

Fine Tuning LLM-Based Decompiler Backend for Output Code Quality and Trustworthiness

Muqi Zou¹, Hongyu Cai¹, Hongwei Wu¹, Zion Leonahenahe Basque², Arslan Khan³,
Z. Berkay Celik¹, Dave (Jing) Tian¹, Antonio Bianchi¹, Ruoyu (Fish) Wang², and Dongyan Xu¹

¹Purdue University

²Arizona State University

³Pennsylvania State University

¹{zou116, hongyu, wu1685, zcelik, daveti, antoniob, dxu}@purdue.edu

²{zbasque, fishw}@asu.edu

³arslankhan@psu.edu

Abstract—As a prevalent function in many software security tasks, decompilation reconstructs human-readable source code from binaries. Despite recent advances, decompiler outputs remain difficult to read. With the advent of large language models (LLMs), researchers recently began to explore the potential of LLMs to improve decompiler output. Nevertheless, our study of these approaches reveals their problems, such as introducing new syntactic and semantic errors and relying on unreliable accuracy validation, making the decompiled code less trustworthy. In this paper, we present D-LiFT, an enhanced “decompiler-LLM” pipeline with a fine-tuned LLM using *code quality-aware* reinforcement learning. Unlike prior work that overlooks preserving accuracy, D-LiFT adheres to a key principle for enhancing the quality and trustworthiness of decompiled code: *preserving accuracy while improving readability*. Central to D-LiFT, we propose D-SCORE, an integrated code quality assessment system to score the decompiled source code from multiple aspects, and use it to guide reinforcement learning-based LLM fine-tuning and to select the best output during inference. In line with our principle, D-SCORE assigns low scores to any inaccurate output and only awards higher scores to code that passes the accuracy check. Our implementation, based on Ghidra and a range of LLMs, demonstrates significant improvements for the accurately decompiled code from the coreutils and util-linux projects. Compared to baseline LLMs without D-SCORE-driven fine-tuning, our trained LLMs produce 55.3% more improved decompiled functions. Overall, D-LiFT improves the quality of 68.2% of all the functions produced by the native decompiler.

Index Terms—Large Language Model, Decompiler, Code Quality and Trustworthiness

I. INTRODUCTION

Binary decompilation, as an important program analysis function, is the process of generating high-level human-comprehensible source code from a binary program. Decompilation supports numerous software management and security tasks, such as reverse engineering [66], vulnerability discovery [46], [29], program patching and hardening [55], and cyber forensics [12]. However, source code generated by decompilers is often hard for human users to read [9], [80], [16], [42], [82], [76], [75], [11], [40]. Moreover, decompiled outputs often suffer from syntactic and semantic correctness issues [43], [83], [15], [23], [77], [27]:

A piece of decompiled code may not compile or preserve semantics of the original binary, thereby undermining the trustworthiness of the decompiled output.

In recent years, with the advent of Large Language Models (LLMs), researchers have started to explore the potential of LLMs to generate and improve decompiled code [72], [62], [32], [37], [26], [35], [70], [74], [21], [22]. For example, to improve the readability of decompiled outputs, LLM4Decompile [62] trains new LLMs via supervised fine-tuning (SFT), and DeGPT [32] introduces a three-role mechanism to help verify LLMs’ output during inference. These LLM-assisted decompilation improvement tools have rapidly gained attention. For instance, LLM4Decompile was ranked 4th in GitHub Daily Trending.

Unfortunately, applying LLMs to improve decompiled code may introduce inaccuracy, further undermining its trustworthiness. For instance, Figure 6 in the Appendix shows how GPT-o4-mini [50] can easily introduce semantic errors. Furthermore, our analysis of the above state-of-the-art LLM-based methods for decompiled code improvement also shows significant limitations in preserving the original semantics. For LLM4Decompile, our study (detailed in Section VI-A3) reveals that, even when the input is error-free, new errors are found in 93.2% of the functions generated by its LLM. For DeGPT, it cannot enhance the LLM’s capability to generate more accurate decompiled code, since it only provides a non-deterministic semantic check after inference. These findings show a clear need for a more rigorous, LLM-driven decompiled-code enhancement pipeline, one that incorporates a novel training paradigm to produce smarter models and a rigorous inference-time accuracy check to ensure high-readability and semantics-preserving outputs.

Hence, we propose D-LiFT¹, an enhanced decompiler-LLM pipeline with a fine-tuned LLM using code quality-aware reinforcement learning. D-LiFT takes a baseline LLM,

¹“D-LiFT” reflects the decompilation pipeline with a “Decompiler” front-end and an “LLM with Fine Tuning” back-end.

a decompiler, and a set of real-world binaries as inputs, and outputs decompiled code of higher quality using an enhanced decompiler-LLM pipeline. To this end, D-LIFT first leverages reinforcement learning (RL) to fine-tune the baseline LLM and then, at inference, selects the top-scored output from the baseline LLM, the fine-tuned LLM, or the native decompiler.

Central to D-LIFT is D-SCORE, a novel code quality and trustworthiness assessment function, which provides multi-aspect scoring for LLM-generated decompiled code based on accuracy and readability. Unlike prior work, D-SCORE follows a key principle: *preserving accuracy while improving readability*. Specifically, first, to assess the accuracy of decompiled code, D-SCORE employs a compiler to generate the *syntax* feedback and symbolic execution to compare the *semantics* of the generated code against the corresponding function in the original binary. After that, to assess the readability of decompiled code, D-SCORE computes a score by applying two established readability metrics to compare the LLM’s output with the original decompiled code. By using D-SCORE as the reward function in reinforcement learning, D-LIFT enhances the LLMs’ ability to generate higher-quality decompiled code. Meanwhile, by using D-SCORE at inference, D-LIFT comparatively selects the best result among the native decompiler, the baseline LLM, and the fine-tuned LLM.

We implement D-LIFT using Ghidra [3] and a number of LLMs, and evaluate D-LIFT using functions from the coreutils [18] and util-linux [65] projects. As measured by D-SCORE, models fine-tuned by D-LIFT improve the quality of 55.3% more functions compared to the baseline LLMs. Interestingly, we observe a significant “improve-ability gap” between the accurate and inaccurate decompiled functions originally produced by the decompiler. All LLMs, including baseline and fine-tuned models, face challenges in improving the originally inaccurate decompiled code, with improvement rates of just 8.02% for the originally inaccurate functions, in contrast to 86.2% for the originally accurate ones. Overall, for functions that are accurately decompiled, D-LIFT improves the quality (measured by D-SCORE) of 68.2% of them, where 47.3% of them are improved by the D-LIFT fine-tuned LLM, and only 20.9% by the baseline LLM. In summary, our main contributions include the following:

- We design D-LIFT, an enhanced decompiler-LLM pipeline with a fine-tuned LLM using code quality-aware RL to improve the quality and trustworthiness of the decompiled code, adhering to the *preserving accuracy while improving readability*.
- We propose D-SCORE, an integrated scoring scheme to quantitatively assess the quality of decompiled code. The scheme incorporates existing code analysis techniques and established readability metrics to deliver a multi-aspect evaluation of decompiled code, assessing both syntactic and semantic correctness as well as readability properties to provide effective feedback to the LLM for training and a quantitative selection metric for inference.
- We implement D-LIFT based on Ghidra and three LLMs, and achieve significant decompiled code improvement for

the benchmark functions.

II. MOTIVATION

Though improving decompiled code via LLMs is prevalent, it has limitations.

LLMs introduce inaccuracy during the decompiled code improvement, making the code less trustworthy. LLMs, operating as probabilistic models that generate text through token-level predictions, systematically introduce inaccuracies in code quality improvement tasks, as evidenced by recent empirical studies documenting pervasive hallucination phenomena [81], [53], [34]. In the decompiled-code improvement scenario, we also observe that LLM-refined output frequently shows new errors. Here, we follow the previous work [43] to define the inaccuracy in the decompiled code as either *syntax errors* that prevent successful compilation or *semantic deviations* between the LLM-generated code and the original binary’s behavior. As shown in Table IV, an average of 44.2% of functions that were originally accurate become inaccurate after LLM processing. For the root cause, as case study examples shown later in Section VI-D, we observed that LLMs frequently make errors on small details, such as omitting brackets or instructions, or referencing wrong variables, that are hard to spot yet vital for accuracy.

These observations reveal that though LLMs may improve readability, they often introduce inaccuracy – hence untrustworthiness – of decompiled code, a factor that prior studies [83], [43], [15], [23], [78] have identified as vital for effective decompilation. Hence, we propose an overarching principle for improving decompiled code quality and trustworthiness: *preserving accuracy while improving readability*.

However, the existing "decompiler-LLM" pipeline does not follow the above principle. Specifically, LLM4Decompile, while effectively boosting LLM’s capability of generating more readable code through training, it compromises the accuracy of its output, as shown in Table IV. While running its MSSC, a mutated unit test, to check LLM output code semantics among inference, DeGPT uses random inputs that produce non-deterministic semantic check results and omits the check for syntax errors. In light of these limitations, we are motivated to develop a more effective "decompiler-LLM" pipeline, where the LLM backend is specifically designed for improving the readability of the decompiled code while preserving its syntactic and semantic accuracy for trustworthiness.

III. DESIGN CHALLENGES

A. Framework Design

1) *Training*: Improving the code quality of LLMs through training is not new [39], [60], [62], [52], [20], [21]. However, many of them are challenging to adapt to decompiled code improvement. Unlike general code improvement tasks, enhancing decompiled code faces a unique challenge: the existence of multiple valid and trustworthy ground truths. Specifically, a single binary may be compiled from multiple semantically equivalent source programs, each of which should be considered a valid ground truth. For instance, simple transformations

such as variable renaming or converting a for loop into an equivalent while loop produce the same binary outputs while significantly increasing source-level edit distance. In practice, however, researchers typically have access to only one of these source variants as the ground truth. Moreover, many fine-tuning methods, such as supervised fine-tuning (SFT) and the actor-critic paradigm in RL, accept only a single reference to compute the training loss, overlooking the possible existence of a full set of correct alternatives. This constraint may degrade a model’s effectiveness, as exemplified by LLM4Decompile’s use of SFT on one source code.

2) *Inference*: In scenarios where we have multiple enhanced code outputs, from the decompiler itself or various LLMs, an objective and quantitative metric is essential to rank quality. Unfortunately, the only existing framework, DeGPT, offers no automated readability metric, relying on human participants’ assessments.

B. Assessment of Decompiled Code Quality

As noted above, the existence of multiple valid ground truths (during training) and enhanced candidates (at inference) needs a quantitative code-quality metric to evaluate each candidate. Nevertheless, to the best of our knowledge, *NO* existing metric meets the dual requirement of maintaining accuracy and trustworthiness while enhancing readability.

