Self-Constructed Context Decompilation with Fined-grained Alignment Enhancement

Yunlong Feng, Dechuan Teng, Yang Xu, Honglin Mu,
Xiao Xu, Libo Qin, Qingfu Zhu*, Wanxiang Che**
Harbin Institute of Technology, China
{ylfeng,dcteng,yxu,hlmu,xxu,lbqin,qfzhu,car}@ir.hit.edu.cn

Abstract

Decompilation transforms compiled code back into a high-level programming language for analysis when source code is unavailable. Previous work has primarily focused on enhancing decompilation performance by increasing the scale of model parameters or training data for pre-training. Based on the characteristics of the decompilation task, we propose two methods: (1) Without fine-tuning, the Self-Constructed Context Decompilation (sc^{2}dec) method recompiles the LLM's decompilation results to construct pairs for in-context learning, helping the model improve decompilation performance. (2) Fine-grained Alignment Enhancement (FAE), which meticulously aligns assembly code with source code at the statement level by leveraging debugging information, is employed during the fine-tuning phase to achieve further improvements in decompilation. By integrating these two methods, we achieved a Re-Executability performance improvement of approximately 3.90% on the Decompile-Eval benchmark, establishing a new state-of-the-art performance of 52.41%. The code, data, and models are available at https://github.com/AlongWY/sccdec.

1 Introduction

Decompilation is the process of converting compiled machine code or bytecode back into a high-level programming language. This process is typically used to analyze how the software works when the source code is not accessible (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). Many tools have been developed for decompilation, like Ghidra (Ghidra, 2024) and IDA Pro (Hex-Rays, 2024). However, these tools often struggle to generate human-readable code. The main challenge in decompilation is that it's hard to fully reconstruct
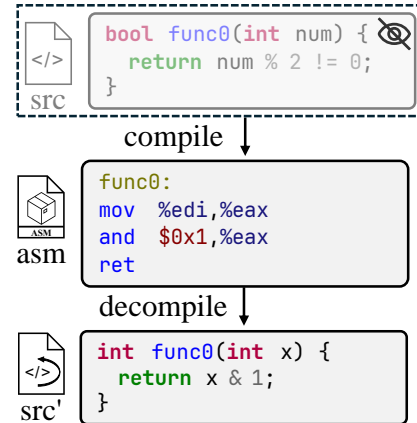

Figure 1: Pipeline of the decompilation. The input for decompilation tasks is typically assembly, with the source code invisible. Additionally, the source code obtained through decompilation usually does not exactly match the original code.

the source code, especially details such as variable names (Lacomis et al., 2019) and primary structures (Wei et al., 2007), which are frequently lost during the compilation process.

Recent advances in large language models (LLMs) have prompted researchers to view programming languages as distinct linguistic systems, utilizing pre-trained code LLMs to accomplish various coding tasks (Lippincott, 2020; Rozière et al., 2023; Guo et al., 2024; OpenAI, 2023). These models have demonstrated significant performance improvements over traditional techniques (Zeng et al., 2022; Xu et al., 2022), making it feasible to apply LLMs to decompilation challenges. For instance, Transformer-based models such as Slade (Armengol-Estapé et al., 2023) and Nova (Jiang et al., 2023) have shown potential in using language models to translate binary code back into more readable and structured source code. Recently, Tan et al. (2024) developed and released the first open-source LLM focused on decompilation, alongside the construction of the first

---
*Corresponding Author

decompilation benchmark to evaluate the decompilation capabilities of models. These efforts primarily treat the decompilation task as a translation task, training a robust decompilation model on extensive synthetic data (Hosseini and Dolan-Gavitt, 2022; Armengol-Estapé et al., 2023; Jiang et al., 2023; Tan et al., 2024).

By analyzing the characteristics of decompilation tasks, we have found two features:

- The decompiled code generated by the model is usually compilable.
- The compiler can produce external debugging information for the debugger.

Based on the characteristics of the decompilation task, we propose two approaches: (1) *Self-Constructed Context Decompilation* (sc^{2}dec), a tuning-free approach, leverages the decompilation results from the model and specific compiler to derive ground-truth $(assembly\ code, source\ code)$ pair closely related to the test sample. By incorporating the obtained pair as a demonstration example within the context, we can fully leverage the model's in-context learning capabilities to enhance performance. (2) *Fine-grained Alignment Enhancement* (FAE) introduces a new step-by-step decompilation training objective in addition to the end-to-end decompilation training objective that relies on large-scale pre-training for implicit alignment. By leveraging debugging information designed for debuggers, we can align assembly code with high-level code at a finer statement level. This allows the model to more faithfully preserve the original functionality after decompilation.

Combining these two methods, we have achieved an approximately 3.90% improvement in re-executability performance based on llm4decompile model (Tan et al., 2024), reaching a new state-of-the-art performance of 52.41% re-executability.

