DeGPT: Optimizing Decompiler Output with LLM

Peiwei Hu[1,2], Ruigang Liang[1,2], and Kai Chen[1,2,⋆]
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
{hupeiwei,liangruigang,chenkai}@iie.ac.cn

***Abstract*—Reverse engineering is essential in malware analysis, vulnerability discovery, etc. Decompilers assist the reverse engineers by lifting the assembly to the high-level programming language, which highly boosts binary comprehension. However, decompilers suffer from problems such as meaningless variable names, redundant variables, and lacking comments describing the purpose of the code. Previous studies have shown promising performance in refining the decompiler output by training the models with huge datasets containing various decompiler outputs. However, even datasets that take much time to construct cover limited binaries in the real world. The performance degrades severely facing the binary migration.**

In this paper, we present DeGPT, an end-to-end framework aiming to optimize the decompiler output to improve its readability and simplicity and further assist the reverse engineers in understanding the binaries better. The Large Language Model (LLM) can mitigate performance degradation with its extraordinary ability endowed by large model size and training set containing rich multi-modal data. However, its potential is difficult to unlock through one-shot use. Thus, we propose the three-role mechanism, which includes referee (R_ref), advisor (R_adv), and operator (R_ope), to adapt the LLM to our optimization tasks. Specifically, R_ref provides the optimization scheme for the target decompiler output, while R_adv gives the rectification measures based on the scheme, and R_ope inspects whether the optimization changes the original function semantics and concludes the final verdict about whether to accept the optimizations. We evaluate DeGPT on the datasets containing decompiler outputs of various software, such as the practical command line tools, malware, a library for audio processing, and implementations of algorithms. The experimental results show that even on the output of the current top-level decompiler (Ghidra), DeGPT can achieve 24.4% reduction in the cognitive burden of understanding the decompiler outputs and provide comments of which 62.9% can provide practical semantics for the reverse engineers to help the understanding of binaries. Our user surveys also show that the optimizations can significantly simplify the code and add helpful semantic information (variable names and comments), facilitating a quick and accurate understanding of the binary.

I. Introduction

Reverse engineering, aiming to obtain the program logic and algorithms of target software through reverse analysis, has received increasing attention for its role as an enabler in vulnerability discovery [28, 33, 43, 48, 53, 62, 67], malware analysis [22, 26, 39, 60], closed-source comprehension [20, 31, 38, 50], etc. For example, security researchers leverage disassemblers to obtain the assembly code of malware samples that do not have available source code for further analysis. However, as the mnemonic, assembly code is closer to machine code than high-level Program Language (PL), which is more conducive to human understanding. Decompilers, such as Hex-Rays [13] and Ghidra [11], appear to convert the low-level assembly language to high-level PL, which is more conducive to human understanding.

State-of-the-art decompilers can recover the control flow structures and guess the possible primitive variable types based on the memory layout. However, since the binary does not contain semantic[1] information such as variable names and comments, the decompiler output lacks effective semantic information recovery. The debug information can provide some help. Unfortunately, since debug information is unnecessary in software execution, it is typically removed before the software is released for space-saving or security protection. Besides lacking the semantic information that boosts the program comprehension, the decompiler output also suffers from other drawbacks, as discussed in Section II.

Although the semantic information is dropped during the compilation, several work has attempted to help security researchers better understand reversed binaries by reconstructing semantic information to refine readability. Specifically, the previous studies can be divided into two categories, depending on the objects imposed by the code analysis. (i) Embed information obtained from disassembled code such as mnemonic sequence and control flow graph for model-based methods [27, 29, 32, 41, 45]. The routine is to calculate the embedding of the information from the assembly function with the help of various program analyses and send the embedding to the prediction model, which usually belongs to encoder-decoder architecture. (ii) Facilitate reverse engineering by enhancing the output of the decompiler rather than the disassembler [25, 40, 44]. This is because the output of the decompiler can provide additional information that facilitates semantic recovery compared to the disassembler.

However, the previous studies suffer from the following limitations. **L1:** *The decrease in performance in the presence of unseen binaries.* Limited by the training set size and model capabilities, previous studies face performance degradation when the binary for testing comes from software that has

⋆ Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24401
www.ndss-symposium.org

[1]We use *semantic* in two categories of meaning. When it is used for "semantic information", it refers to the semantics of information like comments and variable names. When it is used for "function/code semantics", it refers to the behaviors of the function/code.

never been seen before. As discussed in NFRE [29], the size of datasets can heavily impact the performance of optimization. Previous frameworks are trained on small-scale datasets compared with the current fast-growing large models. Besides, the prediction models used in the previous studies are also not big enough compared with the recent large models, which limits the efficacy. **L2:** *Limited decompiler output optimizations.* Previous studies concentrate on augmenting the decompiler output's variable names and types. Considering the situations of realistic reverse engineering, proper comments describing the purposes of the code snippets and easy-to-read code style will also boost the experience of analyzing binaries. Moreover, previous methods tend to handle only one or two optimization types. We expect to use one framework for diverse optimization instead of sundry plugins for different improvements.

Due to its potential in general artificial intelligence, *Large Language Model* (LLM) has become a recent technology hotspot with the strength of supporting multiple downstream tasks, which means it can also optimize the decompiler output with multiple forms of optimization. Moreover, as its name implies, the LLM is large from two aspects including the datasets used for training and the model size (i.e. the number of parameters) compared with the models involved in the previous studies, which endow the ability to cope with the performance degradation caused by binary migration while adopting the LLM to optimize the decompiler output. Unfortunately, there are still more challenges when adopting the LLM for optimizing decompiler output, as discussed in the following.

C1: *The one-shot use of LLM for optimizing the decompiler output leads to limited performance.* The most intuitive way for optimizing decompiler output is to provide a direct prompt to the LLM, such as "Augment the following decompiler output to improve the readability", and collect the response. However, according to our experiments in Section V-E, this highly limits what the LLM can do. The recommended method of leveraging the LLM is to split the tasks into different pieces and guide it step by step to maximize its potential for enhancing decompiler output [63].

C2: *The LLM may give the wrong response for the optimization task.* The uncertainty of LLM's behavior may compromise the fidelity[2] of the decompiler output. LLMs such as ChatGPT [4] are generative models, and their responses are not always guaranteed to be correct. Moreover, the ambiguity and vagueness of natural language, which is how we interact with the LLM, also hinder the LLM from understanding and correctly conducting our prompts. Therefore, the LLM may incorrectly manipulate the decompiler output and compromise the function semantics of the decompiler output, which is undesirable and will confuse the reverse engineers.

DeGPT. In this paper, we present a novel framework for lifting decompiler output called DeGPT, which supports various optimizations including structure simplification, variable renaming, and appending comments to improve the readability of the decompiler output. DeGPT assists the reverse engineers in better understanding and analyzing executable programs. DeGPT is a code readability enhancement tool between the decompiler output and the reverse engineers. It accepts the decompiler output and sends the corresponding output to the reverse engineers after readability enhancement that LLM and lightweight program analysis drive. In particular, we address the aforementioned challenges based on the following insights. Firstly, we found that the decompiler output optimization can be divided into three steps to maximize the potential of the LLM. To implement these steps, we propose a three-role mechanism including referee (R_ref), advisor (R_adv), and operator (R_ope) to maximize the LLM optimizing capability of the decompiler output (for C1). Specifically, we define the R_ref to provide the optimization scheme, R_adv to accept the R_ref's scheme and present specific rectification measures for the decompiler output to achieve the optimization scheme, and R_ope to censor the measures from the R_adv and give the final verdict to adopt those that do not affect the original function semantics to the decompiler output. The R_ope is designed to uphold the correctness of the response from LLM. We split the optimization into several tasks and guide the LLM step by step through the three-role mechanism. According to our experiment results in Section V-E, which is conducted on the decompiler outputs of the current top-level decompiler (Ghidra), the three-role mechanism can boost the DeGPT perform 2.5 times more structural simplification optimization, 3.1 times more variable renaming optimization compared to the one-shot experiment. The appending comments optimization also covers more test cases in the datasets.