For *decompiler accuracy*, as introduced in Section II, two aspects of accuracy, syntactic and semantic, need to be evaluated. For syntax validation, relying on standard compiler checks is a reliable choice. However, there is no widely adopted standard for semantic validation between decompiled code and its corresponding binary. Specifically, since the decompiled code assessment function must be deterministic, methods such as fuzzing [78] and its derivatives, like MSSC [15] from DeGPT, are excluded. Additionally, to ensure the semantics are compared directly with the binary instead of the original source code, we exclude approaches that rely on source-dependent checks, such as unit tests or Alive2 [45]. That leaves D-helix [83], which uses an iterative re-compiler to reveal syntax errors and symbolic execution to check the code’s semantics against the original binary. However, integrating D-helix directly into our framework introduces its own challenge. Notably, D-helix still relies on certain decompiled code output artifacts, e.g., external function calls, to analyze. Since LLM-generated code frequently omits instructions, these artifacts may be missing or unreliable, undermining D-helix’s effectiveness. Furthermore, in the absence of the original source code, there is no authoritative ground truth for the number, types, or ordering of arguments of the external function.

For *readability*, the most direct approach is to run a human study, e.g., pairwise comparisons on platforms such as CodeArena [24], and then train a model on the collected preferences to score readability (an RLHF-style pipeline). However, such scoring models are expensive to build at scale: for instance, as shown in OpenAI’s study [61], training a model for scoring the text summarization required nearly

64k human pairwise preferences and “thousands of labeler hours and significant researcher time”, illustrating the cost of large, high-quality preference datasets. Moreover, most human-comparison settings yield binary outcomes (0–1 preferences), whereas our RL framework requires scalar readability scores per candidate.

Regarding human-validated, automatic readability metrics, most existing metrics [13], [8], [54], [58] ignore decompiler-specific artifacts (e.g., the number of references/dereferences). To the best of our knowledge, R2I [25] is the only metric tailored specifically for decompiled code. However, since it is a relative measure, whose features are derived to compare between multiple decompilers, it cannot assign an absolute score to a single piece of LLM-generated code. For instance, the feature, “number of unnecessary goto labels”, is calculated by contrasting angr [67] or RetDec [6] outputs against those from Ghidra [3] and Hex-Rays [2], which cannot be computed when only one decompilation exists.

In summary, our principle, preserving accuracy while improving readability, cannot be satisfied by any single code quality metric currently available.

IV. D-LIFT DESIGN

D-LIFT is an automatic decompilation pipeline that consists of a decompiler front-end and an LLM back-end capable of improving the quality and the trustworthiness of decompiled code. The workflow of D-LIFT, illustrated in Figure 1, consists of a training phase (Section IV-A1) and an inference phase (Section IV-A2). Specifically, in the training phase, D-LIFT takes three inputs: a baseline model (i.e., the LLM to train), a set of training binaries, and a decompiler. It employs reinforcement learning to fine-tune the LLM, ultimately producing a model with improved decompilation capabilities. Then, in the inference phase, D-LIFT first uses the decompiler to generate the initial decompiled code for an input binary. It then applies both the baseline and the fine-tuned LLMs to refine the code and selects the best-quality result among all outputs. To guide the training and inference phase, D-LIFT introduces a multi-aspect code quality assessment metric, D-SCORE, that evaluates a piece of candidate code with respect to both accuracy and readability (Section IV-B). Specifically, for accuracy, D-SCORE adopts the stepwise approach: first verifying syntax (Section IV-B1), then validating semantics against the original binary (Section IV-B2). Only if a candidate passes these accuracy checks will D-SCORE move on to assess readability (Section IV-B3).

A. System Workflow

D-LIFT works by first fine-tuning the LLM via GRPO to train the LLM with accuracy-aware rewards. Then at runtime, it will select the best-quality decompiled code for each input binary, among the outputs of the decompiler, the baseline LLM, and the fine-tuned LLM, by comparing their D-SCORE.

1) *Training Phase*: The challenge described in Section III-A highlights a critical limitation in current approaches:

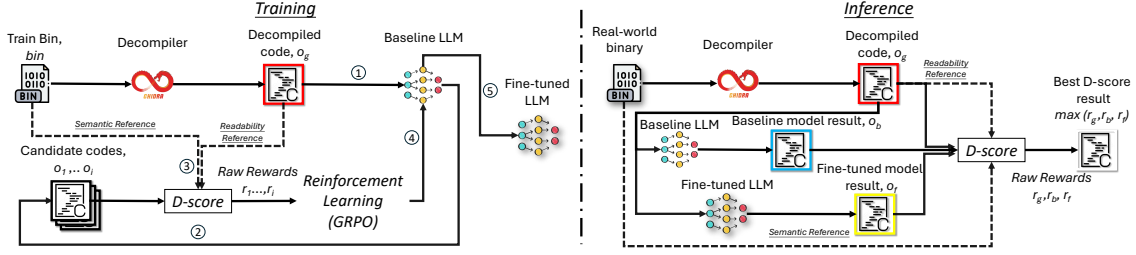


Fig. 1: This figure shows how D-LIFT fine-tunes the model and selects outputs.

decompiled code can map to multiple semantically equivalent source representations, each representing a valid ground truth. This multiplicity poses significant training difficulties for supervised fine-tuning (SFT) approaches, such as LLM4decompile, which are designed to work with only one correct answer. Although reinforcement learning generates multiple candidates during training, theoretically providing access to various valid solutions, conventional RL policies maintain the single ground truth constraint when computing the loss. Fortunately, Group Relative Policy Optimization (GRPO) [59] overcomes this constraint by generating the loss from the normalized rewards across multiple candidate outputs.

Figure 1 illustrates how D-LIFT integrates GRPO into the training. ① Given a training binary, denoted as bin , D-LIFT invokes the decompiler to produce an original decompiled output, denoted as o_g . ② Using the o_g as input, the baseline LLM generates a set of candidate refinements, $\{o_1, o_2, \dots, o_i\}$. ③ For each candidate, o_i , D-LIFT computes a reward score, r_i , using D-SCORE (see Section IV-B for details). ④ Next, GRPO normalizes these rewards, $\mathbf{r} = \{r_1, r_2, \dots\}$ within each candidate group according to its standard formulation:

$$\hat{A}_{i,t} = \tilde{r}_i = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})} \quad (1)$$

and uses these normalized rewards to compute the loss for fine-tuning the model. Given that the decompilation task accepts multiple valid ground truths, D-LIFT provides no ground truth code; it relies exclusively on D-SCORE-generated scores, removing any dependence on a particular reference solution, e.g., source code. Beyond reducing memory and computation, this choice aligns with recent results showing that reference-free training can be more effective [31], [44]. ⑤ Finally, once training iterations complete, D-LIFT outputs the fully fine-tuned LLM.

2) *Inference Phase*: As described in Section II, LLMs often introduce errors when improving code, creating a need for a quantitative metric-guided selection system. To address this, we designed the selection system as shown in Figure 1. This system works in two steps: ① It first takes a user's binary file and runs a decompiler to get the original code. It then feeds that code to both the baseline LLM and our fine-tuned LLM to generate two improved versions. ② By leveraging D-SCORE, the same scoring tool applied during training, it

evaluates all three outputs (the original, the baseline's, and the fine-tuned's), selecting the highest-scoring one as the final result for the user.

B. D-SCORE Design

Figure 2 shows the overall procedure of D-SCORE. As a code quality/trustworthiness assessment function, D-SCORE takes three inputs: a candidate code waiting for evaluation, o_i , its original binary, bin , and its original decompiled code from the decompiler, o_g , and outputs an absolute score r_i reflecting both the accuracy and readability of o_i . Following our principle, inaccuracies are penalized more heavily than readability issues:

$$\max(R_{syn}) < \min(R_{sem}) \text{ and } \max(R_{sem}) < \min(R_{read}) \quad (2)$$

To generate r_i , specifically, ① D-SCORE checks for syntax errors (See Section IV-B1). If not passed, it returns a syntax penalty score. ② Otherwise, it verifies semantic equivalence between o_i and the original binary, bin . If o_i fails the semantic check, D-SCORE returns a semantic penalty score, based on the symbolic matching with the bin (See Section IV-B2). ③ Only when o_i passes both checks, D-SCORE defers to the readability component, compares o_i to the original decompiled code, o_g , and returns a readability score (See Section IV-B3). This design ensures D-SCORE only awards readability once the accuracy has been verified, thereby adhering to our core principle and maintaining decompiled code trustworthiness.

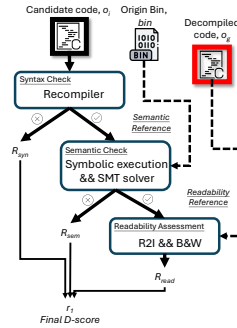


Fig. 2: D-SCORE, following our principle of *preserving accuracy while improving readability*.

1) *Syntax Metric*: As prior work observes [70], [83], decompiler outputs frequently are not directly recompilable even

at the function level, necessitating nontrivial post-processing before they can be compiled by standard toolchains. To address this and feed cleaner code into downstream semantic validation, D-SCORE adopts an iterative recompilation on each function: it places each decompiled function into its own translation unit, repeatedly compiles it, and fixes the compiler-reported syntax issues. Concretely, D-SCORE automatically initializes undefined variables, injects required system libraries, and translates special pseudo-instructions (e.g., CONCAT) into function calls. For each candidate code o_i from the baseline model, D-SCORE iteratively submits it to the compiler and obtains a syntax score, R_{syn} , based on the compiler’s feedback. Specifically, D-SCORE assigns $R_{syn}(o_i)$ to syn_{pen} when the o_i cannot be recompiled and $pass$ otherwise:

$$R_{syn}(o_i) = \begin{cases} syn_{pen}, & \text{if } o_i \text{ cannot be compiled.} \\ pass, & \text{otherwise} \end{cases} \quad (3)$$

Note that syn_{pen} contributes to D-SCORE, a syntax error ensures a clear penalty.

2) *Semantics Metric*: Once the candidate code o_i passes the syntax check, D-SCORE runs symbolic execution to conduct the semantic check between the binary and the intermediate representation (IR) code of o_i . To better describe, we split into: return value and external function call.

To confirm that the return values of the original binary and any candidate code match across different inputs, we employ symbolic execution and SMT solver. Specifically, as shown on the left side of Figure 7 in the Appendix, given the candidate code, D-SCORE first converts each function argument into a symbolic variable and performs symbolic execution to build a `Symbolic-Model-Ret`. This model captures the function’s semantics by linking symbolic inputs to arguments and symbolic outputs to return values, effectively encoding how each conditional transformation affects the final result. After that, D-SCORE runs the solver on `SMT-RET` to compare symbolic models between the original binary and the decompiled code.

When verifying external function calls, however, relying on the arguments provided from the LLM-generated candidate code (i.e., o_i) can yield many false negatives. The fundamental issue is that the binary itself does not encode any definitive ground truth regarding the arguments of external function calls. As a result, if we model or validate such calls based solely on the argument lists produced by the LLM, the assessment becomes highly unreliable. In particular, an LLM can trivially circumvent these checks by generating calls with incomplete or even empty argument lists, which may still pass superficial validation despite being semantically incorrect. More broadly, accurately recovering function prototypes for external calls remains an open and well-recognized challenge in binary analysis and decompilation [41]. The number of arguments, their types, and their semantics are typically lost during compilation and are not recoverable from the binary alone without additional assumptions or debugging information. Consequently, this information cannot serve as a trustworthy

reference during semantic verification.