To summarize, our contributions are as follows:

- We introduce *Self-Constructed Context Decompilation*, which leverages the compilability of decompilation results to construct better contexts.

- We introduce *Fine-grained Alignment Enhancement*, which fine-tunes the model using fine-grained alignment data between assembly code and source code derived from debug information. We also propose how to synthesize its training data automatically.

- The experiments on the Decompile-Eval benchmark show that our proposed model achieves new state-of-the-art results.

2 Related Works

Decompilation has a long history, centered on converting binary files into human-readable source code. Traditional tools such as Hex-Rays IDA Pro (Hex-Rays, 2024) and Ghidra (Ghidra, 2024) predominantly rely on analyzing a program's control flow graph (CFG) to perform this task (Brumley et al., 2013). These tools operate by scrutinizing instructions within assembly code to construct CFGs and identify common programming structures like if-else statements and loops (Wei et al., 2007). However, they rely on intricate and error-prone expert-built rule systems that frequently fail with optimized binary code, a common practice in commercial software development. Moreover, traditional tools often generate outputs that closely resemble assembly code representations, translating variables into registers and utilizing low-level operations such as goto statements. This output is not only challenging to comprehend but also occasionally cannot be recompiled (Liu and Wang, 2020).

Inspired by neural machine translation, researchers have begun to conceptualize decompilation as a translation task, wherein machine-level instructions are converted into readable source code using neural networks. Initial endeavors employing recurrent neural networks (RNNs) demonstrated limited success (Katz et al., 2018). However, recent advancements in natural language processing (NLP) have facilitated the application of pre-trained language models (LMs) in decompilation. Notable examples include BTC (Hosseini and Dolan-Gavitt, 2022), Slade (Armengol-Estapé et al., 2023), and Nova (Jiang et al., 2023), which mark significant strides in the field. Recently, Tan et al. (2024) created and released the first open-source large language models specifically for decompilation and established an evaluation benchmark that considers the re-compilability and re-executability of decompiled code.

3 Method

In this section, we introduce the Self-Constructed Context Decompilation method and Fine-grained Alignment Enhancement fine-tuning. Figures 2 and 3 illustrate the methods.
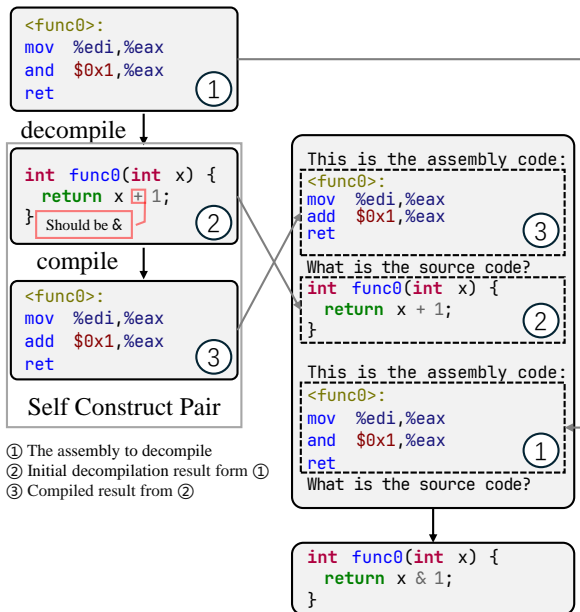
Figure 2: The pipeline for Self-Constructed Context Decompilation operates as follows: when the LLM decompiles and generates compilable code, we compile this code to construct the context, and then use it to decompile the initial assembly code again.

3.1 Self-Constructed Context Decompilation

In this section, we introduce the **S**elf-**C**onstructed **C**ontext **Dec**ompilation (sc^{2}dec) method designed to enhance the performance of decompilation models. The method recompiles the decompiled code generated by the model to construct the context. The sc^{2}dec method involves the following key steps, as the Figure 2 shows:

1. **Decompile**: Initially, the model decompiles the assembly code back to the target programming language. We call it the initial decompilation result.
2. **Compile and Disassembly**: Then we recompile the initial decompilation result to obtain the corresponding assembly code if the generated code is compilable.
3. **Constructed Context**: The self-constructed context is then formed by concatenating the generated C code with its corresponding assembly code.
4. **Decompile Again**: Decompile the assembly code with the self-constructed context again.