Further, we observe that the symbol value changes in each execution path can reflect the function semantics. Since changes in control flow and data flow will affect the symbol values in some execution paths, we leverage this to check whether the response from LLM changes the original function semantics, thus upholding the correctness of the LLM response. Based on the above observation, we propose *Micro Snippet Semantic Calculation* (MSSC for short), which is a program analysis method and placed as a component of R_ope, to assess the symbol value changes of a code snippet (for C2). Concretely, MSSC checks whether the modified code snippet follows the same symbolic value change on some non-local symbols compared to the pre-modified snippet. DeGPT leverages MSSC to filter out code changes that compromise the fidelity of the decompiler output. The experiments in Section V-B and Section V-E show that MSSC can successfully detect harmful code changes which account for 21.9% of all simplification-related responses from the LLM with 84% accuracy and 85% recall rate.

After testing DeGPT on the dataset containing the decompiler output from various software including practical command line tools, an IoT virus, a library for audio processing, and implementations of algorithms, the results show that from the aspect of structure simplification, DeGPT can achieve 24.4% reduction in the cognitive burden of understanding the decompiler outputs. DeGPT also adds comments of which 62.9% can provide practical semantics for the reverse engineers to help the understanding of binaries. From the aspect of variable renaming, 30.3% of the variables renamed by DeGPT are able to find correspondence with the variable names in the source code, while only 11.1% of DIRTY which is the current state-of-the-art variable renaming framework (see Section V-C). This shows as an end-to-end framework, DeGPT can highly facilitate the analysis of reverse engineers. Moreover, our user surveys also show that the optimizations

[2]We use "fidelity" to describe how close the decompiler output is to the actual semantics of the decompiled binary.

can significantly simplify the code and add helpful semantic information including variable names and comments, which aids in understanding the binary (see Section V-D).

Contributions. Our contributions are summarized as follows:

• *New techniques.* We propose a novel LLM-based end-to-end decompiler output optimization framework that enhances the readability of decompiled high-level PL by reconstructing semantic information (e.g., variable names) and structural simplification. DeGPT addresses several critical challenges that prior work has not effectively overcome, including designing a three-role mechanism (referee, advisor, and operator) to maximize LLM's potential and building MSSC to check the function semantic changes in optimizing decompiler output, which can effectively assist the reverse engineers in understanding the binary program better.

• *Implementation.* We implement our ideas as a framework called DeGPT. We evaluate DeGPT on datasets containing decompiler outputs from various software, including practical command line tools, an IoT virus, a library for audio processing, and implementations of algorithms. The results show that DeGPT can achieve 24.4% reduction in the cognitive burden of understanding the decompiler outputs, which highly facilitate the analysis of reverse engineers. DeGPT also has decent performance in reconstructing semantic information. Moreover, we conduct user surveys and participants have a positive attitude toward the DeGPT's ability to aid in binary comprehension. We will open source DeGPT to the community to facilitate the following research [3].

II. Background

Decompilers play an essential role in reverse engineering by assisting the analysts in understanding the binary by boosting the executables to a high-level PL. However, some factors distinguish the decompiler output from the manually written code, hindering the analysts' understanding and analysis of the target program. In this section, we will discuss these factors as the prelude to boosting the decompiler output to assist the reverse engineers better. We leverage Figure 1, which contains the source code and corresponding decompiler output, as an example for illustration. Compared to the source code, the decompiler output has the following drawbacks.

Redundant Structures. The decompiler output has redundant elements compared with the source code. For example, the local variables "iVar1" and "iVar2" in decompiler output are needless for achieving functionality. Moreover, the recursive part in decompiler output can also be simplified like in the source code. Unlike humans, who can simplify the code according to specific situations, the decompiler lifts the assembly to high-level PL based on the fixed, manually designed algorithms and rules, lacking the ability to simplify the code structures. The redundant structures increase the burden of understanding and decrease the analysis efficiency.

Meaningless Identifiers. An identifier provides semantic information describing the properties and functions of a variable, such as the identifier "number" in the source code. However,

[3]Open source address: https://github.com/PeiweiHu/DeGPT

Source Code

```
// Calculate Fibonacci numbers.
int Fibon(int number){

   if (number == 1 || number == 2) {
      return 1;
   } else{
      return Fibon(number - 1) +
Fibon(number - 2);
   }
}
```

Decompiler Output

```
int Fibon(int param_1) {
   int iVar1;
   int iVar2;

   if ((param_1 == 1) || (param_1 == 2)) {
      iVar2 = 1;
   } else {
      iVar1 = Fibon(param_1 + -1);
      iVar2 = Fibon(param_1 + -2);
      iVar2 = iVar2 + iVar1;
   }
   return iVar2;
}
```

Fig. 1: Example of Decompiler Output (Ghidra). The source code is compiled with -O0, and the binary contains no debug information.

the variables in the decompiler output lack meaningful identifiers. Take the parameter "param_1" as an example. Although it can be used to indicate that the variable is a parameter by "param", this is still a limited expression compared with the source code. Moreover, this phenomenon is more obvious in the more sophisticated code snippets. The main reason for this phenomenon is that the information about the identifier is lost during the compilation. Assigning valid identifiers to variables in the decompiler output is still challenging, even if the binary contains debugging information and the source code is available. This is because the decompiler output usually owns different structures compared with the source code, and its variables do not have a simple one-to-one mapping relationship.

Lacking Comments. Comments are explanations and descriptions of the source code that provide essential information describing the purpose and logic, which can also significantly enhance the readability and comprehensibility of the source code. Writing proper comments is an important principle for programmers. However, comments are discarded during the preprocessing step of compilation, making it impossible for the decompiler to recover the original comments. Moreover, some studies [36, 37, 66] try to automatically generate the comments for code that heavily depends on semantic information, such as the variable names. However, the identifiers in decompiler output are meaningless. Lacking comments makes the analysts have to understand the code's inner logic by reading the code alone without hints.

III. Approach

In this section, we present the design of DeGPT, an LLM-based framework for boosting the decompiler output and assisting the reverse engineers in understanding the binary program better. First, we give an overview of its workflow, mainly discussing the three-role mechanism and a code example to show how the different components of DeGPT work together. Then, we describe the details of R_ref, R_adv, and R_ope.

A. Overview

Figure 2 shows the workflow of DeGPT, which works as an end-to-end optimization tool between the decompiler and reverse engineers. After reading the output from the decompiler,

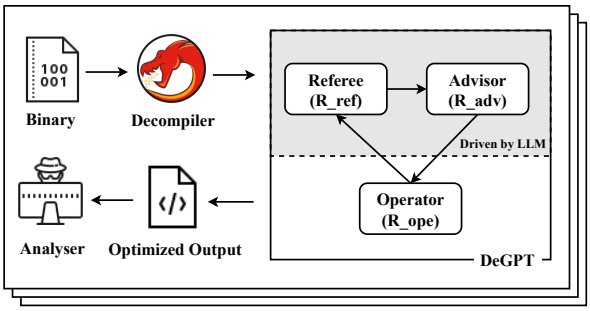

Fig. 2: Workflow of DeGPT. We use R_ref to represent the referee, R_adv to represent the advisor, and R_ope to represent the operator for readability in the following sections.