Hence, to model the external function call in LLM-generated code in a better way, D-SCORE adopts a strategy that does not rely on unavailable prototype information. To do this, D-SCORE checks whether the number of external function calls being invoked, under varying input conditions, matches between the original binary and the LLM-generated code, as shown on the right side of Figure 7 in the Appendix. Specifically, to identify valid external calls in each decompiled function, we first extract and save the names of all invoked external functions to a file as our ground truth. After that, during the symbolic execution of the LLM-generated code, we build a symbolic model called `Symbolic-Model-Call` to model the external function call behavior. Similar to `Symbolic-Model-Ret`, this model is built through symbolic execution with symbolic inputs. However, instead of tracking return values, it counts how many ground-truth external function calls are invoked along each execution path. This count is then used as a behavioral signature of function-call activity. Finally, we run SMT solver on `SMT-CALL` to compare the function call-count symbolic models between the original binary and the decompiled code. By using this approach, we successfully avoid relying on potentially unreliable function prototype information.

By sequentially integrating the return-value check and the external-call check, we define our semantics metric as follows:

$$R_{sem}(o_i, bin) = \begin{cases} ret_{pen}, & \text{if } (check_{ret}(o_i, bin) == false) \\ call_{pen}, & \text{if } (check_{ret}(o_i, bin) == true, \\ & \text{and } check_{call}(o_i, bin) == false) \\ pass, & \text{otherwise} \end{cases} \quad (4)$$

, where $check_{ret}(o_i, bin)$ is the boolean result of running SMT solver on `SMT-RET`, $check_{call}(o_i, bin)$ is the result of running SMT solver on `SMT-CALL`, ret_{pen} and $call_{pen}$ are the penalties applied when the return-value check or the external-call check fails, respectively.

3) *Readability Metric*: For the reinforcement learning over multiple LLM-generated candidates, D-SCORE must assign each candidate o_i a scalar, objective readability score that supports reliable ranking. However, existing options do not directly fit our setting. Specifically, RLHF-style human studies primarily yield pairwise preferences (“which is better?”) rather than calibrated scalar scores, while individual software-engineering readability metrics each have limitations. To address this limitation, we score each o_i directly against the original decompiler output o_g and build a hybrid assessor by customizing established software engineering readability models. Specifically, we integrate selected features from the only decompiler-related metric, R2I [25], and the most classic readability metric, B&W [13] via a weighted linear combination (i.e., multiply each feature count by its weight and sum), following the same principle used by both R2I and B&W. Moreover, to blend them effectively, we normalize their individual scores to a fixed interval, allowing precise control over their relative influence. We leverage the fact that both

B&W and R2I are defined and validated via user studies [25], [13], reflecting readability features of decompiled code such as the number of references and do-while constructs.

To compute $R_{b\&w}(o_i, o_g)$, we follow the B&W framework, which defines a feature set, $\mathbf{f}_{b\&w} = \{f_1, f_2, \dots\}$, e.g., average number of commas per line, and their associated weights, $\mathbf{w}_{b\&w} = \{w_1, w_2, \dots\}$. Specifically, we first apply B&W to o_i and o_g to obtain two absolute scores. We then compute their relative difference and pass this value through a sigmoid function to map it into the range $[-1, 1]$, yielding a normalized, relative readability score. Formally,

$$R_{b\&w}(o_i, o_g) = \text{sigmoid}\left(\frac{R_{mul}(o_g) - R_{mul}(o_i)}{\min(R_{mul}(o_i), R_{mul}(o_g))}\right),$$

where $R_{mul}(x) = \left(\sum \mathbf{f}_{b\&w}(x) \cdot \mathbf{w}_{b\&w}\right)$ (5)

Nevertheless, certain decompilation-specific features remain outside the scope of $R_{b\&w}(o_i, o_g)$. Fortunately, $R_{R2I}(o_i, o_g)$ includes these features, $\mathbf{f}_{R2I}$ and their associated weights $\mathbf{w}_{R2I}$. Our implementation follows the methodology established in the original paper to generate the $R_{R2I}(o_i, o_g)$. Through renormalization of the feature-associated weights, $R_{R2I}(o_i, o_g)$ produces values within the range $[-1, 1]$, calculated using the following approach:

$$R_{R2I}(o_i, o_g) = \sum \mathbf{w}_{R2I} \cdot r_{\log}(\mathbf{f}_{R2I}(o_i) - \mathbf{f}_{R2I}(o_g)),$$

where $r_{\log}(x) = r \cdot e^{-\log_{10}(1+x)}$ (6)

$$+ (1-r) \cdot \left(1 - e^{-\log_{10}(1+x)}\right)$$

, where r is a binary indicator taking values in 0,1 to distinguish two types of feature groups ($r=1$, when features belongs to smaller values imply better readability, $r=0$, when larger values imply better readability). By combining B&W and R2I, we define an overall readability score in which γ and δ weight each metric's contributions as follows:

$$R_{read}(o_i, o_g) = \gamma \cdot R_{b\&w}(o_i, o_g) + \delta \cdot R_{R2I}(o_i, o_g) \quad (7)$$

V. IMPLEMENTATION

D-LiFT is a Python framework, using Ghidra (v11.2).

A. Reinforcement Learning

We run TRL [69] (version 0.18) from Huggingface to train models. We set *num_generations* to 3, *num_iterations* to 10, *per_device_train_batch_size* to 3, *vllm_gpu_memory_utilization* to 0.7, leaving all other parameters as default.

B. Semantic Check-related Settings

For the Recompiler, we set the maximum iteration to 10. For the semantics checking, we implement our semantic check framework on angr [67] (version 9.2), z3 [19] (version 4.13), and prompt [79] (version 1.0). To reduce the training time, we set the execution timeout to 30 seconds.

C. D-SCORE Metric Settings

We set *syn_{pen}* to -3, *ret_{pen}* to -2, and *call_{pen}* to -1.5. For readability metric, we set γ to 0.25 and δ to 0.75 for Equation (7), making $R_{read}(o_i, o_g) \in (-1, 1)$. These values follow Equation (2) and experimental observations to prevent sparse rewards [56]. Specifically, we selected $(\gamma, \delta)=(0.25, 0.75)$ after a modest grid search: We swept γ from 0.1 to 0.9 in steps of 0.05 (setting $\delta = 1 - \gamma$) and evaluated the aggregate objective on a held-out set of 100 samples. After that, we weight pair the summed (normalized) B&W and R2I scores on each sample. The chosen setting produced the highest aggregate readability signal while maintaining non-sparse rewards for RL training.

VI. EVALUATION

We organize this section as follows: In Section VI-A, we describe our experimental setup, including the evaluation of D-SCORE, dataset details, and baseline model selection. In Section VI-B and Section VI-C, we demonstrate how effectively D-LiFT enhances LLM-generated decompiled code via the above setup. In Section VI-D, we provide concrete case studies illustrating how D-LiFT refines decompiled snippets to improve both accuracy/trustworthiness and readability.

A. Experiment Setup

In this section, we begin by evaluating D-SCORE along two dimensions, applicability and precision, in Section VI-A1. After that, we describe our training and evaluation dataset in Section VI-A2. Finally, we detail our baseline model selections and reasons behind them in Section VI-A3.

1) *D-SCORE Evaluation*: We conduct experiments on a cluster node with an NVIDIA A100 Tensor Core GPU (80GB) [49], two 32-core AMD EPYC 7543 CPUs, and 400 GB RAM.

Dataset. Our metric evaluation uses a dataset of 1,948 decompiled functions sourced from binaries in two benchmarks widely used in software security research [83], [9], [32], [11], [80], [76], [75], [82], [40]: *coreutils* [18] (v9.5) and *util-linux* [65] (v2.41). Specifically, we first use GCC [1] to compile these projects under x86 Linux platform and then run Ghidra [3] (version 11.2) on 578 resulting binaries and object files, yielding 5,385 unique decompiled functions (1,792 from *coreutils* and 3,593 from *util-linux*). To focus on functions with sufficient complexity and room for improvement, we first filter this set to include only those with at least 20 lines of code (LOC) and a cyclomatic complexity [47] greater than 3. Nevertheless, since excessively complex or massive monolithic functions increase the binary's information entropy, which can easily be detected by heuristic indicators utilized by modern anti-virus scanners [4], we cap functions at less than 200 LOC. As a result, applying these filters yields 1,948 (36.2%) functions (653 from *coreutils* and 1,295 from *util-linux*).

Methodology. D-SCORE contains two core components, accuracy check and readability assessment. Because our readability metric is an aggregation of two previously validated

#	Categories of Errors	Pct.
1	Underlying tool errors	43%
2	Timeout	34%
3	D-SCORE errors	14%

TABLE I: The percentage of decompiled functions that cannot be analyzed by D-SCORE due to the listed errors.

Categories	Originally Accurate(OA)	Originally Inaccurate(OIA)
(Training, Evaluation)	(300, 725)	(0, 548)
(Syntax Errs, Semantic Errs)	(0, 0)	(307, 241)

TABLE II: Dataset overview. The table reports the number of functions per category, including counts used for training and evaluation, and the incidence of each error type in the original decompiled code.

metrics [13], [25], we accept its validity. Hence, our validity evaluation focuses on the metrics of decompiled code accuracy. To do this, we prompt `Qwen-Coder-2.5-3B` [7] with the above 1,948 decompiled functions and then pass each of its generated code completions, along with the corresponding original binary, through D-SCORE for an accuracy check. Specifically, we evaluate the accuracy check of D-SCORE along two dimensions: (1) *Applicability*: The proportion of functions for which D-SCORE can successfully derive the symbolic models from the binary. (2) *Precision*: Among these analyzable functions, the fraction for which D-SCORE correctly determines that the generated code is semantically equivalent to the original.

Applicability result. D-SCORE successfully finishes the accuracy check of 80.7% (1,573/1,948) of functions, i.e., generates the symbolic models of these functions and gets the result from the SMT solver without errors. To understand the limitation, we randomly sample 50 functions, where D-SCORE fails, without replacement, and manually analyze them. Table I categorizes the failures encountered by D-SCORE and reports their occurrence frequency. These errors fall into three categories: (1) Errors from the underlying tool, e.g., bugs in angr or incorrect identification of function boundaries when encountering no-return calls, (2) Timeout due to the scale of a function, and (3) Internal errors of D-SCORE, e.g., unsupported floating-point instructions.

Precision result. We evaluate the precision of D-SCORE by randomly sampling 100 functions without replacement from the 1,573 that D-SCORE can analyze, and manually compare each LLM-generated output against its original binary’s semantics to verify whether the decision made by D-SCORE was correct. Table III presents our results in terms of accuracy, precision, recall, and F1 score. Our analysis uncovers that: (1) False positives occur because the memory model of D-SCORE allocates fresh memory addresses for symbolized pointers. Even when the generated code is semantically correct, the values of pointers differ from those in the original binary, causing the symbolic executor to report a spurious mismatch. (2) False negatives occur because we assume all external function calls return a constant value. When using a constant return value in a conditional instruction, one branch may never be explored during the symbolic execution of the binary. However, since users can vary the assumed constant, rerunning

Accuracy	0.9450
Precisions	0.9100
Recall	0.9785
F1	0.9430

TABLE III: D-SCORE on LLM output.

