D-LIFT: Improving LLM-based Decompiler Backend via Code Quality-driven Fine-tuning

Muqi Zou[1], Hongyu Cai[1], Hongwei Wu[1], Zion Leonahenahe Basque[2], Arslan Khan[3], Z. Berkay Celik[1], Dave (Jing) Tian[1], Antonio Bianchi[1], Ruoyu (Fish) Wang[2], and Dongyan Xu[1]

[1]Purdue University
[2]Arizona State University
[3]Pennsylvania State University
[1]{zou116, hongyu, wu1685, zcelik, daveti, antoniob, dxu}@purdue.edu
[2]{zbasque,fishw}@asu.edu
[3]arslankhan@psu.edu

***Abstract*—As one of the key tools in many security tasks, decompilers reconstruct human-readable source code from binaries. Yet, despite recent advances, their outputs often suffer from syntactic and semantic errors and remain difficult to read. Recently, with the advent of large language models (LLMs), researchers began to explore the potential of LLMs to refine decompiler output. Nevertheless, our study of these approaches reveals their problems, such as introducing new errors and relying on unreliable accuracy validation.**

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 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 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 for readability to code that passes the accuracy check. Our implementation, based on Ghidra and a range of LLMs, demonstrates significant improvements for the accurate 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, as measured by D-SCORE. Overall, D-LIFT improves the quality of 68.2% of all the functions produced by the native decompiler.

I. INTRODUCTION

Binary decompilation is the process of generating high-level human-comprehensible source code from a binary program. As a fundamental component in security, decompilation supports extensive tasks, such as software reverse engineering [61], vulnerability discovery [40], [22], program patching and hardening [49], and cyber forensics [9]. However, source code generated by decompilers often has problems with syntactic and semantic correctness [37], [75], [11], [16]: A piece of decompiled code may not compile or preserve the semantics of the original binary. Moreover, despite advances in the past decade [7], [72], [12], [35], [74], [69], [68], [8], [33], even with syntactic and semantic correctness, the decompiled code may still be hard for human users to read.

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 [66], [56], [25], [29], [18], [28], [64], [67]. For example, to improve decompiled output readability, LLM4Decompile [56] trains new LLMs via supervised fine-tuning (SFT), and DeGPT [25] 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 Day Trending.

Unfortunately, applying LLMs to improve decompiled code may trade accuracy for readability. For instance, as shown in Figure 1, while the GPT-o4-mini [44] improves readability by recovering better variable names and collapsing nested loops into a single loop, it erroneously replaces the condition hex value 0x16 (line 18 of Ghidra's decompilation) with EBUSY (16 in decimal) instead of the correct constant (EINVAL), introducing a semantic discrepancy. Furthermore, our analysis of the above state-of-the-art LLM-based decompiled code improvement methods also shows significant limitations in preserving the original semantics. For LLM4Decompile, our study (detailed in Section VI-A3) reveals that, even when the original decompiled code is error-free, new inaccuracies are found in 93.2% of the functions generated by its fine-tuned LLM. For DeGPT, though it provides a semantic check during inference, its check is non-deterministic and never part of training; it cannot enhance the LLM's inherent capability to generate more accurate decompiled code. These findings demonstrate a clear need for a more rigorous, LLM-driven decompiled-code enhancement pipeline, one that incorporates

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240977
www.ndss-symposium.org

Original Decompiled Code:

```
1  long FUN_0011c6ec(undefined4
       param_1,undefined8 param_2,long
       param_3)
2  {
3    long lVar1;
4    int *piVar2;
5    long lStack_30;
6
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 *)__errno_location();
16     } while (*piVar2 == 4);
17     piVar2 = (int *)__errno_location();
18     if (*piVar2 != 0x16) break;
19     if (lStack_30 < 0x7ff00001) {
20       return lVar1;
21     }
22     lStack_30 = 0x7ff00000;
23   }
24   return lVar1;
25 }
```

GPT-o4-mini Improved Code:

```
1  #include <unistd.h>
2  #include <errno.h>
3
4  ssize_t write_with_retry(int fd, const
       void *buf, size_t count) {
5      const size_t MAX_CHUNK = 0x7FF00000;
6      size_t to_write = count;
7      ssize_t result;
8
9      while (1) {
10         result = write(fd, buf,
               to_write);
11         if (result >= 0) {
12             return result;
13         }
14         if (errno == EINTR) {
15             continue;
16         }
17         if (errno == EBUSY) {
18             if (to_write > MAX_CHUNK) {
19                 to_write = MAX_CHUNK;
20                 continue;
21             }
22         }
23         return result;
24     }
25 }
```

Source Code:

```
1  ptrdiff_t
2  safe_rw (int fd, bufptr buf, idx_t
       count)
3  {
4    for (;;)
5      {
6        ssize_t result = rw (fd, buf,
               count <= (size_t) -1 ?
               count : (size_t) -1);
7
8        if (0 <= result)
9          return result;
10       else if (IS_EINTR (errno))
11         continue;
12       else if (errno == EINVAL &&
               SYS_BUFSIZE_MAX < count)
13         count = SYS_BUFSIZE_MAX;
14       else
15         return result;
16     }
17 }
```

Fig. 1: The code snippet shows the original decompiled code, its corresponding source code, and the version improved by the GPT-o4-mini, where the line 17 of the GPT-o4-mini-refined code is misinterpreted, causing the function's semantics to diverge from both the original decompilation (line 18) and the true source (line 12). To generate the improved output, we fed GPT-4o-mini (text format, default medium reasoning) the prompt: "help me make the following code more readable without comments:{original decompiled code}".

a novel training paradigm to produce sturdier models and a rigorous inference-time accuracy check to ensure high-readability and semantics-preserving outputs.

