of 660,559 paired decompiled and source functions from 2,032 packages spanning 8,722 binaries from the Gentoo package tree (Gentoo, 2026). We decompile them using Hex-Rays decompiler (Hex-Rays SA, 2026) in IDA Pro 9.2. We split the dataset by package name, assigning all versions of each package to the same split to prevent overlap between training and evaluation packages. The binaries were compiled at `-O0`, and debug information is available to the decompiler. DECOMPBENCH therefore evaluates debug-assisted decompilation refinement rather than source recovery from fully stripped binaries.

We first perform supervised fine-tuning (SFT) on Qwen (Yang et al., 2025) and Llama (Grattafiori et al., 2024) models on the paired functions, then apply GRPO with a reward derived from the graph edit distance (GED) between the control-flow graphs of generated and reference functions (Sanfeliu & Fu, 1983; Gao et al., 2010). This reward provides graded feedback on structural reconstruction: lower GED indicates closer agreement in control-flow organization. A distance of zero denotes a structural match under the metric, not exact source recovery or functional equivalence. On held-out test packages, GRPO reduces mean GED over successfully scored outputs from 7.23 to 5.53 for Qwen and from 9.30 to 6.03 for Llama, relative to SFT. These improvements are accompanied by modest increases in the rate of zero-GED reconstructions.

We next study test-time scaling under a fixed budget of 20 candidates per function. On the validation set, GRPO produces better individual samples, but SFT benefits more from repeated sampling and achieves lower oracle minimum GED starting from a budget of five candidates in both model families (see Fig 3). We compare these single-checkpoint strategies with cross-checkpoint sampling and use a learned selector to choose a final reconstruction without access to the reference source. This separates the quality available within a candidate pool from the quality that can be identified at inference time. On the held-out test set, cross-checkpoint sampling achieves lower failure-penalized mean GED than either single-checkpoint sampling strategy.

Finally, we evaluate whether refinement learned from IDA outputs transfers to other decompilers. On outputs from Ghidra and angr_sailr, the evaluated cross-checkpoint strategy increases the rate of zero-GED reconstructions but also increases mean GED, revealing limitations in cross-decompiler generalization. These findings underscore the need to assess both structural improvements and regressions when evaluating decompilation refinement. We release DECOMPBENCH, our code, trained models, and analysis to support further research.

2 Related Work

Decompilation and Source Recovery. Traditional decompilers recover high-level program representations from compiled binaries using disassembly, control-flow recovery, data-flow analysis, type inference, and control-flow structuring (National Security Agency, 2023; Shoshitaishvili et al., 2016). Prior work improves individual components of this process, including structured control-flow recovery (Brumley et al., 2013; Yakdan et al., 2015), type inference (Noonan et al., 2016; Chen et al., 2022), and variable-name recovery (He et al., 2018; Pal et al., 2024). Learning-based approaches further explore reconstruction from binaries and intermediate representations (Liang et al., 2021; Fu et al., 2021). Recent LLM-based methods include both direct decompilation and refinement of existing decompiler outputs. LLM4Decompile supports both settings, including models fine-tuned to refine Ghidra output (Tan et al., 2024), while Wong et al. (2023) investigate refinement aimed at making decompiled code recompilable. SK2Decompile separates structure recovery from identifier naming and applies RL objectives to both stages (Li et al., 2026). Our work builds on this literature by studying structural refinement under a debug-assisted decompilation setting, with particular emphasis on how CFG-based RL affects reconstruction quality and test-time sampling.

Reinforcement Learning for Coding Tasks. Reinforcement learning with verifiable rewards enables language models to learn from automatically evaluated outcomes rather than solely from supervised demonstrations (Lambert et al., 2024). In coding tasks, compilation and test-case execution can provide feedback for optimizing task success (Guo et al., 2025). Our work uses GRPO (Shao et al., 2024) to optimize a different target: agreement between the control-flow graphs of generated and reference functions. The resulting GED-based reward provides graded feedback on structural reconstruction, rather than verifying functional correctness. We therefore evaluate improvements in control-flow recovery without treating it as evidence of semantic equivalence. By comparing supervised fine-tuning with subsequent RL, we investigate how optimizing this structural objective affects both individual reconstructions and the candidate pools available through repeated sampling.

Test-Time Scaling. Test-time scaling allocates additional inference computation to reasoning and candidate exploration. Chain-of-thought prompting and its variants encourage intermediate reasoning (Wei et al., 2022; Kojima et al., 2022), while self-consistency samples multiple reasoning paths and aggregates their answers through majority voting (Wang et al., 2022). Structured search and sampling methods, including tree-of-thought (Yao et al., 2023), Monte Carlo tree search (Zhang et al., 2024), REBASE (Wu et al., 2024), and GuidedSampling (Handa et al., 2026), explore alternative ways to use this additional computation. Ghosal et al. (2025) show that independently sampled reasoning paths with majority voting can outperform extended reasoning traces under the same inference budget. Relatedly, Chen et al. (2026) report that while RL-trained models can outperform their base models at small sampling budgets, they achieve lower pass@k at larger budgets. We investigate this relationship in decompilation refinement by comparing single-checkpoint and cross-checkpoint sampling under a fixed candidate budget. Our evaluation distinguishes oracle candidate-pool quality, measured by $\mathrm{minGED}\ @k$, from the quality of the final reconstruction chosen by a learned selector without access to the ground truth reference source. This separates improvements in candidate generation from improvements that can be realized through selection at inference time.

3 Method

3.1 Preliminaries

Decompilation and structural representation. We study decompilation refinement, in which an LLM transforms the output of an existing decompiler into source-like code. Given a decompiled function f_{dec}, the model generates a reconstruction $\hat{f}$, while the corresponding original function f_{src} is the reference. Our objective is to recover source-like program structure rather than reproduce the original source text exactly. We characterize this structure using a control-flow graph (CFG), whose nodes represent basic blocks and whose directed edges represent possible transfers of control. CFGs capture branching and looping structure while abstracting away surface-level differences such as identifier names and formatting, making them useful for structural analysis of decompiled programs (Brumley et al., 2013; Basque et al., 2024).

