Poster: Enhancing LLM4Decompile with Obfuscated Source Code for Optimized Assembly

Ryota Saito*, Takuya Kataiwa*, Tetsushi Ohki*†
*Shizuoka University, †RIKEN AIP
Email: saito.ryota.23@shizuoka.ac.jp

***Abstract*—Decompilation is a critical technique in cybersecurity contexts. Recent machine-learning-based approaches have improved accuracy and readability but struggle with optimized inputs. We propose a novel method to enhance LLM4Decompile, an end-to-end Large Language Model-based decompiler, by leveraging obfuscation techniques for training dataset creation. Experiments demonstrate the effectiveness and limitations of our method in improving decompilation accuracy.**

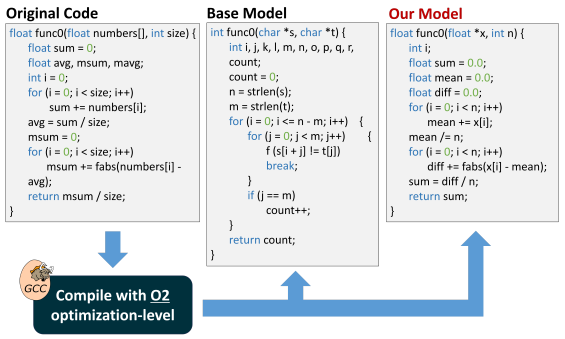
I. Introduction

Decompilation is a fundamental technique used in cybersecurity contexts, including malware analysis and vulnerability research. Existing decompilers widely used in industry, such as Ghidra and IDA Pro, rely on rule-based approaches to carry out the decompilation process. However, rule-based decompilers face two major challenges: (1) the accuracy of decompilation and (2) the readability of the generated code.

The compilation process leads to the loss of critical information from the source code, such as control structures and symbol names, making accurate decompilation a challenging task. If decompiled code is functionally accurate, readability is often poor in many cases due to the inability of recovering lost information such as symbol names, types, etc. In recent years, machine-learning-based decompilers have emerged as promising alternatives. Some models have achieved remarkably high accuracy in decompilation tasks. Furthermore, with regard to readability, they have demonstrated the ability to address NP-hard problems, such as symbol name inference, further advancing the capabilities of decompilers.

Among existing approaches, LLM4Decompile [1] stands out, particularly its variant LLM4Decompile-End, which is the first to replace an entire decompiler workflow with a Large Language Model (LLM). Despite some successes, it still struggles to handle inputs that are optimized by compilers, making this a critical area for further improvement. To address these challenges, we propose a novel method that incorporates obfuscation techniques to enhance the decompilation accuracy of LLM4Decompile. By generating obfuscated training datasets, our approach improves the model's ability to interpret optimized assembly code and mitigates existing limitations.

Our contributions can be summarized as follows: (1) We propose a systematic method for generating obfuscated training datasets, reducing manual overhead while enhancing data diversity. (2) We fine-tune LLM4Decompile-End using the constructed dataset, achieving significant accuracy improvements in optimized assembly code. Fig. 1 is a example of the outpus from base and our model. (3) We identify and analyze key limitations of the proposed approach, offering insights for future enhancements in symbol resolution.



```
float func0(float numbers[], int size) {
    float sum = 0;
    float avg, msum, mavg;
    int i = 0;
    for (i = 0; i < size; i++)
        sum += numbers[i];
    avg = sum / size;
    msum = 0;
    for (i = 0; i < size; i++)
        msum += fabs(numbers[i] -
    avg);
    return msum / size;
}
```

```
int func0(char *s, char *t) {
    int i, j, k, l, m, n, o, p, q, r,
    count;
    count = 0;
    n = strlen(s);
    m = strlen(t);
    for (i = 0; i <= n - m; i++)   {
        for (j = 0; j < m; j++)     {
            f (s[i + j] != t[j])
            break;
        }
        if (j == m)
            count++;
    }
    return count;
}
```

```
float func0(float *x, int n) {
    int i;
    float sum = 0.0;
    float mean = 0.0;
    float diff = 0.0;
    for (i = 0; i < n; i++)
        mean += x[i];
    mean /= n;
    for (i = 0; i < n; i++)
        diff += fabs(x[i] - mean);
    sum = diff / n;
    return sum;
}
```

Fig. 1 Example Output of Base and Enhanced Models: Comparison of decompiled outputs from the base model and our model, highlighting improved symbol resolution and control flow reconstruction.

II. Proposed Method

This section presents a method to enhance the decompilation accuracy of the LLM4Decompile-End model [1]. Our approach consists of two key components: **Dataset Construction** and **Fine-Tuning**, each tailored to address the limitations of handling optimized inputs.

A. Dataset Construction

We use Code-to-Code obfuscation to enhance the diversity of training data without requiring new source code. This approach eliminates the need for manual dataset creation, thereby reducing the overall cost of data preparation.