To address these problems, we propose D-LIFT[1], an enhanced decompiler-LLM pipeline with a fine-tuned LLM using code quality-aware reinforcement learning. D-LIFT takes a baseline LLM, a decompiler, and a set of real-world binaries as inputs, trains a quality-aware LLM as the decompiler's backend. The enhanced decompiler-LLM pipeline will then output decompiled code of higher quality. Unlike prior work, D-LIFT follows a key principle: *preserving accuracy while improving readability*, throughout both its training and inference. To this end, D-LIFT first leverages reinforcement learning 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 assessment function, which provides multi-aspect scoring for LLM-generated decompiled code based on accuracy and readability measurements. Specifically, 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. 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 leveraging D-SCORE in reinforcement learning, D-LIFT enhances the LLMs' ability to generate higher-quality decompiled code. Meanwhile, by applying D-SCORE at inference, D-LIFT reliably 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 [13] and util-linux [59] 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. We have anonymously released our training scripts and evaluation results at [2]. 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 of the decompiled code, adhering to the principle of *preserving accuracy while improving readability*.
- We propose D-SCORE, an integrated scoring mechanism to quantitatively assess the quality of decompiled code.

[1]"D-LIFT" reflects the decompilation pipeline with a "**D**ecompiler" front-end and an "**L**LM w**i**th **F**ine **T**uning" back-end.

[2]https://github.com/XXXXX/xxx/tree/main

The framework incorporates existing analytical tools and proven 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 reliable selection metric for inference.
- We implement D-LIFT based on Ghidra and three LLMs, and achieve significant decompiled code improvement for commonly used benchmark functions.

II. Background and Motivation

A. Decompiled Code Accuracy

The accuracy of the decompiled code, including *syntactic* recompilation failures and *semantic* deviations between the decompiled code and the original source code, was first systematically evaluated using Equivalence Modulo Input (EMI) tests [37] at the function level. Since then, various methods, including symbolic execution [75], random testing [11], fuzzing [70], and manual inspection [16], have been employed to assess decompiled function's accuracy.

B. Decompiled Code Readability

Quantitative assessment of code readability has long been a topic in software engineering. Buse and Weimer (B&W) [10] led the way by recruiting 120 human evaluators to rate 100 short code function snippets, then building a mathematical model, featuring metrics like the number of variables per function, to evaluate the readability.

For decompilation, R2I [17] was the first approach to measure readability specifically for decompiled code at the function level. It defines a set of features that can be categorized into five feature groups: code quality, user preference, conflicting features, erroneous syntax, and general features. Analyzing these features extracted from the code from user surveys with 22 participants provides relative readability scores across different decompilers. Nevertheless, since R2I focuses on the comparison across different decompilers, many of its features cannot be generically applied to the LLM-generated code.

C. LLM application and training

1) Applications of LLMs: Large language models (LLMs) have seen wide adoption and application, particularly in the domain of code-related tasks. Industry has released numerous LLM-powered tools, e.g., GitHub Copilot [19], Google Gemini Code Assistant [20], and Meta LLaMA Coder [51], to streamline different programming tasks. Academia has mirrored this enthusiasm: the number of publications on LLM code programming rose from 11 in 2022 to 75 in 2023 and then to 140 in 2024—a 1,272% increase [27]. In the security domain, people have also leveraged LLMs to improve the decompiled code. For instance, during inference, DeGPT [25] introduces a three-role framework, optimization-scheme referee, rectification-measures advisor, and semantic-check operator, to assist the LLM in producing more readable code. Most recently, people use Model Context Protocol (MCP) servers to infer variable and function names ; one such tool, GhidraMCP [31], garnered 4.3K stars on GitHub in the first month after release.

2) Training LLMs: During LLM training, the most common strategy is to begin with a broad pre-training phase and then follow up with task-specific fine-tuning [36]. The pre-training allows the model to learn linguistic knowledge, such as C coding conventions, into its parameters. Fine-tuning then adapts the pre-trained model to the specific task using two main approaches: instruction tuning and reinforcement learning.

Instruction tuning uses the supervised learning paradigm, aiming to align the model's output with a single desired completion. For instance, LLM4Decompile [56] applies it to align the desired output with the original source code.

Reinforcement learning (RL), on the other hand, refines the model through iterative actions and feedback: the model generates candidate outputs (actions), which will be assessed by a reward function that provides feedback, guiding the model toward higher-quality results. In code generation, the feedback usually comes from frameworks, such as unit tests and compilers. Some RL policies, such as PPO [53], also require a single ideal completion to train a critic model that predicts long-term rewards, helping the model choose better actions over time. More recently, Group Relative Policy Optimization (GRPO) [54] achieved promising results on math-related tasks. By normalizing the rewards of candidate outputs instead of relying on a critic model, GRPO eliminates the limitation of accepting only one ideal completion.

D. Motivations

Although LLM-based decompiled code improvement is increasingly prevalent, it still has several major limitations.

LLMs introduce inaccuracy in the decompiled code. 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 [73], [47], [27]. In the decompiled-code improvement scenario, we also observe that LLM-refined output frequently shows new errors. Here, we follow the previous work [37] 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 III, an average 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 the wrong variable, that are hard to spot yet vital for accuracy.

These observations reveal that though LLMs may improve readability, they often introduce inaccuracy of decompiled code, a factor that prior studies [75], [37], [11], [16], [70] have identified as vital for effective decompilation. Hence, we propose an overarching principle for improving decompiled code quality: *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

```
int main() {
 char a[100];
 unsigned int b=0;
 printf("Enter text:");
 if(fgets(a,sizeof(a),stdin)
      != NULL) {
  while(b<10){
   printf("%c",a[b]);
   b++;
  }
 }
 else
  return 2;
 return 1+1+1;
}
```

```
int main() {
 char c[100];
 int d=0;
 printf("Enter text:");
 if(fgets(c,sizeof(c),
      stdin) != NULL) {
  for(d=0;d<10;d++){
   printf("%c",c[d]);}
 } else {
  return 1+1;
 }
 return 3;
}
```

Fig. 2: Two code snippets generate the same binary code. Notably, differences appear in every line except the first and fourth lines.

readable code through training, it compromises the accuracy of its output, as shown in Table III. 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.

