

Article

Interactive End-to-End Decompilation via Large Language Models

Yunlong Feng ¹, Bohan Li ¹, Xiaoming Shi ², Qingfu Zhu ^{1,*} and Wanxiang Che ^{1,*}¹ Research Center for Social Computing and Interactive Robotics, Harbin Institute of Technology, Harbin 150001, China; ylfeng@ir.hit.edu.cn (Y.F.); bhli@ir.hit.edu.cn (B.L.)² School of Computer Science and Technology, East China Normal University, Shanghai 200062, China; xmshi@cs.ecnu.edu.cn

* Correspondence: qfzhu@ir.hit.edu.cn (Q.Z.); car@ir.hit.edu.cn (W.C.)

Abstract

The goal of decompilation is to convert compiled low-level code (e.g., assembly code) back into high-level programming languages, enabling analysis in scenarios where source code is unavailable. This task supports various reverse engineering applications, such as vulnerability identification, malware analysis, and legacy software migration. The end-to-end decompilation method based on large language models (LLMs) reduces reliance on additional tools and minimizes manual intervention due to its inherent properties. However, previous end-to-end methods often lose critical information necessary for reconstructing control flow structures and variables when processing binary files, making it challenging to accurately recover the program's logic. To address these issues, we propose the ReF Decompile method, which incorporates the following innovations: (1) The Relabeling strategy replaces jump target addresses with labels, preserving control flow clarity. (2) The Function Call strategy infers variable types and retrieves missing variable information from binary files. Experimental results on the Humaneval-Decompile Benchmark demonstrate that ReF Decompile surpasses comparable baselines and achieves state-of-the-art (SOTA) performance of 61.43%.

Keywords: large language model; code generation

Academic Editor: Arkaitz Zubiaga

Received: 2 November 2025

Revised: 10 November 2025

Accepted: 12 November 2025

Published: 14 November 2025

Citation: Feng, Y.; Li, B.; Shi, X.; Zhu, Q.; Che, W. Interactive End-to-End Decompilation via Large Language Models. *Electronics* **2025**, *14*, 4442. <https://doi.org/10.3390/electronics14224442>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Decompilation [1–3] is the reverse process of converting compiled binary code back into a high-level programming language, with the goal of recovering source code that is functionally equivalent to the original executable. Decompilation has alluring application value in performing various reverse engineering tasks, such as vulnerability identification, malware analysis, and legacy software migration [1,3–6].

Despite the development of various rule-based decompilation tools, such as Ghidra and IDA Pro, the decompilation process continues to face significant challenges. These include the inherent loss of information during compilation [7,8] and the heavy reliance on manual effort to analyze and summarize assembly code patterns for rule-based approaches. Moreover, uncovered or misinterpreted patterns can lead to inaccuracies in the decompilation results [1,4,7–9]. Recent work such as SAILR [10] demonstrates that compiler-aware structuring algorithms can improve decompilation quality by reconstructing complex control flow graphs (CFGs). However, such approaches require sophisticated CFG construction and pattern matching, which can be brittle. To address these limitations, researchers explore

the use of large language models (LLMs) in decompilation tasks [1–3,6,11,12]. By automatically learning the mapping patterns between low- and high-level code from aligned corpora, LLMs can reduce human labor and improve the readability of the generated output.

The LLM-based decompilation methods are typically divided into two categories: refine-based methods [1,2,13] and end-to-end methods [3,6,11,12]. As shown in Figure 1, the refine-based method focuses on recovering the original code (a) from the pseudo code (d) produced by existing decompilation tools, whereas the end-to-end method aims to directly reconstruct the original code (a) from assembly code (c). Compared to the refine-based approach, the end-to-end method eliminates reliance on rule-based decompilers, significantly reducing manual labor and tool maintenance costs. This work focuses on improving end-to-end methods.

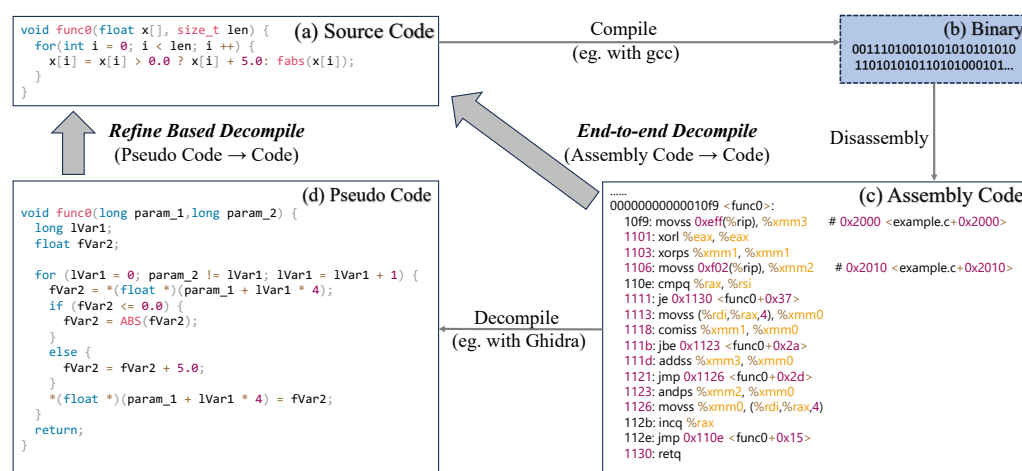


Figure 1. LLM-based decompilation methods can primarily be categorized into two types: (1) refine-based methods, which aim to refine the pseudo code generated by decompilers such as Ghidra to recover the original code; and (2) end-to-end methods, which aim to reconstruct the original code directly from assembly code.

However, despite its advantages, previous end-to-end methods encounter significant challenges in reconstructing control flow and variable values, limiting their accuracy and practical applicability. Unlike rule-based approaches that can directly access and analyze the binary file structure (including CFG construction as in SAILR [10]), end-to-end LLM methods process only textual assembly code, which inherently loses critical information during the text conversion process. These limitations arise primarily from two issues: (1) During data preprocessing, address information is directly and simplistically removed to reduce input complexity, making it difficult to recover control flow information from the processed assembly code; and (2) end-to-end methods rely solely on the processed assembly code, which lacks a significant amount of variable value information stored in data sections (e.g., floating-point numbers, strings, etc.) of the binary file. These limitations hinder the accuracy and completeness of the decompiled results.

To address the above issues, we propose *ReF Decompile* method, which is designed to optimize the decompilation process of the end-to-end method. (1) To tackle the problem of missing control flow information, the *Relabeling* strategy is introduced to restructure the data format, which preserves the control flow information through simple label-based preprocessing. Compared to rule-based decompilers with complex CFG construction (e.g., Ghidra), our method achieves superior results through simpler strategies that maintain assembler-compatible syntax while enabling LLMs to learn control flow patterns directly from labeled assembly code. This simplicity is an advantage rather than a limitation,

as it avoids the brittleness of rule-based CFG construction. (2) To solve the problem of missing variable information, the *Function Call* strategy enables the model to actively interact with the binary file to retrieve variable values stored in data sections through interactive binary data access. Importantly, Function Call operates as true inference-time interaction: the model generates requests for missing data and receives responses from the binary file during decompilation. This transforms the LLM from a passive translator into an active analysis agent that can query the binary environment for missing information. By combining these two strategies, we leverage both control flow and variable information from the binary file, improving the precision of the decompiled results.

On the Humaneval-Decompile Benchmark, the ReF Decompile method outperforms the strongest refine-based baseline by 8.69% (from 52.74% to 61.43%), demonstrating that our simple yet effective strategies can surpass methods that rely on sophisticated rule-based decompilers. It achieves a new state-of-the-art (SOTA) performance of 61.43% and the highest readability score of 3.69. These results demonstrate the effectiveness of the two strategies (Relabeling and Function Call) in improving the correctness and readability of decompiled outputs.

Our contributions are as follows:

- We systematically analyze the information loss problem in end-to-end LLM decompilation and propose effective solutions to preserve critical control flow and variable information.
- We introduce two strategies for LLM-based end-to-end decompilation: a *relabeling* approach to preserve control flow information in LLM-processable format, and a *function call* mechanism enabling interactive binary data access, which allows LLMs to query and retrieve variable information from binary files during decompilation.
- We achieve state-of-the-art performance with 61.43% re-executability rate and 3.69 readability score, demonstrating that simple yet effective strategies can surpass methods relying on complex rule-based decompilers.

2. Background & Related Work

Decompilation is the process of reversing a binary file back into its source code form. This process can be used to analyze the functionality of software when the source code is unavailable. The typical decompilation tools, such as Hex-Rays IDA Pro and Ghidra, typically rely on the analysis of the program's data flow or control flow [4]. These decompilation tools analyze the instructions in the executable section (.text section) of the assembly code and then construct the program's Control Flow Graph (CFG). They identify patterns that correspond to standard programming structures (such as if-else, while loops, etc. [8]) and perform type inference to resolve information in the read-only data section (.rodata section).

However, the construction of these tools heavily relies on rule systems created by experts, and the process of constructing these rules is highly challenging. It is also difficult to cover the entire CFG, and errors are common. Furthermore, these rules tend to fail when facing optimized binary code, yet optimization is a common practice in commercial compilers. Recent advances such as SAILR [10] propose compiler-aware structuring algorithms that improve upon traditional approaches by better handling compiler-generated patterns. SAILR demonstrates that sophisticated CFG reconstruction and pattern matching can enhance decompilation quality, particularly for structured control flow recovery. However, these methods still face challenges where compiler patterns may be obscured or non-standard. While other rule-based techniques like Ramblr [14] (for binary reassembly) and Divine [15] (for variable discovery) address specific binary analysis tasks, they do not target LLM-based decompilation. Additionally, the output of these decompilation

tools is often a source-code-like representation of the assembly code, such as directly translating variables to registers, using goto statements, and other low-level operations. This makes the output code difficult to read and understand, and it may not be sufficient to support recompilation.

