LLM4Decompile: Decompiling Binary Code with Large Language Models

Anonymous ACL submission

Abstract

Decompilation aims to convert binary code to high-level source code, but traditional tools like Ghidra often produce results that are difficult to read and execute. Motivated by the advancements in Large Language Models (LLMs), we propose LLM4Decompile, the first and largest open-source LLM series (1.3B to 33B) trained to decompile binary code. We optimize the LLM training process and introduce the LLM4Decompile-End models to decompile binary directly. The resulting models significantly outperform GPT-4o and Ghidra on the HumanEval and ExeBench benchmarks by over 100%. Additionally, we improve the standard refinement approach to fine-tune the LLM4Decompile-Ref models, enabling them to effectively refine the decompiled code from Ghidra and achieve a further 16.2% improvement over the LLM4Decompile-End. LLM4Decompile[1] demonstrates the potential of LLMs to revolutionize binary code decompilation, delivering remarkable improvements in readability and executability while complementing conventional tools for optimal results.

1 Introduction

Decompilation, the reverse process of converting machine code or binary code into a high-level programming language, facilitates various reverse engineering tasks such as vulnerability identification, malware research, and legacy software migration (Brumley et al., 2013; Katz et al., 2018; Hosseini and Dolan-Gavitt, 2022; Xu et al., 2023; Armengol-Estapé et al., 2023; Jiang et al., 2023; Wong et al., 2023; Hu et al., 2024). Decompilation is challenging due to the loss of information inherent in the compilation process, particularly finer details such as variable names (Lacomis et al., 2019) and fundamental structures like loops and conditionals (Wei et al., 2007). To address these challenges, numerous tools have been developed for decompilation, with Ghidra (Ghidra, 2024) and IDA Pro (Hex-Rays, 2024) being the most commonly used. Although these tools have the capability to revert binary code to high-level pseudo-code, the outputs often lack readability and re-executability (Liu and Wang, 2020a; Wang et al., 2017), which are essential for applications like legacy software migration and security instrumentation tasks (Wong et al., 2023; Dinesh et al., 2020).

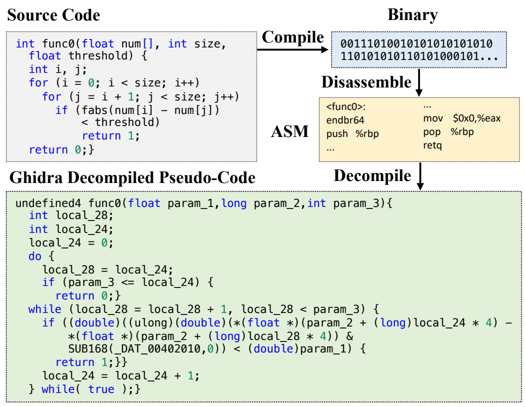

Figure 1: Illustration of compiling source code to binary, disassembling binary to assembly code (ASM), and decompiling ASM to pseudo-code with Ghidra. The pseudo-code is hard to read and not executable.

Figure 1 illustrates the transformation from the source C code to a binary file, assembly code (ASM), and pseudo-code decompiled from Ghidra. In this pseudo-code, the original nested `for` structure is replaced with a less intuitive combination of a `do-while` loop inside another `while` loop. Furthermore, array indexing like `num[i]` is decompiled into complicated pointer arithmetic such as `*(float *)(param_2 + (long)local_24 * 4)`. The decompiled output also exhibits syntactical errors, with the function return type being converted to `undefined4`. Overall, traditional decompilation tools often strip away the syntactic clarity provided by high-level languages and do not ensure the correctness of syntax, posing significant challenges

[1]https://github.com/anonepo/LLM4Decompile

even for skilled developers to reconstruct the algorithmic logic (Wong et al., 2023; Hu et al., 2024)

Recent advancements in Large Language Models (LLMs) have greatly improved the process of decompiling code.There are two primary approaches to LLM-based decompilation—*Refined-Decompile* and *End2end-Decompile*. In particular, *Refined-Decompile* prompts LLMs to refine the results from traditional decompilation tools (Hu et al., 2024; Wong et al., 2023; Xu et al., 2023). However, LLMs are primarily optimized for high-level programming languages and may not be as effective with binary data. *End2end-Decompile* fine-tunes LLMs to decompile binaries directly. Nevertheless, previous open-source applications of this approach were limited by the use of smaller models with only around 200 million parameters and restricted training corpus (Hosseini and Dolan-Gavitt, 2022; Armengol-Estapé et al., 2023; Jiang et al., 2023), In contrast, utilizing larger models trained on broader datasets has proven to substantially improve the performance (Hoffmann et al., 2024; Kaplan et al., 2020; Rozière et al., 2023; OpenAI, 2023).

To address the limitations of previous studies, we propose LLM4Decompile, the first and largest open-source LLM series with sizes ranging from 1.3B to 33B parameters specifically trained to decompile binary code. To the best of our knowledge, there's no previous study attempts to improve the capability of LLM-based decompilation in such depth or incorporate such large-scale LLMs. Based on the *End2end-Decompile* approach, we introduce three critical steps: data augmentation, data cleaning, and two-stage training, to optimize the LLM training process and introduce the LLM4Decompile-End models to decompile binary directly. Specifically, our LLM4Decompile-End-6.7B model demonstrates a successful decompilation rate of 45.4% on HumanEval (Chen et al., 2021) and 18.0% on ExeBench (Armengol-Estapé et al., 2022), far exceeding Ghidra or GPT-4o by over 100%. Additionally, we improve the *Refined-Decompile* strategy by examining the efficiency of Ghidra's decompilation process, augmenting and filtering data to fine-tune the LLM4Decompile-Ref models, which excel at refining Ghidra's output. Experiments suggest a higher performance ceiling for the enhanced *Refined-Decompile* approach, with 16.2% improvement over LLM4Decompile-End. Additionally, we assess the risks associated with the potential misuse of our model under obfuscation conditions commonly used in software protection. Our findings indicate that neither our approach nor Ghidra can effectively decompile obfuscated code, mitigating concerns about unauthorized use for infringement of intellectual property.