DeGPT's three roles start working, focusing on different tasks for optimizing the decompiler output. Finally, DeGPT outputs the optimized decompiler output, which is more readable and concise to the analyst. Moreover, the function semantics of the optimized output remains the same as the original decompiler output.

Three-role Mechanism. The most straightforward way of using LLM for optimizing the decompiler output is to pass the decompiler output directly to LLM with a prompt like "*Boost the following decompiler output to improve the readability*" and collect the response, which is called one-shot use. However, this approach will suffer from the following problems. (i) Prompts containing multiple tasks lead to limited optimization effectiveness. Logically, the aforementioned prompt assigns two tasks to the LLM, including what optimization to do and how to do it. However, previous studies [63] show that splitting the tasks into small pieces and guiding the LLM step by step maximizes the ability of the LLM. Our experimental results in Section V-E also conform this argument. (ii) Various optimizations cannot help each other, which limits the optimization effect. The aforementioned one-shot use compresses various optimizations into one interaction with the LLM, eliminating the possibility of different optimizations helping each other. For example, if DeGPT handles optimizations one by one, like first appending the comments describing the purpose of the code in decompiler output and then performing variable renaming, the newly appended comments can provide additional information for variable renaming, which in turn improves the effectiveness of the decompiler output. (iii) All-in-one optimization hinders function semantic checking. One of the goals of DeGPT is to retain the function semantics of the original decompiler output to uphold the fidelity of the boosted output compared with the source code. DeGPT adopts MSSC to perform function semantic checking. However, the above one-shot use will harm the execution of function semantic checks. For example, if LLM performs both variable renaming and structure simplification on the decompiler output, the structure of the decompiled code is vastly altered, thus making function semantic consistency checking much more difficult.

In order to improve the readability and comprehensibility of the original decompiler output while maintaining its functional semantics, we propose a three-role mechanism that splits the optimization task into fine-grained parts to gradually guide the LLM toward the refactoring of information such as meaningful variable names and comments. Specifically, the three-role mechanism contains three roles: R_ref, R_adv, and R_ope. DeGPT focuses on three optimizations including variable renaming, appending comments, and structure simplification to mitigate the drawbacks of decompiler output as discussed in Section II. The R_ref interacts with the LLM focusing on whether the three optimizations are necessary for the decompiler output to avoid unnecessary costs. The LLM's response to the R_ref is a list of *Yes* and *No*, which indicates the necessity of the corresponding optimizations. The R_ref then sends the required optimizations, which we call the optimization scheme, to the R_adv. The R_adv queries the LLM for rectification measures, i.e., how to edit the decompiler output to achieve the optimization scheme from the R_ref. As discussed before, various optimizations can help each other. Thus, the R_adv will sort the received various optimizations in the scheme and prioritize the optimizations that can provide additional guidance for subsequent optimizations. The output of the R_adv is the latest version of the decompiler output, which adopts the rectification measures from LLM. The R_adv will send the latest version of the decompiler output to the R_ope, which is responsible for checking whether the function semantic of the new decompiler output remains the same as the original decompiler output. To achieve this, R_ope conducts MSSC on two versions of decompiler output. If the new decompiler output passes the function semantic checking, DeGPT accepts it as the latest decompiler output and continues the subsequent optimization.

Example. Figure 3, a detailed example of DeGPT, illustrates how different components of DeGPT work together. Note that we use the decompiler output in Figure 1 as the input of DeGPT. First, R_ref sends the decompiler output to LLM for the optimization scheme, and it gets three *Yes* indicating all three optimizations are required, as shown in ①. Then, R_ref sends the optimization scheme and decompiler output to the next role R_adv. The R_adv first performs the sort, and optimizations that can provide additional guidance for subsequent optimizations are given relatively higher priority. After sorting, the structure simplificaiton is in the first place. The principles of sorting are discussed in Section III-C. Next, the R_adv chooses the proper predefined prompt for the optimization and sends the prompt containing the decompiler output to LLM for the specific editing operations of the optimization. The R_adv then adopts the editing operations on the decompiler output and sends a more simplified new version of the decompiler output to the R_ope, as shown in the top area of ③. The R_ope runs MSSC on the new version of decompiler output coming from the R_adv for function semantics checking. If MSSC concludes that the R_adv's rectification measures has not changed the function semantics, DeGPT accepts the new decompiler output as the latest one and continues the subsequent optimization, adding comments here. Figure 3-④ shows the final version of the optimized decompiler output after conducting all optimizations, which is more readable, concise, and more accessible to understand than the decompiler output in Figure 1.

B. Design of R_ref

While DeGPT focuses on three optimizations including variable renaming, comment appending, and structure simplification, not all decompiler outputs need all three optimizations. Directly applying three optimizations on all decompiler outputs will lead to unnecessary API token costs. For example, when

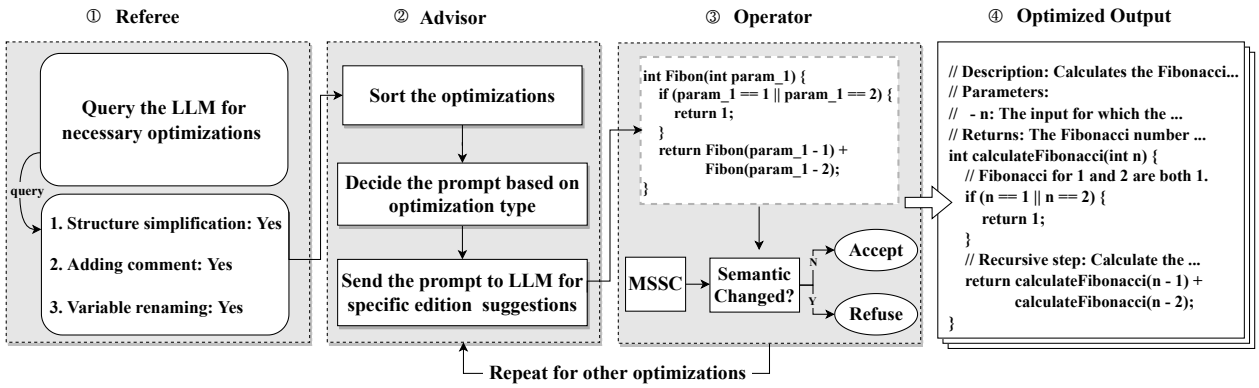

Fig. 3: Example of DeGPT. Use the decompiler output in Figure 1 as the input.

we ask LLM to perform structure simplification, for the functions that don't require structure simplification, LLM will simply return the same functions as input functions, which consumes additional API tokens. Considering this, DeGPT leverages the R_ref to check whether each optimization is necessary for the decompiler output. Specifically, the R_ref will query LLM with the following prompt[4] in the box. The response of LLM is a list of *Yes* and *No*, which costs a small number of tokens and indicates the necessity of the corresponding optimizations. The R_ref then sends the required optimizations, which we call optimization scheme, to the R_adv for the rectification measures to achieve the corresponding optimizations. Moreover, our evaluation in Section V shows that with the help of the R_ref, DeGPT can save up to 21.9% API tokens.

Prompt of R_ref

Do you think the following C code needs structure simplification, comment appending, and variable renaming? Answer three yes or no in order. No explanation.

TABLE I: Prompts of Correspond Optimizaitons.

Optimization	Prompt
Appending comments	Help me add code comments for the code snippet in the following C code. **No explanation**.
Variable renaming	Help me rename the variables for the code snippet in the following C code. **Output the old and new names in JSON format like {'old name': 'new name'}. No explanation**.
Structure simplification	Simplify the following C function by removing redundant variables and unnecessary code. **No explanation**.