In this context, the assembly code precedes the C code, resembling an example in the context. The final input to the model is a concatenation of the self-constructed context (with assembly code first and C code second) and the original assembly code. This
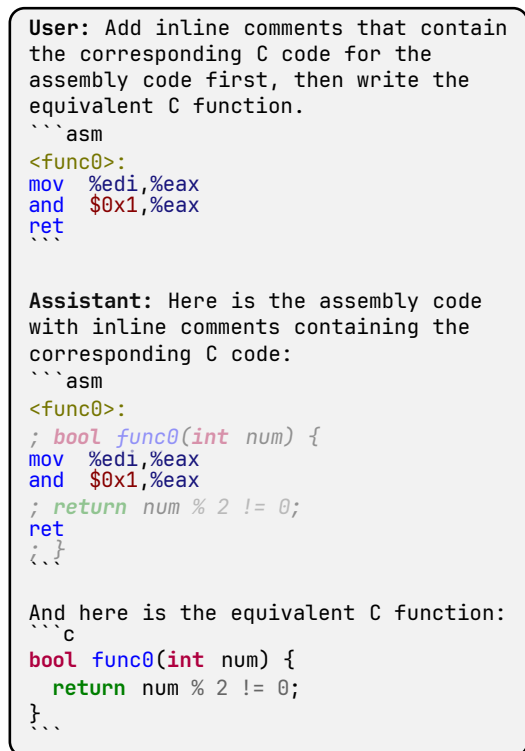

Figure 3: An example of step-by-step decompilation is presented, where the training objective requires the model to generate C code progressively after each assembly block. Fine-tuning the model with this objective aids in learning the fine-grained correspondences between assembly and C code.

method leverages the additional context provided by the generated C code and its corresponding assembly code to improve the overall decompilation performance.

3.2 Fine-grained Alignment Enhancement

To enhance the model's ability to perceive the correspondence between assembly code and its corresponding source code, we propose a Fine-grained Alignment Enhancement (FAE) method, which constructs fine-grained alignment data between assembly code and source code derived from debug information. The method primarily includes two training objectives: the End-to-end Decompilation Objective and the Step-by-Step Decompilation Objective.

- **End-to-end Decompilation Objective**: In the end-to-end decompilation task, the model takes assembly code as input and directly generates the corresponding decompiled C code. The goal of this task is to enable the model to translate from assembly language to C language without any intermediate steps. This ob-

jective aims to prevent catastrophic forgetting in the model. During training, we increase the context length and ensure that the same sample is not truncated, which enhances the model's ability to align assembly and C code.

- **Step-by-step Decompilation Objective**: As the Figure 3 shows, in the step-by-step decompilation process, an intermediate form containing both the assembly code and the corresponding C code is first generated, followed by the generation of the complete C code. This step-by-step approach allows the model to better understand the correspondence between assembly code and C code. Training the model with the step-by-step decompilation objective can enhance the model's ability to finely align assembly code with C code. It is important to note that this form is used only during training and not during inference.

During the training process, we employed two different training objectives to enhance the model's decompile performance: an end-to-end decompile objective and a step-by-step decompile objective. They all take assembly code as input, but their output formats are different.

3.2.1 Data Processing

By following the steps below, we can synthesize some fine-grained alignment training data. The synthesized data, as shown in Figure 3, is in a format of interleaved assembly and code output. We expect the model to learn finer-grained alignment between assembly and code from such data.

1. **Compilation**: We employed the "gcc" compiler to compile the selected function code into binary shared libraries at various optimization levels (including O0, O1, O2, and O3). The debug option was enabled during compilation to ensure the inclusion of DWARF debugging information in the compiled output. This debugging information is crucial for subsequent disassembly and parsing steps.

2. **Disassembly**: Subsequently, we used the "objdump" tool to disassemble the compiled binary shared libraries. The "-S" option was utilized to include source code in the disassembly output. Additionally, we used the "-source-comment=;" option to prepend each line of source code with a ";" symbol, facilitating easier parsing in later stages.

3. **Data Parsing**: The disassembly process yielded a mixed output containing both assembly code and source code, organized in an alternating sequence of source code followed by assembly code. We parsed the output to extract the corresponding assembly and source code for each function, ensuring accurate alignment between the two.

4. **Data Reorganization**: Finally, we reorganized the parsed content into a format where the assembly code precedes the corresponding source code in an alternating sequence.

Through these steps, we constructed a high-quality training dataset that ensures each sample contains comprehensive source code information along with the corresponding assembly code. This dataset will be used to train our model, aiming to enhance its performance in tasks such as code understanding and compilation optimization.

4 Experiments

4.1 Evaluation

In this section, we will introduce the benchmark used in our evaluation process, Decompile-Eval (Tan et al., 2024), which is specifically designed to assess the decompilation capabilities of large language models.

The Decompile-Eval benchmark is adapted from the HumanEval benchmark, which includes 164 problems initially designed for code generation tasks. These problems were translated into the C programming language, and the corresponding assembly code was generated at four optimization levels (O0, O1, O2, and O3). The correctness of the decompilation results was tested using the test cases from HumanEval. The primary metrics of the Decompile-Eval benchmark are as follows:

- **Re-compilability**: This metric evaluates whether the decompiled code produced by our model can be successfully recompiled into executable binary without errors. A high recompilability rate indicates that the decompiled code is syntactically correct and adheres to the constraints of the target language (in this case, C).