Inspired by neural machine translation, researchers redefine decompilation as a translation task, which converts machine-level instructions into human-readable source code. Initial attempts use Recurrent Neural Networks (RNNs) [5] for decompilation, supplemented by error correction techniques to improve results. However, these efforts are limited in effectiveness. Recent advancements in Natural Language Processing (NLP) enable large language models (LLMs) to be applied to code-related tasks [16–18]. These models typically adopt the Transformer architectures [19], use self-attention mechanisms, and are pre-trained on large-scale text datasets. This approach allows LLMs to capture subtle contextual nuances and contributes to a general understanding of language. Currently, when introducing LLMs into the binary decompilation domain, the methods are categorized based on whether relying on existing decompilation tools: end-to-end methods and refine-based methods. We will respectively introduce them below.

Specifically, refine-based methods work with the output of decompilation tools. DeGPT [13] designs an end-to-end framework to improve the readability of decompiler output. DecGPT [2] uses LLMs combined with compiler information and runtime program information to enhance the compilability of decompiler output. Recently, LLM4Decompile [1] releases the first open-source large language model specifically for decompilation, which includes both refine-based models and end-to-end models. LLM4Decompile also introduces a readability evaluation framework using LLM-as-a-Judge [20] to assess decompiled code quality, which we adopt in our evaluation (see Appendix A). Refine-based methods reuse human-encoded rules from existing decompilers, reducing the difficulty of decompilation, but they also introduce additional dependencies.

End-to-end methods decompile directly from assembly code. BTC [12] is one of the earliest methods to fine-tune LLMs for this purpose, extending the decompilation task to multiple languages. Slade [11] expand the model size and trained an LLM-based decompiler with 200 million parameters. Nova [6] proposes a hierarchical attention mechanism and a contrastive learning approach to improve the decompilation ability of the model. LLM4Decompile [1], in addition to releasing open-source decompilation models, provides a new benchmark for decompilation tasks. However, their preprocessing directly removes address information, discarding valuable jump target information. SCC and FAE [3] further improve the performance of end-to-end decompilation based on LLM4Decompile with self-constructed context and fine-grained alignment techniques.

We observe that existing end-to-end decompilation methods face a fundamental challenge: they process only textual assembly code, losing access to the rich structural information available in binary files. Unlike rule-based methods (e.g., Ghidra, SAILR [10]) that can directly analyze binary structure and construct CFGs, end-to-end LLM methods must work with preprocessed text, which inherently discards critical information. Specifically, they lose jump target information in the executable section and fail to access data stored in separate sections (e.g., .rodata) of the binary file.

Our approach differs fundamentally from both traditional rule-based methods and previous end-to-end methods. Rather than constructing explicit CFGs like SAILR [10], we use simple label-based preprocessing that preserves control flow information in a format directly processable by LLMs. This simplicity is intentional: it avoids the complexity and brittleness of rule-based CFG construction while enabling the LLM to learn control flow patterns from data. Furthermore, we introduce an interactive mechanism that allows the LLM to query the binary file for missing data, transforming it from a passive translator

into an active analysis agent. To address these challenges, we redesign the end-to-end decompilation process in *ReF Decompile*, incorporating a relabeling strategy to preserve control flow information and a binary data access strategy to retrieve variable information.

3. Methodology

Existing end-to-end decompilation methods often lose critical information needed to reconstruct control flow structures and variables when processing binary files, making it difficult to accurately recover the program’s logic. For example, in Figure 2, the processed assembly of the previous method lost variable information (“3.14” in the source code) and the jump target (“1109” of “jbe 1109” in the raw assembly).

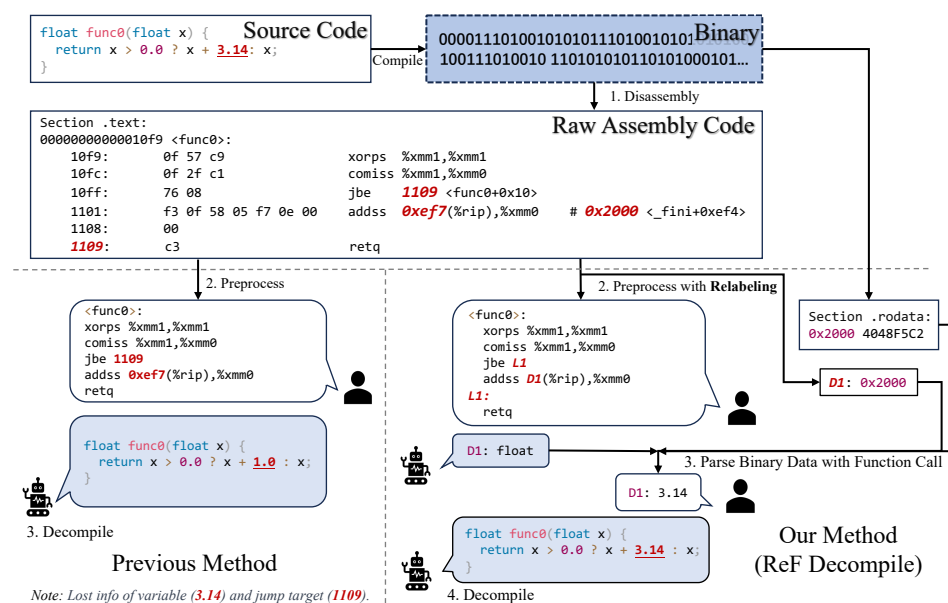


Figure 2. Comparison of the previous method and our method (ReF Decompile). Previous end-to-end methods rely solely on information from the executable segment, leading to “information loss” during decompilation. For example, the processed assembly here lost variable information (“3.14” in the source code) and the jump target (“1109” of “jbe 1109” in the raw assembly). This results in code reconstructions that appeared plausible but are actually incorrect. By incorporating Relabeling information (Relabeling) and leveraging relevant tools (Function Call), the model can now gain a deeper understanding of code jump logic and access valuable information stored outside the executable segment. This enhancement allows the model to accurately reconstruct the original code, significantly improving the precision and reliability of the decompilation process.

To solve the above challenge, in this paper, we propose the Relabeling and Function Call Enhanced Decompilation method (ReF Decompile), which updates and optimizes the end-to-end decompilation process and applies it to both the training and inference stages. The method includes using the Relabeling strategy to identify address information, and leveraging the function call strategy to complete variable information using the binary file.

In this section, we first introduce our proposed ReF Decompile and then describe how the training data are constructed.

3.1. ReF Decompile: Relabeling and Function Call Enhanced Decompilation

In this subsection, we present the overall process of the ReF Decompile method in the Algorithm 1, which consists of the following four steps during the inference-time process:

Algorithm 1: ReF Decompile

Input: Binary file B , Function name or address range F
Output: High-level source code C

```

// Step 1: Disassembly
1  $raw\_asm \leftarrow Disassemble(B, F)$ 

// Step 2: Preprocess with Relabeling
2  $(asm, label\_map) \leftarrow Relabeling(raw\_asm)$ 

// Step 3: Iterative Function Call (inference-time interaction)
3  $context \leftarrow asm;$ 
4 while LLM needs more data do
5    $request \leftarrow LLM\_GenerateRequest(context);$       /* Generate request */
6   if request is not empty then
7      $data \leftarrow RetrieveData(request, label\_map, B);$       /* Retrieve data */
8      $context \leftarrow Concatenate(context, data);$ 
9   else
10    break;

// Step 4: Final decompilation
11  $C \leftarrow LLM\_Decompile(context);$ 
12 return  $C;$ 

```

1. Disassembly Binary File (Line 1): By employing disassembly tools such as “Objdump” or “Capstone”, and by specifying concrete function names or address ranges, we can translate the machine code of functions into raw assembly code that is more interpretable. Raw assembly code includes the complete source code information that will be used in the following steps.
2. Preprocess Assembly with Relabeling (Line 2): In this phase, we preprocess the raw assembly code to simplify its structure for data construction. The processed result also satisfies the syntax accepted by the assembler. Initially, all jump instruction target addresses are replaced with labels (e.g., address 0x1109 is replaced with “L1”), and these labels are inserted at the corresponding target locations while removing all address information preceding non-target addresses. Secondly, addresses in instructions related to data access are also labeled (e.g., *0xef7(%rip)* is replaced with *D1(%rip)*), and a mapping between these labels and their actual addresses is recorded. The preprocessed assembly code is then utilized as input for the model.
3. Process Function Call Request (Lines 3–10): Our system incorporates a mechanism that allows the model to generate structured requests for function calls or accessing data. Upon feeding the preprocessed assembly code into the model, if it contains memory access instructions, the model typically issues a function call request that includes labels from the memory access instructions along with their associated data types. Leveraging the label-to-address mapping established in the previous step, we can identify the actual addresses represented by these labels and parse the corresponding sections of the binary file (commonly the .rodata section) according to the data types predicted by the model. The labels and parsing results are then fed back into the model as input.
4. Decompile Finally (Line 11): After completing the aforementioned steps, the model accumulates both the assembly code corresponding to the function and the data values at the addresses referenced by memory access instructions. Based on this information, the model produces a final high-level language representation, thereby

achieving the transformation from low-level machine code to high-level programming language constructs.

3.2. Data Construction

In this section, we introduce the data construction process of *Relabeling* and *Function Call* in our proposed method. The constructed data trains the model to understand the format of relabeled data and to interact through function calls.