#	Model Name	Synt Errs	Sem Errs	Total Errs
1	Qwen2.5-Coder-1.5B	288 (39.7%)	61 (8.41%)	349 (48.1%)
2	Qwen2.5-Coder-3B	196 (27.0%)	83 (11.4%)	279 (38.5%)
3	Llama3.2-3B	279 (38.5%)	114 (15.7%)	393 (54.2%)
4	LLM4Decompile-1.3B	636 (87.7%)	46 (6.34%)	682 (94.0%)

TABLE IV: Different baseline models’ performance on the (OA) dataset.

D-SCORE with multiple return values may exercise most branches and thus eliminate these false negatives.

2) *Dataset*: We present a dataset overview, then explain the training and evaluation datasets.

Dataset overview. Given that D-SCORE supports 1,573 functions (see Section VI-A1), we center our overview on this filtered set. To better understand our dataset, we run D-SCORE on the raw Ghidra outputs of these functions and examine their results distribution. Based on the accuracy check, the functions were split into two groups: originally accurate (i.e., error-free) and originally inaccurate. Table II shows the results: Of the 1,573 functions, 1,025 are classified as originally accurate (i.e., pass D-SCORE’s accuracy check), while the remaining 548 fall into originally inaccurate (i.e., fail the D-SCORE’s accuracy check), 307 exhibiting syntax errors and 241 containing semantic errors.

Training and evaluation. We randomly choose 300 functions from the originally accurate group as the training data. This selection criterion is based on the fact that LLMs demonstrate a limited ability to correct decompiler-introduced inaccuracies when provided only with decompiled code, as listed in Section VI-C. We apply this template uniformly during both training and inference: “prompt”: [{ “role”: “system”, “content”: “You are a helpful assistant for improving the decompiled result from the user. The user will input the decompiled result from Ghidra. Please improve its readability while preserving its semantics. Please do not add comments. Please just output the improved code.” }, { “role”: “user”, “content”: “[original decompiled code]” }].

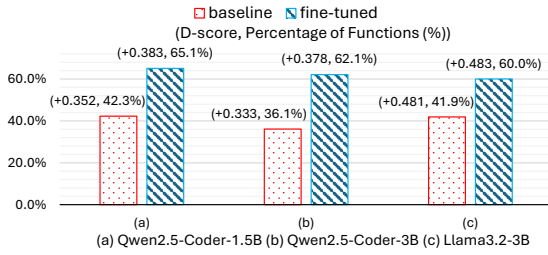
We evaluate D-LIFT on the remaining 1,273 functions with two subsets: 725 functions that are **originally accurate (OA)** and 548 functions that are **originally inaccurate (OIA)**.

3) *Baseline Model*: Due to computational resource constraints, we restrict our LLM to fewer than 3 billion parameters. Specifically, as a reinforcement learning framework, GRPO is particularly memory-intensive, since it requires simultaneous inference to generate candidate outputs and backpropagation to update model parameters, both of which consume significant GPU memory. Additionally, the reward normalization in GRPO mandates generating at least two candidates per input, further increasing memory demands. In our experiments, even after using vLLM [38] with a memory-efficient setting of 0.7, we still encountered out-of-memory errors when running on models larger than 3 billion parameters.

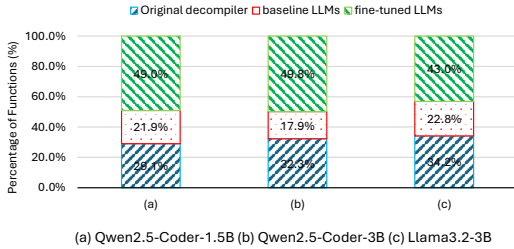
Therefore, we select two code-focused LLMs, Qwen2.5-Coder [33] (1.5B and 3B variants) and one general-purpose model, Llama3.2-3B [28]. As shown in Table IV, due to the poor performance of LLM4Decompile-End-1.3B [62] on our dataset, where 93% of the functions go wrong after applying it, we excluded it from our main evaluation. More specifically, we ask all four above models to improve the functions in our OA evaluation dataset, score the outputs using D-SCORE and show the results in Table IV. Since being fine-tuned based on the model released from November 2023, LLM4Decompile-End-1.3B made 682 functions (94.0%) inaccurate. In contrast, the three newer LLMs (released around September 2024) made only 46.9% of functions inaccurate. Given this marked difference, we omit LLM4Decompile-End-1.3B from further analysis.

B. D-LiFT Performance on OA

We show the evaluation of D-LiFT on originally accurate functions (OA) in two steps. We demonstrate changes brought from D-LiFT in Section VI-B1. Section VI-B2 analyzes its underlying reasons.



(a) The performance of different models on the OA dataset, before and after fine-tuning by D-LiFT.



(b) Overall performance of the selection system within D-LiFT on the OA dataset.

1) *Result*: To clearly present, we split the result into two parts: (1) baseline and fine-tuned LLMs' comparisons, and (2) an overall performance assessment.

Fine-tuned LLMs vs. baseline LLMs. Figure 3a summarizes the number of original decompiled functions within the OA dataset that are improved by the baseline and fine-tuned models and their average D-SCORE. To this end, we compare the output from each model with the original decompiled code. Compared with the baseline LLMs without D-SCORE-driven fine-tuning, our trained LLMs on average improve 55.3% more decompiled functions. By selecting the best output

among all three baseline LLMs, 509 (70.2%) functions (avg. +0.465) show improvement, while fine-tuned LLMs achieve improvement in 616 functions (84.9%), averaging +0.517.

Overall improvement by D-LiFT. Figure 3b shows the result of the selection system (i.e., selecting the version with the highest D-SCORE score among the fine-tuned model, the baseline model, and the original decompiled output). Specifically, for Qwen2.5-Coder-1.5B, fine-tuned LLM's output is the best for 355 (49.0 %) functions (avg. +0.399), baseline LLM for 159 (21.9 %) functions (avg. +0.419). For Qwen2.5-Coder-3B, fine-tuned LLM's output is best for 361 (49.8%) functions (avg. +0.402) and baseline LLM for 130 (17.9%) functions (avg. +0.398). For Llama3.2-3B, fine-tuned LLM's output is best for 312 (43.0%) functions (avg. +0.523) and baseline LLM for 165 (22.8%) functions (avg. +0.561). Moreover, by taking the best output among all six LLMs and the original decompiled code, 625 (86.2%) functions showed improvement (avg. +0.585), while 100 functions remained unimproved.

#	Model Name	Improvements				Regressions			
		Syn	Sem	Read	Tot	Syn	Sem	Read	Tot
1	Qwen2.5-Coder-1.5B	135	18	202	355	25	2	132	159
2	Qwen2.5-Coder-3B	88	17	256	361	12	2	116	130
3	Llama3.2-3B	116	38	114	268	19	9	150	178

TABLE V: This table shows how many functions get improved and regressed by the training of D-LiFT.

2) *Findings*: To better understand the training part of the D-LiFT's influence on LLM performance, we conducted an additional analysis by comparing the fine-tuned output and the baseline model output. Specifically, we analyze how the training within D-LiFT modifies function performance through two distinct categories: improvements, where fine-tuned models successfully resolve issues present in baseline outputs, and regressions, where previously error-free baseline functions develop new problems following the fine-tuning process. Based on it, we classify these changes into six categories: syntax fixes, semantic fixes, syntax regressions, semantic regressions, readability improvements, and readability regressions. Table V reports the number of functions falling into each category. These results show that LLMs fine-tuned by D-LiFT achieve considerable accuracy enhancements, with regression instances remaining at acceptably low levels.

To investigate the root causes of both improvements and regressions, we randomly selected 50 functions from each group (i.e., those that improved and those that regressed) and performed a manual code review. Our findings show that fine-tuning corrects five main categories of errors (Table VI): (1) Missing instructions, e.g., missing goto labels, variable declarations, and value assignments. (2) Incorrect brackets in conditional expressions, e.g., missing parentheses in if statements or misplacement leading to incorrect pointer dereference. (3) Incorrect variable naming and casting, e.g., using `mode_t` when only `__mode_t` is defined. (4) Unwanted instruction insertions, e.g., duplicate goto labels and unwanted function calls. (5) Incorrect literal value. e.g., constant values are inconsistent with the original values. However, fine-

Perform	Error Type	No.
Improve	Missing instruction	24
	Incorrect brackets	12
	Incorrect casting	5
	Unwanted instruction	7
	Incorrect literal value	2
Degrade	Missing instruction	14
	Incorrect brackets	24
	Unwanted instruction	12

TABLE VI: The root causes of both improvements and regressions caused by D-LiFT.

#	Model Name	No. of Func	Avg. D-S.
1	Qwen2.5-Coder -1.5B-baseline	8 (1.48%)	-0.591
2	Qwen2.5-Coder -3B-baseline	12 (2.23%)	-0.936
3	Llama3.2 3B-baseline	14 (2.6%) (2.6%)	-0.614
4	All base- line models	31 (5.66%)	-1.170
5	Qwen2.5-Coder -1.5B-fine-tuned	6 (1.12%)	+0.113
6	Qwen2.5-Coder -3B-fine-tuned	9 (1.67%)	-0.396
7	Llama3.2 -3B-fine-tuned	8 (1.49%)	-0.243
8	All fine- tuned models	19 (3.47%)	-0.469

TABLE VII: Different models’ performance on the OIA dataset, before and after fine-tuning by D-LiFT.

tuning also introduces regressions in three areas: (1) Missing instructions. (2) Incorrect brackets in conditional expressions. (3) Unwanted instruction insertions. More examples of the enhancements that D-LiFT enables are in Section VI-D.

Summary: When the original decompiled code does not have syntax or semantic errors, D-LiFT yields substantial improvement, in both the number of functions improved and their average D-SCORE.

C. D-LiFT Performance on OIA

We show the evaluation of D-LiFT on originally inaccurate functions (OIA) in two steps. We first compare outputs from the baseline and fine-tuned models and then analyze the underlying reasons for these observed scores.

1) *Result:* Table VII summarizes, for both baseline and fine-tuned models, how many functions achieve a higher D-SCORE than the original decompiled output and the average score improvement for those functions. As shown, both the baseline model and the fine-tuned models struggle with improving the original inaccurate functions. Moreover, by selecting the best among all six LLMs’ output and the original decompiled code, only 44 functions showed improvement (average -0.759), while 504 functions remained unimproved (average -2.50).

2) *Findings:* To investigate why our LLMs perform poorly on the OIA dataset, we randomly select 50 functions, where 25 are successfully improved by any LLMs (including baseline LLMs and fine-tuned LLMs) and 25 are not improved by LLMs, and manually analyze the underlying factors.