- **Re-executability**: 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

6606

is determined using the testing methodology provided by the HumanEval dataset, ensuring a comprehensive evaluation of the logical accuracy of the decompiled code.

In our experiment, we pay more attention to the Re-executability, as it more accurately reflects the overall decompilation capability of the model.

4.2 Training Details

Implementation We utilize Lora (Hu et al., 2022) to fine-tune the llm4decompile-6.7b obtained on Hugging Face (Wolf et al., 2019)., with rank set to 32, alpha set to 64 and target set to all projection layers [1]. The optimizer is AdamW (Loshchilov and Hutter, 2019), with a learning rate of 5e-5. The maximum sequence length is set to 16384, and the learning rate scheduler type is cosine, with a warm-up period of 20 steps. The training process was conducted for one epoch. LlamaFactory (Zheng et al., 2024) and FlashAttention 2 (Dao, 2024) were used for the fine-tuning of the model. All experiments were done with an A100-SXM4-80GB. We use greedy decoding for all experiments.

Training data We selected 10,000 samples from the train_real_compilable subset of Exebench (Armengol-Estapé et al., 2022) to synthesis the training data. 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.

4.3 Baselines

We present several baselines in our experiments and demonstrate the effectiveness of our proposed methods.

4.3.1 Models

- **GPT 4**: One of the most powerful OpenAI models, known for its advanced language understanding and generation capabilities (OpenAI, 2023).
- **deepseek-chat**: A powerful conversational AI model from DeepSeek, excelling in generating coherent and contextually relevant dialogues (DeepSeek-AI, 2024).
- **llm4decompile-6.7b**: The 6.7B open-access decompilation LLM pre-trained on 15 billion

```c
bool func0(int num) {
  if (num <= 1) {
    return false;
  }
  for (int i = 2; i * i <= num; i++) {
    if (num % i == 0) {
      return false;
    }
  }
  return true;
}
```

Figure 4: The example for 1-shot learning. In this example, we have tried to cover common control logic such as if statements, loops, and early returns as much as possible. The example will be compiled with the same optimization level as the target assembly code.

tokens of C source code and the corresponding assembly code (Tan et al., 2024)[2].

- **llm4decompile-6.7b+FAE**: The new model was obtained by applying the Fine-grained Alignment Enhancement method for further fine-tuning based on the llm4decompile-6.7b [2].

4.3.2 Methods

- **vanilla**: Using specific prompts for different models to directly request decompilation.

```
Prompt For GPT-4 and Deepseek Chat

What is the C source code of
↪  the assembly code below:
```asm
<asm>
```
```

```
Prompt For llm4decompile

# This is the assembly code:
<asm>

# What is the source code?
```

- **1-shot**: We write a fixed example as the context (Brown et al., 2020). The example will be compiled with the same optimization level as the target assembly code. The example shown in Figure 4. In this example, we have tried to cover common control logic such as if statements, loops, and early returns as much as possible.

---

[1]q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj

[2]We use the latest version, llm4decompile-6.7b-v1.5.

Method	Re-Compilability					Re-Executability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
GPT-4 (OpenAI, 2023)										
vanilla	76.83	67.68	67.07	57.93	67.38	20.73	13.41	13.41	10.98	14.63
+sc^{2}dec	76.22	67.03	65.24	56.10	66.15	32.93	20.12	19.51	13.41	21.49
1-shot	76.22	78.05	73.78	70.73	74.70	31.71	20.12	19.51	21.34	23.17
+sc^{2}dec	74.39	76.22	71.95	68.90	72.87	40.24	25.61	23.78	23.17	28.20
Deepseek Chat (DeepSeek-AI, 2024)										
vanilla	62.20	51.83	47.56	45.73	51.83	10.37	5.49	4.88	5.49	6.55
+sc^{2}dec	57.93	50.00	46.95	45.12	50.00	13.41	6.71	6.71	7.32	8.54
1-shot	67.68	70.73	64.63	56.71	64.94	13.41	10.37	7.93	7.32	9.76
+sc^{2}dec	65.85	70.12	62.20	53.66	62.96	17.68	11.59	10.37	9.15	12.20
llm4decompile-6.7b (Tan et al., 2024)										
vanilla	92.80	93.05	90.73	**94.02**	92.65	70.24	45.49	40.61	38.05	48.60
+sc^{2}dec	92.68	92.80	90.12	93.41	92.26	71.34	**47.56**	45.98	**41.71**	**51.65**
1-shot	93.05	**93.29**	**90.98**	93.90	**92.80**	70.37	44.88	41.46	37.68	48.60
+sc^{2}dec	93.05	92.68	90.37	93.29	92.35	**71.59**	47.20	**46.34**	41.46	**51.65**
retrieval	**94.02**	88.05	84.51	85.12	87.93	65.73	32.68	38.66	36.22	43.32
llm4decompile-6.7b + FAE										
vanilla	**92.68**	**92.44**	**93.66**	**93.17**	**92.99**	67.80	47.68	45.73	42.32	50.88
+sc^{2}dec	92.07	91.59	91.71	92.32	91.92	**70.24**	**48.54**	**47.56**	**43.29**	**52.41**
1-shot	**92.68**	92.32	**93.66**	92.80	92.87	67.68	47.20	45.61	41.95	50.61
+sc^{2}dec	92.07	91.71	91.46	91.95	91.80	**70.24**	48.05	47.20	43.17	52.16