3.2.1. Relabeling

The relabeling process aims to remove specific address information (including jump and memory access addresses) from the assembly code while preserving the program's jump logic to ensure control flow integrity. As the Figure 3 and Algorithm 2 shows, the specific steps are as follows:

Algorithm 2: Relabeling

Input: Raw assembly code *raw_asm*
Output: Preprocessed assembly *asm*, Label-to-address mapping *label_map*

```

1 label_map  $\leftarrow \emptyset$ , jump_labels  $\leftarrow \emptyset$ , data_labels  $\leftarrow \emptyset$ ;
   // Step 1: Collect addresses and assign labels
2 foreach instruction inst in raw_asm do
3   if inst is jump instruction (e.g., jbe, jmp) then
4     target_addr  $\leftarrow$  ExtractJumpTarget(inst);
5     if target_addr  $\notin$  jump_labels then
6       label  $\leftarrow$  GenerateLabel("L", |jump_labels| + 1);
7       jump_labels[target_addr]  $\leftarrow$  label;
8       label_map[label]  $\leftarrow$  target_addr;
9   else
10    if inst is memory access instruction then
11      mem_addr  $\leftarrow$  CalculateAddress(inst); /* e.g., 0xef7(%rip) */
12      if mem_addr  $\notin$  data_labels then
13        label  $\leftarrow$  GenerateLabel("D", |data_labels| + 1);
14        data_labels[mem_addr]  $\leftarrow$  label;
15        label_map[label]  $\leftarrow$  mem_addr;

   // Step 2: Replace addresses with labels
16 asm  $\leftarrow$  raw_asm;
17 foreach (addr, label) in jump_labels  $\cup$  data_labels do
18   asm  $\leftarrow$  ReplaceAddressWithLabel(asm, addr, label);

   // Step 3: Insert labels before jump targets
19 foreach (target_addr, label) in jump_labels do
20   asm  $\leftarrow$  InsertLabelBeforeAddress(asm, target_addr, label);
21 return (asm, label_map);

```

1. Collection of Address Information and Label Assignment (Lines 2–15): First, we perform a disassembly analysis of the program. (a) We identify all target addresses associated with jump instructions (e.g., “jbe”, Line 3), as these addresses control the program's flow of execution and are crucial for the relabeling process. These addresses are recorded and labeled (e.g., L1, L2, etc.). (b) We identify all instructions related to

- memory access (e.g., “addss 0xef7(%rip),%xmm0”, Line 10), record these addresses, and assign corresponding labels (e.g., D1, D2, etc.).
2. Replacement of Specific Addresses with Labels (Lines 16–18): All addresses recorded in the previous step, including both jump and memory access addresses, are replaced with the assigned labels.
3. Insertion of Labels Before Jump Targets (Lines 19–20): To eliminate specific memory address information while maintaining the jump logic, we insert the corresponding label before each jump target instruction.

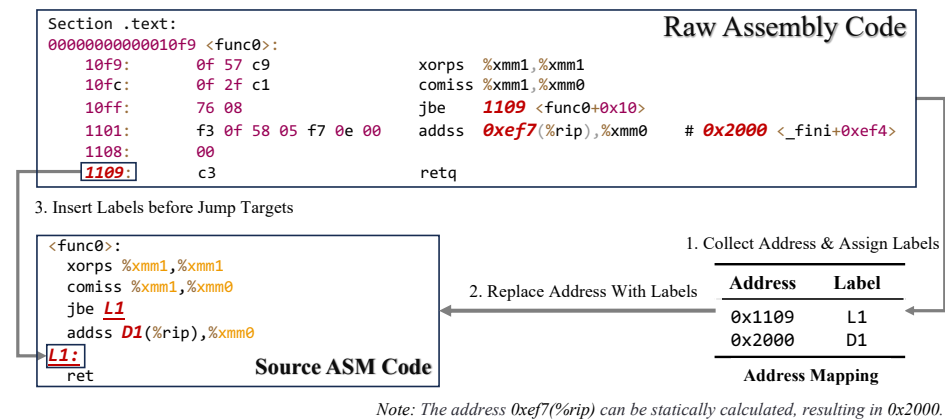


Figure 3. The processing details of Relabeling.

Through this process, we remove the address information while preserving the integrity of the program’s jump logic and ensuring that critical jump information is not lost.

3.2.2. Binary Data Access via Function Call

We introduce a novel mechanism that enables the LLM to actively query data from the binary file, transforming it from a passive translator into an active analysis agent. We term this mechanism “Binary Data Access via Function Call” because it follows a request-response paradigm similar to modern tool-calling capabilities in LLMs: the model generates structured requests for data (analogous to function calls), and the system responds with the requested values retrieved from the binary file. This approach addresses a fundamental limitation of end-to-end methods: while traditional rule-based decompilers can directly access all sections of a binary file (including .rodata, .data, etc.), end-to-end LLM methods typically process only textual assembly code, losing access to variable values stored outside the executable section.

To enable the model to perform accurate function calls during the inference process described in Section 3.1, we construct training data following Algorithm 3. The described data construction process in Algorithm 3 enables the model to directly learn the mapping between source code literals and their binary representations, rather than relying on manual binary analysis for type inference. This involves parsing source code and analyzing binary files to accurately extract and match literals with their storage addresses within the binaries. As Figure 4 shows, the detailed process is as follows:

Algorithm 3: Function Call

Input: Preprocessed assembly *asm*, Label mapping *label_map*, Binary file *B*
Output: Data mapping *data_map* (label $\rightarrow$ value)

```

1 data_map  $\leftarrow \emptyset$ , literals  $\leftarrow \emptyset$ ;

  // Step 1: Collect literals from source code
2 source_code  $\leftarrow$  GetSourceCode();
3 foreach literal lit in ParseLiterals(source_code) do
4   | bytecode  $\leftarrow$  ConvertToBytecode(lit);
5   | literals[bytecode]  $\leftarrow$  (lit.value, lit.type);

  // Step 2: Collect memory access addresses from assembly
6 mem_labels  $\leftarrow \emptyset$ ;
7 foreach instruction inst in asm do
8   | if inst contains data label (e.g., D1, D2) then
9   |   | label  $\leftarrow$  ExtractLabel(inst);
10  |   | mem_labels  $\leftarrow$  mem_labels  $\cup$  {label};

  // Step 3 & 4: Construct data-address mapping and function calls
11 foreach label in mem_labels do
12   | addr  $\leftarrow$  label_map[label];
13   | bytecode  $\leftarrow$  ReadBytesFromBinary(B, addr);
14   | if bytecode  $\in$  literals then
15   |   | (value, type)  $\leftarrow$  literals[bytecode];           /* Match to infer type */
16   | else
17   |   | type  $\leftarrow$  InferTypeFromContext(asm, label);
18   |   | value  $\leftarrow$  ParseBytecode(bytecode, type);
19   | data_map[label]  $\leftarrow$  (value, type);

20 return data_map;

```

1. Collect Literals from Source Code (Lines 2–5): Initially, we employ the clang compiler toolchain to parse C language code, extracting literal data such as strings, floats, and doubles. This step identifies all constants within the source code, clarifying how these constants manifest post-compilation. To enable subsequent bytecode matching, we convert the extracted literals into corresponding bytecode based on their types (Line 4). This conversion ensures that the storage format of literals matches the form present in the binary files, allowing us to later compare the bytecode read from memory access locations with these literal bytecodes to infer data types and values.
2. Collecting Memory Access Addresses (Lines 6–10): Next, we disassemble the compiled binary files to identify memory addresses related to memory access instructions, assigning labels (e.g., D1, D2) to these addresses. This step mirrors the relabeling process, specifically focusing on address mapping pertinent to memory accesses.
3. Constructing Data Addresses Mapping (Lines 11–19): Finally, we compare the bytecode representation of literals with the memory load addresses obtained from the disassembly analysis. For each memory access label, we first read the bytecode from the binary file at the mapped address (Line 13). We then match this bytecode against the literals collected in Step 1 (Line 14): if a match is found, we can infer both the data type and value from the matched literal; otherwise, we infer the type from the assembly context (Line 17) and parse the value accordingly. Through this bytecode comparison approach, we determine the specific storage addresses and types of liter-

als within the binary files, thereby constructing a mapping relationship among labels, addresses, types, and data.

4. Constructing Function Call Training Data: Based on the established mapping relationships, we construct training examples that teach the model to generate structured requests for binary data access. Each training example includes: (1) the assembly code with data labels (e.g., D1, D2), (2) the model's generated request specifying which labels and data types are needed, and (3) the system's response containing the actual values retrieved from the binary file. This request-response format mirrors modern tool-calling paradigms, where the LLM learns to interact with external systems to gather missing information.

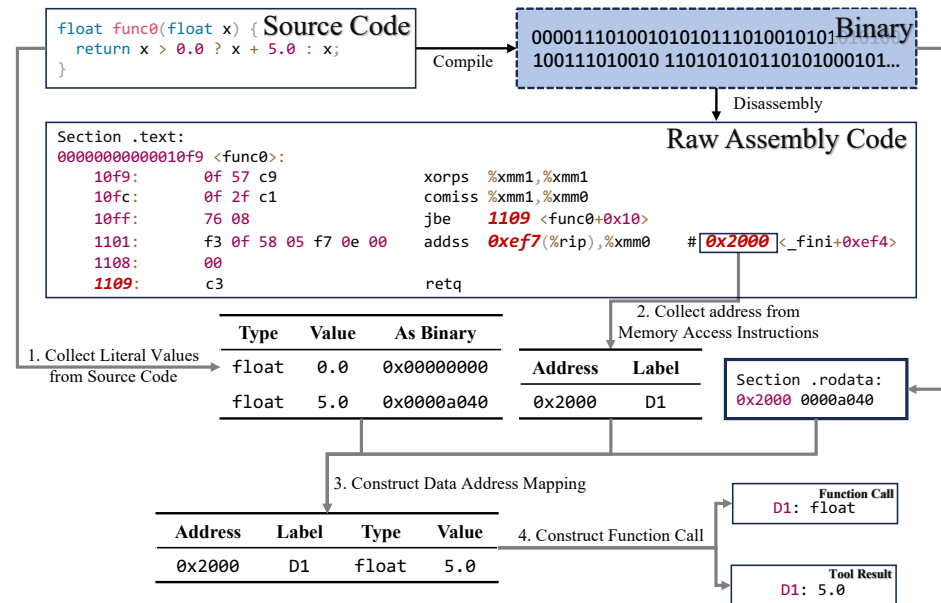


Figure 4. Overview of the Data Construction for Function Call. The mechanism enables the LLM to actively query data values stored in the binary file through a structured request-response paradigm, similar to modern tool-calling capabilities in LLMs.