Structural evaluation. Following prior work (Basque et al., 2024), we evaluate structural reconstruction by comparing the CFGs of $\hat{f}$ and f_{src}. We extract CFGs using Joern through `pyjoern` (Joern) and compute their graph edit distance using `vj_ged` from `cfgutils` (Basque, 2026). We denote the resulting distance by $d_{\text{CFG}}(\hat{f}, f_{\text{src}})$. Node-matching costs depend on local connectivity rather than exact source text, so lower distances indicate closer agreement in control-flow organization. A distance of zero denotes a structural match under this metric, not recovery of the original source text or verification of functional equivalence. In particular, the metric does not establish the correctness of data flow, constants, or expressions. We therefore interpret improvements in d_{CFG} specifically as improvements in structural reconstruction, rather than exact source recovery.

3.2 DECOMPBENCH

We construct DECOMPBENCH, a dataset of paired decompiled and source functions for training and evaluating decompilation refinement. Each example consists of a decompiled function f_{dec} and its corresponding source function f_{src}.

Data sources and decompilation setting. We build on the x86-64 C/C++ binaries and associated source archives released with VarBERT (Pal et al., 2024). The binaries were compiled at `-O0`. Rather than using VarBERT's function-level dataset, which targets variable-name recovery, we reconstruct paired decompiled and source functions from the released artifacts. We regenerate all decompiler outputs using the Hex-Rays decompiler (Hex-Rays SA, 2026) in IDA Pro 9.2. The binaries retain debug information, and IDA has unrestricted access to this metadata. Consequently, the decompiled inputs already contain function names, parameter names, and structure types recovered from debug information. DECOMPBENCH therefore evaluates refinement in a debug-assisted setting rather than source recovery from fully stripped binaries.

Constructing function pairs. We match each binary to its corresponding source archive using package and version identifiers. Debug metadata is then used to identify the relevant source files within each archive, and binaries for which the corresponding source files are missing or ambiguous are excluded. We decompile each remaining binary and discard unsuccessful decompilation runs.

For each successfully decompiled binary, we extract functions from both the decompiler output and matched source files, pairing them by exact, case-sensitive function name. We exclude automatically generated IDA names, which account for 2% of extracted functions, and names associated with multiple distinct function bodies on either side. The resulting dataset therefore covers functions that can be matched unambiguously by name, rather than every function in each binary file. Finally, we remove comments, apply a consistent formatting style to both sides, and remove exact duplicate pairs to reduce superficial differences unrelated to program reconstruction.

Dataset characteristics. Figure 2 characterizes both the function lengths and pre-refinement structural distance. Median decompiled and source lengths are 178 and 136 tokens respectively in the training set and 149 and 143 tokens respectively in the test set. Function lengths are heavily skewed: 1% of training functions exceed 3,749 tokens, and the longest reaches 163,003 tokens. The longest 1% of training functions account for 18% of all tokens, while the longest 10% account for 52%. Decompiled functions are longer than their source counterparts in 71.8% of pairs, and the two lengths are strongly correlated (Spearman $\rho = 0.92$). This reflects how software is commonly written, many short functions and occasionally very large ones (Concas et al., 2007; Hatton, 2008).

We also measure the pre-refinement structural distance $d_{\text{CFG}}(f_{\text{dec}}, f_{\text{src}})$, which characterizes how closely the decompiler's output already matches the reference control-flow structure. Figure 2 reports zero distance for 48.5% of training pairs and 60.9% of test pairs. The substantial fraction of zero-distance inputs highlights the importance of preserving structure already recovered by the decompiler, alongside improving reconstructions that differ from the reference, without any regression.

Splits. We split the dataset by package name after removing version information, assigning all versions of a package to the same split. This prevents different versions of the same package from appearing across training and evaluation sets. The training set contains 643,767 function pairs from 1,972 package groups; the validation set contains 5,954 pairs from 30 groups; and the held-out test

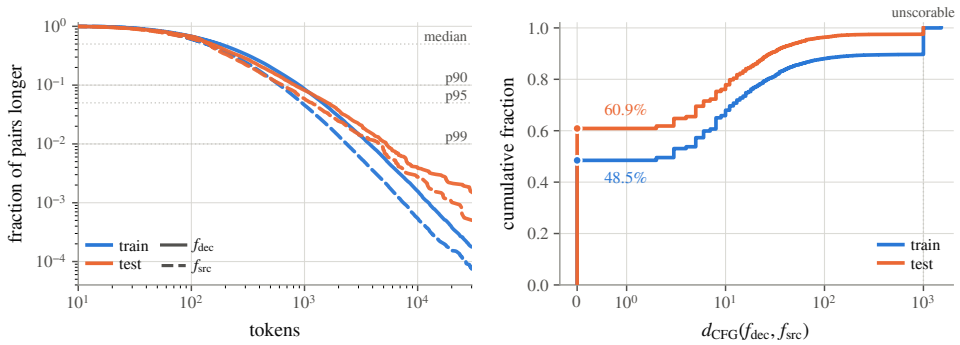


Figure 2: **Left:** Illustrates how long the functions are. Each curve gives, for a length on the horizontal axis, the fraction of functions longer than that. Both axes are logarithmic. Solid lines are the decompiled input f_{dec}, dashed lines the source target f_{src}. Median length is 178/136 tokens on train and 149/143 on test, but the distribution is heavy-tailed. **Right:** Distribution of the control-flow distance $d_{\text{CFG}}(f_{\text{dec}}, f_{\text{src}})$ of instances present in DECOMPBENCH. The decompiler output already reproduces the source CFG exactly for 48.5% of train and 60.9% of test pairs.

set contains 10,838 pairs from 30 additional groups. In total, DECOMPBENCH contains 660,559 function pairs from 2,032 package groups, all processed using the debug-assisted IDA 9.2 pipeline.

3.3 TRAINING

Supervised fine-tuning. For each model family, we initialize from an instruction-tuned checkpoint and fine-tune on paired decompiled and source functions. Each example places the decompiled function in the user message and the corresponding source function in the assistant response. We apply the SFT loss only to assistant-response tokens, including the end-of-turn token.