C. Design of R_adv

R_ref transfers the optimization scheme that LLM suggests to R_adv, which aims to figure out the specific methods to implement the suggested optimizations on code with the help of LLM. To achieve this, an intuitive approach is to send the optimization type to LLM after joining it with a prompt like "*how to edit the code to achieve that*". However, according to our experience, the response from LLM in this way will become unpredictable because of the randomness of LLM. Thus, we prepare the prompt for each optimization to make the response from LLM is predictable and analyzable. Moreover, we observe that some optimizations can provide additional information for the other optimizations. Thus, a proper optimization sequence will boost performance instead of randomly conducting optimizations. We discuss these steps in detail in the following.

Choose Prompts. The first step of the R_adv is to choose the appropriate prompt based on the types of suggested optimizations. Since the LLM is a generative model, its output format is full of randomness. Besides describing the task, another critical role of the prompt is to limit the output of LLM to facilitate the following analysis. Considering this, we carefully design our prompts based on the types of optimizations, as shown in the column "Prompt" in Table I. We use bold fonts to highlight the sentences that control the formatting of the output. For variable renaming, we leverage an example {'old name': 'new name'} to ask the LLM to output the response in JSON format, which is convenient for later code optimization. We also limit the additional explanation of LLM by appending the sentence "*No explanation*" to prompts. The experimental results show that these prompts can work well, and the response is easy to analyze.

Sort Optimizations. As discussed before, by splitting the whole optimization task into small pieces, various optimizations can help each other. DeGPT achieves this by adequately arranging the order of the optimizations and letting the previous optimization provide additional information for the succeeding optimizations. Generally speaking, the optimizations mentioned above can be divided into two classes. One is the structure-related optimizations, which will change the abstract syntax tree of the code, such as removing the redundant variables. The other is semantic-related optimizations, which will not change the abstract syntax tree but the semantic information that facilitates understanding the code, such as renaming variable names and appending comments. The structure-related optimizations can affect the semantic-related optimizations. For example, one may assign the name "counter" to the loop's induction variable. However,

[4]We use "C code" since the decompiler output corresponds to the syntax of C language.

the structure-related optimizations depend on the PL's syntax and the code's logical structure. Based on this observation, structure-related optimizations occur before semantic-related optimizations. Moreover, we observe that the variable names can be treated as simplified comments. The comments can assist the LLM in choosing the appropriate words for variable renaming. Thus, it is reasonable to put the optimization of adding comments before variable renaming.

D. Design of R_ope

As a generative model, LLM generates the response based on understanding the provided prompt and past learning experience. While it currently shows impressive performance, there are some arguments that it does not ensure the correctness of the response [23, 42]. DeGPT is designed to improve the readability and understandability of the decompiler output without changing the semantics of the original function. Thus, while R_ref indicates the optimization scheme and R_adv gives the specific rectification measures to achieve the optimizations, R_ope is responsible for checking the correctness of the optimized decompiler output. We define the "correctness" of optimized decompiler output as it keeps the same function semantics as the original decompiler output. In other words, we define the original decompiler output as F and the optimized decompiler output as F', and they satisfy the following equations:

$$\begin{cases} F(i) = F'(i) \\ SideEffect(F(i)) = SideEffect(F'(i)) \end{cases} \tag{1}$$

while $i \in Input(F)$, i.e., i belongs to the input set of F, and $SideEffect$ represents the side effect[5] of the function invocation [24]. It is not easy to verify the correctness of the optimized decompiler output. The straightforward and effective way to conform to the Equations 1 is to dynamically execute the decompiler output with as many inputs as possible. However, there is some distance away from being able to run the decompiler outputs directly [21, 61, 65]. Verifying the correctness of the optimized decompiler output by static analysis is more feasible but also suffers from some challenges. Firstly, it is difficult to simulate the code semantics without running it. We cannot obtain the return values and side effects needed for correctness checking. Secondly, tracing the side effect hidden in the call chain requires heavy overhead. As the call chain grows, the execution path increases exponentially. However, call chains appear frequently, leading to a rapid increase in overhead.

MSSC. To cope with the aforementioned challenges, we propose *Micro Snippet Semantic Calculation* (MSSC), which aims to check whether the optimized decompiler output keeps the same function semantics as the original decompiler output. MSSC reads in a pair of code snippets containing the optimized decompiler output and original decompiler output and outputs a boolean value indicating whether the function semantics change after optimizations. While deciding the function semantic changes is not easy, checking Behavioral Equivalence [54] is one common method in practice and Equation 1 describes the behavioral equivalence in programming languages. The rationale behind MSSC is to simulate the execution of the function and check the changes of return values and side effects of the code before and after optimization to finally check the behavioral equivalence. In detail, MSSC simulates the function execution by assigning symbols like variables and callees (arguments of the callees are also considered) with unique random numbers and updating the symbols' values along the statements in execution paths. So we can get comparable values of symbols like the returned variable for later comparison, which addresses the aforementioned first challenge. Moreover, MSSC represents the invocations with unique random numbers during the simulation instead of tracing the long call chains, which addresses the aforementioned second challenge. The main insufficiency of this method is that in theory, different random numbers that experience different statements may lead to the same final values, making a symbol that has changed be mistaken for unchanged. But that is with small probability, and we don't meet this case during the evaluation of MSSC. Other common issues of static program analysis such as point-to issue may also harm the performance of MSSC, as discussed in Section V-E. We evaluate MSSC in Section V-E and it achieves an accuracy of 84% and a recall of 85%, showing it can effectively help filter out unreasonable optimization. MSSC consists of two phases: the calculation and comparison phases. The former calculates each code fragment to collect the necessary information, while the latter compares the computation results of two code fragments to check for function semantic changes.

Algorithm 1: Calculation Stage

Input: $Code$: Code snippet;
Output: $SymTables$: A set storing the symbol value tables of different paths;
$CallLogs$: A set storing the invocations with the argument values of different paths;

```
1  SymTables ← ∅;
2  CallLogs ← ∅;
3  Paths = getExecutionPath(Code)
4  SymTable = NewTable();
5  Syms = CollectSymbols(Code);
6  foreach S of Syms do
       SetRandomValue(SymTable, S);
7  for Path in Paths do
8      ST = CopyTable(SymTable);
9      CL = NewCallLog();
10     for Statement in Path do
11         UpdateSymTable(ST, Statement);
12         if hasInvocation(Statement) then
13             LogInvocation(CL, Statement);
14         end
15     end
16     Add(SymTables, ST);
17     Add(CallLogs, CL);
18 end
```

Calculation Phase. Algorithm 1 shows the workflow of the Calculation Phase, whose primary purpose is to collect the value changes during the execution and to provide information for the following comparison phase. The calculation's result consists of a set that stores the symbolic values after the exe-

[5]In computer science, an operation, function or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, which is to say if it has any noticeable impact other than its primary effect of returning a value to the invoker of the operation. [18]

cution (*SymTables*) and another set that stores the invocations with their argument values during the execution (*CallLogs*). Specifically, MSSC first generates the execution paths (line 3) and collects the symbols of the target code (line 5). These symbols include all variables and memory locations accessed by pointers. MSSC then assigns unique random numbers to these symbols (line 6). Note that the assigned random numbers of the symbols will be shared between the original decompiler output and the optimized decompiler output to enable the comparison between the two outputs in the next phase. Next, MSSC iterates over the statements for each path while updating the symbol values (line 11). If the statement contains the invocation, MSSC records it with the argument values in *CL* for the comparison in the second phase (lines 12-14). Moreover, every invocation in *CL* will also be assigned a unique random number in case it participates in the computation of the rvalue of a statement such as assigning a variable with the return value of an invocation. After iterating all paths, MSSC outputs the collected information ($SymTables$ and $CallLogs$) to the next phase.