(1) Source Code Collection: We utilize the AnghaBench dataset [2], which consists of functions written in C collected from various open-source projects. As one of the largest datasets available, it is widely used for its diversity and reliability. This dataset is used to produce LLM4Decompile-End [1].

(2) Code Obfuscation: This process generates obfuscated C functions based on predefined obfuscation techniques. Using Tigress [3], a Code-to-Code obfuscation tool, we obfuscate the functions in the AnghaBench dataset. Tigress supports multiple obfuscation methods that are relevant to real-world scenarios. It is suitable obfuscator for generating diverse functions simulate obfuscated or optimized code in this process.

Model	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37
Our Model	68.29	**43.29**	**44.51**	**40.85**	49.23

TABLE I Re-executability rate on the HumanEval benchmark: O0 ˜O3 represent inputs with different levels of compiler optimization.

Model	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	19.00	17.69	17.69	17.54	17.98
Our Model	**17.18**	**14.89**	**14.06**	**14.14**	**15.07**

TABLE II Re-executability rate on the ExeBench benchmark

(3) Dataset Optimization: Obfuscated C functions produced by Tigress often include excessive comments and expanded `#include` statements, increasing file size. We use a Python script to remove unnecessary comments and compress `#include` statements, reducing dataset size and improving usability.
(4) Compilation to Assembly Code: We compile the optimized dataset using GCC with four optimization levels (`O0` to `O3`). Each function in the dataset is compiled individually, producing and paired with four assembly codes per function.

B. Fine-Tuning

Using the constructed dataset, fine-tuning is performed on the LLM4Decompile-End model. The fine-tuning process allows the model to better generalize to optimized assembly code, enhancing its ability to decompile accurately across various optimization levels.

III. Evaluation and Results

We evaluate the proposed method by fine-tuning the `LLM4Decompile-End-6.7b` model and assessing its performance on open datasets. Our primary goals is to validate the effectiveness of obfuscation-based dataset augmentation in improving decompilatioin accuracy for optimized code.
Training data: As described in Section II, the training data was constructed by obfuscating the AnghaBench dataset by Tigress. We randomly selected 2% of the files from the AnghaBench and used them as the training dataset.
Metrics and Evaluation Datasets: We use two widely recognized datasets, HumanEval [4] and ExeBench [5] to evaluate the model. Both datasets provide pairs of source code and test cases. We adopt re-executability rate as an evaluation metric. Re-executability rate refers to the ability of the generated code to compile successfully and pass provided test cases, ensuring both syntactic and functional correctness.
Results: Results are shown in Table I and II. Re-executability rate, especially for optimized inputs got improved in Humaneval benchmark, on the contrary, accuracy decreased for all optimization level. Figure 1 illustrates decompiled code examples (ours and bases).

IV. Discussions and Future Plans

The improvement observed in HumanEval benchmark indicate that obfuscated datasets effectively emulate compiler optimization. This emulation enhances the model's ability to generalize to optimized assembly code, as evidenced by an 3.86% average. The improvements demonstrate the effectiveness of obfuscated datasets for inference on optimized inputs, which is crucial for practical applications.

In contrast, performance on ExeBench benchmark declined 2.91% average. A key distinction between the two datasets is the presence of external symbols. While HumanEval does not include external symbols, ExeBench frequently does. The performance degradation can be attributed to issues with inferring external static symbols. LLMs tend to generate context-dependent symbol names, which may lead to unresolved symbol errors during compilation. We conducted preliminary experiments to count up the each kind of error messages and confirmed that the fine-tuned model exhibited more compilation errors due to unresolved symbols than the base model. This indicates that the issue stems from the architecture of LLM4Decompile rather than the training data.

The findings from this study provide important implications for the development of LLM-based decompilers. The proposed method demonstrates that code obfuscation can emulate compiler optimizations, improving LLMs' ability to generalize to optimized assembly code, as shown in the HumanEval benchmark. This highlights the potential of obfuscated datasets to bridge the gap between non-optimized and optimized binaries. Future work will address challenges such as handling external symbols using techniques like Retrieval-Augmented Generation (RAG) and investigate the impact of various obfuscation methods on model performance to identify the most effective strategies.

V. Summary

In this study, we proposed a method to improve the accuracy of the LLM4Decompile by performing additional fine-tuning using obfuscated code. Experimental results demonstrated both the effectiveness and limitation of the proposed method. This study contributes to advancing practical, end-to-end decompilation by leveraging obfuscation as a form of data augumentation. For future work, we aim to explore solutions such as RAG for better symbol resolution and to evaluate the impact of different obfuscation techniques on model performance.

VI. Acknowledgment

This study was supported in part by JST Moonshot JPMJMS2215.