III. Design Challenges

Section II-D motivates us to propose a "decompiler-LLM" pipeline without sacrificing accuracy. In this section, we summarize the design-level challenges.

A. Framework Design

As our framework is built around LLMs, we divide the challenges into two parts: training and inference.

1) Training: Improving the code quality of LLMs through training is not new [32], [55], [56], [46], [15]. However, many of these methods are challenging to adapt for decompiled code improvement. Unlike general code improvement tasks, enhancing decompiled code faces a unique challenge: the existence of multiple valid 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, as illustrated in Figure 2, two different source files, each around 300 characters long and with an edit distance of 74, produce the same binary output using GCC [1] with `-O2` optimization. In practice, however, researchers typically have access to only one of these source variants as the ground truth. Moreover, as mentioned in Section II-C, 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 only.

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 their quality. Unfortunately, the only existing framework, DeGPT, offers no automated readability metric, relying instead 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) necessitates a quantitative code-quality metric to evaluate each candidate. To the best of our knowledge, no existing metric meets the dual requirement of maintaining syntactic and semantic accuracy while enhancing readability.

For decompiler accuracy, as introduced in Section II-D, two aspects of accuracy, syntactic and semantic, need to be evaluated. Since the decompiled code assessment function must be deterministic, methods such as fuzzing [70] and its derivatives, like `MSSC` [11] 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` [39]. That leaves `D-helix` [75], 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.

For readability, most existing metrics [10], [42], [48], [52] ignore decompiler-specific artifacts (e.g., the number of references/dereferences). To the best of our knowledge, R2I [17] is the only metric tailored specifically for decompiled code. However, as described in Section II-B, since it's 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 [62] or RetDec [5] outputs against those from Ghidra [3] and Hex-Rays [2], which cannot be computed when only one piece of decompiled code is given.

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 of decompiled code. The workflow of D-LiFT, illustrated in Figure 3, 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

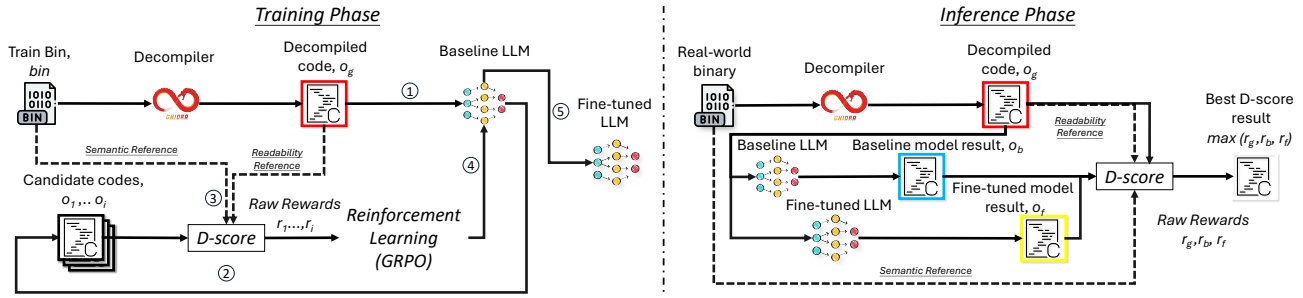


Fig. 3: This figure shows how D-LIFT fine-tunes the baseline model and selects the output. It applies GRPO to overcome "single decompiled code segments can correspond to multiple semantically equivalent source representations" in training and uses D-SCORE to select the best quality code among inference.

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, all based on the outputs' D-SCORE.

1) Training Phase: The challenge described in Section III-A highlights a critical limitation in current approaches: 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) [54] overcomes this constraint by generating the loss from the normalized rewards across multiple candidate outputs.

Figure 3 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,$\{o_1, o_2, ...o_i\}$. (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, $\mathbf{r} = \{r_1, r_2, ...\}$ within each candidate group according to its standard formulation:

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

and uses these normalized rewards to compute the loss for fine-tuning the model as follows:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}} \left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\left(r_t(\theta)\hat{A}_{i,t}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_{i,t}\right) \right] - \beta D_{\text{KL}}\left(\pi_\theta \| \pi_{\text{ref}}\right) \tag{2}$$

Given that the decompilation task accepts multiple valid ground truths, D-LIFT eliminates the weight related to the reference by setting β to zero, thereby removing dependence on a single reference solution. In practice, setting β to zero not only reduces memory and computational overhead but also improves training effectiveness, in line with recent findings [24], [38]. The additional variables referenced in Equation (2) serve to modify the baseline model parameters, with detailed definitions available in the original GRPO research publication [54]. (5) Finally, once the training iterations are complete, D-LIFT outputs the fully fine-tuned LLM.

2) Inference Phase: As described in Section II-D, 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 3. 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 4 shows the overall procedure of D-SCORE. As a code quality assessment function, D-SCORE takes three inputs: the original decompiled code from the decompiler, o_g, the original binary, *bin*, and a candidate code waiting for evaluation, o_i,

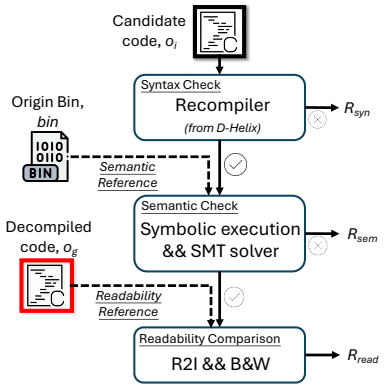


Fig. 4: D-SCORE, following our principle of *preserving accuracy while improving readability*. It performs three sequential checks on each candidate code: syntax/semantic verification, and readability assessment. If any of the accuracy checks fail, D-SCORE halts immediately and returns a penalty score.