In summary, our contributions are as follows:

- We introduce the LLM4Decompile series, the first and largest open-source LLMs (ranging from 1.3B to 33B parameters) fine-tuned on 15 billion tokens for decompilation.

- We optimize the LLM training process and introduce LLM4Decompile-End models, which set a new performance standard of direct binary decompilation, significantly surpassing GPT-4o and Ghidra by over 100% on the HumanEval and ExeBench benchmarks.

- We improve the *Refined-Decompile* approach to fine-tune the LLM4Decompile-Ref models, enabling them to effectively refine the decompiled results from Ghidra and achieve further 16.2% enhancements over LLM4Decompile-End.

2 Related Work

The practice of reversing executable binaries to their source code form, known as decompilation, has been researched for decades (Miecznikowski and Hendren, 2002; Nolan, 2012; Katz et al., 2019). Traditional decompilation relies on analyzing the control and data flows of program (Brumley et al., 2013), and employing pattern matching, as seen in tools like Hex-Rays Ida pro (Hex-Rays, 2024) and Ghidra (Ghidra, 2024). These systems attempt to identify patterns within a program's control-flow graph (CFG) that corresponding to standard programming constructs such as conditional statements or loops. However, the output from such decompilation processes tends to be a source-code-like representation of assembly code, including direct translations of variables to registers, use of gotos, and other low-level operations instead of the original high-level language constructs. This output, while often functionally similar to the original code, is difficult to understand and may not be re-executable (Liu and Wang, 2020b; Wong et al., 2023). Drawing inspiration from neural machine translation, researchers have reformulated decompilation as a translation exercise, converting machine-level instructions into readable source code (Katz et al., 2019). Initial attempts in this area utilized recurrent neural networks (RNNs) (Katz et al.,

2018) for decompilation, complemented by error-correction techniques to enhance the outcomes.

Motivated by the success of Large Language Models (Li et al., 2023; Rozière et al., 2023; Guo et al., 2024), researchers have employed LLMs for decompilation, primarily through two approaches—*Refined-Decompile* and *End2end-Decompile*. In particular, *Refined-Decompile* prompts the LLMs to refine results from traditional decompilation tools like Ghidra or IDA Pro. For instance, DeGPT (Hu et al., 2024) enhances Ghidra's readability by reducing cognitive load by 24.4%, while DecGPT (Wong et al., 2023) increases IDA Pro's re-executability rate to over 75% by integrating error messages into its refinement process. These approaches, however, largely ignore the fact that LLMs are designed primarily for high-level programming languages (Li et al., 2023; Rozière et al., 2023; Guo et al., 2024), and their effectiveness with binary files is not well-established. *End2end-Decompile*, on the other hand, fine-tunes LLMs to decompile binaries directly. Early open-source models like BTC (Hosseini and Dolan-Gavitt, 2022) and recent development Slade (Armengol-Estapé et al., 2023) adopt the language model with around 200 million parameters (Lewis et al., 2020) to fine-tune for decompilation. While Nova (Jiang et al., 2023), which is not open-sourced, develops a binary LLM with 1 billion parameters and fine-tunes it for decompilation. Consequently, the largest open-source model in this domain is limited to 200M. Whereas utilizing larger models trained on broader datasets has proven to substantially improve the performance (Hoffmann et al., 2024; Kaplan et al., 2020; Rozière et al., 2023).

Therefore, our objective is to present the first and most extensive open-source LLM4Decompile series, aiming at comprehensively advancing the decompilation capability of LLMs. Initially, we optimize the *End2end-Decompile* approach to train the LLM4Decompile-End, demonstrating its effectiveness in directly decompiling binary files. Subsequently, we enhance the *Refined-Decompile* frameworks to integrate LLMs with Ghidra, augmenting traditional tools for optimal effectiveness.

3 LLM4Decompile

First, we introduce our strategy for optimizing LLM training to directly decompile binaries, the resulting models are named as LLM4Decompile-End. Following this, we detail our efforts for enhancing the *Refined-Decompile* approach, the corresponding fine-tuned models are referred to as LLM4Decompile-Ref, which can effectively refine the decompiled results from Ghidra.

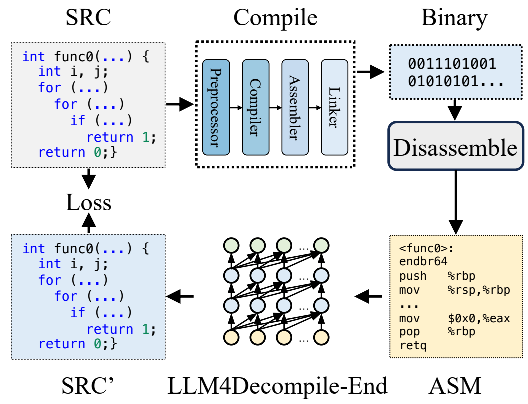

Figure 2: *End2end-Decompile* framework. The source code (SRC) is compiled to binary, disassembled to assembly instructions (ASM), and decompiled by LLM4Decompile to generate SRC'. Loss is computed between SRC and SRC' for training.