Table 1: The main results of different methods across four optimization levels (O0, O1, O2, O3), as well as their average scores (AVG). The results in bold represent the optimal performance, while those underlined indicate the second-best performance. More results can be found in the Table 5.

- **retrieval**: We use the retrieval method (BM25 (Lù, 2024)) to get the context. The context is generated by searching for the most similar assembly code in the training set.
- **sc^{2}dec**: The self-constructed context decompilation we described in the Section 3.1.
- **1-shot+sc^{2}dec**: We apply the sc^{2}dec on the 1-shot method. Note that the 1-shot example can only be applied to generate the initial decompilation result in Figure 2.

4.4 Results

In this section, we present and analyze the performance of different models. Our experiments aim to evaluate the impact of various techniques, including one-shot, self-constructed context decompilation, and fine-tuning, on the model's performance. The main results are shown in Table 1.

Performance of the Base Model Through a comparative analysis of the experimental results in Table 1, it is evident that different models exhibit significant differences in re-executability across various optimization levels (O0, O1, O2, O3). The base model, llm4decompile-6.7b, performs best at the O0 optimization level with a score of 70.24%, but its performance gradually declines at other optimization levels, with an average score of 48.60%. Among models not fine-tuned with decompilation tasks, GPT-4 achieved the best performance, with an average re-executability score of 14.64%. Generally, their results on re-compilability metrics are significantly higher than their performance on re-executability metrics.

Context Sensitivity By comparing vanilla, 1-shot, and retrieval methods based on the llm4decompile-6.7b, it is observed that the model's performance can even decline after providing a sample. In contrast, the dynamic samples generated through our self-constructed context method yield an improvement in executability of over 2%

Method	Re-Compilability					Re-Executability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
sc^2dec	92.07	91.59	91.71	92.32	91.92	**70.24**	**48.54**	47.56	43.29	**52.41**
sc^2dec (O0)	**92.68**	**92.20**	91.71	92.32	92.23	**70.24**	48.90	47.68	42.93	51.83
sc^2dec (O1)	**92.68**	91.59	**93.54**	**92.93**	**92.68**	68.41	**48.54**	47.44	43.05	51.86
sc^2dec (O2)	**92.68**	**92.20**	91.71	92.32	92.23	67.80	48.90	47.68	42.93	51.68
sc^2dec (O3)	**92.68**	91.59	92.93	92.32	92.38	67.80	47.07	**47.80**	**43.41**	51.52

Table 2: The results of our method across optimization levels. For cases such as "sc^2dec (O0)", we assume that the optimization level of asm code is entirely unknown and select a specific optimization level to compile the code to construct the context.

```
int func0(int a) {
  for (int i = 0; i * i * i <= abs(a); i++)
    if (i * i * i == abs(a))
      return 1;
  return 0;
}                              Source code
```

```
int func0(int n) {
  int i;
  if (n < 0) n = -n;
  for (i = 1; i * i * i <= n; i++)
    if (i * i * i == n)
      return 1;
  return 0;
}                              Decompile
```

```
int func0(int n) {
  int i;
  if (n < 0) n = -n;
  if (n == 0)
    return 1;
  for (i = 1; i * i * i <= n; i++)
    if (i * i * i == n)
      return 1;
  return 0;
}                              SC^2Dec
```

Figure 5: A case study for sc^2dec, which is based on the llm4decompile-6.7b with FAE. By applying the sc^2dec method, the model detects and fixes the mismatch between the code and assembly.

before and after fine-tuning. This indicates that inappropriate samples may even have a detrimental effect in the decompilation domain, while our method achieves a nearly stable performance enhancement without the need for retrieval.

1-shot is a Strong Baseline As shown in Table 1, utilizing the 1-shot method significantly enhances the model's performance. Specifically, for GPT-4, the model's re-executability performance increased from 14.64% to 23.17%, an approximate improvement of 7%. In the open-source model Deepseek Chat, performance improved from 6.55% to 9.76%, reflecting a 3.21% enhancement. However, in the case of the llm4decompile model, which had already been trained on decompilation tasks, there was no performance improvement. Moreover, in the llm4decompile model fine-tuned with Fine-grained Alignment Enhancement, performance actually decreased from 50.88% to 50.61%, a reduction of approximately 0.27%.