and outputs an absolute score r_i reflecting both the accuracy and readability of o_i. Formally, r_i is defined as follows:

$$\begin{aligned} r_i = & \mathbb{1}(\ \text{cond}_{syn}\ ) \cdot \big(R_{syn}(o_i)\big) \\ & + \mathbb{1}(\neg\ \text{cond}_{syn}\ ) \cdot (\mathbb{1}(\ \text{cond}_{sem}\ ) \cdot R_{sem}(o_i, bin) \\ & + \mathbb{1}(\neg\ \text{cond}_{sem}\ ) \cdot R_{read}(o_i, o_g))), \\ \text{where} \quad & \text{cond}_{syn} = \big(R_{syn}(o_i)\ !=\ pass\big), \\ & \text{cond}_{sem} = \big(R_{sem}(o_i, bin)\ !=\ pass\big) \end{aligned} \tag{3}$$

Within the above equation, R_Y represents the reward return from each specific check in field Y and $\mathbb{1}(\ \text{cond}_x)$ is an "if-then-else" statement that returns 1 if the cond_x is True and returns 0 otherwise. Meanwhile, following our principle, inaccuracies are penalized more heavily than readability issues:

$$max(R_{syn}) < min(R_{sem}) \text{ and } max(R_{sem}) < min(R_{read}) \tag{4}$$

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 syntax and semantic 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 overall design ensures D-SCORE only awards readability once the accuracy has been verified, thereby adhering to our core principle.

1) Syntax Metric: As prior work has shown [64], [75], decompiled output often fails to compile directly, even at the function level. To address this and to feed cleaner code for downstream semantic validation, D-SCORE adopts `D-helix`'s `Recompiler`. As an iterative recompilation tool, `Recompiler` 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 submits it to `Recompiler` and obtains a syntax score, R_{syn}, based on `Recompiler`'s feedback. Specifically, D-SCORE assign $R_{syn}(o_i)$ to syn_{pen} when the o_i cannot be recompiled and *pass* otherwise. Formally:

$$R_{syn}(o_i) = \begin{cases} syn_{pen}, & \text{if } o_i \text{ cannot be recompiled.} \\ pass, & \text{otherwise} \end{cases} \tag{5}$$

Note that syn_{pen} contributes directly to D-SCORE (See Equation (3)), when there is a syntax error, ensuring a clear penalty.

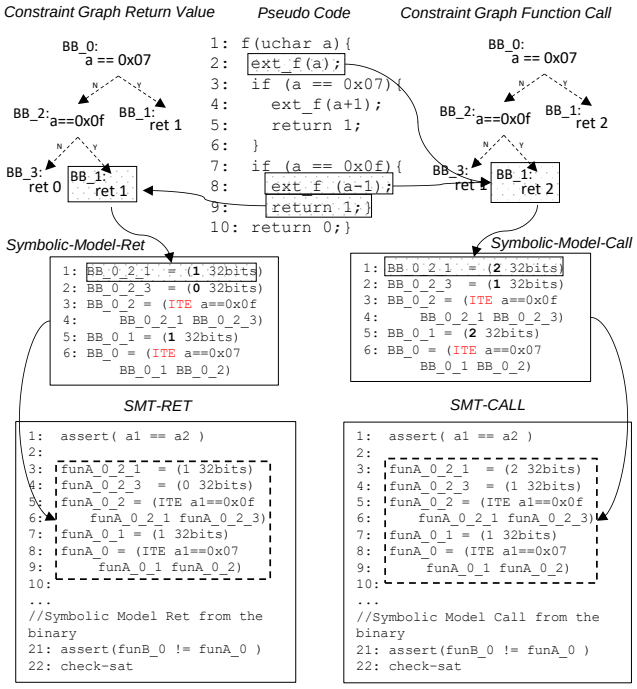


Fig. 5: The workflow of semantic check in D-SCORE. The left side shows how the function return value is checked, and the right side shows how the external function call is examined. For both scenarios, D-SCORE runs symbolic execution with symbolic inputs. After that, it constructs symbolic models, representing the function's behaviors, by using If-Then-Else statements, which connect the inputs to the function's return values/external calls.

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 original binary and the intermediate representation (IR) code of o_i. To better describe, we split into two parts: 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 mostly adopt `D-helix`'s approach, which employs symbolic execution and `SMT` solver. Specifically, as shown on the left side of Figure 5, 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 `SMT` solver on `SMT-RET` to compare the symbolic models between the original binary and the decompiled code.

When verifying external function calls, however, directly applying `D-helix` to test LLM-generated candidate code (i.e.,

o_i) can yield many false negatives. This limitation stems from `D-helix`'s fundamental dependence on decompiled code to establish ground truth for verifying external function calls. Specifically, it models external function calls by approximating return values through the sum of the least significant bytes of those arguments. However, this external function call modeling process is only initialized when an external function call is detected within the decompiled code. Since instructions, including function calls, are often omitted in the LLM-generated code, `D-helix` may *not* properly initialize its function call modeling procedures, leading to possible false negatives. In fact, accurately restoring the prototypes for external function calls is an open problem [34], as this "ground truth" information is absent from the binary itself.

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 5. 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:

$$R_{sem}(o_i, bin) = \begin{cases} ret_{pen}, \text{ if } (check_{ret}(o_i, bin) == false) \\ call_{pen}, \text{ if } (check_{ret}(o_i, bin) == true, \\ \qquad \text{and } check_{call}(o_i, bin) == false) \\ pass, \text{ otherwise} \end{cases} \tag{6}$$

, 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: To evaluate the readability of the candidate code, o_i, fairly, we generate a relative readability score by directly comparing it against the original decompiled code, o_g. However, as described in Section III-B, none of the existing ones can be directly applied in our context. To address this limitation, we customize established software engineering readability models by integrating selected features from the only decompiler-related metric, R2I [17], and the most classic readability metric, B&W [10] as follows:

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

, where γ and δ weight each metric's contribution. Our metric adopts the same principle as R2I and B&W, aggregating multiple code features by multiplying each feature's count by its weight and summing the results. Moreover, to blend them effectively, we normalize their individual scores to a fixed interval, allowing precise control over their relative influence.