Reinforcement learning. We initialize GRPO (Shao et al., 2024) from the SFT checkpoint and retain a frozen copy of that checkpoint as the reference model for KL regularization. GRPO compares the rewards of multiple reconstructions generated for the same input to update the policy. We define the reward using the structural distance between a generated function $\hat{f}$ and its reference source f_{src}:

$$r(\hat{f}, f_{\text{src}}) = \begin{cases} \max\left\{0,\ 1 - \dfrac{d_{\text{CFG}}(\hat{f}, f_{\text{src}})}{30}\right\}, & \text{if scoring succeeds,} \\ 0, & \text{otherwise.} \end{cases}$$

The reward is one at zero distance, decreases linearly for distances between zero and 30, and remains zero for distances of 30 or greater. We also assign zero reward when CFG extraction fails, the required function cannot be found, or scoring exceeds the eight-second budget.

3.4 TEST-TIME SCALING

Candidate generation. We compare three strategies under a fixed budget of 20 candidates per input. Single-checkpoint sampling draws all candidates from either the SFT checkpoint or the GRPO checkpoint. Cross-checkpoint sampling combines candidates from the SFT and GRPO checkpoints and can include the unchanged IDA output as one candidate within the same budget. We evaluate candidate allocations on the validation set and fix the allocation before test-set evaluation. This comparison tests whether combining candidate sources improves reconstruction relative to repeated sampling from a single checkpoint.

Candidate selection. Because the reference source is unavailable at inference time, we train a selector to estimate candidate quality from the decompiled input and candidate reconstruction alone. The selector consists of an LLM with a scalar regression head. Given an input x and candidate $\hat{f}$, it predicts the transformed structural distance

$$z(\hat{f}, f_{\text{src}}) = \log\left(1 + \min\left\{d_{\text{CFG}}(\hat{f}, f_{\text{src}}),\ 200\right\}\right).$$

Table 1: Results on held-out packages of DECOMPBENCH containing 10.8k instances. The proprietary models are evaluated on 3k samples due to high cost. All metrics evaluate one final output per function, after selection for test-time methods. Bold marks the best reported value per column.

		GED ↓		Rates (%)	
Method	k	Mean GED	Pen. Mean	Zero GED ↑	Failure ↓
(a) Decompiler and frontier-model baselines					
IDA	1	8.27	32.71	60.88	2.46
GPT-6-Astra (Max) *(On 3k instances)*	1	7.59	21.15	62.47	1.37
Claude-Opus-4.8 (Max) *(On 3k instances)*	1	7.06	30.59	63.63	2.37
(b) Fine-tuned models (single candidate)					
Llama-3.2-3B-Instruct (SFT)	1	9.30	38.00	67.35	2.90
Llama-3.2-3B-Instruct (GRPO)	1	6.03	34.09	68.22	2.82
Qwen-3-4B-Instruct (SFT)	1	7.23	36.91	68.73	2.99
Qwen-3-4B-Instruct (GRPO)	1	5.53	32.41	69.08	2.70
(c) Test-time methods (sampling + selection)					
Llama SFT sampling + selector	20	6.38	32.79	67.90	2.66
Llama GRPO sampling + selector	20	6.04	31.08	68.62	2.52
Llama Cross-checkpoint sampling	20	6.05	29.80	68.68	**2.39**
Qwen SFT sampling + selector	20	5.25	32.42	68.96	2.73
Qwen GRPO sampling + selector	20	**4.83**	30.63	70.28	2.59
Qwen Cross-checkpoint sampling	20	4.85	**29.00**	**70.29**	2.43

We construct selector-training data from 6,000 training-split prompts, sampled to emphasize longer inputs. For each prompt, we collect 12 candidates from each trained model and include the unchanged decompiler output. We label candidates using their structural distance to the reference source. At inference time, the selector scores each candidate in the pool and returns the one with the lowest predicted transformed distance, without access to the reference source.

4 EXPERIMENTS & RESULTS

4.1 EXPERIMENTAL SETUP

Evaluation Setting. We evaluate final reconstructions on the held-out DECOMPBENCH test split, which contains 10,838 function pairs from 30 package groups. As described in § 3.2, all versions of each package are assigned to the same split. Inputs are generated by Hex-Rays in IDA Pro 9.2 from x86-64 binaries compiled at `-O0`, with debug information available to the decompiler.

Models and Baselines We evaluate Qwen-3-4B-Instruct-2507 (Yang et al., 2025) and Llama-3.2-3B-Instruct (Grattafiori et al., 2024) after SFT and after subsequent GRPO training. The unchanged IDA output serves as a primary baseline, allowing us to assess whether model-based refinement improves on the structure already recovered by the decompiler. We additionally evaluate GPT-6-Astra (OpenAI, 2026) and Claude-Opus-4.8 (Anthropic, 2026) with `max` reasoning effort. Because of evaluation cost, these frontier-model baselines are evaluated on a subset of 3,000 test functions[1].

Evaluation Metrics We measure structural reconstruction using the CFG distance d_{CFG} defined in § 3.1. For each method, let N denote the number of evaluation inputs, V the subset whose outputs are successfully scored, and d_x the distance between the output and reference functions for input x. We report the mean GED over successfully scored outputs and a failure-penalized mean:

$$\mathrm{MeanGED} = \frac{1}{|V|}\sum_{x\in V} d_x, \qquad \mathrm{PenMeanGED} = \frac{\sum_{x\in V} d_x + 1000(N-|V|)}{N}.$$

[1] On the 3k samples, GPT-6-Astra incurred the cost of $248.52 and Claude-Opus-4.8 of $716.70

MeanGED measures structural distance among scored outputs, whereas PenMeanGED additionally assigns a GED of 1,000 to each instance that fails during scoring. A scoring failure occurs when CFG extraction fails, the required function cannot be found, or evaluation exceeds the eight-second budget. We also report the failure rate and the Zero GED rate, defined as the percentage of outputs with $d_{\mathrm{CFG}} = 0$. Zero GED denotes a structural match under the metric, not functional equivalence or exact recovery.

Sampling and selection. We compare the three 20-candidate strategies described in § 3.4: SFT sampling, GRPO sampling, and cross-checkpoint sampling. Cross-checkpoint sampling combines reconstructions from the SFT and GRPO checkpoints with the unchanged IDA output, which occupies one slot in the 20-candidate pool. A learned selector scores each candidate using only the decompiled input and candidate reconstruction, then returns the candidate with the lowest predicted structural distance. The test-time rows in Table 1 therefore evaluate one final output per input.