Sc²dec Brings Further Performance Improvement The Table 1 shows that applying sc²dec independently results in performance improvements of 6.86% and 1.99% on GPT-4 and Deepseek Chat, respectively. These improvements are lower than those achieved by the 1-shot method, likely due to the model's low Re-Compilability metric. In other words, a substantial portion of the code obtained through decompilation is un-compilable, which impedes the application of the sc²dec method. Furthermore, when combined with the 1-shot method, sc²dec can further enhance performance, achieving an additional 5% improvement for GPT-4, bringing it to 28.20%, and a 2.44% improvement for Deepseek Chat, bringing it to 12.20%. In the case of llm4decompile-6.7b, due to the models' relatively high Re-Compilability metric, the sc²dec method significantly outperformed the 1-shot method, increasing performance from 48.60% to 51.65% and from 50.88% to 52.41%.

Best Performance with Combined Methods As the Table 1 shows, by combining our Step-by-Step Decompile tasks and context-based decompilation methods, we ultimately achieved approximately a 3.90% performance improvement, surpassing the performance of these methods when applied independently. This demonstrates that fine-tuning and self-construct context methods are orthogonal. Most notably, the combined method of fined-

Model	Re-Compilability					Re-Executability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
w/o tune	92.80	**93.05**	90.73	94.02	92.65	70.24	45.49	40.61	38.05	48.60
FAE (End-to-End & Step-by-Step Decompilation Objective)										
vanilla	92.68	92.44	**93.66**	**93.17**	**92.99**	67.80	47.68	45.73	42.32	50.88
+sc^{2}dec	92.07	91.59	91.71	92.32	91.92	70.24	**48.54**	**47.56**	**43.29**	**52.41**
1-shot	92.68	92.32	**93.66**	92.80	92.87	67.68	47.20	45.61	41.95	50.61
+sc^{2}dec	92.07	91.71	91.46	91.95	91.80	70.24	48.05	47.20	43.17	52.16
End-to-End Decompilation Objective										
vanilla	**93.9**	90.61	90.61	91.34	91.62	69.51	43.29	42.2	40.73	48.93
+sc^{2}dec	92.8	90.24	90.00	90.12	90.79	**71.34**	46.10	43.78	42.44	50.91
1-shot	**93.9**	90.73	90.37	91.22	91.55	69.51	44.02	42.68	40.61	49.21
+sc^{2}dec	92.8	90.24	89.76	90.00	90.7	71.22	46.46	44.27	42.56	51.13

Table 3: The results of ablation study. All results are based on the llm4decompile-6.7b model.

grained alignment enhancement and self-construct context decompilation achieves the highest scores across all optimization levels expect "O0", with scores of 48.54%, 47.56%, and 43.29% at the O1, O2, and O3 levels, respectively, and an overall average score of 52.41%.

4.5 Analysis

In this section, we will further analyze the experimental results to study the effectiveness and sensitivity of our method:

Mismatched Optimization Levels Leads to Performance Degradation As the Table 2 shows, the performance can degrade by 1% to 2% overall when constructing contexts that mismatch the optimization levels of the target assembly code. However, specific optimization levels may exhibit different behaviors. For instance, using a fixed O3 optimization level to construct contexts while decompiling assembly optimized with O2 does not result in significant performance degradation and even better. ***In practical scenarios, using O3 to construct contexts for binaries with unknown optimization levels might be a reasonable choice***.

Case study of sc^{2}dec As the Figure 5 shows, the compiler has expanded the abs function, reducing the overhead of function calls. The model initially generated a piece of compilable but incorrect code during direct decompilation, omitting the case where the variable n equals 0. We compiled and disassembled this code to construct a new sample. By applying the sc^{2}dec method with the context, the model successfully identified the missing part

in the short segment of assembly code and generated the correct code, thus rectifying the error from the previous inference.

Contribution of FAE The results in Table 1 show that the application of Fine-grained Alignment Enhancement (FAE) for further fine-tuning the model significantly enhanced its re-executability performance across all optimization levels except "O0". Notably, at optimization levels O2 and O3, the scores improved by 5.12% and 4.27%, reaching 45.73% and 42.32%, respectively. The average performance increased by 2.28%, achieving 50.88%. This demonstrates the effectiveness of the Fine-grained Alignment Enhancement method.