To compute $R_{b\&w}(o_i, o_g)$, we follow the B&W framework, which defines a feature set, $\mathbf{f}_{b\&w} = \{f_1, f_2, ...\}$, e.g., average number of commas per line, and their associated weights, $\mathbf{w}_{b\&w} = \{w_1, w_2, ...\}$. 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,

$$\begin{aligned} R_{b\&w}(o_i, o_g) = & sigmoid(\frac{R_{mul}(o_g) - R_{mul}(o_i)}{min(R_{mul}(o_i), R_{mul}(o_g))}), \\ & \text{where} \quad R_{mul}(x) = (\sum \mathbf{f}_{b\&w}(x) \cdot \mathbf{w}_{b\&w}) \end{aligned} \tag{8}$$

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 research 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:

$$\begin{aligned} R_{R2I}(o_i, o_g) = & \sum \mathbf{w}_{R2I} \cdot r_{elog}(\mathbf{f}_{R2I}(o_i) - \mathbf{f}_{R2I}(o_g)), \\ \text{where} \quad & r_{elog}(x) = r \cdot e^{-\log_{10}(1+x)} \\ & + (1-r) \cdot \left(1 - e^{-\log_{10}(1+x)}\right) \end{aligned} \tag{9}$$

V. IMPLEMENTATION

We implemented D-LIFT as a modular Python framework, using the SOTA open-source decompiler, Ghidra (version 11.2).

A. Reinforcement learning

We run TRL [60] (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 follow its default configuration by setting the maximum iteration count to 10.

For the semantics checking, we implement our semantic check framework on angr [62] (version 9.2), z3 [14] (version 4.13), and prompt [71] (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 (4) and experimental observations to prevent sparse rewards [50]. We leverage the fact that both B&W and R21 are defined and validated via user studies [17], [10], reflecting key readability features of decompiled code such as the number of references, do-while constructs, and goto labels.

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 specific decompiled snippets to improve both accuracy 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) [43], two 32-core AMD EPYC 7543 CPUs, and 400 GB of RAM.

Dataset. Our metric evaluation uses a dataset of 1,948 decompiled functions sourced from binaries that the most classic previous literature uses, which are `coreutils` [13] (v9.5) and `util-linux` [59] (v2.41). Specifically, we first use GCC [1] to compile these projects with -O2 optimization 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 then filter this set to include only those with at least 20 lines of code and a cyclomatic complexity [41] greater than 3, resulting in 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 established metrics [10], [17], 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` [6] 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.

#	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.

Accuracy	0.9450
Precisions	0.9100
Recall	0.9785
F1	0.9430

TABLE II: The accuracy, precision, recall, and F1 of D-SCORE on LLM-generated output.

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 II 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 D-SCORE with multiple return values may exercise most branches and thus eliminate these false negatives. If the LLM's output omits code for those unvisited branches, D-SCORE cannot detect the discrepancy and thus overlooks the error.

2) Dataset: We first present an overview of the dataset, then explain how we construct our training and evaluation datasets.

Dataset overview. Given that D-SCORE supports only 1,573 functions (see Section VI-A1), we center our overview on this filtered set. To better understand our dataset, we first 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. Of the 1,573 functions, 1,025 are classified as originally accurate (i.e., pass

#	Model Name	Synt Errs	Sem Errs	Total Errs
1	Qwen2.5-Coder-1.5B	373 (36.4%)	92 (8.98%)	465 (45.4%)
2	Qwen2.5-Coder-3B	238 (23.2%)	110 (10.7%)	348 (34.0%)
3	Llama3.2-3B	371 (36.2%)	180 (17.6%)	551 (53.8%)
4	LLM4Decompile-End-1.3B	887 (86.5%)	68 (6.63%)	955 (93.2%)

TABLE III: Different baseline models' performance on the originally accurate (OA) decompiled code. Specifically, it shows how many functions become inaccurate *after* being processed by the baseline LLM (i.e., without fine-tuning).

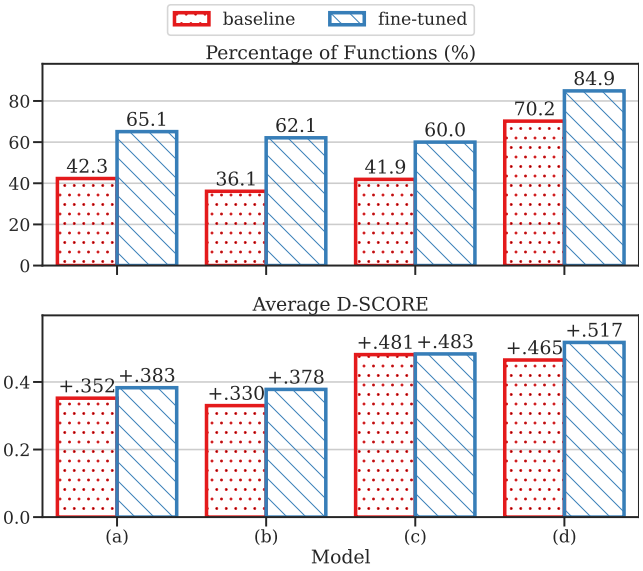
D-SCORE's accuracy check), while the remaining 548 fall into originally inaccurate (i.e., fail the D-SCORE's accuracy check), 307 exhibiting syntax errors and 241 containing semantic errors.

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 [30] with a memory-efficient setting of 0.7, we still encountered out-of-memory errors when running D-LIFT on models larger than 3 billion parameters.