Separately, we characterize candidate-pool quality on the validation set using minGED @k: the expected minimum GED among a uniformly chosen k-candidate subset of a 20-candidate pool. Because minGED @k assumes the best candidate in the subset is identified, it is a lower bound on the expected GED achievable by a selection rule.

4.2 Results Discussion

GRPO vs SFT. Table 1 shows that GRPO improves single-output structural reconstruction in both model families. Relative to SFT, mean GED decreases from 7.23 to 5.53 (23.5% ↓) for Qwen and from 9.30 to 6.03 (35.2% ↓) for Llama. The Zero GED rate increases more modestly, by 0.35 percentage points for Qwen and 0.87 percentage points for Llama. GRPO also reduces the failure rate and failure-penalized mean in both families. Its benefits therefore extend beyond the frequency of zero-distance outputs to the graded structural quality of successfully scored reconstructions.

The comparison with unchanged IDA output is more nuanced. Both GRPO models achieve lower valid-only mean GED and higher Zero GED rates than IDA, but their failure rates are higher: 2.70% for Qwen and 2.82% for Llama, compared to 2.46% for IDA. Consequently, Qwen GRPO only slightly improves the failure-penalized mean, from 32.71 to 32.41, while Llama GRPO increases it to 34.09. These results illustrate why structural quality among scored outputs and the incidence of scoring failures must be evaluated together.

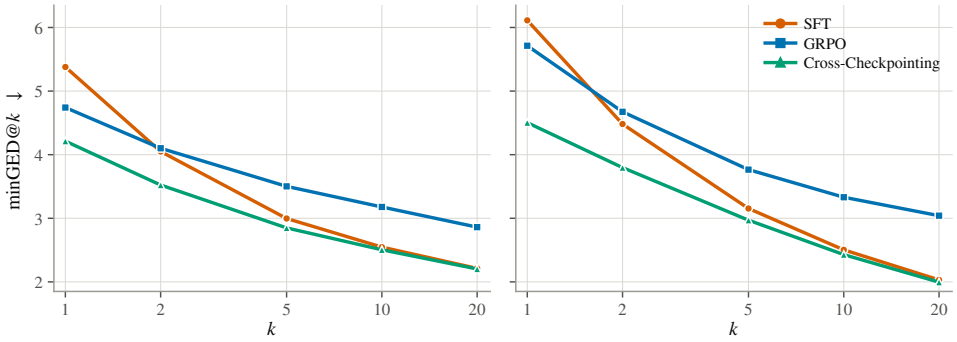


Figure 3: minGED @k for Qwen3-4B-Instruct (left) and Llama-3.2-3B-Instruct (right), on the validation set. SFT starts worse than GRPO at $k = 1$ but overtakes it by $k = 5$; cross-checkpointing gets the best of both, beating GRPO at small k and matching SFT at large k.

Performance across function lengths. The length-binned results in Appendix C reveal that improvements are localized on smaller functions. Qwen GRPO has a lower mean GED than IDA for inputs of at most 512 tokens, at 2.71 versus 4.30 (37.0% ↓), but higher mean GED for inputs exceeding 2,048 tokens, at 47.16 versus 39.74 (18.7% ↑). Relative to Qwen SFT, however, GRPO reduces mean GED in all four reported length bins, including from 90.77 to 47.16 in the longest-input bin.

Sampling and learned selection. Sampling multiple candidates and applying the learned selector improves selected-output quality in several settings. For Qwen GRPO, a 20-candidate pool reduces

mean GED from 5.53 to 4.83, a 12.7% reduction, while increasing the Zero GED rate from 69.08% to 70.28%, or 1.20 percentage points. Sampling and selection also reduce mean GED for the SFT models, from 7.23 to 5.25 for Qwen and from 9.30 to 6.38 for Llama. The effect is not uniform across configurations: Llama GRPO's mean GED changes from 6.03 to 6.04, although its Zero GED rate increases and its failure rate decreases. Additional candidates therefore do not guarantee an improvement in every final-output metric.

Separately, on the validation set, the learned selector selects a minimum-GED candidate from the 20-candidate pool in 78.4% of cases for Qwen SFT, 87.2% for Qwen GRPO, 77.1% for Llama SFT, and 84.5% for Llama GRPO. The selector therefore attains the pool minimum more frequently for GRPO than for SFT in both model families.

Test-Time scaling effectiveness. Figure 3 examines the quality available through repeated sampling independently of the selector. On the validation set, GRPO achieves lower $\mathrm{minGED}\,@1$ than SFT: 4.74 versus 5.38 for Qwen and 5.71 versus 6.11 for Llama. However, SFT benefits more from additional samples and achieves lower $\mathrm{minGED}\,@k$ by $k = 5$ in both families. Its advantage persists at $k = 20$. Thus, GRPO's stronger individual samples do not translate into better oracle candidate-pool quality at larger sampling budgets.

Cross-checkpoint sampling also produces favorable oracle curves, but these improvements must be distinguished from gains in selected-output quality. On the test set, cross-checkpoint sampling lowers the failure-penalized mean relative to GRPO sampling, from 30.63 to 29.00 for Qwen and from 31.08 to 29.80 for Llama. However, valid-only mean GED is slightly higher: 4.85 versus 4.83 for Qwen and 6.05 versus 6.04 for Llama. The improvement in the penalized metric is therefore due to fewer scoring failures, rather than a lower mean distance among successfully scored outputs. Candidate generation and selection consequently remain distinct contributors to test-time performance.

Comparison with frontier LLMs. On the shared subset of 3,000 test functions, GPT-6-Astra and Claude-Opus-4.8 reduce mean GED from the IDA baseline of 10.56 to 7.59 and 7.06, respectively. Qwen cross-checkpoint sampling achieves a mean GED of 7.23, lower than GPT but higher than Claude, while attaining a higher Zero GED rate than both GPT (2.33% ↑) and Claude (1.17% ↑). However, Qwen exhibits larger upper-tail structural errors in this comparison: its 99th-percentile GED is 143, compared with 98 for GPT and 82 for Claude. These results distinguish the frequency of zero-GED reconstructions from average structural distance and upper-tail errors.