3.1 LLM4Decompile-End

In this section, we describe the general *End2end-Decompile* framework, and present details on our strategy to optimize the training of LLM4Decompile-End models.

3.1.1 The End2End-Decompile Framework

Figure 2 illustrates the *End2end-Decompile* framework from compilation to decompilation processes. During compilation, the Preprocessor processes the source code (SRC) to eliminate comments and expand macros or includes. The cleaned code is then forwarded to the Compiler, which converts it into assembly code (ASM). This ASM is transformed into binary code (0s and 1s) by the Assembler. The Linker finalizes the process by linking function calls to create an executable file. Decompilation, on the other hand, involves converting binary code back into a source file. LLMs, being trained on text, lack the ability to process binary data directly. Therefore, binaries must be disassembled by `Objdump` into assembly language (ASM) first. It should be noted that binary and disassembled ASM are equivalent, they can be interconverted, and thus we refer to them interchangeably. Finally, the loss is computed between the decompiled code and source code to guide the training.

3.1.2 Optimize LLM4Decompile-End

We optimize the training of LLM4Decompile-End Models through three key steps: 1) augmenting the training corpus, 2) improving the quality of the data, 3) and incorporating two-state training.

Training Corpus. As indicated by the Scaling-Law (Hoffmann et al., 2024; Kaplan et al., 2020), the effectiveness of an LLM heavily relies on the size of the training corpus. Consequently, our initial step in training optimization involves incorporating a large training corpus. We construct asm-source pairs based on ExeBench (Armengol-Estapé et al., 2022), which is the largest public collection of five million C functions. To further expand the training data, we consider the compilation optimization states frequently used by developers. The compilation optimization involves techniques like eliminating redundant instructions, better register allocation, and loop transformations (Muchnick, 1997), which perfectly acts as data augmentation for decompilation. The key optimization levels are O0 (default, no optimization) to O3 (aggressive optimizations). We compile the source code into all four stages, i.e., O0, O1, O2, and O3, and pair each of them with the source code.

Data Quality. Data quality is critical in training an effective model (Li et al., 2023). Therefore, our second step is to clean our training set. We follow the guidelines of StarCoder (Li et al., 2023) by computing MinHash (Broder, 2000) for the code and utilizing Locally Sensitive Hashing (LSH) to remove duplicates. We also exclude samples that are less than 10 tokens.

Two-Stage Training. Our final step for training optimization aims to educate the model with binary knowledge, and includes two-stage training. In the first stage, we train the model with a large corpus of compilable but not linkable (executable) data. Note that it's significantly easier to extract C code that is compilable but not linkable (da Silva et al., 2021; Armengol-Estapé et al., 2022). Such not-executable binary object code will closely resemble its executable version except it lacks linked addresses for external symbols. Therefore, in the first stage, we use the extensive compilable codes to ground our model in binary knowledge. In the second stage, we refine the model using executable code to ensure its practical applicability. We also conduct an ablation study for the two-stage training in Section 4.1.2.

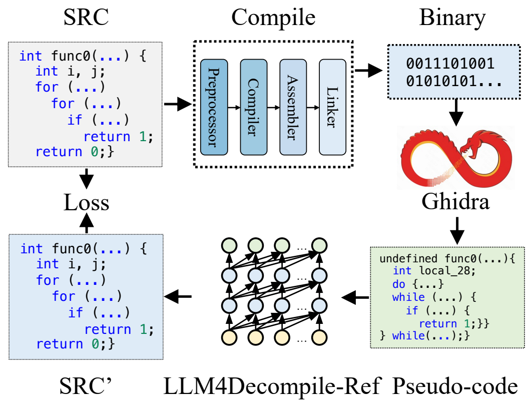

Figure 3: *Refined-Decompile* framework. It differs from *End2end-Decompile* (Figure 2) only in the LLM's input, which is pseudo-code decompiled from Ghidra.

3.2 LLM4Decompile-Ref

We now examine how the conventional decompilation tool, Ghidra, can be significantly improved by integrating it with LLMs. Note that our approach aims at refining entire outputs from Ghidra, offering a broader strategy than merely recovering names or types (Nitin et al., 2021; Xu et al., 2024). We begin by detailing the general *Refined-Decompile* framework, and discuss our strategy to enhance Ghidra's output by LLM4Decompile-Ref.

3.2.1 The Refined-Decompile Framework

The *Refined-Decompile* approach is shown in Figure 3. This approach differs from that in Figure 2 only in terms of the LLM's input, which in the case of *Refined-Decompile* comes from Ghidra's decompilation output. Specifically, Ghidra is used to decompile the binary, and then the LLM is fine-tuned to enhance Ghidra's output. While Ghidra produces high-level pseudo-code that may suffer from readability issues and syntax errors, it effectively preserves the underlying logic. Refining this pseudo-code significantly mitigates the challenges associated with understanding the obscure ASM.

3.2.2 Refine Ghidra by LLM4Decompile-Ref

Decompiling using Ghidra. Decompiling the executable code with Ghidra (Figure 3) is time-consuming due to the complex nature of the executables in ExeBench, which include numerous external functions and IO wrappers. Ghidra Headless requires 2 seconds per sample using 128-core multiprocessing. Given such a high computational load, and the high similarities between non-executable and executable binaries, we choose to decompile the non-executable files using Ghidra. This choice significantly reduces the time to 0.2 seconds per sample, enabling us to efficiently gather large amounts of training data.