References

[1] H. Tan *et al.*, "LLM4Decompile: Decompiling binary code with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 3473–3487.
[2] A. F. da Silva *et al.*, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 378–390.
[3] C. Collberg, "The Tigress C obfuscator," https://tigress.wtf, Accessed: 2025-01-06.
[4] M. chen *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374
[5] Armengol-Estapé *et al.*, "ExeBench: An ML-Scale Dataset of Executable C Functions." New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3520312.3534867

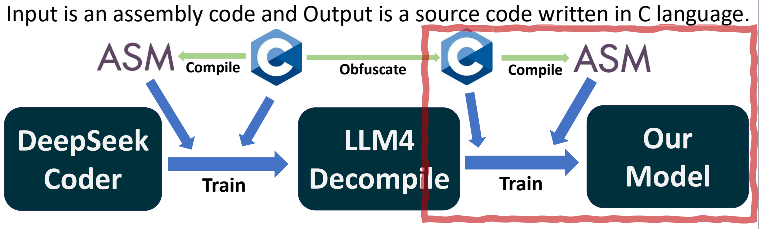
Poster: Enhancing LLM4Decompile with Obfuscated Source Code for Optimized Assembly

Ryota Saito[1] Takuya Kataiwa[1] Tetsushi Ohki[1 2]

[1] Shizuoka University [2] RIKEN AIP

Introduction and Motivation

LLM4Decompile is an End-to-End Large Language Model-based Decompiler. Input is an assembly code and Output is a source code written in C language.



LLM4Decompile has been struggling to handle **optimized input**[1] as Table 1. describes, however in real-world scenarios, input is often optimized.

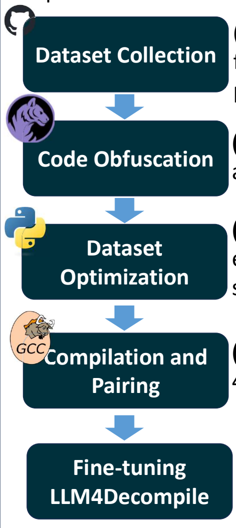
Model/Benchmark	O0	O1	O2	O3	AVG
LLM4Decompile-End-1.3B	47.20	20.61	21.22	20.24	27.32
LLM4Decompile-End-6.7B	**68.05**	**39.51**	**36.71**	**37.20**	**45.37**
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54

Table 1. Performance comparison on HumanEval-Decompile benchmarks.

To address this challenge, we leverage **obfuscation techniques** for training dataset creation.

Proposed method

We present a method to enhance the accuracy of LLM4Decompile-End. Proposed method are described as following steps:



(1) Utilize **AnghaBench**, one of the largest dataset formed by C functions collected from open-source projects.

(2) Obfuscate collected source code by **Tigress**, a Code-to-Code obfuscation tool.

(3) Remove unnecessary comments and compress expanded #include statements to reduce dataset size.

(4) Compile each obfuscated function by GCC with 4 optimization level (O0 ~ O3) to get assembly code.

(5) Fine-tune **LLM4Decompile** with the pairs of obfuscated source code and assembly code.

Metrix and Evaluation Dataset

Re-executability:

- Model is evaluated by re-executability which is generated code is **compilable and executable as functionally correct.**

Evaluation Dataset:

- Two datasets are used for evaluation.
- **Human-Eval** and **ExeBench**.

Human-Eval example

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int func0(float numbers[], int size, float threshold) {
  int i, j;

  for (i = 0; i < size; i++) {
    for (j = i + 1; j < size; j++) {
      if (fabs(numbers[i] - numbers[j]) < threshold) {
        return 1;
      }
    }
  }
  return 0;
}
```

ExeBench example

```
typedef struct TYPE_2__ TYPE_1__;
struct TYPE_2__ {
     scalar_t__ InterruptVector;
     scalar_t__ MIE;
     scalar_t__ PointerBits;
     scalar_t__ SCC_Interrupt_Type;
};
TYPE_1__ SCC;
size_t Wire_SCCInterruptRequest;
size_t Wire_VIA1_iA7_SCCwaitrq;
int* Wires;
void SCC_Reset(void) {
     Wires[Wire_VIA1_iA7_SCCwaitrq] = 1;
     SCC.SCC_Interrupt_Type = 0;
     Wires[Wire_SCCInterruptRequest] = 0;
     SCC.PointerBits = 0;
     SCC.MIE = 0;
     SCC.InterruptVector = 0;
     SCC_InitChannel(1);
     SCC_ResetChannel(0);
}
```

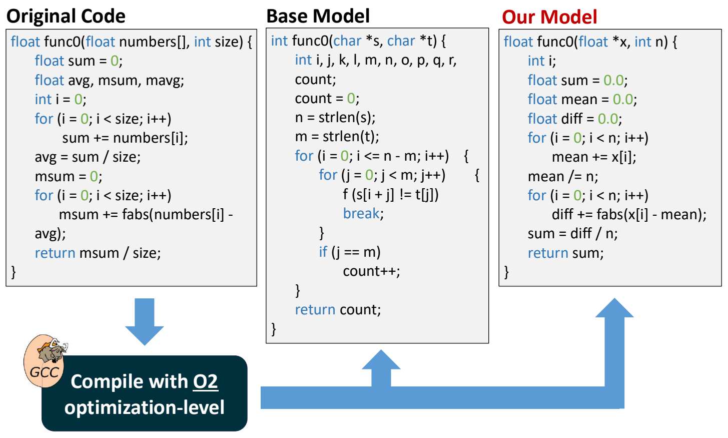
Evaluation Discussion

For Human-Eval, the model got improved, particularly for **optimized inputs** as Table 2. describes.
This suggests that obfuscation may trace the effects of optimization.

Model	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37
Our Model	68.29	43.29	44.51	40.85	49.23

Table 2. Re-executability rate on the HumanEval benchmark.

Example output below demonstrate the ability of our model to restore code.



Original Code