This systematic approach provides high-quality training data that teaches the LLM to actively interact with binary files during decompilation. Unlike previous end-to-end methods that passively translate assembly text, our approach enables the model to recognize when critical information is missing and proactively request it from the binary file. By learning to generate appropriate data access requests and integrate the retrieved values into the decompilation process, the model gains a deeper understanding of program semantics, leading to more accurate and complete code reconstruction.

4. Experiments

In this section, we introduce the experimental details and analysis of our results. Following Tan et al. [1] and Feng et al. [3], we utilize a subset of Exebench [21] as the training set and Decompile-Eval [1] as the test set. We compare our method with six baseline methods and experimental results demonstrate that our method achieves state-of-the-art performance among models of the same size.

4.1. Training Details

4.1.1. Training Data

Following Tan et al. [1] and Feng et al. [3], we utilize a subset of Exebench [21] as the training set. ExeBench is the largest public collection of five million C functions, and we select 150k samples from the train_real_compilable subset to synthesis the training data

(about 4b tokens). The selected functions exclusively utilize the standard C library and do not include additional data structures. The training data were synthesized with gcc 11.4 provided by Ubuntu 22.04.

4.1.2. Implementation

Following Feng et al. [3], we employ LoRA [22] to fine-tune the llm4decompile-6.7b-end v1.5 model obtained from Hugging Face [1,23]. The rank is set to 32, alpha to 64, and the target includes embedding layer, lm head, and all projection layers (embed_tokens, lm_head, q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj). The model is trained for one epoch using the AdamW optimizer [24] with a learning rate of 5×10^{-5} . The maximum sequence length is set to 4096, and the learning rate scheduler type is cosine, with a warm-up period of 20 steps. The fine-tuning process leverages LlamaFactory [25], FlashAttention 2 [26], and DeepSpeed [27]. All experiments are conducted on an A100-SXM4-80GB GPU. Following Tan et al. [1] and Feng et al. [3], the greedy decoding is utilized during the experiments.

4.2. Evaluation Details

4.2.1. Benchmark

Following Tan et al. [1] and Feng et al. [3], we employ Decompile-Eval [1] as our evaluation benchmark, which is specifically designed to assess the decompilation capabilities of large language models. The Decompile-Eval benchmark [1] is adapted from the HumanEval benchmark [28], which includes 164 problems initially designed for code generation tasks. These problems are translated into the C programming language, and the corresponding assembly code is generated at four optimization levels (O0, O1, O2, and O3). The correctness of the decompilation results is tested using the test cases from HumanEval.

4.2.2. Metrics

The primary metrics of the Decompile-Eval benchmark are as follows:

- **Re-executability Rate:** This metric assesses the functional correctness of the decompiled code. Specifically, it measures whether the recompiled binaries produce the expected outputs when executed. The correctness of the output is determined using the testing methodology provided by the HumanEval dataset, ensuring a comprehensive evaluation of the logical accuracy of the decompiled code. We adopt re-executability as our primary metric because it directly measures functional correctness and has been widely used in recent LLM-based decompilation work [1,3]. As our experiments show, models typically achieve high re-compilability (>95%), making re-executability more discriminative for evaluating decompilation quality.
- **Readability:** Following Tan et al. [1], we evaluate the readability of decompiled code—defined as how easily reverse engineers can understand and work with the code based on three practical dimensions: variable naming quality, control flow clarity, and structural organization. We use GPT-4o with a structured template (see Appendix A for the complete template) to assess:
 - *Syntax Similarity:* Variable naming quality (meaningful names vs. generic like var1), preservation of control flow structures (loops, conditionals), and maintenance of array indexing and structure operations.
 - *Structural Integrity:* Ease of understanding at a glance, logical structure preservation, and absence of redundant code snippets.

Based on these detailed aspects, GPT-4o assigns an overall score from 1 (Poor) to 5 (Excellent). A score of 4 or higher indicates that the decompiled code is nearly identical to the original in terms of readability. While this evaluation uses an LLM-based

approach, the assessment dimensions effectively capture key readability indicators that have been emphasized in prior decompilation work [10], including variable naming quality, control structure clarity (which relates to goto statement usage), and overall code organization.

4.3. Baselines

To demonstrate the effectiveness of our proposed methods, we compare them with several baselines, including Rule-Based Decompilers, Refine-Based Methods, and End-to-End Methods. In this section, we introduce these methods (For the baselines listed below, all models are assumed to be of size 6.7B unless otherwise specified, except for Ghidra and GPT-4o. We also report the performance of models with other sizes in Figure 5).

- **Rule-Based Decompiler** relies on manually crafted rules and techniques such as control flow and data flow analysis to transform assembly code into high-level language code.
 - Ghidra: A free and open-source reverse engineering tool (decompiler). It serves not only as the baseline for comparison but also as the preprocessing tool for the Refine-Based decompilation method.
- **Refine-Based Methods** builds upon the output of rule-based decompilers, leveraging large models to refine and enhance the decompilation results for improved accuracy and readability.
 - GPT 4o: One of the most powerful language models developed by OpenAI, which is used to refine the Ghidra decompilation output.
 - LLM4Decompile-Ref: A series of pre-trained refine-based models [1], which refine pseudo-code decompiled by Ghidra.
- **End-to-End Methods** directly process assembly code using large models to generate high-level language code.
 - LLM4Decompile-End: A series of pre-trained end-to-end models [1], which directly decompile binaries into high-level code.
 - FAE Decompile: A model obtained by applying the Fine-grained Alignment Enhancement method to further fine-tune the llm4decompile-End-6.7b [3].

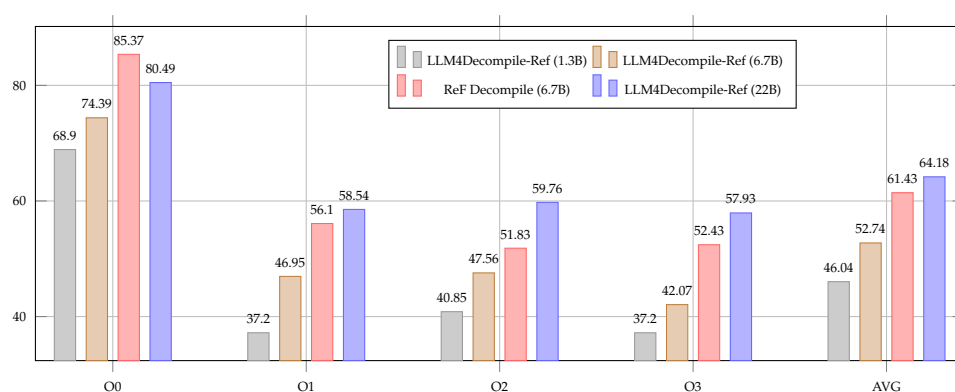


Figure 5. Re-executability rate comparison between ReF Decompile (6.7B) and LLM4Decompile-Ref models of varying sizes (1.3B, 6.7B, and 22B parameters). Despite having only one-third the parameters of the 22B baseline, ReF Decompile achieves competitive performance (average gap of only 2.75%), demonstrating the effectiveness of our Relabeling and Function Call strategies. At O0 level, our 6.7B model even surpasses the 22B baseline (85.37% vs. 80.49%).

4.4. Main Results

In this paper, we evaluate our ReF Decompile and the backbones in Section 4.3 on Decompile-Eval. The main results of our experiments are shown in Table 1 and Figure 5.

Table 1. Main comparison of different approaches and models for re-executability rate and readability across different optimization levels (O0, O1, O2, and O3) on HumanEval-Decompile benchmark. **Bold** denotes the best performance. Underline denotes the second-best performance.

Model/Metrics	Re-Executability Rate (%)					Readability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Rule Based Decompiler										
Ghidra	34.76	16.46	15.24	14.02	20.12	2.98	2.41	2.52	2.38	2.57
Refine-Based Method										
GPT-4o	46.95	34.15	28.66	31.10	35.22	2.82	2.35	2.29	2.31	2.44
LLM4Decompile-Ref	<u>74.39</u>	46.95	<u>47.56</u>	<u>42.07</u>	<u>52.74</u>	<u>4.08</u>	3.38	3.34	3.19	3.50
End-to-End Method										
LLM4Decompile-End	69.51	44.51	39.63	38.41	48.02	4.07	<u>3.46</u>	<u>3.40</u>	3.23	<u>3.54</u>
FAE Decompile	67.68	<u>48.78</u>	45.73	42.07	51.07	3.94	<u>3.46</u>	<u>3.40</u>	<u>3.25</u>	3.51
ReF Decompile	85.37	56.10	51.83	52.43	61.43	4.13	3.60	3.54	3.49	3.69

- Our method is the best and surpasses all other approaches to become the state of the art. As shown in Table 1, the decompiler-based method has the lowest performance, with a re-compilability of 20.12, because manually crafted rule systems cannot guarantee that the generated code is fully compilable. In the refine-based methods, LLM4decompile, which is trained on 20B tokens of decompilation data, outperforms the untrained GPT-4o, achieving a re-compilability of 52.74%. As analyzed in the introduction, it corrects the decompilation results from the decompiler, surpassing the two other end-to-end methods, and becomes the strongest baseline.
- As an end-to-end approach, ReF Decompile surpasses refine-based baselines. It improves the Re-executability metric by 8.69% and the readability metric by 0.19. This demonstrates the effectiveness of our two strategies, Relabing and Function Call, which reverse the trend where end-to-end methods typically perform worse than refine-based methods. Notably, our simple strategies outperform refine-based methods that rely on complex CFG construction and rule-based decompilers, while significantly reducing manual effort for rule maintenance and tool integration.
- The Relabing and Function Call strategies better leverage the potential of end-to-end methods. As shown in Figure 5, the performance of 6.7B ReF Decompile not only significantly surpasses both 1.3B and 6.7B Refine-Based LLM4Decompile-Ref, but it is also comparable to the 22B Refine-Based LLM4Decompile-Ref, with an average gap of only 2.75%. Despite having only one-third the parameters of the 22B model, our method achieves competitive performance, demonstrating the effectiveness of our targeted strategies. Notably, at optimization level O0, the performance of ReF Decompile (6.7B) even exceeds that of the 22B Refine-Based LLM4Decompile-Ref model, indicating that the model can automatically learn patterns beyond those defined by humans from large-scale corpora.
- ReF Decompile surpasses other end-to-end baselines in readability, becoming the new SOTA. Besides a significant improvement in Re-executability (10.36%), it achieves a readability score of 3.69. This shows that our two strategies effectively avoid the loss of crucial information needed to reconstruct control flow structures and variables, leading to a more accurate recovery of the program's logic. To further validate the generalizability of our approach, we conducted an extended evaluation with 7 additional powerful LLMs (see Appendix B).