5 Analysis & Discussion

Exact Function Match. Structural recovery is an important objective in binary analysis (Brumley et al., 2013; Basque et al., 2024), but CFG agreement does not imply recovery of the source function. To complement our structural evaluation, we measure exact function match by removing whitespace from each output and its reference function and comparing the resulting strings. On the held-out test set, raw IDA output achieves an exact-match rate of 1.58%. Qwen SFT and GRPO achieve 4.52% and 4.64%, respectively, while Llama SFT and GRPO achieve 3.49% and 4.02%. Thus, the trained models produce more exact matches than the unchanged decompiler output, with GRPO providing additional increases of 0.12 and 0.53 percentage points over SFT for Qwen and Llama, respectively.

Cross-checkpoint sampling with learned selector achieves exact-match rates of 4.10% for Qwen and 3.55% for Llama, both below their corresponding single-output GRPO models. These results show that improvements under structural evaluation do not necessarily lead to higher exact-match rates.

Cross-Checkpointing Variations We examine how allocating a fixed 20-candidate pool among SFT, GRPO, and unchanged IDA output affects oracle and selected-output quality on the validation set. Table 2 shows that, among allocations without IDA, increasing the share of GRPO candidates raises $\mathrm{minGED}\,@20$ in both model families. An SFT-only pool achieves $\mathrm{minGED}\,@20$ of 2.21 for Qwen and 2.03 for Llama, compared with 2.86 and 3.04 for GRPO-only pools. These results concern the best reconstruction available within each pool, rather than the quality of individual samples.

The ordering differs after learned selection. For Qwen, allocating four candidates to SFT and 16 to GRPO yields a selected-output GED of 4.48, compared with 5.78 for SFT-only sampling and 4.59 for GRPO-only sampling. For Llama, eight SFT and 12 GRPO candidates yield 4.91, compared

Table 2: Allocating a fixed 20-candidate pool allocated among SFT, GRPO, and raw IDA output. $\mathrm{minGED}\ @20$ represent the best GED, while *Selector* is the GED value of the candidate the selector model actually returns.

	Qwen3-4B		Llama-3.2-3B	
Allocation	$\mathrm{minGED}\ @20 \downarrow$	Selected GED $\downarrow$	$\mathrm{minGED}\ @20 \downarrow$	Selected GED $\downarrow$
20 SFT + 0 GRPO + 0 IDA	2.21	5.78	2.03	6.08
16 SFT + 4 GRPO + 0 IDA	2.28	5.74	2.06	5.75
12 SFT + 8 GRPO + 0 IDA	2.34	5.41	2.12	5.72
8 SFT + 12 GRPO + 0 IDA	2.39	4.50	2.38	4.91
4 SFT + 16 GRPO + 0 IDA	2.57	4.48	2.75	5.97
0 SFT + 20 GRPO + 0 IDA	2.86	4.59	3.04	5.90
8 SFT + 11 GRPO + 1 IDA	2.20	4.50	1.99	4.93

with 6.08 and 5.90 for the respective single-checkpoint strategies. Thus, some mixed allocations improve selected-output quality despite having higher oracle GED than SFT-only sampling. The allocation with the lowest oracle distance need not produce the best result under learned selection.

Including unchanged IDA output further illustrates this distinction. Replacing one GRPO candidate in the eight-SFT/twelve-GRPO allocation with IDA output lowers $\mathrm{minGED}\ @20$ from 2.39 to 2.20 for Qwen and from 2.38 to 1.99 for Llama. However, the selected-output GED remains 4.50 for Qwen and changes from 4.91 to 4.93 for Llama. Including the decompiler output therefore improves oracle candidate-pool quality in this comparison without a corresponding reduction in selected-output GED. We use validation results to fix the candidate allocations before test-set evaluation.

Generalization to other Decompilers To assess transfer beyond the IDA outputs, we decompile using Ghidra (National Security Agency, 2023) and angr_sailr (Basque et al., 2024), then evaluate Qwen's cross-checkpoint sampling strategy on their outputs. After filtering matching pairs, the resulting evaluation sets contain 11,681 functions for Ghidra and 7,410 for angr_sailr.

For Ghidra, refinement increases the Zero GED rate from 26.4% to 46.3%, while mean GED rises from 13.08 to 17.85. For angr_sailr, the Zero GED rate increases from 22.6% to 29.3%, while mean GED rises from 12.23 to 22.69. These results indicate uneven cross-decompiler transfer: the evaluated pipeline produces zero-GED reconstructions more frequently, but worsens average structural distance on both evaluation sets. Consequently, a higher Zero GED rate alone does not establish an overall improvement in reconstruction quality when transferring refinement learned from IDA outputs to other decompilers.

6 Conclusion

We presented DecompBench, a large-scale benchmark for studying LLM-based refinement of decompiler output, containing more than 660K paired decompiled and source functions from over 2,000 C/C++ package groups. Using this benchmark, we studied supervised fine-tuning, reinforcement learning with a control-flow graph edit-distance reward, and test-time sampling for structural source reconstruction. GRPO consistently improves single-output structural quality over supervised fine-tuning across both Qwen and Llama models, reducing mean GED while modestly increasing the rate of zero-GED reconstructions. Our results also show that improvements to individual generations do not necessarily translate into better performance under increased test-time compute. Although GRPO produces stronger individual samples, SFT candidate pools achieve lower oracle structural distance at larger sampling budgets, revealing a tradeoff between per-sample quality and the range of useful reconstructions available through repeated sampling. Moreover, this available candidate quality is difficult to exploit without access to the reference source: learned selection realizes only a fraction of the oracle improvement, and different candidate allocations can affect oracle and selected performance differently.

AI DISCLOSURE SECTION

In this work, we used generative AI tools for paper writing and coding our experiments. Our method and baselines themselves require LLMs as part of their methods. We have **not** used generative AI tools for interpreting results, refining hypothesis, or creating scientific images. We have reviewed all AI-assisted work. We thoroughly verify all AI generated artifacts. We take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our results, we release the entire code and data at `https://anonymous.4open.science/r/RLforDecomp-AA69/`. We note that LLMs are inherently probabilistic in nature, and some results may vary upon each run. We hope our data aid in future research.

ETHICS STATEMENT

In accordance with ICLR policy, we disclose that AI assistants, specifically Grammarly for grammar correction and ChatGPT for sentence restructuring and paraphrasing, were utilized during the preparation of this manuscript. The authors have reviewed, edited, and take full responsibility for all final content presented in this paper.