Therefore, we select two code-focused LLMs, `Qwen2.5-Coder` [26] (`1.5B` and `3B` variants) and one general-purpose model, `Llama3.2-3B` [21]. As shown in Table III, due to the poor performance of `LLM4Decompile-End-1.3B` [56] 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 our 1,025 originally accurate functions, score the outputs using D-SCORE (i.e., -3 if the syntax check is not passed, -2 or -1.5 if the semantic check is not passed and score in (-1,1) for the readability) and show the results in Table III. Since being fine-tuned based on the model released from November 2023, `LLM4Decompile-End-1.3B` made 955 functions (93.2%) inaccurate. In contrast, the three newer LLMs (released around September 2024) made only 44.4% of



(a) Qwen2.5-Coder-1.5B (b) Qwen2.5-Coder-3B (c) Llama3.2-3B (d) All

Fig. 6: The performance of Different models on the OA dataset, before and after fine-tuning by D-LIFT. Specifically, it shows how many functions are improved by the model(s) compared with the original decompiled code, and these functions from D-SCORE.

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 the underlying reasons for the differences.

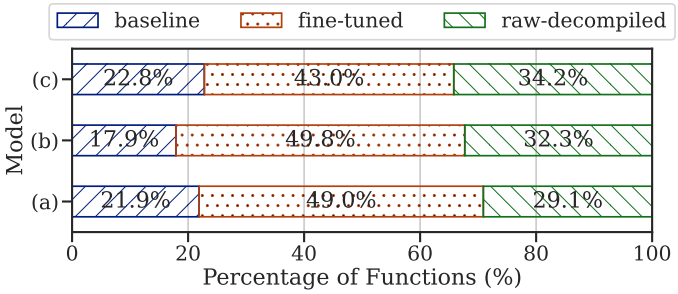
1) Result: To clearly present, we split the result into two parts: (1) a comparison between baseline and fine-tuned LLMs, and (2) an assessment of D-LIFT 's overall performance.

Fine-tuned LLMs vs. baseline LLMs. Figure 6 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 7 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

#	Model Name	Improvements				Regressions			
		Syntax	Semantic	Readability	Total	Syntax	Semantic	Readability	Total
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 IV: This table shows how many functions get improved and regressed by the training of D-LIFT. This comparison happens between the output of the fine-tuned model and the baseline model, where the improvement refers to the function getting a higher score from the fine-tuned model and the regression refers to the function getting a higher score from the baseline model.



(a) Qwen2.5-Coder-1.5B (b) Qwen2.5-Coder-3B (c) Llama3.2-3B

Fig. 7: Overall performance of the selection system within D-LIFT on the OA dataset. This chart shows, for each function, which source, baseline LLM, fine-tuned LLM, or original decompiler achieves the highest D-SCORE score and reports the percentage of functions in each category.

Performance	Error Type	No. Func
Improvement	Missing instruction	24
	Incorrect brackets	12
	Incorrect variable naming and casting	5
	Unwanted instruction insertions	7
	Incorrect literal value	2
Degradation	Missing instruction	14
	Incorrect brackets	24
	Unwanted instruction insertions	12

TABLE V: This table shows the root causes of both improvements and regressions caused by D-LIFT, where the result is concluded by manually reviewing randomly selected 100 functions.

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 (average +0.585), while only 100 functions remained unimproved.

2) Findings: To better understand the training part of the D-LIFT's influence on LLM performance, we conducted an additional analysis by directly 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 IV 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

#	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%)	-0.614
4	All baseline 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

[1] "No. of Func" shows how many functions were improved and the percentage relative to the total functions.
[2] "Avg. D-S." is the average D-SCORE difference (improved vs. raw decompiled).

TABLE VI: Different models' performance on the OIA dataset, before and after fine-tuning by D-LIFT. Specifically, it shows how many functions are improved by the model(s) compared with the raw decompiled code and these functions' score from D-SCORE

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, summarized in Table V, show that fine-tuning corrects five main categories of errors: (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-tuning also introduces regressions in three areas: (1) Missing instructions. (2) Incorrect brackets in conditional expressions. (3) Unwanted instruction insertions. More representative 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 error, 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. In Section VI-C1, we compare outputs from the baseline and fine-tuned models. In Section VI-C2, we analyze the underlying reasons for these observed scores.

1) Result: Table VI 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

Source:

```
void uuid_copy(uuid_t dst,
     const uuid_t src)
{
 unsigned char *cp1;
 const unsigned char *cp2;
 int i;

 for (i=0, cp1 = dst, cp2
      = src; i < 16; i++)
 *cp1++ = *cp2++;
}
```

Original Decompiler:

```
void uuid_copy (long
     param_1,long param_2)

{
 long lVar1;

 lVar1 = 0;
 do {
 *(undefined *) (param_1 +
      lVar1) = *(undefined
      *)(param_2 + lVar1);
  lVar1 = lVar1 + 1;
 } while (lVar1 != 0x10);
 return;
}
```

Baseline model:

```
void uuid_copy (long
     dest, long src)
{
 long i;

 for (i = 0; i < 16;
      i++)
 {
  dest[i] = src[i];
 }
}
```

Fine-tuned model:

```
void uuid_copy(long dest,
     long src)
{
 long i;

 for (i = 0;i < 0x10; i++)
 {
 *(unsigned char *)(dest +
      i) = *(unsigned char
      *)(src + i);
 }
}
```

Fig. 8: This code snippet shows how D-LIFT helps `Qwen2.5-Coder-3B` enhance its ability to generate more readable code while correcting the syntax error *"error: subscripted value is neither array nor pointer nor vector"* that appears at line 7 of the baseline output.

Source:

```
if(!tb ||
     !tb->colors_wanted ||
     tb->format !=
     SCOLS_FMT_HUMAN)
 return NULL;
...
 tb->cur_color = color;
```

Original decompiler:

```
if (param_1 == 0) {
 return;
}
if ((*(byte *)(param_1 +
     0x140) & 2) == 0) {
 return;
}
if (*(int *)(param_1 + 0x108)
     != 0) {
 return;
}
...
*(char **)(param_1 + 0x120) =
     param_3;
```

Baseline model:

```
if (!param_1) return;
if (!(*(byte*)(param_1 +
     0x140) & 2)) return;
if (*(int*)(param_1 + 0x108)!=
     0) return;
...
*(char **)param_1 + 0x120 =
     param_3;
```

Fine-tuned model:

```
if (param_1 == 0) return;
if (((*(byte *)(param_1 +
     0x140) & 2) == 0) ||
     (*(int *)(param_1 +
     0x108)!= 0)) return;
...
*(char **)(param_1 + 0x120)
     = param_3;
```

Fig. 9: 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 *"error: lvalue required as left operand of assignment."* that appears at line 5 of the baseline output.