For the 25 functions where LLMs achieve improvements, surprisingly, we observed a consistent pattern: each involved variables declared as undefined [16], a 16-byte type with unknown signedness. The LLMs converted the variables with these opaque declarations into fixed-length arrays and updated member accesses. For instance, we observe that the declaration changed from undefined [16] auVar1; to an array, ulong auVar1[2] and this variable’s first eight-byte assignment is modified from auVar1._0_8_ = 0; to auVar1[0] = 0;. Although these transformations sometimes introduce subtle semantic inaccuracies, they consistently eliminate syntax errors, resulting in higher D-SCORE.

For the 25 functions where LLMs fail to achieve any improvements, we find three main causes: (1) Syntax errors due to undefined function pointer types. In 13 cases, errors stem from unresolved function pointer types like (code *)puVar3[4]. (2) Syntax errors from unsolvable type patterns. In 7 cases, issues arise from constructs like CONCAT31((int3)XX), where the type int3 is not well-formed. (3) Semantic errors from uninitialized global variables. In 5 cases, the decompiled code fails to initialize global variables to the correct value.

Summary: When the original decompiled code contains syntax or semantic errors, neither baseline LLMs nor our fine-tuned LLMs can deliver real improvements, indicating the importance of the decompiler front-end and the limitation of current LLMs in fixing those errors introduced by the decompiler.

D. Case Study

In this section, we present several illustrative cases that demonstrate how D-LiFT effectively improves the readability of decompiled code while preserving the accuracy. Specifically, we show how D-LiFT fixes syntax errors introduced by the LLM in Section VI-D1. Moreover, we provide an example about how D-LiFT fixes semantic errors introduced by the LLM in Section D in the Appendix.

1) *Syntax Error Fixes: Fixing Incorrect Variable Casting and Refactor Loop.* Figure 4 in the Appendix illustrates how D-LiFT enhances the output of Qwen2.5-Coder-3B model for the decompiled snippet of the uuid_copy function (util-linux).

Specifically, this function simply copies 16 bytes from the second parameter (src) to the first (dst). Since the decompiler cannot reconstruct the original struct or pointer types for the two input parameters, it falls back to manual pointer casting these parameters to byte pointers and uses pointer offsetting by (lVar1) to copy each byte (see line 8 of the original decompiled output).

Meanwhile, the decompiler also uses a do-while loop that checks lVar1 != 0x10, rather than the more natural for (i = 0; i < 16; i++) construct. Note that implementing a rule or heuristic inside Ghidra to systematically convert such do-while loops into for loops is non-trivial. In particular, Ghidra first normalizes for loops into while loops in ActionStructureTransform, and in this case the loop is not even recognized as a while loop because it fails the checks in CollapseStructure::ruleBlockWhileDo, which relies on several heuristics (e.g., the condition block must have exactly two outgoing edges, and a body-start block should typically have only one incoming/parent block).

In contrast, an LLM can easily convert this pattern into a canonical for loop, and the correctness of such a transformation can be validated by D-SCORE. Specifically, when the baseline LLM refactors the loop, though it successfully rewrites it as a for with i < 16, it mistakenly applies the array subscript operator to the input parameters of type

long, leading to the compiler error. D-LiFT, however, fine-tunes the model to insert the correct byte-pointer casts and maintain the `for` with `i < 16`, which eliminates the syntax error while yielding a more readable, semantically faithful implementation. As a result, D-LiFT improves the D-SCORE from -3.000 to +0.8887.

Fixing Missing Parentheses and Consolidating Conditionals. Figure 5 in the Appendix demonstrates another example about how D-LiFT corrects the syntax errors while enhancing readability in the `fputs_color_cell_close` (`util-linux`).

Specifically, for inaccuracy, line 5 in baseline Qwen2.5-Coder-1.5B model output lacks the necessary parentheses, resulting in the compilation error, *lvalue required as left operand of assignment*. Regarding readability, the original source code combines three checks with two `or` expressions (line 1). The original decompiled output, however, expands this into three separate `if` statements. For the output from the baseline model, though it removes redundant braces, it still uses three `if` blocks. Our fine-tuned model not only adds the necessary parentheses in line 4 but also merges the second and third checks into one consolidated `if` statement in line 2, mirroring the source code’s succinct logic. As a result, D-LiFT raises the D-SCORE for this function from -3.000 to +1.275.

VII. DISCUSSION

Applicability on OIA dataset. As illustrated in Section VI-C2, D-LiFT performs poorly when the decompiler front-end fails to produce accurate code. Moreover, because D-LiFT relies on function-level decompiled output, it cannot help if the decompiler is unable to process the binary. Hence, D-LiFT performs poorly in addressing internal decompiler issues, such as bugs that cause decompiler crashes or persistent decompilation challenges, such as function boundaries identification, type recovery, or indirect calls. We point out that decompilation is *only one of multiple aspects of binary reverse engineering*, focusing on presenting users with more readable source code of the subject binary. The other aspects of reverse engineering are better supported by other specialized reverse engineering tools [48], [10], [57], [42], [16], [9], [71] to complement decompilation. Moreover, as future work, we will explore the *integration* of such specialized tools (some implemented as agents) into the decompilation pipeline, to iteratively improve the decompiled code in multiple aspects (e.g., types, indirect calls, etc.).

Underlying tools. Since D-LiFT employs `angr` [67], `klee` [14] to run the symbolic execution, some features, including floating-point instructions, double pointers (due to the memory model), and large binaries or long context decompiled code because of timeouts, are currently not supported. We leave them as future work.

Readability metrics. We acknowledge that current approaches for assessing code readability are not perfect. However, R2I remains the state-of-the-art available metric for automatically evaluating the quality of decompiled code. This

automation is a critical prerequisite for our RL pipeline. Furthermore, our framework is designed to be extensible; it can seamlessly integrate superior metrics as they emerge in the future. Since the development of novel readability metrics falls outside the scope of this paper, we leave it for future work.

VIII. RELATED WORK

Regarding LLM-based decompiled-code enhancement, several approaches have been proposed. Besides LLM4Decompile [62], researchers have also proposed DecGPT [70], focusing on making decompiled code more recompilable. Nevertheless, this approach does not handle the semantic errors, e.g., hallucination errors, introduced by the LLM. To the best of our knowledge, DeGPT [32] is the only existing work that attempts to validate the accuracy of LLM-generated decompiled code. DeGPT introduces MSSC, a static analysis framework that assigns random values to inputs and observes the resulting changes in symbolic values in both the decompiled code and the LLM-generated code, aiming to detect discrepancies and identify inaccuracies. However, this method has notable drawbacks. For instance, MSSC does not recompile the code; it cannot detect syntax errors. Meanwhile, by testing with random inputs, the validation result can be inconsistent across runs.

Fine-tuning LLM to generate better quality code is not new. PPOCoder [60] uses structural differences, measured via Data Flow Graphs (DFGs) and Abstract Syntax Trees (ASTs) between the source code and generated code, as its reward signal, to improve the performance of LLM in multiple code generation tasks. StepCoder, CodeRL, and Palit [20], [52], [39] utilize compiler and unit tests as feedback for reinforcement learning. Nevertheless, all the above approaches accept only one unique ground truth, which makes them inappropriate in the decompilation scenario.

Researchers have also investigated the use of symbolic execution tools to validate the semantics of LLM-generated code. Taneja [63] applies Alive2 [45] on the vectorized code generated by LLM to verify its semantic correctness. Similarly, Wang [68] integrates Alive2 into a compiler’s translation-validation pipeline to ensure semantic fidelity. Nevertheless, since Alive2 is primarily designed to detect bugs, such as undefined behaviors, arising from compiler optimizations, it may miss decompiler-specific errors.

Regarding the code readability metric, subsequent studies are proposed based on the B&W framework. Specifically, researchers [30], [58], [54], [8], [64] have shifted toward the broader concept of code comprehensibility, which extends readability by also considering elements beyond the code itself, such as associated documentation, during evaluation. However, because these metrics assume the presence of comments and external documentation, features that decompiled code typically lacks, they are not well suited for assessing decompiled output.

IX. CONCLUSION

We design D-LIFT, an enhanced decompiler-LLM pipeline with a fine-tuned LLM using code quality-aware RL to improve the trustworthiness and quality of the decompiled code, adhering to the principle of *preserving accuracy while improving readability*. We propose D-SCORE, an integrated scoring mechanism designed specifically for decompilation recovery tasks. We implement D-LIFT based on Ghidra and fine-tune three LLMs, and achieve significant decompiled code improvement for widely used benchmark functions. Compared to the baseline LLMs, on average, our fine-tuned LLMs improve the quality of 55.3% more functions. Overall, D-LIFT generates 68.2% better quality functions than the native decompiler, where 47.3% of functions come from the D-SCORE-driven fine-tuned model and only 20.9% from the baseline model.

ETHICS CONSIDERATIONS

We note that our found bugs do not pose an immediate threat to users or developers since they mainly affect the accuracy of LLM-generated code.

We acknowledge the use of Chat-GPT [51] and Claude-AI [5] solely as a paraphrasing aid to improve clarity and readability; at no point did we allow it to generate new content, ideas, or arguments. All substantive work and original insights remain entirely our own.

APPENDIX

In this section, we show an example about how the LLM may make mistakes (Section A), examine training-size effects (Section B), compare two models with identical architecture but different parameter counts (1.5B and 3B) (Section C), and an example about how D-LiFT fixes the semantic error introduced by LLMs (Section D).

A. Code Example

As shown in Figure 6, while GPT-o4-mini [50] improves readability by recovering better variable names and collapsing nested loops into a single loop (boxes with solid lines), it erroneously replaces the condition hex value 0x16 with EBUSY instead of the correct constant (boxes with dashed lines), introducing a semantic error.

B. Training Efficiency

To determine the optimal training set size, we evaluated LLM performance with training sets of 50, 100, 150, 200, 250, and 300 functions. Figure 8a shows how many functions from the OA dataset scored higher after being processed by the fine-tuned LLM, compared to the raw decompiled code under D-SCORE. As shown in Figure 8a, performance levels off once the training set reaches 200 functions.