Comparison Phase. The Comparison Phase aims to check whether the optimized decompiler output keeps the same function semantics as the original decompiler output by simulating the function semantics based on the information from the Calculation Phase. The Comparison Phase contains the following two steps. Step 1: *Invocation checking*. Since DeGPT does not provide the source code of the callee to the LLM while optimizing the caller, the LLM has no basis for change invocations. Thus, if the result $CallLogs$ of optimized decompiler output differs from the original decompiler output, DeGPT concludes that the function semantics change. By invocation checking, DeGPT checks the correctness of the side effects hidden in the call chain. Besides, by invocation checking, DeGPT converts inter-procedural analysis to intra-procedural analysis and saves the cost of tracing too many execution paths. Step 2: *Variable checking*. Check whether the values of return variables and non-local variables remain the same in each execution path of the optimized decompiler output compared to the original decompiler output. Suppose the optimized decompiler output successfully passes the invocation and variable checks. In that case, the R_ope will accept the change, thus achieving that the readability and comprehensibility of the original decompiler output are greatly improved while the function semantics remain unchanged.

While the LLM may provide the wrong response, MSSC helps R_ope guard the function semantics after optimizations keep unchanged compared with the original decompiler output, avoiding confusing the reverse engineers in practical usage. We also evaluate the efficacy of MSSC in Section V-B and Section V-E. The results show that MSSC can effectively detect function semantics changes with an accuracy of 84% and a recall rate of 85%.

IV. Implementation

Large Language Model. As an explosive technology, many large language models have been published recently [1, 4, 12, 15]. We choose ChatGPT (gpt-3.5-turbo with the temperature 0.2) as the LLM support of DeGPT for the following reasons. Firstly, it is a leading model and performs well according to various rankings and studies [49, 55, 57]. Secondly, OpenAI provides easy-to-use and reliable interfaces to invoke ChatGPT, which highly facilitates our research. Note that even if we adopt ChatGPT to support DeGPT, the techniques involved in DeGPT are not limited to ChatGPT but are also available for other LLMs. For example, we explore the effectiveness of DeGPT supported by GPT-4 in Section VI. The components of DeGPT designed for interaction with the LLM consist of 900 lines of Python code.

Program Analysis. The typical way [35, 46, 47] to implement program analysis is based on the framework, like LLVM infrastructure [16], CodeQL [6]. However, they require the analyzed code to be compilable, which is unrealistic for the decompiler output. Fortunately, the decompiler output conforms to the syntax of C language, allowing us to analyze it by code parser generated by parser generator like tree-sitter [19]. Considering the aforementioned reasons, we adopt cinspector [5], a tree-sitter-based code analysis framework, to analyze the decompiler output. And our practical experience shows that it can satisfy our analysis requirements. The components of DeGPT designed for program analysis consist of 3,100 lines of Python code. Another possible issue is the path explosion while collecting the execution path of the code waiting for optimization. We avoid the path explosion caused by loops by transforming them into branch statements, as employed in AURC [35]. Path explosion caused by branch statements primarily occurs in inter-procedural analysis, caused by long call chains. MSSC conducts intra-procedural analysis. Based on our analysis, the average execution path of functions in the dataset is 6.2, with the maximum path being 106, which is still within acceptable limits. As a precaution, we also set a time limit to interrupt analyses with path explosion issues. Addressing the issue of path explosion is not the focal point of our research.

V. Evaluation

A. Experiment Setting

Platform. All our experiments are conducted on the server running Ubuntu 20.04 with 8 processors (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 128GB memory, 3TB hard drive, and 2 GPUs (RTX 3090). The compiler used for evaluation is gcc 9.4.0 and the decompiler is Ghidra 10.2.3.

Dataset. Previous studies [25, 29, 44] spend much effort building tremendous datasets to contain various source codes as much as possible. This is because the effectiveness of their model highly depends on the fed training set. One of the strengths of DeGPT is that it relies solely on LLM (like ChatGPT) and does not require training to get decent performance. We only need a relatively small test set to evaluate the performance of DeGPT without requiring a massive training set. We construct our dataset by collecting random functions from various codebases to ensure our experiments are conducted on representative samples. In detail, we collect the code from LeetCode [14], Coreutils [7], Mirai [17], and AudioFlux [2]. We choose these codebases because they are popular and representative of various fields including implementations of algorithms, practical command line tools, IoT viruses, and audio processing. Considering the training set of ChatGPT predates September 2021, we use AudioFlux [2], a library starting from 2023 and getting over 1,900 stars on GitHub,

to test the efficacy of DeGPT on the codebases that ChatGPT has never seen before. Next, we compile the source code with the compiler optimization level "O2" for the collected samples and decompile them with Ghidra [11] to obtain the decompiler output for subsequent optimization tests. We also decompile the stripped version of the aforementioned samples to test the efficacy of DeGPT on stripped binaries. Further, we filter out the functions the decompiler cannot handle and randomly select 620 sets of source code and decompiler outputs of functions from non-stripped and stripped binaries to construct our final dataset. In detail, the dataset contains 170, 40, 310, and 100 sets of source code and decompiler outputs of functions from non-stripped and stripped binaries from LeetCode, Mirai, Coreutils, and AudioFlux, respectively.

Metrics. Unlike previous studies [25, 29, 44], that focus on specific types of optimizations, DeGPT involves several optimizations, including structure simplification, appending comments, and variable renaming. Moreover, some optimizations are adopted for the first time on the output of the decompiler. Therefore, we adopt a variety of metrics to ensure a comprehensive evaluation of the optimization effectiveness. We discuss these metrics below.

• *Meaningful Variable Ratio (MVR):* We also design a metric to evaluate the effectiveness of DeGPT in variable renaming. While evaluating variable renaming is an open problem, the high-level idea is to compare the optimized variable name with the variable name in the source code. Inspired by the previous studies [44], we treat an optimized variable name meaningful if the Levenshtein distance between it and one of the variables in the source code is less than the threshold[6]. Moreover, as discussed in the previous studies [40], abbreviations and synonyms are common in naming variables. To cope with this, we manually construct a list of synonyms to refine the reliability of the evaluation. If the optimized name and one of the variable names in the source code belong to the same group in the synonym list, this optimized variable name will also be treated as meaningful. Specifically, we define

$$\text{MVR} = \frac{\text{Meaningful Variable(optimized output)}}{\text{Variable Number(optimized output)}}$$

where *Meaningful Variable* represents the number of meaningful variables and *Variable Number* represents the number of all variables. The higher MVR is, the better DeGPT behaves in variable renaming.

• *Effort Ratio (ER):* We leverage Effort Ratio (ER for short) to evaluate the effectiveness of structure simplification. It is the ratio of the *Effort* metric from Hastead's Complexity Measures [30] before and after optimization. We adopt Hastead's Complexity Measures because recent research has shown that it has the highest correlation with the cognitive burden of understanding programs among the four categories of program evaluation metrics under test [52]. Moreover, *Effort* is a comprehensive metric in Hastead's Complexity Measures that reflects the effort required to implement or understand a program [10]. Specifically, we define

$$\text{ER} = \frac{\text{Effort(optimized output)}}{\text{Effort(output)}}$$

where *Effort* represents the effort metric in Hastead's Complexity Measures. The lower ER is, the better DeGPT behaves in structure simplification.