Ablation Study of FAE The results of ablation experiments in Table 3 have demonstrated the effectiveness of our proposed Step-by-Step Decompile training objective. Removing the training objective results in approximately a 2% decline in Re-Executability performance, regardless of whether the sc^{2}dec method is used, and notably, a 3.78% decline at the "O2" optimization level. However, the performance of the 1-shot remains unaffected, supporting our hypothesis that the mismatch between the training data form and the inference form leads to performance degradation. Furthermore, retaining only the End-to-end Decompilation training objective still yields little performance improvement, about 0.33%. This improvement can be attributed to the longer context provided during training, as the samples are not truncated during training, allowing the model to fully observe the assembly code and its corresponding C code.

5 Conclusion

In this paper, we introduce two methods based on the characteristics of the decompilation task: Self-Constructed Context Decompilation, which leverages the compilability of decompilation results to construct better contexts, and Fine-grained Alignment Enhancement, which fine-tunes the model using fine-grained alignment data between assembly code and source code derived from debug information. By integrating these methods, we achieved a 3.90% improvement in decompilation performance, reaching a new state-of-the-art level of 52.41%.

Acknowledge

Limitations

The sc^2dec depends on the decompiled code generated by the model. If the decompiled code cannot be recompiled, sc^2dec will not be able to benefit from it, necessitating that the decompiled code has high re-compilability performance. For Fine-grained Alignment Enhancement, we created a Step-by-Step Decompile training objective, training with only 10,000 samples across four optimization levels (O0-O3), without validating its performance on larger datasets or bigger models. Moreover, fine-tuning led to a further decline in performance in the 1-shot scenario.

Potential risks

Decompilation is a technique that converts compiled binary code back into human-readable source code. While decompilation can be legal and useful in certain contexts, it also entails several potential risks. The primary potential risks are as follows:

- Intellectual Property Infringement: Decompilation may violate the copyright and licensing agreements of software. Unauthorized decompilation can lead to copyright infringement, thereby instigating legal disputes.

- Security Risks: Decompiled code may expose the internal structure and implementation details of the software, providing attack vectors for hackers. Malicious actors can exploit this information to identify and exploit vulnerabilities in the software.

- Ethical Concerns: Decompiling and analyzing another's code can be considered unethical, especially when done without authorization.

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.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

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.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

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, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

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

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

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova$^+$: Generative language models for binaries. *Preprint*, arXiv:2311.13721.

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.

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.

Thomas Lippincott. 2020. Starcoder: A general neural ensemble technique to support traditional scholarship, illustrated with a study of the post-atlantic slave trade. In *15th Annual International Conference of the Alliance of Digital Humanities Organizations, DH 2020, Ottawa, Canada, July 20-25, 2020, Conference Abstracts*.

Zhibo Liu and Shuai Wang. 2020. 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.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Xing Han Lù. 2024. Bm25s: Orders of magnitude faster lexical search via eager sparse scoring. *Preprint*, arXiv:2407.03618.

OpenAI. 2022. ChatGPT.

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.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *Preprint*, arXiv:2403.05286.

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.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

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.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 39–51. ACM.

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   Training examples

```
User: # This is the assembly:
<func0>:
mov   %edi,%eax
and   $0x1,%eax
ret
# What is the source code?

Assistant: bool func0(int num) {
  return num % 2 != 0;
}
```

Figure 6: The training example of end-to-end decompile objective.

```
User: Add inline comments that contain the corresponding C code for the
assembly code first, then write the equivalent C function.
```asm
<func0>:
mov   %edi,%eax
and   $0x1,%eax
ret
```

Assistant: Here is the assembly code with inline comments containing the
corresponding C code:
```asm
<func0>:
; bool func0(int num) {
mov   %edi,%eax
and   $0x1,%eax
; return num % 2 != 0;
ret
; }

And here is the equivalent C function:
```c
bool func0(int num) {
  return num % 2 != 0;
}
```

Figure 7: The training example of step-by-step decompile objective. Note that the workflow is only for training.

B   Extra Results

The Model is More Sensitive to Compiler Series than to Versions: As the Table 4 shows, using a compiler different from the one used during training to build the context can lead to significant performance degradation. The model performs better when the context is built using the GCC compiler series, whereas the performance is slightly worse when using the clang series. This may be due to the fact that the context used during both the continued pre-training and fine-tuning stages was constructed with the GCC compiler series. When different versions of GCC are used for construction, the performance of sc^{2}dec stabilizes around 52% for the fine-tuned model except "x86-64 gcc 14.1". In contrast, when the context is built using the clang series, the performance of sc^{2}dec is slightly lower and more consistent, stabilizing around 51% for the fine-tuned model.