Optimization Strategies. Similar to Section 3.1.2, we augment our dataset by compiling with optimization levels O0, O1, O2, and O3. We further filter the dataset using LSH to remove duplicates. As shown in Figure 1, Ghidra often generates overly long pseudo-code. Consequently, we filter out any samples that exceed the maximum length accepted by our model.

4 Experiments

In this section, we discuss the experimental setups and results for LLM4Decompile-End and LLM4Decompile-Ref respectively.

4.1 LLM4Decompile-End

4.1.1 Experimental Setups

Training Data. As discussed in Section 3.1.2, we construct asm-source pairs based on compilable and executable datasets from ExeBench (Armengol-Estapé et al., 2022), where we only consider the decompilation of GCC (Stallman et al., 2003) compiled C function under x86 Linux platform. After filtering, our refined compilable training dataset includes 7.2 million samples, containing roughly 7 billion tokens. Our executable training dataset includes 1.6 million samples, containing roughly 572 million tokens. To train the model, we use the following template: `# This is the assembly code: [ASM code] # What is the source code? [source code]`, where `[ASM code]` corresponds to the disassembled assembly code from the binary, and `[source code]` is the original C function. Note that the template choice does not impact the performance, since we fine-tune the model to produce the source code.

Evaluation Benchmarks and Metrics. To evaluate the models, we introduce HumanEval (Chen et al., 2021) and ExeBench (Armengol-Estapé et al., 2022) benchmarks. HumanEval is the leading benchmark for code generation assessment and includes 164 programming challenges with accompanying Python solutions and assertions. We converted these Python solutions and assertions into C, making sure that they can be compiled with the GCC compiler using standard C libraries and pass all the assertions, and name it HumanEval-Decompile. ExeBench consists of 5000 real-world C functions taken from GitHub with IO examples. Note that the HumanEval-Decompile consists of individual functions that depend only on the standard C library. In contrast, ExeBench includes functions extracted from real-world projects with user-defined structures and functions.

As for the evaluation metrics, we follow previous work to calculate the re-executability rate (Armengol-Estapé et al., 2023; Wong et al., 2023). During evaluation, the C source code is first compiled into a binary, then disassembled into assembly code, and fed into the decompilation system to be reconstructed back into C code. This decompiled C code is then combined with the assertions to check if it can successfully execute and pass those assertions.

Model Configurations. The LLM4Decompile uses the same architecture as DeepSeek-Coder (Guo et al., 2024) and we initialize our models with the corresponding DeepSeek-Coder checkpoints. We employ Sequence-to-sequence prediction (S2S), which is the training objective adopted in most neural machine translation models that aim to predict the output given the input sequence. As illustrated in Equation 1, it minimizes the negative log likelihood for the source code tokens $x_i, ..., x_j$:

$$\mathcal{L} = -\sum_i \log P_i(x_i, ..., x_j | x_1, ..., x_{i-1}; \theta) \quad (1)$$

Where the loss is calculated only for the output sequence $x_i...x_j$, or the source code.

Baselines. We selected two key baselines for comparison. First, GPT-4o (OpenAI, 2023) represents the most capable LLMs, providing an upper bound on LLM performance. Second, DeepSeek-Coder (Guo et al., 2024) is selected as the current SOTA open-source Code LLM. It represents the forefront of publicly available models specifically tailored for coding tasks. While recent work Slade (Armengol-Estapé et al., 2023) fine-tunes language model for decompilation, it relies on intermediate compiler outputs, specifically, the *.s files. In practice, however, such intermediate files are rarely released by developers. Therefore, we focus on a more realistic approach, and consider decompilation only from the binaries, for further discussions please refer to Appendix A.

Implementation. We use the DeepSeek-Coder models obtained from Hugging Face (Wolf et al.,

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
DeepSeek-Coder-6.7B	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
GPT-4o	0.3049	0.1159	0.1037	0.1159	0.1601	0.0443	0.0328	0.0397	0.0343	0.0378
LLM4Decompile-End-1.3B	0.4720	0.2061	0.2122	0.2024	0.2732	0.1786	0.1362	0.1320	0.1328	0.1449
LLM4Decompile-End-6.7B	**0.6805**	**0.3951**	**0.3671**	**0.3720**	**0.4537**	**0.2289**	**0.1660**	**0.1618**	**0.1625**	**0.1798**
LLM4Decompile-End-33B	0.5168	0.2556	0.2415	0.2475	0.3154	0.1886	0.1465	0.1396	0.1411	0.1540

Table 1: Main comparison of *End2end-Decompile* approaches for re-executability rates on evaluation benchmarks.

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Compilable-1.3B	0.4268	0.1646	0.1646	0.1707	0.2317	0.0568	0.0446	0.0416	0.0443	0.0468
Compilable-6.7B	**0.5183**	**0.3354**	**0.3232**	**0.3232**	**0.3750**	0.0752	0.0649	0.0671	0.0660	0.0683
Executable-1.3B	0.1951	0.1280	0.1280	0.1159	0.1418	0.2194	0.1946	0.1931	0.1950	0.2005
Executable-6.7B	0.3720	0.1829	0.2256	0.1707	0.2378	**0.2938**	**0.2598**	**0.2591**	**0.2549**	**0.2669**

Table 2: Ablation study on training dataset. The “Compilable” models are trained on 7.2M non-executable functions, while the “Executable” models are trained on 1.6M executable functions.