4.5. Implementation Efficiency

To address concerns about scalability and practical applicability, we provide detailed statistics on the efficiency of our method. All inference experiments were conducted on a single NVIDIA RTX A6000 GPU using vLLM for optimization.

Interaction Statistics: On the complete test set of 656 functions, our Function Call mechanism demonstrates high efficiency:

- Average interaction rounds: 1.28 rounds per function.
- Minimum interaction rounds: 1 round.
- Maximum interaction rounds: 2 rounds.

These statistics indicate that most functions require only 1–2 interactions, demonstrating that our Function Call mechanism introduces minimal overhead. The low average of 1.28 rounds shows that the model efficiently determines when additional information is needed from the binary file.

Time Performance: The decompilation process is reasonably efficient:

- Average decompilation time: 13.80 s per function.
- Minimum decompilation time: 1.54 s.
- Maximum decompilation time: 78.81 s.
- Total time for 656 functions: 300.39 s (≈ 5 min).

With vLLM's parallel inference capability (batch size 128), our method processed all 656 functions in approximately 5 min on a single GPU, achieving a throughput of approximately 8000 functions per hour. This demonstrates strong scalability for large-scale binary analysis: for instance, decompiling an application with 10,000 functions would require approximately 1.25 h on a single GPU, or proportionally less time with multi-GPU deployment. The average per-function time of 13.80 s reflects the sequential processing baseline. Moreover, with ongoing advances in model architectures (e.g., recent Linear/MoE architectures) and inference optimization techniques, models with similar capabilities are becoming smaller and faster, which will further improve the scalability of our approach.

4.6. Ablation Study

As shown in Table 2, we conduct ablation experiments to analyze the impact of two different strategies (Relabeling and Function Call) on model performance.

When initialized with the LLM4Decompile-End-6.7B model, both Relabeling and Function Call contribute to model performance and readability, and their combination performs best in both metrics. Specifically, the introduction of Relabeling improves performance by 3%, while the introduction of Function Call improves performance by 7%. The model with both Relabeling and Function Call achieves the highest average re-executability at 61.43%. Moreover, these two components also improve the readability of the decompiled results, increasing it by about 0.14 points, which corresponds to a 4% improvement in readability.

In addition to using LLM4Decompile-End-6.7B, in order to minimize the impact of continued pretraining, we also use its initialized model Deepseek-Coder-6.7B-base. Surprisingly, the model with Relabeling and Function Call, trained with only 4B tokens, outperforms the strongest baseline of the same size in Table 1 (52.74). The ablation results further confirm that Relabeling and Function Call both lead to significant performance improvements, with the combined effect being even more pronounced. Specifically, the introduction of Relabeling leads to a 3% performance improvement, Function Call leads to a 5% improvement, and the simultaneous introduction of both components leads to a 10% improvement, surpassing the combined performance improvements of each component individually. Similar to the model initialized with LLM4Decompile-End-6.7B, both components also improve the readability of the decompiled results by about 0.14.

Table 2. Ablation study for tuned models. The ablation study of different components across four optimization levels (O0, O1, O2, O3), as well as their average scores (AVG). The results in bold represent the optimal performance. The  and  means Relabeling and Function Call. **Bold** denotes the best performance.

Components		Re-Executability Rate (%)					Readability				
		O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Initialize with LLM4Decompile-End-6.7B [1]											
		69.51	46.95	50.61	46.34	53.35	3.98	3.41	3.44	3.38	3.55
		75.61	50.61	50.00	50.00	56.55	4.01	3.44	3.39	3.49	3.58
		83.54	56.10	51.22	50.61	60.37	4.05	3.51	3.51	3.42	3.62
		85.37	56.10	51.83	52.43	61.43	4.13	3.60	3.54	3.49	3.69
Initialize with Deepseek-Coder-6.7B-base [18]											
		59.15	35.98	39.02	37.80	42.99	3.71	3.05	3.16	3.05	3.24
		66.46	41.46	38.41	36.59	45.73	3.76	3.17	3.21	3.08	3.31
		70.73	39.63	39.02	40.24	47.41	3.90	3.17	3.08	3.11	3.31
		79.88	45.73	43.90	42.68	53.05	3.96	3.21	3.18	3.19	3.38

4.7. Analysis of Two Components for Untuned Models

This section analyzes the impact of two key components—Relabeling and Tool Integration—on the performance of untuned models. We examine how these strategies enhance decompilation accuracy.

4.7.1. Relabeling Improves the Performance of Untuned Models Significantly

As shown in Table 3, Relabeling enhances the readability of jump addresses in assembly code by assigning more intuitive labels, leading to better decompilation results. For example, performance on the GPT-4o [29] model improves from 21.34% to 27.74% (a gain of 6.40%), while on the Qwen model [30], it improves from 11.28% to 14.63% (a gain of 3.37%). This supports our hypothesis that Relabeling makes assembly code's jump logic easier for models to understand, resulting in better inference. GPT-4o benefits more from this change, likely due to its stronger ability to handle complex logic. Relabeling demonstrates how simple preprocessing improvements can significantly boost performance without requiring additional fine-tuning. Enhancing input readability and logic clarity proves to be a valuable strategy for improving model effectiveness in specific tasks.

Table 3. Ablation study for untuned models. The ablation study of different components across four optimization levels (O0, O1, O2, O3), as well as their average scores (AVG). This study aims to investigate the performance impact of various components on the model without any fine-tuning. The  and  means Relabeling and Function Call. **Bold** denotes the best performance.

Components		GPT-4o					Qwen2.5-Coder-32B-Instruct				
		O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
		30.49	17.68	18.90	18.29	21.34	12.20	12.20	11.59	9.15	11.28
		37.80	26.22	24.39	22.56	27.74	15.24	16.46	14.02	12.80	14.63
		28.66	15.85	14.63	17.07	19.05	23.17	14.02	15.24	10.37	15.70
		31.71	25.61	22.56	25.00	26.22	23.78	16.46	17.07	10.37	16.92
Constructed Function Call Messages											
		35.98	21.95	16.46	18.29	23.17	25.00	15.85	16.46	10.98	17.07
		43.29	29.27	26.83	28.66	32.01	24.39	17.07	16.46	10.37	17.07

4.7.2. Potential of Function Call to Enhance Untuned Model Performance Remains Underexplored

As shown in Table 3, introducing tools does not always lead to performance improvements. In our experiments, although we provide the model with tools for decompilation in the prompt, it consistently returns the decompiled results directly without invoking any tools. For GPT-4o, Function Call slightly reduces performance. This decrease might be due to the model's inability to determine when to invoke tools to retrieve data from unknown addresses. Additionally, prompts related to tool usage may cause interference. In contrast, in the Qwen experiments, Function Call slightly improves performance. Although the model does not actively do function call, the descriptive prompts associated with tools may stimulate some decompilation capabilities.

To further verify the model's ability to utilize function call information, we simulate function call and responses to test whether the model can use tool outputs. The results show that models, despite not calling tools themselves, are able to leverage the provided information to improve performance. For example, tools supply variable information missing from the assembly code, allowing models to avoid "guessing" variable values. This leads to a performance boost of 2% to 5% for GPT-4o.

These findings highlight two key points. First, current models struggle to invoke and utilize tools effectively, limiting their immediate benefits. Second, tools still show significant potential for performance enhancement, as demonstrated by the gains from simulated tool use.

4.8. Dataset Analysis

In this section, to validate the necessity of Relabeling and Function Call, we analyze the distribution of information related to these two strategies within the dataset.

4.8.1. Rodata Is Widely Present in Various Environments

To demonstrate the necessity of introducing function calls, we analyze the Rodata information in Exebench and Humaneval-Decompile in Table 4. In Humaneval-Decompile, 27.90% of the code includes data within the rodata segment. Similarly, in Exebench's train_real_compilable subset, 48% of the code stores certain information in the rodata section rather than the executable code section after compilation. It is commonly believed that Exebench approximates the distribution of real-world code scenarios, while Humaneval is relatively simple. Thus, we can observe that even in relatively simple scenarios like Humaneval, nearly 1/4 of the code includes some rodata segment data. This implies that in real-world scenarios, the introduction of function call enables models to read the contents of the rodata segment, potentially leading to greater benefits.

Table 4. The table shows the proportion of samples with different labels under different optimization levels.

Proportion Rate (%)	Decompile-Eval				Exebench			
	O0	O1	O2	O3	O0	O1	O2	O3
With Data Labels	25.61	26.21	25.00	34.76	47.61	47.62	47.75	49.01
With Jump Labels	96.34	95.73	95.73	95.73	76.66	72.19	71.57	71.20
With Both	25.00	25.61	24.39	34.15	37.67	36.74	36.81	37.88
With Any	96.95	96.34	96.34	96.34	86.61	83.06	82.51	82.33

4.8.2. Relabeling and Function Call Are Widely Applicable

As shown in Table 4, more than 80% of the code includes fixed addresses, such as data memory access addresses and jump targets, demonstrating the broad applicability of the Relabeling strategy. Decompile-Eval contains more branch and jump instructions (Jump Instructions), whereas Exebench exhibits more frequent access to rodata (Data Labels and Load Instructions).

As shown in Table 5, at the O2 optimization level, both datasets exhibit the lowest numbers of memory access and jump instructions, which could explain the relatively smaller performance gains from Relabeling and Function Call at this level. Nonetheless, our methods demonstrate significant improvements across other optimization levels, underscoring their robustness and adaptability in diverse optimization settings.