C. Model Comparison

Recent studies and public benchmarks [73], [36], [17] consistently show a strong correlation between model scale and code-generation quality: larger LLMs generally score substantially higher than small models across code benchmarks, in line with established scaling laws for language modeling. Nevertheless, our computational resource constrains our evaluation to relatively small models (Qwen2.5-Coder-1.5B and Qwen2.5-Coder-3B). To make the most informative comparison within this constraint, we provide a comprehensive comparison between 1.5B and 3B models under D-SCORE. Specifically, on the OA dataset, we conduct three comparisons: (i) for the two baseline models, we select the better output for each instance and report the proportion of wins per model; (ii) we repeat the same procedure for the two fine-tuned models; and (iii) we perform an oracle-style comparison over all four models (two baselines and two fine-tuned) by selecting the best output per instance and analyzing the contribution of each model. Figure 8b shows the result of the above three comparisons. Specifically, for baseline models, the 3B model’s output is the best for 440 (60.7 %) functions (avg. -0.998), the 1.5B model for 285 (39.3 %) functions (avg. +0.217). For fine-tuned models, the 3B model’s output is the best for 401 (55.3 %) functions (avg. -0.324), the 1.5B model for 324 (44.7 %) functions (avg. +0.340). Across all four models, the 3B fine-tuned model’s output is the best for 313 (43.2 %) functions (avg. -0.294), the 3B baseline model for 93 (12.8 %) functions (avg. +0.258), the 1.5B fine-tuned model’s for 250 (34.5 %) functions (avg. -0.334) and the 1.5B baseline model for 69 (9.52 %) functions (avg. +0.551). Overall, the 3B models dominate the 1.5B models in both baseline and

fine-tuned settings, and D-LiFT does not alter this dominance trend, consistent with established scaling-law observations that larger models tend to achieve stronger performance.

D. Semantic Error Fixes

Fixing Incorrect Literal Value and Extracting Functional Variable. Figure 9 shows how the decompiled code snippet from the

`fdisk_delete_all_partitions (util-linux)` is improved by fixing the semantic errors while enhancing the readability.

Semantically, the baseline Qwen2.5-Coder-3B output mistakenly returns the decimal constant -65538 (lines 8 and 11), which equates to 0xffffefffe rather than the intended 0xfffffffffea. Our fine-tuned model fixes this by emitting `return 0xfffffffffea;` at lines 9 and 12, restoring correct behavior. For readability, our fine-tuned model makes two key improvements. First, it removes a redundant `if-else` construct (original decompiled code line 14) and replaces it with a direct `return 0xfffffffffea` without an `else` condition added (fine-tuned model output line 12), yielding a control flow structure that more closely mirrors the original source. Second, it extracts the repeated expression `*(long *) (* (long *) (param_1 + 0x180) + 0x30)` (fine-tuned model output line 14), into a named variable, reducing duplicated usage (original decompiled code line 15 and line 23). As a result, D-LiFT raises this function’s D-SCORE from -2.000 to +0.625.

Source:	Original Decompiler:	Baseline model:	Fine-tuned model:
<pre> 1 void uuid_copy(uuid_t 2 [dst], const uuid_t 3 src) 4 { 5 unsigned char *cp1; 6 const unsigned char *cp2; 7 int i; 8 for (i=0, cp1 = dst, cp2 = 9 src; i < 16; i++) 10 *cp1++ = *cp2++; 11 } </pre>	<pre> 1 void uuid_copy (long 2 [param_1], long param_2) 3 { 4 long lVar1; 5 lVar1 = 0; 6 do { 7 *(undefined *) (param_1 + 8 lVar1) = *(undefined 9 *) (param_2 + lVar1); 10 lVar1 = lVar1 + 1; 11 } while (lVar1 != 0x10); 12 } </pre>	<pre> 1 void uuid_copy (long 2 [dest], long src) 3 { 4 long i; 5 for (i = 0; i < 16; 6 i++) 7 { 8 dest[i] = src[i]; 9 } 10 } </pre>	<pre> 1 void uuid_copy(long [dest], 2 long src) 3 { 4 long i; 5 for (i = 0; i < 0x10; i++) 6 { 7 *(unsigned char *) (dest + 8 i) = *(unsigned char 9 *) (src + i); 10 } </pre>

Fig. 4: This code snippet shows how D-LIFT helps Qwen2.5-Coder-3B enhance its ability to generate more readable code (boxes with solid lines) while correcting the syntax error (boxes with dashed lines).

Source:	Original decompiler:	Baseline model:	Fine-tuned model:
<pre> 1 if (!tb 2 !tb->colors_wanted 3 tb->format != 4 SCOLS_FMT_HUMAN) 5 return NULL; 6 ... 7 [tb->cur_color = color;] </pre>	<pre> 1 if (param_1 == 0) { 2 return; 3 } 4 if ((* (byte *) (param_1 + 5 0x140) & 2) == 0) { 6 return; 7 } 8 if ((* (int *) (param_1 + 9 0x108) != 0) { 10 return; 11 } 12 ... 13 *(char **) (param_1 + 0x120) = 14 param_3; </pre>	<pre> 1 if (!param_1) return; 2 if (!(* (byte *) (param_1 + 3 0x140) & 2)) return; 4 if ((* (int *) (param_1 + 5 0x108) != 0) return; 6 ... 7 *(char **) param_1 + 0x120 = 8 param_3; </pre>	<pre> 1 if (param_1 == 0) return; 2 if (((* (byte *) (param_1 + 3 0x140) & 2) == 0) 4 ((* (int *) (param_1 + 5 0x108) != 0)) return; 6 ... 7 *(char **) (param_1 + 0x120) = 8 param_3; </pre>

Fig. 5: This code snippet shows how D-LIFT helps Qwen2.5-Coder-1.5B enhance its ability to generate more readable code while correcting the syntax error (boxes with dashed lines).

Source Code:	Original Decompiled Code:	GPT-o4-mini Output Code:
<pre> 1 ptrdiff_t 2 safe_rw (int [fd], bufptr buf, 3 idx_t count) 4 { 5 for(;;) 6 { 7 ssize_t [result] = rw (fd, 8 buf, count <= (size_t) 9 -1 ? count : (size_t) 10 -1); 11 if (0 <= result) 12 return result; 13 else if (IS_EINTR (errno)) 14 continue; 15 else if ([errno == EINVAL] && 16 SYS_BUFSIZE_MAX < 17 count) 18 count = SYS_BUFSIZE_MAX; 19 else 20 return result; 21 } 22 } </pre>	<pre> 1 long FUN_0011c6ec(unknown4 [param_1], 2 unknown8 param_2, long param_3) 3 { 4 long lVar1; 5 int *piVar2; 6 long lStack_30; 7 lStack_30 = param_3; 8 while(true) { 9 do { 10 lVar1 = write(param_1, 11 param_2, lStack_30); 12 if (-1 < lVar1) { 13 return lVar1; 14 } 15 piVar2 = (int 16 *) __errno_location(); 17 while (*piVar2 == 4); 18 piVar2 = (int *) __errno_location(); 19 if (*piVar2 != 0x16) break; 20 if (lStack_30 < 0x7ff00001) { 21 return lVar1; 22 } 23 lStack_30 = 0x7ff00000; 24 } 25 return lVar1; 26 } </pre>	<pre> 1 #include <unistd.h> 2 #include <errno.h> 3 4 ssize_t write_with_retry(int [fd], const 5 void *buf, size_t count) { 6 const size_t MAX_CHUNK = 7 0x7ff00000; 8 size_t to_write = count; 9 ssize_t [result]; 10 11 while (1) { 12 result = write(fd, buf, 13 to_write); 14 if (result >= 0) { 15 return result; 16 } 17 if (errno == EINTR) { 18 continue; 19 } 20 if ([errno == EBUSY]) { 21 if (to_write > MAX_CHUNK) { 22 to_write = MAX_CHUNK; 23 continue; 24 } 25 } 26 return result; 27 } </pre>

Fig. 6: The code snippet shows the original decompiled code, its corresponding source code, and the version output from the GPT-o4-mini, where line 17 of the GPT-o4-mini-refined code is misinterpreted.

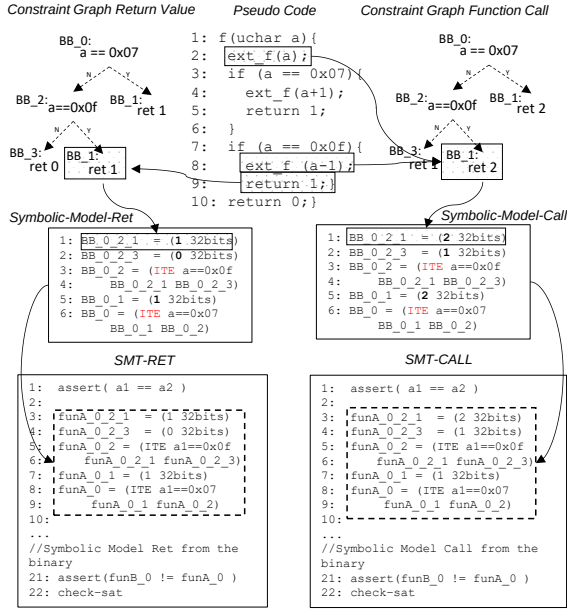
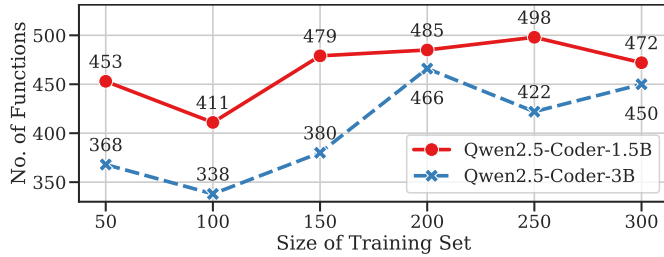
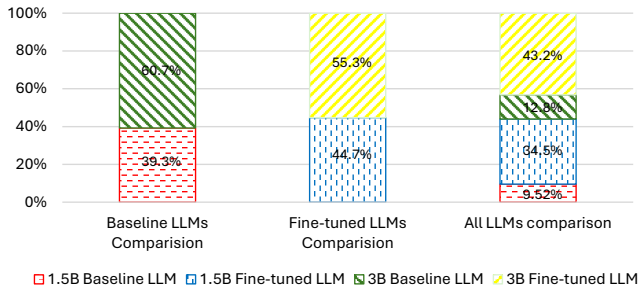


Fig. 7: The workflow of semantic check in D-SCORE.



(a) This table shows how many functions from the OA dataset scored higher (x-axis) after being processed by the fine-tuned LLMs with different sizes of the training sets (y-axis), compared to the raw decompiled code under D-SCORE.



(b) Overall performance of the model with different size on the OA dataset. This chart shows, for each function, which LLM achieves the highest D-SCORE score and reports the percentage of functions in each category.