2019). We train our models using LLaMA-Factory library (Zheng et al., 2024). For 1.3B and 6.7B models, we set a $batch\ size = 2048$ and $learning\ rate = 2e{-}5$ and train the models for 2 epochs (15B tokens). Experiments are performed on NVIDIA A100-80GB GPU clusters. Fine-tuning the 1.3B and 6.7B LLM4Decompile-End takes 12 and 61 days on $8 \times A100$ respectively. Limited by the resources, for the 33B model we only train for 200M tokens. For evaluation, we use the *vllm* (Kwon et al., 2023) to accelerate the generation (decompilation) process. We employ greedy decoding to minimize randomness.

4.1.2 Experimental Results

Main Results. Table 1 presents the re-executability rate under different optimization states for our studied models. The base version of DeepSeek-Coder-33B is unable to accurately decompile binaries. It could generate code that seemed correct but failed to retain the original program semantics. GPT-4o shows notable decompilation skills; it's capable to decompile non-optimized (O0) code with a success rate of 30.5%, though the rate significantly decreases to about 11% for optimized codes (O1-O3). The LLM4Decompile-End models, on the other hand, demonstrate excellent decompilation abilities. The 1.3B version successfully decompiles and retains the program semantics in 27.3% of cases on average, whereas the 6.7B version has a success rate of 45.4%. This improvement underscores the advantages of using larger models to capture a program's semantics more effectively. While attempting to fine-tune the 33B model, we encountered substantial challenges related to the high communication loads, which significantly slowed the training process and restricted us to using only 200M tokens (Section 4.1.1). Despite this limitation, the 33B model still outperforms the 1.3B model, reaffirming the importance of scaling up the model size.

Ablation Study. As discussed in Section 4.1.1, our training data comprises two distinct sets: 7.2 million compilable functions (non-executable) and 1.6M executable functions. We conducted an ablation study using these datasets, and the results are displayed in Table 2. Here, “Compilable” denotes the model trained solely on compilable data, while “Executable” indicates models trained exclusively on executable data. Notably, the binary object from compilable functions lacks links to function calls, which is similar in text distribution to the HumanEval-Decompile data, consisting of single functions dependent only on standard C libraries. Consequently, the 6.7B model trained only on compilable data successfully decompiled 37.5% of HumanEval-Decompile functions, but only 6.8% on ExeBench, which features real functions with extensive user-defined functions. On the other hand, the 6.7B model trained solely on executable data achieved a 26.7% re-executability rate on the ExeBench test set but faced challenges with single functions, with only a 23.8% success rate on HumanEval-Decompile due to the smaller

Model/Metrics	Re-executability Rate					Edit Similarity				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	0.6805	0.3951	0.3671	0.3720	0.4537	0.1557	0.1292	0.1293	0.1269	0.1353
Ghidra										
Base	0.3476	0.1646	0.1524	0.1402	0.2012	0.0699	0.0613	0.0619	0.0547	0.0620
+GPT-4o	0.4695	0.3415	0.2866	0.3110	0.3522	0.0660	0.0563	0.0567	0.0499	0.0572
+LLM4Decompile-Ref-1.3B	0.6890	0.3720	0.4085	0.3720	0.4604	0.1517	0.1325	0.1292	0.1267	0.1350
+LLM4Decompile-Ref-6.7B	**0.7439**	0.4695	**0.4756**	**0.4207**	**0.5274**	**0.1559**	0.1353	0.1342	0.1273	0.1382
+LLM4Decompile-Ref-33B	0.7073	**0.4756**	0.4390	0.4146	0.5091	0.1540	**0.1379**	**0.1363**	**0.1307**	**0.1397**

Table 3: Main comparison of *Refined-Decompile* approaches for re-executability rate and Edit Similarity on HumanEval-Decompile benchmark. "+GPT-4o" refers to enhance the Ghidra results with GPT-4o, "+LLM4Decompile-Ref" means refining Ghidra results with the fine-tuned LLM4Decompile-Ref models.

size of the training corpus. Limited by the space, we present further analysis in Appendix B.

4.2 LLM4Decompile-Ref

4.2.1 Experimental Setups

Experimental Datasets. The training data is constructed using ExeBench, with Ghidra Headless employed to decompile the binary object file. Due to constraints in computational resources, only 400K functions each with optimization levels from O0 to O3 (1.6M samples, 1B tokens) are used for training and the evaluation is conducted on HumanEval-Decompile. The models are trained using the same template described in Section 4.1.1. In addition, following previous work (Hosseini and Dolan-Gavitt, 2022; Armengol-Estapé et al., 2023), we access the readability of decompiled results in terms of Edit Similarity score.

Implementation. Configuration settings for the model are consistent with those in Section 4.1.1. For the 1.3B, 6.7B models, the fine-tuning process involves 2B tokens in 2 epochs, and requires 2, and 8 days respectively on $8 \times A100$ respectively. Limited by the resource, for 33B model we only train for 200M tokens. For evaluation, we first access the re-executability rate of Ghidra to establish a baseline. Subsequently, GPT-4o is used to enhance Ghidra's decompilation result with the prompt, `Generate linux compilable C/C++ code of the main and other functions in the supplied snippet without using goto, fix any missing headers. Do not explain anything.`, following DecGPT (Wong et al., 2023). Finally, we use LLM4Decompile-Ref models to refine the Ghidra's output.

4.2.2 Experimental Results

The results for the baselines and *Refined-Decompile* approaches are summarized in Table 3. For the pseudo-code decompiled by Ghidra, which is not optimized for re-execution, only an average of 20.1% of them pass the test cases. GPT-4o assists in refining this pseudo-code and enhancing its quality. The LLM4Decompile-Ref models offer substantial improvements over Ghidra's outputs, with the 6.7B model yielding a 160% increase in re-executability. Similar to the discussion in Section 4.1.2, the 33B model outperforms the 1.3B model even though it used considerably less training data. And it achieves performance that is only 3.6% below the 6.7B model, which benefited from ten times more training data. When compared to LLM4Decompile-End-6.7B, the LLM4Decompile-Ref-6.7B model, though trained on just 10% of the data in LLM4Decompile-Ref models, shows a 16.2% performance increase, suggesting a greater potential for the *Refined-Decompile* approach.