Table 5. This table lists the average number of different labels (jump labels, data labels) and instructions (jump instructions, load instructions, total instructions) contained in each sample across various compilation optimization levels (O0 to O3) within the Exebench and Decompile-Eval datasets.

Mean Num	Decompile-Eval				Exebench			
	O0	O1	O2	O3	O0	O1	O2	O3
Data Labels	0.80	0.85	0.78	1.10	2.46	2.44	2.43	2.49
Jump Labels	6.59	6.07	5.71	6.80	4.13	3.62	3.49	3.81
Load Instructions	1.00	1.06	0.90	1.27	3.51	3.04	2.85	2.94
Jump Instructions	7.51	7.65	7.04	9.14	5.21	4.46	4.33	5.01
Total Instructions	78.13	50.57	52.30	70.38	69.46	48.05	49.71	55.99

5. Discussion & Future Work

5.1. Discussion

The core contribution of this work is the introduction and validation of a new paradigm for decompilation centered on direct interaction with binary files. By introducing two synergistic strategies, we have demonstrated that evolving Large Language Models (LLMs) from mere “text translators” into “analysis agents” capable of actively probing a binary environment is both feasible and highly effective. The state-of-the-art (SOTA) performance achieved on the Humaneval-Decompile benchmark is more than a numerical improvement; it serves as strong evidence that our interactive approach can reconstruct program control flow and variable semantics more accurately than traditional end-to-end methods that rely solely on assembly text.

This work lays a crucial foundation for the field of LLM-based program analysis. It illuminates a path beyond surface-level textual analysis, tackling the challenge of information asymmetry by empowering the LLM with the ability to use external tools (in this case, to query the binary).

5.2. Limitations

Despite its success, our method has several limitations that present opportunities for future work. To provide concrete insights into these limitations, we have conducted a detailed error analysis (see Appendix C) of failed cases, revealing common failure patterns including complex optimization artifacts, vectorized/SIMD operations, and conditional logic ambiguity:

- **Scalability:** The current experiments were primarily conducted on function-level benchmarks. As detailed in Section 4.5, our method demonstrates strong scalability through parallel inference (approximately 8000 functions per hour on a single GPU), indicating that large-scale binary analysis is practically feasible. However, systematic evaluation on complete large-scale applications (e.g., operating system libraries or enterprise software) would provide further insights into end-to-end performance characteristics.
- **Architectural Generality:** Our model was primarily trained and tested on a specific CPU architecture (e.g., x86-64). Generalizing it to other architectures, such as ARM or MIPS, would require additional adaptation and training data.
- **Resilience to Obfuscation:** Many real-world binaries, particularly malware, employ anti-reversing techniques like code obfuscation and packing. The robustness of our method against these intentionally adversarial binaries is an open question.
- **Complex Data Structures:** While the Function Calling strategy is effective for recovering primitive types (e.g., strings, floats), handling complex, nested data structures, classes, or objects may require more sophisticated interaction protocols and reasoning capabilities.

5.3. Future Work

Based on the discussion above, we outline several directions for future research:

- **Handling Code Obfuscation:** Improve robustness against code obfuscation. This involves training the LLM directly on obfuscated binaries—such as those using packing, virtualization, or encryption—to enable the model to learn and reverse common obfuscation patterns.
- **Whole-Binary Analysis:** The scope of analysis could be expanded from single functions to entire binaries. This would require addressing more complex challenges, such as inter-procedural data-flow analysis, global variable recovery, and the reconstruction of inter-modular dependencies.
- **Integration with Dynamic Analysis (Debugging):** Combine the current static interaction framework with dynamic analysis. Granting the LLM control over a debugger would enable it to resolve runtime-dependent information, such as indirect jumps and calculated variable values, thereby fundamentally improving analysis accuracy for complex programs.

6. Conclusions

Decompilation is the reverse process of converting compiled binary code back into a high-level programming language. In this paper, we revisit previous end-to-end decompilation approaches and identify a critical issue: they often lose crucial information required for reconstructing control flow structures and restoring variables when processing binary files. This limitation makes it challenging for end-to-end methods to accurately recover program logic, resulting in inferior performance compared to refine-based methods. To address this issue, we propose *ReF Decompile*, which involves a redesigned end-to-end decompilation workflow. Specifically, to tackle the loss of control flow information, we introduce a relabeling strategy to reformat data by replacing jump target addresses with labels and placing the corresponding labels before the jump targets for clear identification. To mitigate the loss of variable information, we introduce an interactive function call strategy that enables the model to infer variable types and actively query the binary file to retrieve variable values, completing the information required for variable reconstruction. Experimental results on the Humaneval-Decompile Benchmark demonstrate that ReF Decompile, as an end-to-end approach, outperforms refine-based baselines with the same model size. It

achieves a SOTA performance of 61.43% in deep learning decompilation. In addition, we find that our method not only enhances the performance of the decompilation task but also improves the readability of the decompiled results compared to baselines. We further analyze the effectiveness of the Relabeling and Function Call through ablation studies and dataset analysis.

Author Contributions: Conceptualization, methodology, and writing—original draft preparation, Y.F.; writing—review and editing, Y.F., B.L., X.S., Q.Z. and W.C.; supervision and funding acquisition, Q.Z. and W.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (NSFC), grant numbers 62236004, 62206078, 62441603 and 62476073.

Data Availability Statement: The data, code, and models presented in this study are openly available at <https://github.com/AlongWY/ReF-Dec> (accessed on 29 May 2025).

Acknowledgments: We gratefully acknowledge the support of the National Natural Science Foundation of China (NSFC) and the computational resources provided for this research.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Language Model
SOTA	State of the Art
GCC	GNU Compiler Collection
CFG	Control Flow Graph
RNN	Recurrent Neural Network
NLP	Natural Language Processing
NSFC	National Natural Science Foundation of China

Appendix A. Readability Evaluation Template

Following LLM4Decompile [1], we use GPT-4o to evaluate decompiled code readability. This LLM-as-a-Judge approach [20] assesses variable naming quality, control flow clarity, and structural organization.

This appendix presents the structured template used with GPT-4o to evaluate the readability of decompiled code. The template assesses both syntactic similarity and structural integrity, providing a comprehensive evaluation framework for measuring code quality.

GPT-4o Readability Evaluation Template

1. Original Function

[SRC]

2. Decompiled Function

[DSRC]

3. Evaluation Aspects

Each aspect of the decompiled function should be rated on a scale from 1 (Poor) to 5 (Excellent). Provide brief justifications for each rating.

3.1 Syntax Similarity

- Are the variable names easy to understanding the decompiled function?
- Does the decompiled control flow (loops, conditionals) closely match the original function?
- Does the decompiled function keep the array indexing and structure operation?

Rating: _____

Short Justification: _____

3.2 Structural Integrity

- Is the decompiled function easy to understand at a glance?
- Does the decompiled function maintain the logical structure of the original?
- Are there any redundant or unnecessary code snippets added during decompilation?

Rating: _____

Short Justification: _____

4. Overall Readability Score

Description: Based on the detailed aspects evaluated above, provide an overall numerical score from 1 (Poor) to 5 (Excellent) for the readability of the decompiled function. Do not explain.

Score: _____

This template ensures a systematic and reproducible evaluation process, enabling consistent assessment of decompiled code quality across different optimization levels and decompilation methods.

Appendix B. Extended LLM Baseline Comparison

This appendix presents an extended comparison of our ReF Decompile method against a broader range of powerful large language models. This comprehensive evaluation demonstrates the generalizability and robustness of our approach across different model architectures and sizes.

We evaluated 7 additional large language models on the HumanEval-Decompile benchmark, including both open-source and proprietary models with varying parameter sizes. The models tested include:

- Qwen3-Plus: An enhanced version of the Qwen model series

- Qwen3-Coder-Plus: Code-specialized version of Qwen3-Plus
- Qwen3-Max: Production version of Qwen’s large-scale model
- Qwen3-Max-Preview: Preview version of Qwen’s latest large-scale model
- Gemini-2.5-Flash: Google’s fast, efficient multimodal model
- Gemini-2.5-Pro: Google’s most capable multimodal model
- DeepSeek-Chat: Deepseek’s most powerful model

These models represent a diverse range of architectures, training approaches, and capabilities, providing a comprehensive assessment of the decompilation task complexity and the effectiveness of our fine-tuning approach.

Table A1 presents the detailed comparison of all evaluated models across different optimization levels (O0–O3) on the HumanEval-Decompile benchmark. Each model was evaluated under two configurations: RAW (without our strategies) and ReF (with Relabeling and Function Call strategies).

Table A1. Extended comparison of large language models on the HumanEval-Decompile benchmark. All models were evaluated with two configurations: RAW (without Relabeling and Function Call) and ReF (with Relabeling and Function Call). The Re-executability Rate (%) is reported for each optimization level (O0, O1, O2, O3) and their average (AVG). Avg. Cycles indicates the average number of interaction rounds required per function (all models exhibit 1–2 rounds range). **Bold** denotes the best performance among untuned models.