REFERENCES

Anthropic. Claude opus 4.8 system card. Technical report, Anthropic, May 2026. URL `https://www-cdn.anthropic.com/0f0c97ad20d8005706296bd92aa1c27c6b2f4f61/Claude%20Opus%204.8%20System%20Card.pdf`. May 28, 2026.

Zion Leonahenahe Basque. CFGUtils: Utility library for working with control flow graphs. `https://github.com/mahaloz/cfgutils`, 2026. GitHub repository, accessed September 21, 2026.

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 361–378, 2024.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 353–368, 2013.

Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2765–2782, 2022.

Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 4327–4343, 2022.

Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *Advances in Neural Information Processing Systems*, 38:57654–57689, 2026.

Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Cheng Fu, Kunlin Yang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. N-bref: A high-fidelity decompiler exploiting programming structures. In *International Conference on Learning Representations*, volume 2021, 2021.

Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

Gentoo. Gentoo linux package repository, 2026. URL `https://github.com/gentoo/gentoo`. Accessed: 2026-09-24.

Soumya Suvra Ghosal, Souradip Chakraborty, Avinash Reddy, Yifu Lu, Mengdi Wang, Dinesh Manocha, Furong Huang, Mohammad Ghavamzadeh, and Amrit Singh Bedi. Does thinking more always help? understanding test-time scaling in reasoning models. *arXiv preprint arXiv:2506.04210*, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Divij Handa, Mihir Parmar, Aswin RRV, Md Nayem Uddin, Hamid Palangi, and Chitta Baral. Guidedsampling: Steering llms towards diverse candidate solutions at inference-time. In *International Conference on Learning Representations*, volume 2026, pp. 145052–145077, 2026.

Les Hatton. Power-law distributions of component size in general software systems. *IEEE Transactions on Software Engineering*, 35(4):566–572, 2008.

Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 1667–1680, 2018.

Hex-Rays SA. Hex-Rays Decompiler, 2026. URL `https://hex-rays.com/decompiler`. Accessed 2026-09-16.

Joern. Exporting graphs. URL `https://docs.joern.io/export/`. Joern Documentation.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, Yuqun Zhang, et al. Sk2decompile: Llm-based two-phase binary decompilation from skeleton to skin. In *International Conference on Learning Representations*, volume 2026, pp. 89731–89762, 2026.

Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4(1):5, 2021.

National Security Agency. The ghidra decompiler. https://ghidra-sre.org/, 2023. Accessed: 2026-09-23.

Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 27–41, 2016.

OpenAI. GPT-6 Astra: A new generation of intelligence. https://openai.com/index/gpt-6-astra/, 2026. Accessed: 2026-09-21.

Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4069–4087. IEEE, 2024.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pp. 1–16. IEEE, 2020.

Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (3):353–362, 1983.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pp. 138–157. IEEE, 2016.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models. 2024.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*, 2024.

TABLE OF CONTENTS

A Training Details **15**

A.1 Preliminaries 15

A.2 Setup & Hyperparameters 15

B Prompts **16**

B.1 Supervised Fine-Tuning and Reinforcement Learning 16

B.2 Frontier-Model Baselines 16

B.3 Learned Selector 17

C Additional Results **17**

A Training Details

A.1 Preliminaries

Supervised Fine-Tuning (SFT). Let $\mathcal{D}_{\mathrm{SFT}}$ be a dataset of input–response pairs (x, y), where x is a task prompt and $y = (y_1, \ldots, y_T)$ is the target response. SFT adapts a language model π_θ by minimizing the negative log-likelihood of the target response conditioned on the input:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_{\mathrm{SFT}}}\left[\sum_{t=1}^{T} \log \pi_\theta(y_t \mid y_{<t}, x)\right].$$

RL via Policy Gradients. In reinforcement learning, the language model acts as an autoregressive policy. For an input $x \sim \mathcal{D}_{\mathrm{RL}}$, it generates a response $y = (y_1, \ldots, y_T)$, with each token sampled from $\pi_\theta(\cdot \mid y_{<t}, x)$. Policy-gradient updates use an advantage-weighted term of the form

$$\nabla_\theta J(\theta) = \mathbb{E}_{x\sim\mathcal{D}_{\mathrm{RL}},\, y\sim\pi_\theta(\cdot|x)}\left[A(x, y)\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(y_t \mid y_{<t}, x)\right],$$

where $A(x, y)$ estimates the relative quality of a response under the reward signal. Policy-gradient methods differ in their advantage estimates and constraints on policy updates (Williams, 1992; Schulman et al., 2017).

A.2 Setup & Hyperparameters

In this subsection, we provide the details of our training setup. All of our experiments are conducted on NVIDIA H100 GPUs, with 2 GPUs for SFT and selector training and 4 GPUs for RL training.

Supervised Fine-Tuning. We implement our SFT using the `open-instruct` (Lambert et al., 2024) framework. All models are optimized with the **AdamW** optimizer under a cosine learning rate schedule, with the learning rate set to $\mathbf{1.0 \times 10^{-5}}$ and a warmup ratio of 0.03. We enable sequence packing and gradient checkpointing to reduce memory usage. Both Qwen3-4B-Instruct-2507 and Llama-3.2-3B-Instruct are trained for a single epoch with an effective batch size of **64** using DeepSpeed ZeRO-3 (Rajbhandari et al., 2020).

Reinforcement Learning. We perform RL training using the **GRPO** algorithm in the `verl` (Sheng et al., 2025) framework, starting from the SFT checkpoint of each model. To accelerate rollout generation, we employ vLLM (Kwon et al., 2023). For the RL training data, we sample **12,000** functions from the training split whose decompiled and source functions are between 150 and 3,500 characters long, and hold out 500 functions from separate packages to track the validation reward. We sample **8** responses per prompt using a temperature of **1.0**. Training is conducted with a **batch size of 64** and a **mini-batch size of 32**, i.e., one on-policy and one off-policy update per optimization step. The model is optimized using the **AdamW** optimizer with a constant learning rate of $\mathbf{1.0 \times 10^{-6}}$. We apply KL regularization against the frozen SFT checkpoint as a loss term with a coefficient of **0.001**. Rewards are computed from the GED between the longest code block in the response and the reference function, as described in § 3.3, and all RL experiments are run for **1 epoch**.