An analysis of readability across different methods is also conducted and presented in Table 3, examples are presented in Figure 4. For text similarity, all decompiled outputs diverge from the original source code, with Edit Similarity ranging from 5.7% to 14.0%, primarily because the compilation process removes variable names and optimizes the logic structure. Ghidra generates pseudo-code that is particularly less readable with 6.2% Edit Similarity on average. Interestingly, with refinement from GPT (Ghidra+GPT-4o), there is a marginal decrease in Edit Similarity. GPT assists in refining type errors like `undefined4` and `ulong` (Figure 4), however, it struggles to accurately reconstruct `for` loops and array indexing. In contrast, both LLM4Decompile-End and LLM4Decompile-Ref generate outputs that are more aligned with the format of the source code and easier to comprehend. To summarize, domain-specific fine-tuning is crucial for enhancing re-executability and readability of decompilation outputs.

Model/Obfuscation	Control Flow Flattening					Bogus Control Flow				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	0.0427	0.0488	0.0488	0.0305	0.0427	0.0976	0.0732	0.0793	0.0976	0.0869
Ghidra	0.1220	0.0671	0.0610	0.0671	0.0793	0.0610	0.0427	0.0305	0.0427	0.0442
+LLM4Decompile-Ref-6.7B	0.0671	0.0366	0.0488	0.0549	0.0519	0.1585	0.1402	0.0854	0.0793	0.1159

Table 4: Re-executability rates of different approaches on the HumanEval-Decompile benchmark under obfuscations. Compared to Table 3, the decompilation success rates significantly drop for over 70%.

Source Code

```c
int func0(float num[], int size,
  float threshold) {
  int i, j;
  for (i = 0; i < size; i++)
    for (j = i + 1; j < size; j++)
      if (fabs(num[i] - num[j])
          < threshold)
          return 1;
  return 0;}
```

LLM4Decompile-End-6.7B

```c
int func0(float *x, int n,
  float eps){
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (fabsf(x[i] - x[j]) < eps)
        return 1;
  return 0;}
```

Ghidra

```c
undefined4 func0(float param_1,long param_2,int param_3){
  int local_28;
  int local_24;
  local_24 = 0;
  do {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;}
  while (local_28 = local_28 + 1, local_28 < param_3) {
    if ((double)((ulong)(double)(*(float *)(param_2 + (long)local_24 * 4) -
        *(float *)(param_2 + (long)local_28 * 4)) &
        SUB168(_DAT_00402010,0)) < (double)param_1) {
      return 1;}}
    local_24 = local_24 + 1;
  } while( true );}
```

Ghidra+LLM4Decompile-6.7B

```c
int func0(float *x, int n,
  float eps){
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (fabsf(x[i] - x[j]) < eps)
        return 1;
  return 0;}
```

GPT-4o

```c
int func0(float **arr, int length,
  float threshold) {
  int outer = 0;
  while (outer < length) {
    int inner = 0;
    while (inner < length) {
      float diff = arr[outer][inner]
        - arr[inner][inner];
      if (fabs(diff) <= threshold) {
        return 1;}
      inner++;}
    outer++;}
  return 0;}
```

Ghidra+GPT-4o

```c
typedef unsigned int undefined4;
undefined4 func0(float param_1,
  long param_2, int param_3)
  {
  int local_28;
  int local_24;
  local_24 = 0;
  while (1) {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;
    }
    while (local_28 + 1 < param_3) {
      local_28++;
      if ((double)((*(float *)
        (param_2 + (long)local_24 * 4)
        - *(float *)(param_2 +
        (long)local_28 * 4)))
        < (double)param_1) {
          return 1;
      }
    }
    local_24++;}}
```

Figure 4: Decompilation results of different approaches. GPT-4o output is plausible yet fail to recover the array dimension (incorrect 2D array `arr[outer][inner]`). Ghidra's pseudo-code is notably less readable as discussed in Figure 1. GPT-refined Ghidra result (Ghidra+GPT-4o) marginally enhances readability but fails to correctly render `for` loops and array indexing. Conversely, LLM4Decompile-End and LLM4Decompile-Ref produce accurate and easy-to-read outputs.

5 Obfuscation Discussion

The process of decompilation aims at revealing the source code from binaries distributed by developers, presenting a potential threat to the protection of intellectual property. To resolve the ethical concerns, this section accesses the risks of the possible misuse of our decompilation models.

In software development, engineers typically implement obfuscation techniques before releasing binary files to the public. This is done to protect the software from unauthorized analysis or modification. In our study, we focus on two fundamental obfuscation techniques as suggested in Obfuscator-LLVM (Junod et al., 2015): Control Flow Flattening (CFF) and Bogus Control Flow (BCF). These techniques are designed to disguise the true logic of the software, thereby making decompilation more challenging to protect the software's intellectual property. We present the details of these two techniques in the Appendix C.

Results summarized in Table 4 demonstrate that basic conventional obfuscation techniques are sufficient to prevent both Ghidra and LLM4Decompile from decoding obfuscated binaries. For example, the decompilation success rate for the most advanced model, LLM4Decompile-Ref-6.7B, drops significantly for 90.2% (0.5274 to 0.0519) under CFF and 78.0% (0.5274 to 0.1159) under BCF. Considering the industry standard of employing several complex obfuscation methods prior to software release, experimental results in Table 4 mitigate the concerns about unauthorized use for infringement of intellectual property.