• *Correct Rate of Comments (CR):* We design Correct Rate of Comments (CR for short) to measure the ability of DeGPT in appending comments, which is shown as follows.

$$\text{CR} = \frac{\text{Correct Comments(optimized output)}}{\text{All Comments(optimized output)}}$$

Specially, *Correct Comments* represents the number of correct comments and *All Comments* represents the number of all comments. It's not easy to decide the correctness of the comments automatically. Therefore, we manually calculate CR to ensure its trustworthiness during the evaluation. The higher CR is, the better DeGPT behaves in appending comments.

• *Non-trivial Rate of Comments (NR):* Considering only the correctness of comments is not enough, we also need to consider whether the comments added by DeGPT can help understand the program. We define a comment as *trivial* if it merely states the behavior of the program statement. Otherwise, it's *non-trivial*. For example, the comment "set a to 1" for `a=1` is trivial. Specifically, we define

$$\text{NR} = \frac{\text{Non-trivial Comments(optimized output)}}{\text{Correct Comments(optimized output)}}$$

where *Non-trivial Comments* represents the number of non-trivial comments and *Correct Comments* represents the number of correct comments, to evaluate the quality of the comments appended by DeGPT. Like CR, we also manually calculate NR to ensure its trustworthiness during the evaluation. The higher NR is, the better the quality of the appended comments is.

B. Effectiveness

Here we evaluate the effectiveness of DeGPT from multiple aspects. We first test DeGPT focusing on four metrics including MVR, ER, CR, and NR. They can reflect the efficacy of DeGPT from both the structure and semantic information aspects. Further, we analyze the different optimization types involved and the reasons for the failure of some cases.

Overall Effectiveness. We evaluate the performance of DeGPT on our dataset containing decompiler output of non-stripped and stripped binaries from LeetCode, Mirai, Coreutils, and AudioFlux. The results are shown in Table II. The symbol ① represents non-stripped binaries while ② represents stripped binaries. We first analyze the results of non-stripped binaries (i.e., the data under the column "①"). The column "MVR" in Table II shows the ability of DeGPT to rename variables. The MVRs of different codebases vary from 27.0% to 37.0%, with an average MVR of 30.3%. This means 30.3% of the variable names generated by DeGPT can correspond to the variable names in the source code. However, this does not mean the other variable names are meaningless since the decompiler output and source code do not have the same structures. Other variable names may be meaningful under the context

[6]In the implementation, for names less than 5 characters, only identical names are judged to be correct. For the longer name, if the Levenshtein distance is less than 30% of the length, the assigned name is judged to be correct.

of decompiler output. Considering this, we also conduct user surveys, which involve the effectiveness of variable renaming, to evaluate the performance of DeGPT from another aspect, as discussed in Section V-D. The column "ER" in Table II shows the ability of DeGPT to simplify the decompiler output. The ERs of LeetCode, Mirai, Coreutils, and AudioFlux are 75.5%, 77.5%, 72.0%, and 77.6%, respectively and the average ER is 75.6%, which means 24.4% reduction in the cognitive burden of understanding the decompiler outputs. The column "CR" and "NR" presents the ability of DeGPT to append the comments. We measure these two indicators manually to get a high-quality evaluation of comments. In detail, we analyze all testcases of Mirai manually. For the other three codebases with hundreds of testcases, we randomly sample 50 outputs for both non-stripped and stripped binaries for each codebase and manually analyze these samples. The results show that for all codebases, CRs are very high, close to 100%. This means that DeGPT hardly produces incorrect comments. The NRs of four codebases vary from 53.0% to 71.6% with an average NR of 62.9%, which means that 62.9% of the generated comments can provide practical semantics for the reverse engineers to help the understanding of binaries.

TABLE II: Effectiveness of DeGPT. The symbol ① represents non-stripped binaries while ② represents the stripped binaries.

Codebase	MVR (%)		ER (%)		CR (%)		NR (%)	
	①	②	①	②	①	②	①	②
LeetCode	27.0	23.0	75.5	76.9	98.7	100	64.8	36.8
Mirai	29.1	17.3	77.5	75.8	99.4	96.8	71.6	36.7
Coreutils	28.0	20.2	72.0	74.2	98.9	100	62.0	40.6
AudioFlux	37.0	36.4	77.6	78.5	99.6	100	53.0	38.4
Average	30.3	24.2	75.6	76.3	99.2	99.2	62.9	38.1

There are two things deserving discussion. The first is the influence of stripped symbols of binaries. For four metrics, MVRs of stripped binaries suffer from a decrease compared with non-stripped binaries. This is because of the lack of the semantics provided by the stripped symbols while optimizing the decompiler outputs of stripped binaries. However, DeGPT still performs better on stripped binaries than DIRTY does on non-stripped binaries, which has an MVR of 11.1% on average. ER is not influenced by the stripped symbols. This is because the stripped symbols can hardly affect the structure of the decompiler output. The CRs of stripped binaries also remain similar to the non-stripped binaries, which shows that the LLM has a strong ability to append correct comments. The NRs of stripped binaries suffer from decreases compared with non-stripped binaries, indicating that stripped symbols contribute to the meanings of comments. Note that the MVR of AudioFlux has nearly no reduction since part of its samples come from the shared library containing public symbols that won't be dropped during the stripping. Secondly, we observe there is no significant performance reduction for the codebase that is not in the training set of LLM. As discussed before, the training set of ChatGPT predates September 2021. However, even in the codebase AudioFlux, which starts from 2023, DeGPT has a decent performance. This shows the LLM has good generalization facing unseen codebases.

Analysis of Different Optimizations. We also evaluate the effectiveness of DeGPT from various optimizations. We analyze the distribution of optimization suggestions from R_ref on each dataset and propose a new metric to evaluate the variable renaming. The results are shown in Table III. The column "R_ref" represents that R_ref gives how many cases in this dataset the optimization suggestions belonging to the type in the column header. The column "Succ" represents how many cases pass the checking of R_ope and accept the specific optimization suggestions from R_adv. Note that the datasets LeetCode, Mirai, Coreutils, and AudioFlux have 170, 40, 310, and 100 test cases, respectively.

TABLE III: Distribution of Different Optimizations. Rref - the optimization scheme from R_ref contains this optimization, Succ - this optimization is finally accepted by decompiler output.

Codebase	Simplify		Add Comment		Rename		
	R_ref	Succ	R_ref	Succ	R_ref	Succ	ONR
LeetCode	168	130	161	160	121	118	93.0%
Mirai	40	33	37	37	32	32	94.3%
Coreutils	301	224	278	277	224	220	90.2%
AudioFlux	97	86	81	81	76	76	96.8%
Total	606	473	557	555	453	446	92.4%

The column "Simplify" shows that the R_ref gives optimization recommendations for structural simplification for 606 functions in four datasets containing 620 functions. However, only 473 (78.1%) of them finally accept the specific simplification suggestions from R_adv. In the other 133 (21.9%) cases, the simplification suggestions from LLM fail to pass the MSSC checking from R_ope since they change the original function semantics. This suggests that the LLM, as a generative model, may make mistakes while performing fine-grained tasks like code optimization. Our proposed method MSSC can effectively detect these mistakes and prevent them from mixing into the final optimization results, causing problems for the reverse engineers. We will further evaluate the performance of MSSC in Section V-E. The column "Add Comment" describes the status of the optimization adding comments. This optimization appends additional semantic information describing the decompiler output instead of changing the code structures, thus owning a higher acceptance rate. In detail, R_ref gives the optimization suggestion of adding comments for 557 functions in three datasets, of which 555 (99.6%) eventually accept the changes. The failures are mainly due to the unexpected response from LLM. The failed cases are because the LLM response accidentally changes the function structure instead of just appending comments. The column "Rename" describes the status of the optimization variable renaming. R_ref suggests renaming variables on 453 (73.0%) functions out of all 620 functions in the four datasets, implying the urgency of renaming the variables in decompiler output with more meaningful names. Moreover, 446 (98.5%) of these 453 functions accept the new variable names from R_adv. The failed 7 cases are because the response from the LLM does not conform to the JSON format, which we ask the LLM to follow in the prompt.