Test-Time Scaling. For the single-candidate results, we use greedy decoding with a maximum of **8192** new tokens. For test-time scaling, we sample **20** candidates per function from each checkpoint using a temperature of **0.8** and a top-p of **0.95**. To prevent runaway generations from exhausting the budget, we limit the new tokens for each input to $\min(8192, \max(1536, 6n_{\mathrm{in}} + 512))$, where n_{in} is the number of input tokens, and remove candidates that reach this limit before selection.

Selector Training. We build the selector on **Qwen3-4B-Instruct-2507** with a scalar regression head that predicts the transformed distance z described in § 3.4. To construct its training data, we sample 6,000 prompts from the training split with more weight on longer inputs, as summarized in Table 3, excluding the prompts used for RL. After removing duplicate candidates and pairs whose reference CFG cannot be constructed, we obtain **172,323** labeled candidates from 5,323 prompts. Since the selector has to rank candidates of the same input, we train it with a mean squared error

loss on z together with a pairwise logistic loss between candidates of the same prompt. The model is trained for **1 epoch** (1,322 steps) using DeepSpeed ZeRO-3, with the **AdamW** optimizer, a learning rate of $\mathbf{1.0 \times 10^{-5}}$ under a cosine schedule with a warmup ratio of 0.03, and a maximum input length of **6144** tokens. The selection rule is fixed on 1,476 validation prompts before evaluating on the test set.

Table 3: Number of prompts used to construct the selector training data in each input-length bin.

Input tokens	Prompts
$\leq$512	2,700
513–1,024	1,500
1,025–2,048	1,100
$>$2,048	700
Total	6,000

B PROMPTS

B.1 SUPERVISED FINE-TUNING AND REINFORCEMENT LEARNING

SFT and RL Training Data Format

```
Input: {decompiled_function}
Target: {source_function}

Formatted Prompt:
[User]: {decompiled_function}
[Assistant]: {source_function}
```

Figure 4: SFT and RL training data format. The decompiled function is used as the user message without any additional instruction, and the source function is used as the assistant response. During RL and test-time sampling, the model generates the assistant response.

B.2 FRONTIER-MODEL BASELINES

For the frontier-model baselines, we use the following prompt:

Prompt for *frontier-model decompilation*

```
[System prompt] (Claude-Opus-4.8 only)
You are an expert C programmer. Follow the user instruction exactly.

[User]
Reconstruct the original C source function from the decompiled C function below.

Return only one complete C function. Do not include Markdown fences, explanations,
analysis, alternatives, tests, or surrounding declarations. Preserve the function
name. Infer readable source-level types, variables, expressions, and control flow
while preserving the behavior and control-flow structure. You have no other context.
Do not call or use any tools.

DECOMPILED C FUNCTION:
{decompiled_function}
```

Figure 5: Prompt template for the GPT-6-Astra and Claude-Opus-4.8 baselines. Each function is evaluated in a separate session with max reasoning effort and without tool access, and the longest code block in the response is used as the reconstruction.

B.3 LEARNED SELECTOR

Selector Scoring for Candidate Selection

```
[Model]
Qwen3-4B-Instruct-2507 + scalar regression head

[Input Format]
<DECOMPILED>
{decompiled_function}
</DECOMPILED>
<RECONSTRUCTION>
{candidate_reconstruction}
</RECONSTRUCTION>

[Output]
Predicted transformed distance z = log(1 + min(GED, 200))

[Selection]
For each function with 20 candidate reconstructions:
1. Remove candidates that reach the generation length limit
2. Score the remaining candidates with the selector
3. Select the candidate with the lowest predicted distance
```

Figure 6: Selector scoring pipeline. The selector predicts the transformed GED of each candidate without access to the reference source, and the candidate with the lowest predicted distance is selected as the final reconstruction.

C ADDITIONAL RESULTS

Table 4: Tail behavior of GED over valid pairs on held-out packages of DECOMPBENCH; read jointly with the failure rates in Table 1, since tail statistics over valid pairs improve when hard pairs degrade into failures. Catastrophe rates are the percentage of valid pairs with GED ≥ 50 and ≥ 100. Bold marks the best value per column among methods evaluated on the full 10.8k instances.

	GED percentile ↓				Catastrophe (%) ↓	
Method	p90	p95	p99	Max	GED ≥ 50	GED ≥ 100
(a) Decompiler and frontier-model baselines						
IDA	23	40	113	**438**	3.73	1.21
GPT-6-Astra (Max) *(On 3k instances)*	18	39	98	398	3.62	0.88
Claude-Opus-4.8 (Max) *(On 3k instances)*	13	37	82	362	2.92	0.75
(b) Fine-tuned models (single candidate)						
Llama-3.2-3B-Instruct (SFT)	16	30	128	2691	3.09	1.35
Llama-3.2-3B-Instruct (GRPO)	14	27	93	810	2.35	0.86
Qwen-3-4B-Instruct (SFT)	13	26	94	2792	2.37	0.95
Qwen-3-4B-Instruct (GRPO)	13	24	**79**	2342	1.89	**0.68**
(c) Test-time methods (sampling + selection)						
Llama SFT sampling + selector	14	26	98	2342	2.44	0.96
Llama GRPO sampling + selector	13	26	99	847	2.38	0.99
Llama Cross-checkpoint sampling	13	26	99	847	2.37	0.99
Qwen SFT sampling + selector	13	23	**79**	963	1.92	0.80
Qwen GRPO sampling + selector	**12**	**22**	81	**438**	**1.78**	0.70
Qwen Cross-checkpoint sampling	13	23	81	**438**	1.80	0.70

Tail Behavior of GED. Table 4 reports the tail of the GED distribution over successfully scored outputs on held-out packages. We can observe that all fine-tuned models reduce the 90th and 95th percentile GED relative to the IDA output (e.g., 23 → 13 at p90 for Qwen GRPO), indicating that most reconstructions move closer to the reference structure. However, single-candidate models

Table 5: Results on held-out packages of DECOMPBENCH, binned by input length. Bins are input lengths in tokens under each generator's tokenizer, so bin membership varies across methods. Norm. Med. is the per-bin median of GED divided by the reference CFG size (nodes + edges). Bold marks the best value per column within each bin among methods evaluated on the full 10.8k instances.