6 Conclusions

We propose LLM4Decompile, the first and largest open-source LLM series with sizes ranging from 1.3B to 33B trained to decompile binary code. Based on the *End2end-Decompile* approach, we optimize the LLM training process and introduce the LLM4Decompile-End models to decompile binary directly. The resulting 6.7B model shows a decompilation accuracy of 45.4% on HumanEval and 18.0% on ExeBench, surpassing existing tools like Ghidra and GPT-4o over 100%. Additionally, we improve the *Refined-Decompile* strategy to fine-tune the LLM4Decompile-Ref models, which excel at refining the Ghidra's output, with 16.2% improvement over LLM4Decompile-End. Finally, we conduct obfuscation experiments and address concerns regarding the misuse of LLM4Decompile models for infringement of intellectual property.

Limitations

The scope of this research is limited to the compilation and decompilation of C language targeting the x86 platform. While we are confident that the methodologies developed here could be easily adapted to other programming languages and platforms, these potential extensions have been reserved for future investigation. Furthermore, Our research is limited by financial constraints, with a budget equivalent to using $8 \times A100$ GPUs for one year, which includes all trials and iterations. As a result, we have only managed to fully fine-tune models up to 6.7B, and conducted initial explorations on the 33B models with a small dataset, leaving the exploration of 70B and larger models to future studies. Nonetheless, our preliminary tests confirm the potential advantages of scaling up model sizes and suggest a promising direction for future decompilation research into larger models.

Ethic Statement

We have evaluated the risks of the possible misuse of our decompilation models in Section 5. Basic obfuscation methods such as Control Flow Flattening and Bogus Control Flow have been empirically tested and proven to protect against unauthorized decompilation by both traditional tools like Ghidra and advanced models like LLM4Decompile. This built-in limitation ensures that while LLM4Decompile is a powerful tool for legitimate uses, it does not facilitate the infringement of intellectual property.

In practical applications in the industry, software developers typically employ a series of complex obfuscation methods before releasing their software. This practice adds an additional layer of security and intellectual property protection against decompilation. LLM4Decompile's design and intended use respect these measures, ensuring that it serves as an aid in legal and ethical scenarios, such as understanding legacy code or enhancing cybersecurity defenses, rather than undermining them.

The development and deployment of LLM4Decompile are guided by strict ethical standards. The model is primarily intended for use in scenarios where permission has been granted or where the software is not protected by copyright. This includes academic research, debugging, learning, and situations where companies seek to recover lost source code of their own software.

References

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. 2022. Exebench: An ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 50–59, New York, NY, USA. Association for Computing Machinery.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. 2023. Slade: A portable small language model decompiler for optimized assembler. *CoRR*, abs/2305.12520.

Andrei Z Broder. 2000. Identifying and filtering near-duplicate documents. In *Annual symposium on combinatorial pattern matching*, pages 1–10. Springer.

David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 353–368. USENIX Association.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. ANGHABENCH: A suite with one million compilable C benchmarks for code-size reduction. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 378–390. IEEE.

Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511.

Ghidra. 2024. Ghidra software reverse engineering framework.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Hex-Rays. 2024. Ida pro: a cross-platform multi-processor disassembler and debugger.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. 2024. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the C: retargetable decompilation using neural machine translation. *CoRR*, abs/2212.08950.

Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/-CorpusID*, volume 267622140.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova^{+}: Generative language models for binaries. *CoRR*, abs/2311.13721.

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *Preprint*, arXiv:2001.08361.

Deborah S. Katz, Jason Ruchti, and Eric M. Schulte. 2018. Using recurrent neural networks for decompilation. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 346–356. IEEE Computer Society.

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

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

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 628–639. IEEE.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Zhibo Liu and Shuai Wang. 2020a. 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. Association for Computing Machinery.

Zhibo Liu and Shuai Wang. 2020b. How far we have come: testing decompilation correctness of C decompilers. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 475–487. ACM.

Jerome Miecznikowski and Laurie J. Hendren. 2002. Decompiling java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*.

Steven S. Muchnick. 1997. Advanced compiler design and implementation.

Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT : A transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 48–57, Online. Association for Computational Linguistics.

Godfrey Nolan. 2012. Decompiling android. In *Apress*.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Richard M Stallman et al. 2003. Using the gnu compiler collection. *Free Software Foundation*, 4(02).

Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *NDSS*.

Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A new algorithm for identifying loops in decompilation. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771.

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

Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2023. Lmpa: Improving decompilation by synergy of large language model and program analysis. *CoRR*, abs/2306.02546.

Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2024. Leveraging generative models to recover variable names from stripped binary. *Preprint*, arXiv:2306.02546.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

A ExeBench Setups

For every sample in ExeBench's executable splits, assembly code from *.s file—a compiler's intermediate output as discussed in Section 3.1 and Figure 1—is required to compile the sample into a binary. The specific compilation settings and processing details, however, are not provided by the authors. Consequently, we choose to compile the code in a standard way and manage to compile only half of the samples. This leaves us with 443K out of 797K samples for the executable training set and 2621 out of 5000 samples for the executable test set. Accordingly, we train our model on the 443K samples and conduct the re-executability evaluation on these 2621 samples, the results are shown in Table 1.