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 typically converted the variables with these opaque declarations into fixed-length arrays and updated member accesses accordingly. 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, fixes semantic errors introduced by the LLM in Section VI-D2

1) Syntax Error Fixes: ***Fixing Incorrect Variable Casting and Refactor Loop.*** Figure 8 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

Source:

```
int fdisk_delete_all_partitions
     (struct fdisk_context *cxt)
{
 size_t i;
 int rc = 0;

 if (!cxt || !cxt->label)
  return -EINVAL;

 for (i = 0; i <
      cxt->label->nparts_max; i++) {

  if (!fdisk_is_partition_used(cxt,
       i))
   continue;
  rc = fdisk_delete_partition(cxt,
       i);
  if (rc)
   break;
 }

 return rc;
}
```

Original decompiler:

```
undefined8 fdisk_delete_all_partitions(long param_1)

{
 int iVar1;
 undefined8 uVar2;
 ulong uVar3;

 if (param_1 == 0) {
  return 0xfffffffea;
 }
 if (*(long *)(param_1 + 0x180) == 0) {
  uVar2 = 0xfffffffea;
 }
 else {
  uVar3 = 0;
  if (*(long *)(*(long *)(param_1 + 0x180) + 0x30)!= 0) {
   do {
    iVar1 = fdisk_is_partition_used(param_1,uVar3);
    if ((iVar1 != 0) && (uVar2 = fdisk_delete_partition(param_1,uVar3),
         (int)uVar2 != 0)) {
     return uVar2;
    }
    uVar3 = uVar3 + 1;
   } while (uVar3 < *(ulong *)(*(long *)(param_1 + 0x180) + 0x30));
  }
  uVar2 = 0;
 }
 return uVar2;
}
```

Baseline model:

```
int fdisk_delete_all_partitions(long param_1) {
 int iVar1;
 long lVar2;
 long lVar3;
 long lVar4;

 if (param_1 == 0) {
  return -65538; // Equivalent to 0xfffffffea
 }
 if (*(long *)(param_1 + 0x180) == 0) {
  return -65538; // Equivalent to 0xfffffffea
 }
 lVar2 = 0;
 if (*(long *)(*(long *)(param_1 + 0x180) + 0x30)!= 0) {
  do {
   iVar1 = fdisk_is_partition_used(param_1, lVar2);
   if ((iVar1!= 0) && (lVar3 =
        fdisk_delete_partition(param_1, lVar2),
        (int)lVar3!= 0)) {
     return lVar3;
     // Return error code if partition deletion fails
   }
   lVar2 = lVar2 + 1;
  } while (lVar2 < *(ulong *)(*(long *)(param_1 + 0x180) +
      0x30));
 }
 return 0; // Return success
}
```

Fine-tuned model:

```
int fdisk_delete_all_partitions(long param_1) {
 int iVar1;
 long lVar2;
 long lVar3;
 long lVar4;
 long lVar5;

 if (param_1 == 0) {
  return 0xfffffffea;
 }
 if (*(long *)(param_1 + 0x180) == 0) {
  return 0xfffffffea;
 }
 lVar2 = *(long *)(*(long *)(param_1 + 0x180) + 0x30);
 if (lVar2!= 0) {
  lVar3 = 0;
  do {
   iVar1 = fdisk_is_partition_used(param_1, lVar3);
   if ((iVar1!= 0) && (lVar5 =
        fdisk_delete_partition(param_1, lVar3),(int)lVar5!=
        0)) {
    return lVar5;
   }
   lVar3 = lVar3 + 1;
  } while (lVar3 < lVar2);
 }
 return 0;
}
```

Fig. 10: This code snippet shows how D-LIFT helps `Qwen2.5-Coder-3B` enhance its ability to generate more readable code while correcting the semantic error. Specifically, the return value `-65538` at line 8 and line 11 of the baseline output is not the same as `0xfffffffea`.

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. 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 9 demonstrates another example about how D-LIFT corrects the syntax errors while enhancing readability in the decompiled `fputs_color_cell_close` (`util-linux`).

Specifically, for inaccuracy, line 5 in baseline

`Qwen2.5-Coder-1.5B` model output lacks the necessary parentheses around the assignment target, 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.

2) Semantic Error Fixes: ***Fixing Incorrect Literal Value and Extracting Functional Variable.*** Figure 10 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 `0xfffefffe` rather than the intended `0xfffffffea`. Our fine-tuned model fixes this by emitting `return 0xfffffffea;` 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 0xfffffffea` 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.

VII. DISCUSSION

Applicability. 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, such as those for type recovery [35], [12], control flow graph reconstruction [7], and function boundary recovery [65], which complement decompilation.

Underlying tools. Since D-LIFT employs D-helix, it inherits the limitations of D-helix, including missing support for floating-point instructions, double pointers (due to the memory model), and large binaries because of timeouts.

External function calls. For external function calls, D-SCORE differs from `D-helix`'s approach of modeling return values as the sum of the least significant bytes of its arguments; Instead, D-SCORE counts the number of calls. This simplification can be problematic when a function's return value affects control flow. By default, we assign a constant value (e.g., 0) as the return value of every external function call, which may prevent exploration of branches. Nevertheless, accurately determining whether an external call yields a meaningful return itself is an open challenge [34]. We leave the more precise modeling of external function calls as future work.