```
float func0(float numbers[], int size) {
     float sum = 0;
     float avg, msum, mavg;
     int i = 0;
     for (i = 0; i < size; i++)
          sum += numbers[i];
     avg = sum / size;
     msum = 0;
     for (i = 0; i < size; i++)
          msum += fabs(numbers[i] -
     avg);
     return msum / size;
}
```

Base Model

```
int func0(char *s, char *t) {
     int i, j, k, l, m, n, o, p, q, r,
     count;
     count = 0;
     n = strlen(s);
     m = strlen(t);
     for (i = 0; i <= n - m; i++)   {
          for (j = 0; j < m; j++)      {
               f (s[i + j] != t[j])
               break;
          }
          if (j == m)
               count++;
     }
     return count;
}
```

Our Model

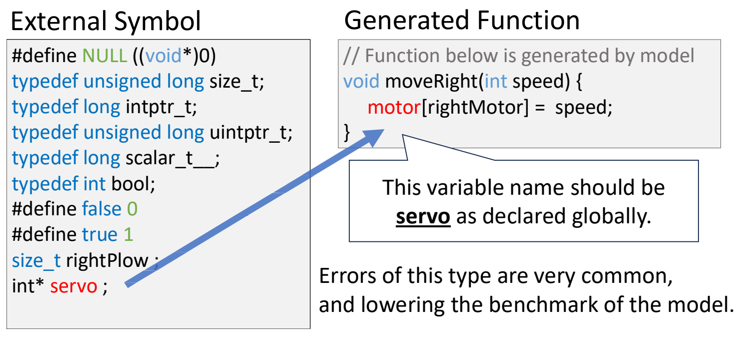
```
float func0(float *x, int n) {
     int i;
     float sum = 0.0;
     float mean = 0.0;
     float diff = 0.0;
     for (i = 0; i < n; i++)
          mean += x[i];
     mean /= n;
     for (i = 0; i < n; i++)
          diff += fabs(x[i] - mean);
     sum = diff / n;
     return sum;
}
```

For ExeBench, the overall accuracy **declined** as Table 3. describes.
This is likely caused by the LLM's context-based inference for **external static symbols**, which are not explicitly provided.
The results indicate that additional fine-tuning does not resolve this issue and may even exacerbate the problem.

Model	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	**19.00**	**17.69**	**17.69**	**17.54**	**17.98**
Our Model	17.18	14.89	14.06	14.14	15.07

Table 3. Re-executability rate on the ExeBench benchmark.

Example of the output below demonstrate symbol resolving issue.



External Symbol

```
#define NULL ((void*)0)
typedef unsigned long size_t;
typedef long intptr_t;
typedef unsigned long uintptr_t;
typedef long scalar_t__;
typedef int bool;
#define false 0
#define true 1
size_t rightPlow ;
int* servo ;
```

Generated Function

```
// Function below is generated by model
void moveRight(int speed) {
   motor[rightMotor] =  speed;
}
```

Errors of this type are very common, and lowering the benchmark of the model.

Future Work

- Developing mechanisms, such as Retrieval-Augmented Generation (RAG), to **better convey information** like relations between symbol names and addresses for improving LLMs for End-to-End Decompilation.
- Additional experiments will investigate how **different obfuscation techniques** impact model performance, aiming to identify methods that enhance decompilation accuracy.

Reference

[1]: H. Tan et al., "LLM4Decompile: Decompiling binary code withlarge language models," in Proceedings of the 2024 Conference onEmpirical Methods in Natural Language Processing, Y. Al-Onaizan,M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Associationfor Computational Linguistics, Nov. 2024, pp. 3473–3487. [Online].Available: https://aclanthology.org/2024.emnlp-main.203/

This study was supported in part by JST Moonshot JPMJMS2215.