The researchers from Slade (Armengol-Estapé et al., 2023), who also developed ExeBench (Armengol-Estapé et al., 2022), have published their decompilation findings on ExeBench. They chose to decompile the intermediate output, or assembly code from *.s file, directly without further compilation into binaries, where in practice, such intermediate output is rarely released by software developers. Their reported results, as seen in Table 5, show a significant difference from ours. Their version of ChatGPT achieved a re-executability rate of 22.2% and an edit similarity of 44.0% under O0 optimization. On the other hand, our GPT-4o model only reached a 4.4% re-executability rate and a 7.9% edit similarity. The approach taken by Slade involves settings not commonly available in practical decompilation scenarios, which explains why their results vary significantly from ours. We adheres to a more realistic setting, decompiling binary files based solely on their intrinsic data, without any external information.

To further illustrate our settings, Figure 5 offers an example where the source function includes specific user-defined types like `Ltc4151State`, `Ltc4151`, and `device`. However, these types are completely lost after compilation, i.e., no information related to these user-definitions can be found in the binary (disassembled ASM code). Conse-

Model/Metrics	Re-executability		Edit Similarity	
Optimization Level	O0	O3	O0	O3
Slade	59.5	52.2	71.0	60.0
ChatGPT	22.2	13.6	44.0	34.0
GPT-4o(ours)	4.4	3.4	7.9	6.6

Table 5: Re-executability and Edit Similarity on Exebench.

Source Code

```
void StateIdle(Ltc4151State next,
  Ltc4151 *device) {
  device->state = next;
}
```

ASM

```
<StateIdle>:
endbr64
push %rbp
mov %rsp,%rbp
mov %edi,-0x4(%rbp)
mov %rsi,-0x10(%rbp)
mov -0x10(%rbp),%rax
mov -0x4(%rbp),%edx
mov %edx,(%rax)
nop
pop %rbp
retq
```

GPT-4o

```
void StateIdle(int a, int *b) {
*b = a;
}
```

Figure 5: Decompilation results of GPT-4o on ExeBench test case.

quently, GPT-4o is unable to reconstruct these types based purely on the ASM (the realistic setting), instead converting them to default types `int` or `pointer`, producing non-executable code. This issue was pervasive across the ExeBench test set, leading to the failure of GPT-4o models in decompiling the ExeBench samples in a realistic setting.

B Further Analysis of LLM4Decompile-Ref

Figure 6 illustrates that the re-executability rate decreases as the input length increases, and there is a marked decline in performance at higher levels of code optimization, highlighting the difficulties in decompiling long and highly optimized sequences. Importantly, the performance difference between the 1.3B and 6.7B models showcased in the figure emphasizes the advantages of larger models in such tasks. Larger models, with their expanded computational resources and deeper learning capabilities, are inherently better at resolving the challenges posed by complex decompilations.

The error analysis presented in Figure 7 for LLM4Decompile-End-6.7B indicates that logical errors are prevalent in the HumanEval-Decompile scenarios, with 64% of errors due to assertions that the decompiled codes do not pass. In the ExeBench dataset, which features real functions with user-defined structures and types, the major challenges are related to reclaiming these user-specific components. Where 50% of the errors come from undeclared functions, and 28% from improper use of structures. Given that these user-defined details are typically lost during the compilation process, reconstructing them can be particularly challenging. Integrating techniques like Retrieval Augmented Generation might supplement the decompilation process with necessary external information.

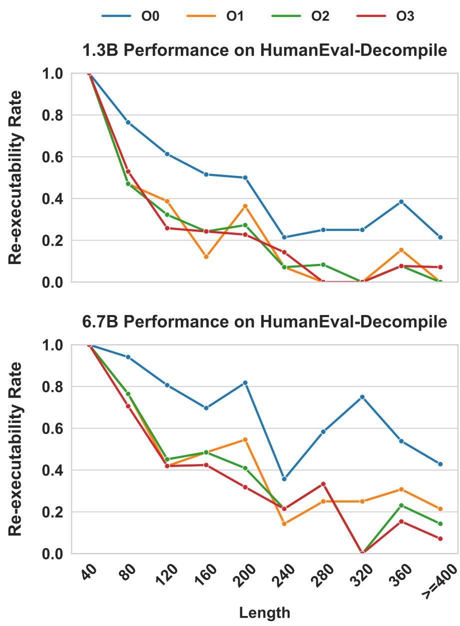

Figure 6: Re-executability rate with the growth of input length. 6.7B model is more robust against input length.

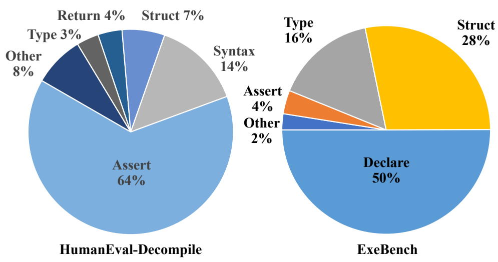

Figure 7: Types of errors identified in the two benchmarks: LLM4Decomile-End-6.7B faces issues with logical errors in HumanEval-Decompile and user-defined components in ExeBench.

C Obfuscation Techniques

We provide the details of two classic obfuscation techniques suggested in Obfuscator-LLVM.

Control Flow Flattening enhances the security of software by transforming its straightforward, hierarchical control flow into a more complex, flattened structure. The workflow involves breaking a function into basic blocks, arranging these blocks at the same level, and encapsulating them within a switch statement inside a loop.

Bogus Control Flow modifies a function's execution sequence by inserting an additional basic blockprior to the existing one. This added block includes an opaque predicate, followed by a conditional jump that leads back to the original block. Additionally, the original basic block is polluted with randomly selected, meaningless instructions.