Compiler	Re-Compilability					Re-Executability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
x86-64 gcc 10.5	92.07	90.98	91.71	92.32	91.77	70.85	47.32	47.80	43.05	52.26
x86-64 gcc 11.4	92.07	91.59	92.32	92.32	92.07	70.85	48.66	47.68	42.68	52.47
x86-64 gcc 12.3	92.07	91.59	92.32	92.32	92.07	70.85	47.68	46.95	42.56	52.01
x86-64 gcc 13.3	92.07	91.59	92.32	92.32	92.07	70.85	47.93	46.59	42.68	52.01
x86-64 gcc 14.1	92.07	91.83	92.44	92.56	92.23	70.85	47.07	45.73	41.95	51.40
x86-64 clang 14.0.0	92.68	92.44	93.66	93.17	92.99	69.02	48.05	45.73	42.44	51.31
x86-64 clang 15.0.0	92.68	92.44	93.66	93.17	92.99	69.02	48.29	45.73	42.32	51.34
x86-64 clang 16.0.0	92.68	92.2	93.66	93.17	92.93	69.02	48.05	46.46	42.32	51.46
x86-64 clang 17.0.1	92.68	92.44	93.66	93.17	92.99	69.02	48.05	46.71	42.32	51.52
x86-64 clang 18.1.0	92.68	92.44	93.66	93.17	92.99	69.02	48.17	46.46	42.32	51.49

Table 4: The re-executability of differnet models across different compilers. We selected the popular three versions of both the clang and gcc series in Compiler Explorer. The compiler of Compiler Explorer may differ from the compilers provided by Ubuntu in some default compilation options, which might result in slight differences in the generated assembly code.

Method	Re-Compilability					Re-Executability				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
GPT-4 (OpenAI, 2023)										
vanilla	76.83	67.68	67.07	57.93	67.38	20.73	13.41	13.41	10.98	14.63
+sc^{2}dec	76.22	67.03	65.24	56.10	66.15	32.93	20.12	19.51	13.41	21.49
1-shot	76.22	78.05	73.78	70.73	74.70	31.71	20.12	19.51	21.34	23.17
+sc^{2}dec	74.39	76.22	71.95	68.90	72.87	40.24	25.61	23.78	23.17	28.20
GPT-3.5 Turbo (OpenAI, 2022)										
vanilla	20.73	25.00	29.27	21.95	24.24	4.27	3.66	3.05	3.05	3.51
+sc^{2}dec	18.29	22.56	28.66	20.12	22.41	5.49	3.66	3.66	4.27	4.27
1-shot	49.39	46.34	42.07	40.85	44.66	10.37	7.32	6.10	5.49	7.31
+sc^{2}dec	42.07	42.68	36.59	34.15	38.87	12.20	9.15	6.71	7.93	8.99
Deepseek Chat (DeepSeek-AI, 2024)										
vanilla	62.20	51.83	47.56	45.73	51.83	10.37	5.49	4.88	5.49	6.55
+sc^{2}dec	57.93	50.00	46.95	45.12	50.00	13.41	6.71	6.71	7.32	8.54
1-shot	67.68	70.73	64.63	56.71	64.94	13.41	10.37	7.93	7.32	9.76
+sc^{2}dec	65.85	70.12	62.20	53.66	62.96	17.68	11.59	10.37	9.15	12.20
Deepseek Coder (Guo et al., 2024)										
vanilla	66.46	57.93	47.56	51.83	55.95	9.15	6.10	6.71	6.98	7.01
+sc^{2}dec	62.80	54.27	43.29	50.00	52.59	12.20	7.93	9.15	8.54	9.45
1-shot	57.93	53.66	52.44	48.17	53.05	11.59	10.37	8.54	7.93	9.60
+sc^{2}dec	56.10	51.83	51.22	46.34	51.37	15.24	11.58	10.37	8.54	11.43
llm4decompile-6.7b (Tan et al., 2024)										
vanilla	92.80	93.05	90.73	94.02	92.65	70.24	45.49	40.61	38.05	48.60
+sc^{2}dec	92.68	92.80	90.12	93.41	92.26	71.34	47.56	45.98	41.71	51.65
1-shot	93.05	93.29	90.98	93.90	92.80	70.37	44.88	41.46	37.68	48.60
+sc^{2}dec	93.05	92.68	90.37	93.29	92.35	71.59	47.20	46.34	41.46	51.65
llm4decompile-6.7b + FAE										
vanilla	92.68	92.44	93.66	93.17	92.99	67.80	47.68	45.73	42.32	50.88
+sc^{2}dec	92.07	91.59	91.71	92.32	91.92	70.24	48.54	47.56	43.29	52.41
1-shot	92.68	92.32	93.66	92.80	92.87	67.68	47.20	45.61	41.95	50.61
+sc^{2}dec	92.07	91.71	91.46	91.95	91.80	70.24	48.05	47.20	43.17	52.16

Table 5: The extra results of different methods across four optimization levels (O0, O1, O2, O3), as well as their average scores (AVG). The results in bold represent the optimal performance, while those underlined indicate the second-best performance.