To fully evaluate the ability of DeGPT to rename variables and to facilitate comparison with the one-shot experiment in Section V-E, we introduce the metric Optimized Name Ratio (ONR for short) to evaluate how many variables names are newly allocated among all variable names in the optimized

decompiler output. In detail, we define

$$\text{ONR} = \frac{\text{New Name Number(optimized output)}}{\text{Name Number(optimized output)}}$$

where *New Name Number* represents the number of the newly allocated variable names and *Name Number* represents the number of all variable names. The higher the ONR is, the more variables that the optimization renames. The results of ONR are shown in the column "ONR" in Table III. The ONRs of different codebases vary from 90.2% to 96.8%, and the overall ONR is 92.4%. This means that DeGPT can rename, on average, 92.4% of the variables in the decompiler output. The high ONR credits the three-role mechanism, which divides the optimization tasks into more concrete and specific pieces. The one-shot experiment in Section V-E shows that the overall ONR drops to 57.0% without the three-role mechanism.

C. Comparison with the State-of-the-Art

To compare the performance of DeGPT with the previous approaches, we select the state-of-the-art tool DIRTY [25]. DIRTY is a transformer-based optimization framework that can rename variables in the decompiler output for better readability, which is trained on a large dataset DIRT [8], constructed by aligning variables in the source code and decompiler output. The authors provide various models for testing. We choose the model "DIRTY-Multitask", which is trained longest (120 hours) for our evaluation. The experiment steps follow the guidelines provided by the authors [9].

TABLE IV: Comparison with DIRTY on non-stripped binaries. The result of AudioFlux is $\star$ since DIRTY crashed on AudioFlux.

Codebase	DeGPT		DIRTY	
	MVR	ONR	MVR	ONR
LeetCode	27.0%	93.0%	6.2%	78.2%
Mirai	29.1%	94.3%	16.5%	76.3%
Coreutils	28.0%	90.2%	10.7%	75.7%
AudioFlux	37.0%	96.8%	$\star$	$\star$
Average	30.3%	93.6%	11.1%	76.7%

Here we mainly focus on two related metrics, MVR and ONR, to compare the performance of DIRTY with DeGPT regarding variable renaming. The results of DIRTY are shown in the column "DIRTY" of Table IV. From the aspect of ONR, DIRTY's value varies from 75.7% to 78.2%. The average ONR of DIRTY is 76.7% while the total ONR of DeGPT is 93.6%. Regarding MVR, the values of DIRTY are between 6.2% and 16.5%, with an average MVR is 11.1%, which is lower than DeGPT's 30.3%. The results show that DeGPT can allocate more meaningful names to more variables than DIRTY. Through in-depth analysis, we found two main reasons for the restricted efficacy of DIRTY. (i) *Limited training dataset leads to its limited prediction capability.* Although DIRTY is trained on a relatively large dataset, it is still not large enough to cope with the multitudinous code snippets. It is difficult for DIRTY to handle cases where code contexts never appear in the training set. For example, regarding MVR, DIRTY performs best on the dataset Coreutils. This is because the training set contains the corresponding source code and decompiler output. (ii) *The alignment makes DIRTY optimize only partial variable names, and the ignored variable names can hinder the comprehensibility of the decompiler output.* As discussed before, the training set DIRT is constructed based on the variable alignment between the source code and decompiler output. DIRTY labels the aligned variables in decompiler output while predicting the variable names. The variables that are not labeled are not optimized but remain with the original meaningless names, affecting the optimization of the decompiler output. From decompilation, the variables between source code and decompiler output are not one-to-one mapping leading to this drawback. DIRTY fails to construct the context information in the training set for the variables that only appear in the decompiler output and do not have a corresponding variable in the source code. The weaknesses of DIRTY make the advantages of DeGPT even more prominent. DeGPT is driven by the LLM, which is famous for the vast and rich datasets it is based on and the large-scale model parameters to cope with various situations. Thus, DeGPT can better cope with complex and variable decompiler output, and the readability improvement against decompiler output is more pronounced than DIRTY, which is heavily dependent on the size of the training set.

D. User Study

DeGPT pays attention to the practical effects of assisting the reverse engineers in understanding the target binaries. To evaluate the extent to which DeGPT helps reverse analysts, we design two user studies including a question-based study and a task-based study based on the results of non-stripped binaries. The IRB of our affiliation has approved our studies.

Question-based Study. We involve 12 participants and divide the participants into three groups, professional, intermediate, and introductory, to explore the practical value of DeGPT for the reverse engineers of different levels. The professional group consists of 4 experienced engineers involved in reverse engineering for more than 4 years. The intermediate group has 4 reverse engineers with an average of 2 years of experience. The introductory group comprises 4 programmers who understand the basic concepts and skills of reverse engineering. Furthermore, we randomly choose 10 pairs <original decompiler output, optimized decompiler output> for each participant. After presenting the pair and providing the reading time, we query the participants focusing on the following questions (Q1-Q5). We also provide a tip "*Please rate on a scale of 0 to 10. (10 – strong agreement, 5 – neutral, 0 – strong disagreement.)*" to guide the scoring of each question. In total, we collect 120 scores for each question in Q1-Q5.

- *Q1:* Do you think the optimized decompiler output is more concise (e.g., fewer redundant variables) and idiomatic compared with the original decompiler output?
- *Q2:* Do you think the optimized decompiler output owns more meaningful and helpful variable names compared with the original decompiler output?
- *Q3:* Do you think the optimized decompiler output owns more meaningful and helpful comments compared with the original decompiler output?
- *Q4:* Do you think the optimized decompiler output retains the function semantics (or behaviors) compared with the original decompiler output?

• *Q5:* Generally, do you think the optimized decompiler output is more helpful in understanding the target binary compared with the original decompiler output?

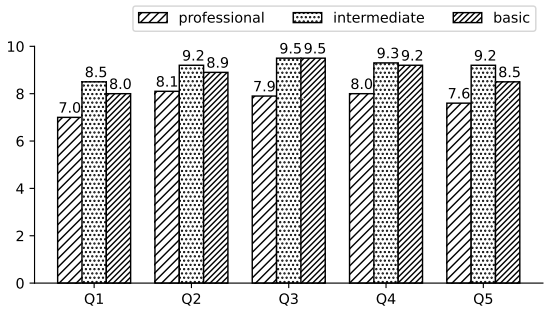

Fig. 4: Results of User Study. 10 - strong agreement, 5 – neutral, 0 – strong disagreement.

We average Q1 to Q5 by each of the three groups, and the final results of the user study are shown in Figure 4. Firstly, considering Q1 to Q5 in all three groups, the scores vary from 7.0 to 9.5. Since score 5 represents a neutral attitude, the score scale reflects that participants show positive attitudes to the optimized decompiler output concerning the five questions related to code structures, variable names, comments, and the effectiveness of assisting the reverse engineers. Secondly, we observe that the professional group scored lower than participants in the other groups from Q1 to Q5, which reflects the limitations of the knowledge of the LLM. The LLM behaves like a domain expert by training on large amounts of data, leading to more positive attitudes among participants in the introductory and intermediate groups. However, the professional group members' attitude towards the results given by the LLM is because they have more experience than the LLM, so the additional information provided to them by LLM is limited. Note that the professional group has a positive attitude towards the results of DeGPT compared to the results of the direct decompiler output since even the lowest score reaches 7. Thirdly, we observe that Q1 possessed lower scores than the other questions in all three groups. After interviewing the participants, we learn this is mainly because some optimized code has too many detailed comments, more than the programmers would write. This adds to the burden of reading the code and makes the participants feel that the optimized code is not idiomatic enough compared with the hand-written code. The problem can be mitigated by, for example, removing comments attached to simple statements.