Model	RAW (Without Strategies)					ReF (with Strategies)					Cycles
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG	
Untuned Models											
Qwen3-Plus	43.29	29.88	30.49	31.10	33.69	35.98	29.27	24.39	21.34	27.74	1.84
Qwen3-Coder-Plus	42.07	24.39	20.73	24.39	27.90	40.85	26.83	28.05	19.51	28.81	1.17
Qwen3-Max	56.10	28.66	33.54	35.98	38.57	56.71	34.76	31.71	29.88	38.26	1.40
Qwen3-Max-Preview	36.59	30.49	25.00	25.61	29.42	58.54	35.98	34.15	34.15	40.70	1.23
Gemini-2.5-Flash	28.66	15.85	17.07	11.59	18.29	34.76	9.15	3.05	5.49	13.11	1.14
Gemini-2.5-Pro	23.78	12.80	18.90	15.24	17.68	52.44	40.85	29.27	27.44	37.50	1.21
DeepSeek-Chat	62.80	42.07	46.34	43.90	48.78	79.27	53.05	46.34	44.51	55.79	1.37
Tuned Model (Ours)											
ReF Decompile (6.7B)	—					85.98	56.10	53.66	54.27	62.50	1.27

ReF Decompile achieves 62.50% average re-executability, demonstrating a three-tier improvement hierarchy: (1) RAW baselines show limited performance (best: DeepSeek-Chat 48.78%), (2) applying Relabeling and Function Call strategies significantly improves all models with gains ranging from 0.91% to 19.82% (e.g., Gemini-2.5-Pro: +19.82%, Qwen3-Max-Preview: +11.28%, DeepSeek-Chat: +7.01%), and (3) fine-tuning provides additional gains of 6.71% over the best untuned ReF model. The consistent RAW → ReF improvements across all seven models, regardless of architecture or specialization, confirm that our strategies address fundamental information loss challenges rather than model-specific limitations. Fine-tuning further optimizes strategy utilization, with ReF Decompile consistently outperforming all untuned ReF models across optimization levels, achieving 85.98% at O0 (vs. 79.27% best untuned) and maintaining superior performance at O3 (54.27% vs. 44.51% best untuned). For detailed ablation analysis on individual strategy components, refer to Table 3.

Performance degrades from O0 to O3 as compiler optimizations increase assembly complexity. However, ReF configurations maintain consistent advantages over RAW across all levels, with strategies remaining effective even at O3 (e.g., Gemini-2.5-Pro: 15.24% → 27.44%). ReF Decompile shows the most stable performance curve (85.98% → 54.27%). The Function Call mechanism demonstrates high efficiency, averaging 1.27 cycles per function (comparable to untuned models), with all models exhibiting 1–2 rounds range, introducing minimal overhead. This evaluation establishes three critical insights: (1) our strategies universally improve diverse model families (DeepSeek, Qwen,

Gemini), confirming they address fundamental rather than model-specific limitations; (2) the approach is complementary to model advances and readily applicable to emerging models; and (3) combining strategic preprocessing with task-specific fine-tuning yields optimal results through a two-stage optimization approach.

Conclusions

This extended evaluation firmly establishes ReF Decompile as the state-of-the-art in LLM-based end-to-end decompilation. The consistent superiority over a diverse range of untuned models—including much larger and code-specialized variants—validates the effectiveness of our Relabeling and Function Call strategies. The results demonstrate that addressing the specific information loss problems in end-to-end decompilation through targeted strategies is more effective than relying solely on model scale or general code understanding capabilities.

Appendix C. Error Case Analysis

This appendix presents a detailed analysis of two representative error cases from our decompilation experiments. These cases illustrate common failure patterns and provide insights for future improvements.

Appendix C.1. Case Study 1: Comparison Operator Error

This case demonstrates a critical error where the model incorrectly reverses the comparison operator in a conditional statement, leading to completely opposite logical behavior.

```

1 // Original Source Code
2 #include <stdio.h>
3
4 int func0(int *arr, int size) {
5     int max = -1;
6     for (int i = 1; i < size; ++i) {
7         if (arr[i] < arr[i - 1]) max = i;
8     }
9     return max;
10 }
```

The function is designed to find the last position where an array element is *less than* its predecessor, returning -1 if no such position exists.

```

1 # Compiled Assembly Code
2 <func0>:
3     endbr64
4     cmp     $0x1,%esi
5     jle     L0
6     lea     -0x2(%rsi),%eax
7     mov     %rdi,%rdx
8     lea     -0x1(%rsi),%r8d
9     mov     %esi,%ecx
10    cmp     $0x2,%eax
11    jbe     L1
12    mov     %r8d,%esi
13    movdqa  D0(%rip),%xmm2
```

```

14     mov     %rdi,%rax
15     pxor    %xmm3,%xmm3
16     shr     $0x2,%esi
17     movdqa  D1(%rip),%xmm4
18     sub     $0x1,%esi
19     shl     $0x4,%rsi
20     lea     0x10(%rdi,%rsi,1),%rsi
21     nopl    0x0(%rax,%rax,1)
22  L2:
23     movdqu  (%rax),%xmm0
24     movdqu  0x4(%rax),%xmm5
25     movdqa  %xmm2,%xmm1
26     add     $0x10,%rax
27     paddb   %xmm4,%xmm2
28     pcmpgtd %xmm5,%xmm0
29     pand     %xmm0,%xmm1
30     pandn   %xmm3,%xmm0
31     movdqa  %xmm0,%xmm3
32     por     %xmm1,%xmm3
33     cmp     %rsi,%rax
34     jne     L2
35     movdqa  %xmm3,%xmm0
36     mov     $0xffffffff,%esi
37     mov     %r8d,%edi
38     psrldq  $0x8,%xmm0
39     movdqa  %xmm0,%xmm1
40     pcmpgtd %xmm3,%xmm1
41     pand     %xmm1,%xmm0
42     pandn   %xmm3,%xmm1
43     por     %xmm0,%xmm1
44     movdqa  %xmm1,%xmm2
45     psrldq  $0x4,%xmm2
46     movdqa  %xmm2,%xmm0
47     pcmpgtd %xmm1,%xmm0
48     pand     %xmm0,%xmm2
49     pandn   %xmm1,%xmm0
50     por     %xmm2,%xmm0
51     movd    %xmm0,%eax
52     test    %eax,%eax
53     cmovbe  %esi,%eax
54     and     $0xffffffffc,%edi
55     lea     0x1(%rdi),%esi
56     cmp     %edi,%r8d
57     je      L3
58  L9:
59     movslq  %esi,%rdi
60     lea     0x1(%rsi),%r9d
61     push    %rbx
62     shl     $0x2,%rdi
63     lea     (%rdx,%rdi,1),%r10

```

```

64     cmp     %r8d,%r9d
65     jge     L4
66     cmp     %esi,%ecx
67     jle     L4
68     mov     (%r10),%r11d
69     cmp     %r11d,-0x4(%r10)
70     mov     0x4(%rdx,%rdi,1),%r10d
71     cmovg   %esi,%eax
72     cmp     %r10d,%r11d
73     lea     0x2(%rsi),%r11d
74     cmovg   %r9d,%eax
75     lea     0x3(%rsi),%r9d
76     cmp     %r9d,%r8d
77     jg      L5
78 L8:
79     movslq  %r11d,%rsi
80     mov     (%rdx,%rsi,4),%r8d
81     lea     0x0(,%rsi,4),%rdi
82     cmp     -0x4(%rdx,%rsi,4),%r8d
83     lea     0x1(%r11),%esi
84     cmovl   %r11d,%eax
85     cmp     %esi,%ecx
86     jle     L6
87     cmp     0x4(%rdx,%rdi,1),%r8d
88     cmovle  %eax,%esi
89     lea     0x2(%r11),%eax
90     cmp     %eax,%ecx
91     jle     L7
92     mov     0x8(%rdx,%rdi,1),%ebx
93     cmp     %ebx,0x4(%rdx,%rdi,1)
94     cmovle  %esi,%eax
95 L6:
96     pop     %rbx
97     ret
98     nopl    (%rax)
99 L5:
100    mov     0x8(%rdx,%rdi,1),%ebx
101    cmp     %ebx,%r10d
102    mov     0xc(%rdx,%rdi,1),%r10d
103    cmovg   %r11d,%eax
104    lea     0x4(%rsi),%r11d
105    cmp     %r10d,%ebx
106    cmovg   %r9d,%eax
107    lea     0x5(%rsi),%r9d
108    cmp     %r9d,%r8d
109    jle     L8
110    mov     0x10(%rdx,%rdi,1),%r8d
111    cmp     %r10d,%r8d
112    cmovge  %eax,%r11d
113    cmp     0x14(%rdx,%rdi,1),%r8d

```

```

114     mov     %r11d,%eax
115     lea     0x6(%rsi),%r11d
116     cmovg   %r9d,%eax
117     jmp     L8
118     nopl    (%rax)
119 L3:
120     ret
121     nopl    0x0(%rax)
122 L0:
123     mov     $0xffffffff,%eax
124     ret
125 L1:
126     mov     $0x1,%esi
127     mov     $0xffffffff,%eax
128     jmp     L9
129 L4:
130     mov     %esi,%r11d
131     jmp     L8
132 L7:
133     mov     %esi,%eax
134     pop     %rbx
135     ret

```

The assembly code contains complex SIMD (Single Instruction Multiple Data) vectorized instructions, which process multiple array elements simultaneously for optimization.

```

1  // Decompiled Output
2  int func0(int *arr, int len) {
3      int i, max = -1;
4      for (i = 1; i < len; i++) {
5          if (arr[i] > arr[i - 1])
6              max = i;
7      }
8      return max;
9  }

```

Error Pattern: The comparison operator is reversed from < to >, completely inverting the function's logic and causing the decompiled code to find increasing positions rather than decreasing ones.

Cause: The decompiler encounters the following critical SIMD vectorization assembly code:

```

23     movdqu  (%rax),%xmm0
24     movdqu  0x4(%rax),%xmm5
25     movdqa  %xmm2,%xmm1
26     add     $0x10,%rax
27     paddb   %xmm4,%xmm2
28     pcmpgtd %xmm5,%xmm0
29     pand    %xmm0,%xmm1
30     pandn   %xmm3,%xmm0

```

```

31     movdqa %xmm0,%xmm3
32     por    %xmm1,%xmm3

```

When processing this code, the decompiler misinterprets the `pcmpgtd` (packed compare greater than) instruction at line 28 combined with subsequent bitwise operations (`pand` at line 29, `pandn` at line 30, `por` at line 32). The compiler replaced the simple scalar comparison with SIMD instructions processing 4 elements simultaneously. While `pcmpgtd` naturally expresses “greater than,” the subsequent logic inverts this semantic through conditional moves (`cmovg` at lines 71, 74 in the full assembly). The model failed to track this semantic inversion through the complex instruction chain, incorrectly outputting the reversed comparison operator.