Source:

```

1 int fdisk_delete_all_partitions
  (struct fdisk_context *cxt)
2 {
3     size_t i;
4     int rc = 0;
5
6     if (!cxt || !cxt->label)
7         return -EINVAL;
8
9     for (i = 0; i <
10         cxt->label->nparts_max; i++) {
11         if
12             (!fdisk_is_partition_used(cxt,
13                                     i))
14             continue;
15         rc = fdisk_delete_partition(cxt,
16                                     i);
17         if (rc)
18             break;
19     }
20
21     return rc;
22 }

```

Original decompiler:

```

1 undefined8 fdisk_delete_all_partitions(long param_1)
2
3 {
4     int iVar1;
5     undefined8 uVar2;
6     ulong uVar3;
7
8     if (param_1 == 0) {
9         return 0xffffffff;
10    }
11    if (*(long *)(param_1 + 0x180) == 0) {
12        uVar2 = 0xffffffff;
13    }
14    else {
15        uVar3 = 0;
16        if (*(long *)((long *)param_1 + 0x180) + 0x30) != 0) {
17            do {
18                iVar1 = fdisk_is_partition_used(param_1, uVar3);
19                if ((iVar1 != 0) && (uVar2 = fdisk_delete_partition(param_1, uVar3),
20                    (int)uVar2 != 0)) {
21                    return uVar2;
22                }
23                uVar3 = uVar3 + 1;
24            } while (uVar3 < *(ulong *)((long *)param_1 + 0x180) + 0x30);
25        }
26        uVar2 = 0;
27    }
28    return uVar2;
29 }

```

Baseline model:

```

1 int fdisk_delete_all_partitions(long param_1) {
2     int iVar1;
3     long lVar2;
4     long lVar3;
5     long lVar4;
6
7     if (param_1 == 0) {
8         return -65538; // Equivalent to 0xffffffff
9     }
10    if (*(long *)(param_1 + 0x180) == 0) {
11        return -65538; // Equivalent to 0xffffffff
12    }
13    lVar2 = 0;
14    if (*(long *)((long *)param_1 + 0x180) + 0x30) != 0) {
15        do {
16            iVar1 = fdisk_is_partition_used(param_1, lVar2);
17            if ((iVar1 != 0) && (lVar3 =
18                fdisk_delete_partition(param_1, lVar2),
19                (int)lVar3 != 0)) {
20                return lVar3;
21                // Return error code if partition deletion fails
22            }
23            lVar2 = lVar2 + 1;
24        } while (lVar2 < *(ulong *)((long *)param_1 + 0x180) +
25            0x30);
26    }
27    return 0; // Return success
28 }

```

Fine-tuned model:

```

1 int fdisk_delete_all_partitions(long param_1) {
2     int iVar1;
3     long lVar2;
4     long lVar3;
5     long lVar4;
6     long lVar5;
7
8     if (param_1 == 0) {
9         return 0xffffffff;
10    }
11    if (*(long *)(param_1 + 0x180) == 0) {
12        return 0xffffffff;
13    }
14    lVar2 = *(long *)((long *)param_1 + 0x180) + 0x30;
15    if (lVar2 != 0) {
16        lVar3 = 0;
17        do {
18            iVar1 = fdisk_is_partition_used(param_1, lVar3);
19            if ((iVar1 != 0) && (lVar5 =
20                fdisk_delete_partition(param_1,
21                lVar3), (int)lVar5 != 0)) {
22                return lVar5;
23            }
24            lVar3 = lVar3 + 1;
25        } while (lVar3 < lVar2);
26    }
27    return 0;
28 }

```

Fig. 9: This code snippet shows how D-LIFT helps Qwen2.5-Coder-3B enhance its ability to generate more readable code (boxes with solid lines) while correcting the semantic error (boxes with dashed lines).

REFERENCES

- [1] Free Software Foundation: Gcc. <https://gcc.gnu.org/>
- [2] Hex-Rays SA: Hex rays decompiler. <https://hex-rays.com/decompiler/>
- [3] National Security Agency: Ghidra. <https://ghidra-sre.org/>
- [4] Alshahwan, N., Barr, E.T., Clark, D., Danezis, G., Menéndez, H.D.: Detecting malware with information complexity. *Entropy* **22**(5) (2020). <https://doi.org/10.3390/e22050575>, <https://www.mdpi.com/1099-4300/22/5/575>
- [5] Anthropic: Claude AI (Claude 3, May 2025 version). <https://claude.ai> (2025), large language model. Accessed: 2025-06-04
- [6] Avast Software: RetDec: A retargetable machine-code decompiler. <https://retdec.com/>
- [7] Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., Ma, J., Men, R., Ren, X., Ren, X., Tan, C., Tan, S., Tu, J., Wang, P., Wang, S., Wang, W., Wu, S., Xu, B., Xu, J., Yang, A., Yang, H., Yang, J., Yang, J., Yang, S., Yao, Y., Yu, B., Bowen, Y., Yuan, H., Yuan, Z., Zhang, J., Zhang, X., Zhang, Y., Zhang, Z., Zhou, C., Zhou, J., Zhou, X., Zhu, T.: Qwen technical report. ArXiv **abs/2309.16609** (2023), <https://api.semanticscholar.org/CorpusID:263134555>
- [8] Muñoz Barón, M., Wyrich, M., Wagner, S.: An empirical validation of cognitive complexity as a measure of source code understandability. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). ESEM '20*, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3382494.3410636>, <https://doi.org/10.1145/3382494.3410636>
- [9] Basque, Z.L., Bajaj, A.P., Gibbs, W., O’Kain, J., Miao, D., Bao, T., Doupé, A., Shoshitaishvili, Y., Wang, R.: 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. USENIX Association, Philadelphia, PA (Aug 2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/basque>
- [10] Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: *International Symposium on Code Generation and Optimization (CGO 2011)*. pp. 213–223 (2011). <https://doi.org/10.1109/CGO.2011.5764689>
- [11] Brumley, D., Lee, J., Schwartz, E.J., Woo, M.: 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)
- [12] Burk, K., Pagani, F., Kruegel, C., Vigna, G.: Decomperson: How humans decompile and what we can learn from it. In: *31st USENIX Security Symposium (USENIX Security 22)*. pp. 2765–2782. USENIX Association, Boston, MA (Aug 2022), <https://www.usenix.org/conference/usenixsecurity22/presentation/burk>
- [13] Buse, R.P., Weimer, W.R.: Learning a metric for code readability. *IEEE Transactions on Software Engineering* **36**(4), 546–558 (2010). <https://doi.org/10.1109/TSE.2009.70>
- [14] Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. p. 209–224. OSDI’08, USENIX Association, USA (2008)
- [15] Cao, Y., Zhang, R., Liang, R., Chen, K.: Evaluating the effectiveness of decompilers. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. p. 491–502. ISSTA 2024, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3650212.3652144>, <https://doi.org/10.1145/3650212.3652144>
- [16] Chen, Q., Lacomis, J., Schwartz, E.J., Goues, C.L., Neubig, G., Vasilescu, B.: Augmenting decompiler output with learned variable names and types. In: *31st USENIX Security Symposium (USENIX Security 22)*. pp. 4327–4343. USENIX Association, Boston, MA (Aug 2022), <https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin>
- [17] Chou, J., Liu, A., Deng, Y., Zeng, Z., Zhang, T., Zhu, H., Cai, J., Mao, Y., Zhang, C., Tan, L., Xu, Z., Zhai, B., Liu, H., Zhu, S., Zhou, W., Lian, F.: Autocodebench: Large language models are automatic code benchmark generators (2025), <https://arxiv.org/abs/2508.09101>
- [18] coreutils: coreutils. <http://git.savannah.gnu.org/gitweb/?p=coreutils.git>
- [19] De Moura, L., Björner, N.: Z3: An efficient smt solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. p. 337–340. TACAS’08/ETAPS’08, Springer-Verlag, Berlin, Heidelberg (2008)
- [20] Dou, S., Liu, Y., Jia, H., Xiong, L., Zhou, E., Shen, W., Shan, J., Huang, C., Wang, X., Fan, X., et al.: StepCoder: Improve code generation with reinforcement learning from compiler feedback. arXiv preprint arXiv:2402.01391 (2024)
- [21] Dramko, L., Goues, C.L., Schwartz, E.J.: Fast, fine-grained equivalence checking for neural decompilers. *ACM Trans. Softw. Eng. Methodol.* **35**(7) (Jun 2026). <https://doi.org/10.1145/3772368>, <https://doi.org/10.1145/3772368>
- [22] Dramko, L., Goues, C.L., Schwartz, E.J.: Idioms: A simple and effective framework for turbo-charging local neural decompilation with well-defined types. In: *33rd Annual Network and Distributed System Security Symposium, NDSS 2026*, San Diego, California, USA, February 23–27, 2026. The Internet Society (2026), <https://www.ndss-symposium.org/ndss2026/>
- [23] Dramko, L., Lacomis, J., Schwartz, E.J., Vasilescu, B., Goues, C.L.: A taxonomy of c decompiler fidelity issues. In: *33rd USENIX Security Symposium (USENIX Security 24)*. pp. 379–396. USENIX Association, Philadelphia, PA (Aug 2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/dramko>
- [24] Du, M., Luu, A.T., Ji, B., Wu, X., Qing, Y., Huang, D., Zhuo, T.Y., Liu, Q., Ng, S.K.: CodeArena: A collective evaluation platform for LLM code generation. In: Mishra, P., Muresan, S., Yu, T. (eds.) *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. pp. 502–512. Association for Computational Linguistics, Vienna, Austria (Jul 2025). <https://doi.org/10.18653/v1/2025.acl-demo.48>, <https://aclanthology.org/2025.acl-demo.48/>
- [25] Eom, H., Kim, D., Lim, S., Koo, H., Hwang, S.: R2i: A relative readability metric for decompiled code. *Proc. ACM Softw. Eng.* **1**(FSE) (Jul 2024). <https://doi.org/10.1145/3643744>, <https://doi.org/10.1145/3643744>
- [26] Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., Zhao, J.: Coda: an end-to-end neural program decompiler. Curran Associates Inc., Red Hook, NY, USA (2019)
- [27] Gorzynski, A., Donaldson, A.F.: FuzzFlesh: Randomised Testing of Decompilers via Control Flow Graph-Based Program Generation. In: Aldrich, J., Silva, A. (eds.) *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 333, pp. 13:1–13:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). <https://doi.org/10.4230/LIPIcs.ECOOP.2025.13>, <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.13>
- [28] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., et al., A.K.: The llama 3 herd of models (2024), <https://arxiv.org/abs/2407.21783>
- [29] Han, H., Kyea, J., Jin, Y., Kang, J., Pak, B., Yun, I.: Queryx: Symbolic query on decompiled code for finding bugs in cots binaries. In: *2023 IEEE Symposium on Security and Privacy (SP)*. pp. 3279–3295 (2023). <https://doi.org/10.1109/SP46215.2023.10179314>
- [30] Holst, G., Dobslaw, F.: On the importance and shortcomings of code readability metrics: A case study on reactive programming (2021), <https://arxiv.org/abs/2110.15246>
- [31] Hu, J., Zhang, Y., Han, Q., Jiang, D., Zhang, X., Shum, H.Y.: Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model (2025), <https://arxiv.org/abs/2503.24290>
- [32] Hu, P., Liang, R., Chen, K.: Degpt: Optimizing decompiler output with llm. *Proceedings 2024 Network and Distributed System Security Symposium* (2024), <https://api.semanticscholar.org/CorpusID:267622140>
- [33] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., Yang, A., Men, R., Huang, F., Quan, S., Ren, X., Ren, X., Zhou, J., Lin, J.: Qwen2.5-coder technical report. ArXiv **abs/2409.12186** (2024), <https://api.semanticscholar.org/CorpusID:272707390>
- [34] Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation (2024), <https://arxiv.org/abs/2406.00515>
- [35] Jiang, L., Jin, X., Lin, Z.: Beyond classification: Inferring function names in stripped binaries via domain adapted llms. *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security* (2025)
- [36] Kaplan, J., McCandlish, S., Henighan, T.J., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws

- for neural language models. ArXiv **abs/2001.08361** (2020), <https://api.semanticscholar.org/CorpusID:210861095>
- [37] Katz, O., Olshaker, Y., Goldberg, Y., Yahav, E.: Towards neural decompilation. ArXiv **abs/1905.08325** (2019), <https://api.semanticscholar.org/CorpusID:160009986>
 - [38] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J.E., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with pagedattention (2023), <https://arxiv.org/abs/2309.06180>
 - [39] Le, H., Wang, Y., Gotmare, A.D., Savarese, S., Hoi, S.C.H.: Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* **35**, 21314–21328 (2022)
 - [40] Lee, J., Avgerinos, T., Brumley, D.: TIE: principled reverse engineering of types in binary programs. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society (2011), <https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs>
 - [41] Lin, Y., Gao, D.: When function signature recovery meets compiler optimization. In: *2021 IEEE Symposium on Security and Privacy (SP)*. pp. 36–52 (2021). <https://doi.org/10.1109/SP40001.2021.00006>
 - [42] Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: *Network and Distributed System Security Symposium* (2010), <https://api.semanticscholar.org/CorpusID:5694229>
 - [43] Liu, Z., Wang, S.: How far we have come: testing decompilation correctness of c decompilers. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. p. 475–487. *ISSTA 2020*, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3395363.3397370>, <https://doi.org/10.1145/3395363.3397370>
 - [44] Liu, Z., Chen, C., Li, W., Qi, P., Pang, T., Du, C., Lee, W.S., Lin, M.: Understanding r1-zero-like training: A critical perspective (2025), <https://arxiv.org/abs/2503.20783>
 - [45] Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for llvm. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. p. 65–79. *PLDI 2021*, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3453483.3454030>, <https://doi.org/10.1145/3453483.3454030>
 - [46] Mantovani, A., Compagna, L., Shoshitaishvili, Y., Balzarotti, D.: The convergence of source code and binary vulnerability discovery – a case study. In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. p. 602–615. *ASIA CCS '22*, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3488932.3497764>, <https://doi.org/10.1145/3488932.3497764>
 - [47] McCabe, T.J.: A complexity measure. *IEEE Transactions on Software Engineering* **SE-2**(4), 308–320 (1976). <https://doi.org/10.1109/TSE.1976.233837>
 - [48] Nethercote, N., Seward, J.: Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.* **42**(6), 89–100 (Jun 2007). <https://doi.org/10.1145/1273442.1250746>, <https://doi.org/10.1145/1273442.1250746>
 - [49] NVIDIA: NVIDIA Data Center Deep Learning Product Performance AI Inference. NVIDIA Developer <https://developer.nvidia.com/deep-learning-performance-training-inference/ai-inference>
 - [50] OpenAI: Gpt-o4-mini (2024), <https://openai.com/zh-Hans-CN/index/introducing-o3-and-o4-mini/>
 - [51] OpenAI: ChatGPT (May 2025 version). <https://chat.openai.com> (2025), large language model. Accessed: 2025-06-04
 - [52] Palit, I., Sharma, T.: Generating refactored code accurately using reinforcement learning. arXiv preprint arXiv:2412.18035 (2024)
 - [53] Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Lost in translation: A study of bugs introduced by large language models while translating code. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. p. 1–13. *ICSE '24*, ACM (Apr 2024). <https://doi.org/10.1145/3597503.3639226>, <http://dx.doi.org/10.1145/3597503.3639226>
 - [54] Posnett, D., Hindle, A., Devanbu, P.: A simpler model of software readability. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. p. 73–82. *MSR '11*, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/1985441.1985454>, <https://doi.org/10.1145/1985441.1985454>
 - [55] Reiter, P., Tay, H.J., Weimer, W., Doupé, A., Wang, R., Forrest, S.: Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation. *IEEE Transactions on Dependable and Secure Computing* **22**, 2270–2282 (2022), <https://api.semanticscholar.org/CorpusID:247155017>
 - [56] Riedmiller, M., Hafner, R., Lampe, T., Neunert, M., Degraeve, J., Wiele, T., Mnih, V., Heess, N., Springenberg, J.T.: Learning by playing solving sparse reward tasks from scratch. In: *International conference on machine learning*. pp. 4344–4353. *PMLR* (2018)
 - [57] Salwan, J., Saudel, F.: Triton: Framework d'exécution concolique et d'analyses en runtime. In: *Proceedings of the 2015 Symposium sur la Sécurité des Technologies de l'Information et des Communications*, ser. SSTIC. vol. 15 (2015)
 - [58] Scalabrino, S., Linares-Vásquez, M., Oliveto, R., Poshvyanyk, D.: A comprehensive model for code readability. *J. Softw. Evol. Process* **30**(6) (Jun 2018). <https://doi.org/10.1002/smr.1958>, <https://doi.org/10.1002/smr.1958>
 - [59] Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y.K., Wu, Y., Guo, D.: Deepseekmath: Pushing the limits of mathematical reasoning in open language models (2024), <https://arxiv.org/abs/2402.03300>
 - [60] Shojaei, P., Jain, A., Tipirneni, S., Reddy, C.K.: Execution-based code generation using deep reinforcement learning. arXiv preprint arXiv:2301.13816 (2023)
 - [61] Stiennon, N., Ouyang, L., Wu, J., Ziegler, D.M., Lowe, R., Voss, C., Radford, A., Amodei, D., Christiano, P.: Learning to summarize from human feedback. *NIPS '20*, Curran Associates Inc., Red Hook, NY, USA (2020)
 - [62] Tan, H., Luo, Q., Li, J., Zhang, Y.: Llm4decompile: Decompiling binary code with large language models. In: *Conference on Empirical Methods in Natural Language Processing* (2024), <https://api.semanticscholar.org/CorpusID:268297213>
 - [63] Taneja, J., Laird, A., Yan, C., Musuvathi, M., Lahiri, S.K.: Llm-vectorizer: Llm-based verified loop vectorizer. arXiv preprint arXiv:2406.04693 (2024)
 - [64] Trockman, A., Cates, K., Mozina, M., Nguyen, T., Kästner, C., Vasilescu, B.: "automatically assessing code understandability" re-analyzed: combined metrics matter. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. p. 314–318. *MSR '18*, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3196398.3196441>, <https://doi.org/10.1145/3196398.3196441>
 - [65] util-linux: util-linux. <https://github.com/util-linux/util-linux>
 - [66] Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse Engineers' processes. In: *29th USENIX Security Symposium (USENIX Security 20)*. pp. 1875–1892. *USENIX Association* (Aug 2020), <https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational>
 - [67] Wang, F., Shoshitaishvili, Y.: Angr - the next generation of binary analysis. In: *2017 IEEE Cybersecurity Development (SecDev)*. pp. 8–9 (2017). <https://doi.org/10.1109/SecDev.2017.14>
 - [68] Wang, Y., Xie, F.: Enhancing translation validation of compiler transformations with large language models (2024), <https://arxiv.org/abs/2401.16797>
 - [69] von Werra, L., Belkada, Y., Tunstall, L., Beeching, E., Thrush, T., Lambert, N., Huang, S., Rasul, K., Galloudec, Q.: TRL: Transformer Reinforcement Learning, <https://github.com/huggingface/trl>
 - [70] Wong, W.K., Wang, H., Li, Z., Liu, Z., Wang, S., Tang, Q., Nie, S., Wu, S.: Refining decompiled c code with large language models. ArXiv **abs/2310.06530** (2023), <https://api.semanticscholar.org/CorpusID:263829777>
 - [71] Xia, T., Hu, H., Wu, D.: DEEPTYPE: Refining indirect call targets with strong multi-layer type analysis. In: *33rd USENIX Security Symposium (USENIX Security 24)*. pp. 5877–5894. *USENIX Association*, Philadelphia, PA (Aug 2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/xia>
 - [72] Xie, D., Zhang, Z., Jiang, N., Xu, X., Tan, L., Zhang, X.: 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*. p. 4554–4568.

- CCS '24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3658644.3670340>, <https://doi.org/10.1145/3658644.3670340>
- [73] Xu, R., Cao, J., Lu, Y., Wen, M., Lin, H., Han, X., He, B., Cheung, S.C., Sun, L.: Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 23762–23779 (2025)
 - [74] Xu, X., Zhang, Z., Su, Z., Huang, Z., Feng, S., Ye, Y., Jiang, N., Xie, D., Cheng, S., Tan, L., Zhang, X.: Unleashing the power of generative model in recovering variable names from stripped binary (01 2025). <https://doi.org/10.14722/ndss.2025.240276>
 - [75] Yakdan, K., Dechand, S., Gerhards-Padilla, E., Smith, M.: Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22–26, 2016. pp. 158–177. IEEE Computer Society (2016). <https://doi.org/10.1109/SP.2016.18>, <https://doi.org/10.1109/SP.2016.18>
 - [76] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M.: No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2015. The Internet Society (2015). <https://www.ndss-symposium.org/ndss2015/no-more-gotos-decompilation-using-pattern-independent-control-flow-structuring-and-semantics>
 - [77] Yang, Z., Nagy, S.: Bin2Wrong: a unified fuzzing framework for uncovering semantic errors in Binary-to-C decompilers. In: 2025 USENIX Annual Technical Conference (USENIX ATC 25). pp. 1161–1179. USENIX Association, Boston, MA (Jul 2025), <https://www.usenix.org/conference/atc25/presentation/yang-zao>
 - [78] Yang, Z., Nagy, S.: Bin2wrong: a unified fuzzing framework for uncovering semantic errors in binary-to-c decompilers (Aug 2025)
 - [79] Yavuz, T., Bai, K.Y.: Analyzing system software components using api model guided symbolic execution. *Journal of Automated Software Engineering* (2020)
 - [80] Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.c., Kwon, Y., Aafer, Y., Zhang, X.: Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 813–832 (2021). <https://doi.org/10.1109/SP40001.2021.00051>
 - [81] Zhang, Z., Wang, C., Wang, Y., Shi, E., Ma, Y., Zhong, W., Chen, J., Mao, M., Zheng, Z.: Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proc. ACM Softw. Eng.* 2(ISSTA) (Jun 2025). <https://doi.org/10.1145/3728894>, <https://doi.org/10.1145/3728894>
 - [82] Zhu, C., Li, Z., Xue, A., Bajaj, A.P., Gibbs, W., Liu, Y., Alur, R., Bao, T., Dai, H., Doupé, A., Naik, M., Shoshitaishvili, Y., Wang, R., Machiry, A.: TYGR: Type inference on stripped binaries using graph neural networks. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 4283–4300. USENIX Association, Philadelphia, PA (Aug 2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/zhu-chang>
 - [83] Zou, M., Khan, A., Wu, R., Gao, H., Bianchi, A., Tian, D.J.: D-Helix: A generic decompiler testing framework using symbolic differentiation. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 397–414. USENIX Association, Philadelphia, PA (Aug 2024), <https://www.usenix.org/conference/usenixsecurity24/presentation/zou>