VIII. RELATED WORK

Regarding LLM-based decompiled-code enhancement, several approaches have been proposed. Besides LLM4Decompiler [56], researchers have also proposed DecGPT [64], 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 [25] 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 [55] 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 [15], [46], [32] 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 [57] applies Alive2 [39] on the vectorized code generated by LLM to verify its semantic correctness. Similarly, Wang [63] 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 have been proposed based on the B&W framework. Specifically, researchers [23], [52], [48], [42], [58] 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 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 [45] and Claude-AI [4] 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.

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] Anthropic. Claude AI (Claude 3, May 2025 version). https://claude.ai, 2025. Large language model. Accessed: 2025-06-04.

[5] Avast Software. RetDec: A retargetable machine-code decompiler. https://retdec.com/.

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenhang Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, K. Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Yu Bowen, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xing Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *ArXiv*, abs/2309.16609, 2023.

[7] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! there is no need to DREAM of c: A Compiler-Aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, Philadelphia, PA, August 2024. USENIX Association.

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

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

[10] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[11] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 491–502, New York, NY, USA, 2024. Association for Computing Machinery.

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

[13] coreutils. coreutils. http://git.savannah.gnu.org/gitweb/?p=coreutils.git.

[14] Leonardo De Moura and Nikolaj Bjørner. 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*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

[16] Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of c decompiler fidelity issues. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 379–396, Philadelphia, PA, August 2024. USENIX Association.

[17] Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. R2i: A relative readability metric for decompiled code. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[18] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. *Coda: an end-to-end neural program decompiler*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[19] GitHub and OpenAI. Github copilot. https://github.com/features/copilot, 2021. Available at https://github.com/features/copilot.

[20] Google. Gemini Code Assist. https://developers.google.com/gemini-code-assist, 2025. Accessed: 2025-06-04.

[21] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and Abhishek Kadian et al. The llama 3 herd of models, 2024.

[22] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. Queryx: Symbolic query on decompiled code for finding bugs in cots binaries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3279–3295, 2023.

[23] Gustaf Holst and Felix Dobslaw. On the importance and shortcomings of code readability metrics: A case study on reactive programming, 2021.

[24] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.

[25] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[26] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186, 2024.

[27] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[28] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 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.

[29] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *ArXiv*, abs/1905.08325, 2019.

[30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[31] LaurieWired. Ghidramcp: Mcp server for ghidra. https://github.com/LaurieWired/GhidraMCP, 2025. Accessed: 2025-06-04.

[32] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[33] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 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.

[34] Yan Lin and Debin Gao. When function signature recovery meets compiler optimization. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 36–52, 2021.

[35] Zhiqiang Lin, X. Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium*, 2010.

[36] Yiheng Liu, Hao He, Tianle Han, Xu Zhang, Mengyuan Liu, Jiaming Tian, Yutong Zhang, Jiaqi Wang, Xiaohui Gao, Tianyang Zhong, Yi Pan, Shaochen Xu, Zihao Wu, Zhengliang Liu, Xin Zhang, Shu Zhang, Xintao Hu, Tuo Zhang, Ning Qiang, Tianming Liu, and Bao Ge. Understanding llms: A comprehensive overview from training to inference, 2024.

[37] Zhibo Liu and Shuai Wang. 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*, ISSTA 2020, page 475–487, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025.

[39] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 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*, ASIA CCS '22, page 602–615, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[42] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understand-

ability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[43] NVIDIA. NVIDIA Data Center Deep Learning Product Performance AI Inference. *NVIDIA Developer*.

[44] OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, and Aditya Ramesh et al. Gpt-4o system card, 2024.

[45] OpenAI. ChatGPT (May 2025 version). https://chat.openai.com, 2025. Large language model. Accessed: 2025-06-04.

[46] Indranil Palit and Tushar Sharma. Generating refactored code accurately using reinforcement learning. *arXiv preprint arXiv:2412.18035*, 2024.

[47] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 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*, ICSE '24, page 1–13. ACM, April 2024.

[48] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation. *IEEE Transactions on Dependable and Secure Computing*, 22:2270–2282, 2022.

[50] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *International conference on machine learning*, pages 4344–4353. PMLR, 2018.

[51] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023. Accessed: 2025-06-04.

[52] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *J. Softw. Evol. Process*, 30(6), June 2018.

[53] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[54] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

[55] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

[56] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Conference on Empirical Methods in Natural Language Processing*, 2024.

[57] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. Llm-vectorizer: Llm-based verified loop vectorizer. *arXiv preprint arXiv:2406.04693*, 2024.

[58] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: combined metrics matter. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 314–318, New York, NY, USA, 2018. Association for Computing Machinery.

[59] util-linux. util-linux. https://github.com/util-linux/util-linux.

[60] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformer Reinforcement Learning.

[61] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse Engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892. USENIX Association, August 2020.

[62] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, 2017.

[63] Yanzhao Wang and Fei Xie. Enhancing translation validation of compiler transformations with large language models, 2024.

[64] Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *ArXiv*, abs/2310.06530, 2023.

[65] Tianrou Xia, Hong Hu, and Dinghao Wu. DEEPTYPE: Refining indirect call targets with strong multi-layer type analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5877–5894, Philadelphia, PA, August 2024. USENIX Association.

[66] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4554–4568, New York, NY, USA, 2024. Association for Computing Machinery.

[67] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. Unleashing the power of generative model in recovering variable names from stripped binary. 01 2025.

[68] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 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*, pages 158–177. IEEE Computer Society, 2016.

[69] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[70] Zao Yang and Stefan Nagy. Bin2wrong: a unified fuzzing framework for uncovering semantic errors in binary-to-c decompilers. August 2025.

[71] Tuba Yavuz and Ken (Yihang) Bai. Analyzing system software components using api model guided symbolic execution. *Journal of Automated Software Engineering*, 2020.

[72] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832, 2021.

[73] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025.

[74] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. TYGR: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, Philadelphia, PA, August 2024. USENIX Association.

[75] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-Helix: A generic decompiler testing framework using symbolic differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 397–414, Philadelphia, PA, August 2024. USENIX Association.