Key Insights: (1) SIMD-aware training with vectorization patterns is essential for handling optimized code, (2) explicit comparison semantic tracking through instruction chains is needed, (3) symbolic verification mechanisms should be incorporated for operator correctness validation.

Appendix C.2. Case Study 2: Conditional Branch Exchange Error

This case illustrates a subtle error where the model correctly identifies the control flow structure but incorrectly associates operations with their corresponding conditional branches.

```

1  // Original Source Code
2  #include <stdio.h>
3
4  int func0(int* lst, int size) {
5      int sum = 0;
6      for (int i = 0; i < size; i++) {
7          if (i % 3 == 0) sum += lst[i] * lst[i];
8          else if (i % 4 == 0) sum += lst[i] * lst[i] * lst[i];
9          else sum += lst[i];
10     }
11     return sum;
12 }

```

The function accumulates array elements with different transformations based on the index: squares for multiples of 3, cubes for multiples of 4, and direct values otherwise.

```

1  # Compiled Assembly Code
2  <func0>:
3      endbr64
4      mov     %rdi,%r8
5      test    %esi,%esi
6      jle     L0
7      mov     %esi,%esi
8      mov     $0x0,%edx
9      mov     $0x0,%ecx
10     jmp     L1
11  L4:
12     test     $0x3,%dl
13     jne     L2
14     mov     (%r8,%rdx,4),%edi

```

```

15     mov     %edi,%eax
16     imul    %edi,%eax
17     imul    %edi,%eax
18     add     %eax,%ecx
19 L5:
20     add     $0x1,%rdx
21     cmp     %rsi,%rdx
22     je      L3
23 L1:
24     movslq   %edx,%rax
25     imul     $0x55555556,%rax,%rax
26     shr     $0x20,%rax
27     mov     %edx,%edi
28     sar     $0x1f,%edi
29     sub     %edi,%eax
30     lea     (%rax,%rax,2),%eax
31     cmp     %edx,%eax
32     jne     L4
33     mov     (%r8,%rdx,4),%eax
34     imul    %eax,%eax
35     add     %eax,%ecx
36     jmp     L5
37 L2:
38     add     (%r8,%rdx,4),%ecx
39     jmp     L5
40 L0:
41     mov     $0x0,%ecx
42 L3:
43     mov     %ecx,%eax
44     ret

```

The compiler optimized the modulo operations using multiplication and bit manipulation, reordering the conditional checks.

```

1  // Decompiled Output
2  int func0(int *x, int c) {
3      int result = 0;
4      int i;
5      for (i = 0; i < c; ++i) {
6          if (i % 3 == 0) {
7              result += x[i] * x[i] * x[i];
8          } else if (i % 4 == 0) {
9              result += x[i] * x[i];
10         } else {
11             result += x[i];
12         }
13     }
14     return result;
15 }

```

Error Pattern: Operations in conditional branches are swapped—multiples of 3 get cubed instead of squared, while multiples of 4 get squared instead of cubed. This results in correct control flow structure but wrong semantic bindings, producing incorrect computational results.

Cause: The decompiler encounters the following critical assembly code with optimized modulo computation and reordered branches:

```

11  L4:
12      test    $0x3,%dl
13      jne     L2
14      mov     (%r8,%rdx,4),%edi
15      mov     %edi,%eax
16      imul    %edi,%eax
17      imul    %edi,%eax
18      add     %eax,%ecx
19  L5:
20      add     $0x1,%rdx
21      cmp     %rsi,%rdx
22      je      L3
23  L1:
24      movslq   %edx,%rax
25      imul     $0x55555556,%rax,%rax
26      shr     $0x20,%rax
27      mov     %edx,%edi
28      sar     $0x1f,%edi
29      sub     %edi,%eax
30      lea     (%rax,%rax,2),%eax
31      cmp     %edx,%eax
32      jne     L4
33      mov     (%r8,%rdx,4),%eax
34      imul    %eax,%eax
35      add     %eax,%ecx
36      jmp     L5

```

When processing this code, the decompiler faces confusion from the optimized modulo computation where `imul` at line 25 (with magic constant `$0x55555556`) combined with subsequent shifts replaces `i%3==0`. The compiler has reordered the branches: the code starting at line 14 handles the `i%3==0` case (which should square the value) but actually performs cubing (`imul` at lines 16–17), while the direct path at line 33 (which should cube for `i%4==0`) only performs squaring (`imul` at line 34). The model correctly identified the control structure but misaligned which arithmetic operations correspond to which conditional checks due to branch reordering during strength-reduction optimization.

Key Insights: (1) Training on diverse compiler optimization patterns including branch reordering is crucial, (2) explicit operation-to-branch binding tracking mechanisms are needed to maintain semantic associations, (3) data flow analysis should be incorporated to maintain semantic accuracy through compiler transformations.

References

1. Tan, H.; Luo, Q.; Li, J.; Zhang, Y. LLM4Decompile: Decompiling Binary Code with Large Language Models. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, Miami, FL, USA, 12–16 November 2024; pp. 3473–3487.

2. 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* **2023**, arXiv:2310.06530. [[CrossRef](#)]
3. Feng, Y.; Teng, D.; Xu, Y.; Mu, H.; Xu, X.; Qin, L.; Zhu, Q.; Che, W. Self-Constructed Context Decompilation with Fined-grained Alignment Enhancement. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, FL, USA, 12–16 November 2024; pp. 6603–6614.
4. Brumley, D.; Lee, J.; Schwartz, E.J.; Woo, M. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, 14–16 August 2013; pp. 353–368.
5. Katz, D.S.; Ruchti, J.; Schulte, E.M. Using recurrent neural networks for decompilation. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, 20–23 March 2018*; Oliveto, R., Penta, M.D., Shepherd, D.C., Eds.; IEEE Computer Society: Washington, DC, USA, 2018; pp. 346–356. [[CrossRef](#)]
6. Jiang, N.; Wang, C.; Liu, K.; Xu, X.; Tan, L.; Zhang, X. Nova⁺: Generative Language Models for Binaries. *arXiv* **2023**, arXiv:2311.13721.
7. Lacomis, J.; Yin, P.; Schwartz, E.J.; Allamanis, M.; Goues, C.L.; Neubig, G.; Vasilescu, B. DIRE: A Neural Approach to Decompiled Identifier Naming. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, 11–15 November 2019; pp. 628–639. [[CrossRef](#)]
8. Wei, T.; Mao, J.; Zou, W.; Chen, Y. A New Algorithm for Identifying Loops in Decompilation. In *Proceedings of the Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, 22–24 August 2007*; Nielson, H.R., Filé, G., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4634, pp. 170–183. [[CrossRef](#)]
9. Kirchner, K.; Rosenthaler, S. bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation. In Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, 29 August–1 September 2017; pp. 45:1–45:7. [[CrossRef](#)]
10. 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 Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 361–378.
11. Armengol-Estapé, J.; Woodruff, J.; Cummins, C.; O’Boyle, M.F.P. SLDe: A Portable Small Language Model Decompiler for Optimized Assembler. *arXiv* **2023**, arXiv:2305.12520. [[CrossRef](#)]
12. Hosseini, I.; Dolan-Gavitt, B. Beyond the C: Retargetable Decompilation using Neural Machine Translation. *arXiv* **2022**, arXiv:2212.08950. [[CrossRef](#)]
13. Hu, P.; Liang, R.; Chen, K. DeGPT: Optimizing Decompiler Output with LLM. In Proceedings of the 2024 Network and Distributed System Security Symposium (2024), San Diego, CA, USA, 26 February–1 March 2024; Volume 267622140.
14. Wang, R.; Shoshitaishvili, Y.; Bianchi, A.; Machiry, A.; Grosen, J.; Grosen, P.; Kruegel, C.; Vigna, G. Ramblr: Making Reassembly Great Again. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.
15. Balakrishnan, G.; Reps, T. Divine: Discovering variables in executables. In *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 1–28.
16. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. Code Llama: Open Foundation Models for Code. *arXiv* **2023**, arXiv:2308.12950. [[CrossRef](#)]
17. Lippincott, T. StarCoder: A general neural ensemble technique to support traditional scholarship, illustrated with a study of the post-Atlantic slave trade. In Proceedings of the 15th Annual International Conference of the Alliance of Digital Humanities Organizations, DH 2020, Ottawa, ON, Canada, 20–25 July 2020.
18. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv* **2024**, arXiv:2401.14196.
19. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA, 4–9 December 2017; pp. 5998–6008.
20. Weyssow, M.; Kamanda, A.; Zhou, X.; Sahraoui, H. CodeUltraFeedback: An LLM-as-a-Judge Dataset for Aligning Large Language Models to Coding Preferences. In *ACM Transactions on Software Engineering and Methodology*; Association for Computing Machinery: New York, NY, USA, 2025.
21. Armengol-Estapé, J.; Woodruff, J.; Brauckmann, A.; Magalhães, J.W.d.S.; O’Boyle, M.F.P. ExeBench: An ML-Scale Dataset of Executable C Functions. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022; pp. 50–59. [[CrossRef](#)]
22. Hu, E.J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W. LoRA: Low-Rank Adaptation of Large Language Models. In Proceedings of the International Conference on Learning Representations, Online, 25–29 April 2022.

23. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv* **2019**, arXiv:1910.03771.
24. Loshchilov, I.; Hutter, F. Decoupled Weight Decay Regularization. In Proceedings of the 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019.
25. Zheng, Y.; Zhang, R.; Zhang, J.; Ye, Y.; Luo, Z.; Ma, Y. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. *arXiv* **2024**, arXiv:2403.13372.
26. Dao, T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, 7–11 May 2024.
27. Wang, G.; Qin, H.; Ade Jacobs, S.; Holmes, C.; Rajbhandari, S.; Ruwase, O.; Yang, F.; Yang, L.; He, Y. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, 7–11 May 2024.
28. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [[CrossRef](#)]
29. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774. [[CrossRef](#)]
30. Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. Qwen2.5-Coder Technical Report. *arXiv* **2024**, arXiv:2409.12186. [[CrossRef](#)]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.