		GED ↓		Rates (%)	
Method	Input tokens	Mean GED	Norm. Med.	Zero GED ↑	Failure ↓
(a) Decompiler and frontier-model baselines					
IDA	≤512	4.30	0.000	67.5	**1.9**
	513–1,024	21.74	0.125	32.7	**2.5**
	1,025–2,048	34.96	0.127	22.9	**1.9**
	>2,048	39.74	0.078	23.8	17.1
GPT-6-Astra (Max)	≤512	4.16	0.000	67.8	1.1
(On 3k instances)	513–1,024	22.80	0.101	28.9	1.0
	1,025–2,048	38.41	0.079	19.5	5.8
	>2,048	53.34	0.427	16.2	5.0
Claude-Opus-4.8 (Max)	≤512	3.93	0.000	69.1	1.5
(On 3k instances)	513–1,024	21.02	0.062	29.9	5.2
	1,025–2,048	42.17	0.085	20.7	17.2
	>2,048	48.92	0.381	12.5	7.5
(b) Fine-tuned models (single candidate)					
Llama-3.2-3B-Instruct (SFT)	≤512	3.00	0.000	73.5	2.3
	513–1,024	14.65	0.042	45.4	3.2
	1,025–2,048	25.89	0.075	31.2	2.1
	>2,048	141.73	0.096	22.0	16.5
Llama-3.2-3B-Instruct (GRPO)	≤512	2.83	0.000	74.5	2.0
	513–1,024	14.04	0.047	44.9	2.6
	1,025–2,048	22.47	0.063	31.2	2.3
	>2,048	49.62	0.064	24.0	22.5
Qwen-3-4B-Instruct (SFT)	≤512	2.80	0.000	74.7	2.5
	513–1,024	13.34	0.030	47.1	3.1
	1,025–2,048	19.96	0.061	32.7	2.5
	>2,048	90.77	0.047	27.6	15.2
Qwen-3-4B-Instruct (GRPO)	≤512	2.71	0.000	75.1	2.1
	513–1,024	11.73	0.030	47.5	2.8
	1,025–2,048	18.19	0.064	**33.1**	2.3
	>2,048	47.16	**0.031**	26.4	18.1
(c) Test-time methods (sampling + selection)					
Qwen SFT sampling + selector	≤512	2.63	0.000	75.0	2.1
	513–1,024	10.16	0.025	47.4	3.1
	1,025–2,048	17.22	**0.051**	**33.1**	2.3
	>2,048	46.10	0.039	26.4	17.8
Qwen GRPO sampling + selector	≤512	**2.48**	0.000	**76.3**	2.0
	513–1,024	**9.78**	**0.000**	**50.1**	2.7
	1,025–2,048	**15.63**	0.057	**33.1**	2.1
	>2,048	39.10	0.036	26.1	16.5
Qwen Cross-checkpoint sampling	≤512	2.50	0.000	76.1	2.0
	513–1,024	9.80	**0.000**	49.9	2.7
	1,025–2,048	**15.63**	0.057	**33.1**	2.1
	>2,048	**37.59**	0.032	**30.5**	**11.9**

occasionally produce reconstructions with very large distances, with the maximum GED reaching 2792 for Qwen SFT compared to 438 for the IDA output. GRPO reduces these extreme errors in both model families, lowering the rate of GED $\geq$ 100 for Qwen (0.95% $\rightarrow$ **0.68%**) and Llama (1.35% $\rightarrow$ 0.86%). Since GRPO also lowers the failure rate (Table 1), this improvement does not come from hard functions turning into scoring failures.

We also observe that sampling with the learned selector mainly reduces the extreme tail, e.g., the maximum GED of Qwen GRPO drops from 2342 to **438**. We hypothesize that this may stem from the selector avoiding runaway reconstructions whenever the pool contains a closer candidate. Cross-checkpoint sampling has nearly identical tail statistics to GRPO sampling with the selector, which indicates that its lower failure-penalized mean in Table 1 comes from fewer scoring failures rather than smaller distances. Although the frontier models are evaluated on a harder subset of 3k instances, neither of them produces distances in the thousands, and Claude-Opus-4.8 has the lightest tail among the baselines with a p99 of 82. Taken together, the results indicate that fine-tuning improves the typical reconstruction, while the extreme tail remains the main weakness of the smaller models.

Performance Across Function Lengths. Table 5 reports the results on held-out packages binned by the number of input tokens. From Table 5, we observe that reconstruction becomes harder for all methods as the input length increases, with the Zero GED rate of the IDA output dropping from 67.5% for inputs of at most 512 tokens to 23.8% for inputs longer than 2,048 tokens. For inputs of up to 2,048 tokens, all fine-tuned models improve over the IDA output on both mean GED and Zero GED rate. The gains are largest on medium-length functions, e.g., for inputs of 513–1,024 tokens, Qwen GRPO lowers the mean GED (21.74 $\rightarrow$ **11.73**) and raises the Zero GED rate (32.7% $\rightarrow$ **47.5%**). In contrast, the frontier models are competitive with the IDA output only on short inputs and fall below it on Zero GED rate in all longer bins.

For inputs longer than 2,048 tokens, all single-candidate models have a higher mean GED than the IDA output (39.74), ranging from 47.16 for Qwen GRPO to 141.73 for Llama SFT. However, the normalized median of Qwen GRPO in this bin (0.031) remains lower than that of the IDA output (0.078), which indicates that the higher mean is driven by a small number of reconstructions with very large distances, consistent with Table 4. GRPO reduces the mean GED of SFT in this bin (90.77 $\rightarrow$ 47.16 for Qwen) but also increases the failure rate (15.2% $\rightarrow$ 18.1%), so part of this reduction may come from hard functions turning into scoring failures. Test-time methods close this gap, and both GRPO sampling with the selector and cross-checkpoint sampling improve over the IDA output on all metrics in this bin. Cross-checkpoint sampling shows the largest improvement in mean GED (39.74 $\rightarrow$ **37.59**), Zero GED rate (23.8% $\rightarrow$ **30.5%**), and failure rate (17.1% $\rightarrow$ **11.9%**), since the unchanged decompiler output is available as a fallback candidate. Overall, these results indicate that the benefits of fine-tuning are concentrated on short and medium-length functions, while test-time scaling helps the most on the longest ones.