Task-based Study. We involve 22 participants for a task-based user study to evaluate the practical efficacy of DeGPT in helping reverse analysts comprehend the decompiler outputs. In particular, each participant receives 5 code snippets from the direct output of decompiler (group-1) and 5 code snippets from DeGPT (group-2). For each participant, group-1 and group-2 are non-overlapping and of similar scale[7]. Moreover, for each decompiler output that appears in group-1, the DeGPT-optimized version of it will appear in group-2 of another participant to reduce the overall difficulty difference between the two groups. The participants are tasked with answering the question, "What's the purpose of this code snippet?". We manually analyze the time consumption and accuracy of participants analyzing code snippets. The results show that, on average, the answers of group-1 achieve an accuracy rate of 39% consuming 1584 seconds while group-2 achieve 88% consuming 840 seconds. The results show that DeGPT improves accuracy by 126% and saves 47% of time. With the help of DeGPT, the understanding of code is faster and more accurate.

[7]The difference in Lines Of Code is within 10%.

E. Evaluation of Individual Components

To maximize the potential capabilities of the LLM, we design a three-role mechanism that splits the optimization tasks into different small pieces and guides the LLM to finish the work step by step. We also design the role R_ref in the three-role mechanism to save API tokens. Moreover, as a generative model, we observed that the LLM might make mistakes on the assigned tasks and fail to "tell the truth". Thus we design MSSC to check the function semantic changes and uphold the fidelity of the optimized decompiler output. In this section, we evaluate the effectiveness of these three components focusing on whether the three-role mechanism leads to better performance in optimizing the decompiler output, whether the role R_ref helps save tokens, and whether MSSC successfully detects the mistakes made by LLM with decent accuracy and recall.

Prompt of DeGPT (Without Three-role Mechanism)

Optimize the following C function to improve readability and simplicity. **Output the new code and no explanation.**

Three-role Mechanism. We evaluate the three-role mechanism to verify that by splitting the task into different components, DeGPT achieves better optimization performance than optimization in a one-shot manner. In detail, we combine the prompt in the above box with the decompiler output and send it to the LLM. Like the prompts we design for DeGPT, the used prompt includes the sentence "*Output the new code and no explanation*" to control the response. We run the optimization in a one-shot manner on the non-stripped datasets consisting of LeetCode, Mirai, Coreutils, and AudioFlux. The results are shown in Table V.

TABLE V: Results of One-shot Experiment. The symbol ① represents DeGPT with the three-role mechanism and the symbol ② represents the one-shot experiment.

Codebase	Simplify		Comment		Rename		ONR	
	①	②	①	②	①	②	①	②
LeetCode	168	74	161	21	121	55	93.0%	46.0%
Mirai	40	20	37	6	32	16	94.3%	44.5%
Coreutils	301	104	278	109	224	38	90.2%	53.8%
AudioFlux	97	46	81	24	76	35	96.8%	83.6%
Total	606	244	557	160	453	144	92.4%	57.0%

The results in Table V illustrate that the three-role mechanism leads to better optimization performance than the one-shot optimization. Firstly, the three-role mechanism leads to higher optimization frequency. Of all three optimization types, DeGPT performs optimization on more functions in the datasets than the one-shot experiment. For example, while the

comments are essential to describe the code snippets, only 160 of 620 functions in datasets enjoy this optimization in the one-shot experiment. However, by deciding the optimization directions by the R_ref first, DeGPT applies this optimization to 557 functions in datasets. Secondly, the three-role mechanism leads to higher optimization quality. For example, 144 functions enjoy the variable renaming optimization in the one-shot experiment. However, the ONR, which measures how many variable names are optimized among all variable names, of the one-shot experiment is only 57.0% while DeGPT can reach 92.4%. This shows that DeGPT can achieve complete variable renaming through the R_ref and R_adv. Thirdly, the three-role mechanism refines the correctness of optimization. As we discussed in Table III about the optimizations of DeGPT, among all 606 functions requiring structure simplification, only 473 of them finally accept the specific changes suggested by R_adv, and 21.9% of the suggestions from R_adv destroy the original function semantics. Although 244 functions received the simplified optimization in a one-time experiment, reverse engineers have to determine whether their modifications are correct due to the lack of R_ope for function semantic checking. This would give misleading results to the final optimization and hinder the reverse engineer from analyzing the target binary. Overall, compared to a single experiment, DeGPT with a three-role mechanism can perform 2.5 times more structural simplification optimizations, 3.1 times more renaming optimizations, and up to 3.5 times more additional commits optimizations, effectively improving the readability of the decompiled output.

Role R_ref. We design the R_ref to avoid unnecessary optimizations wasting the API tokens. Lower usage of API tokens means lower costs. In this section, we evaluate the effectiveness of the R_ref in saving tokens. Specifically, we randomly selected 100 functions from the non-stripped binaries in the datasets for the experiment. First, we optimize these functions without including the R_ref, which means applying three optimizations directly (expr-1). Then, we optimize the same set of functions with the inclusion of the R_ref (expr-2). Finally, we compare the performance metrics and token consumption between these two experiments.

TABLE VI: The effectiveness of the R_ref. The column "Token" represents the token consumption. The token consumption of expr-2 is 78.1% of expr-1.

Experiment	MVR	ER	CR	NR	Token
expr-1	31.9%	80.5%	98.2%	62.4%	253,998 (100%)
expr-2	29.2%	79.4%	99.1%	64.0%	198,387 (78.1%)

The result in Table VI shows that after removing some unnecessary optimizations through the R_ref in expr-2, there is no significant decrease in performance metrics compared to optimizing all functions with three optimizations in expr-1. This shows that not all functions require three optimizations. Furthermore, according to statistics, with the help of the R_ref, the token consumption can be reduced by up to 21.9%. This demonstrates the role of the R_ref in cost savings.

MSSC. As mentioned above, MSSC is a function semantic checking method that aims to mitigate the semantic changes caused by LLM responses. We evaluate its performance by the following two main metrics. The first is *accuracy*, reflecting how many cases MSSC can review correctly. It is calculated by dividing the number of cases correctly analyzed using MSSC by the total number of cases. The second is the *recall rate*, which reflects how many problematic cases MSSC can report among all problematic cases of LLM. It is calculated by dividing the number of cases correctly reported as problematic using MSSC by the total number of problematic cases. Note that we use the problematic cases to represent the case that R_adv gives the wrong suggestions, i.e., suggestions that change the original function semantics. We randomly select 80 cases, including 53 cases that MSSC reports as non-problematic and 27 cases that MSSC reports as problematic in the datasets. After manually analyzing, we find that MSSC correctly checks 67 cases, while the other 13 cases are wrongly analyzed. The overall accuracy is 84%. Out of the 13 cases, 10 cases are wrongly reported as problematic, and the other 3 problematic cases are improperly classified as non-problematic. The overall recall rate is 85%. We also analyze the reasons leading to the wrong results. Among 13 cases that MSSC wrongly analyzes, 3 cases are because the LLM changes the variable names and types during the simplification, which is unexpected and disturbs the symbol comparison. 1 case is because the LLM changes the hexadecimal numbers to corresponding characters, affecting the assignment to variables (see Appendix). Moreover, 9 cases are left due to the implementation drawback, including the analysis engine generated by the tree-sitter giving wrong parsing results, flaws in the processing of nested ternary R_ope, and the influence of the points-to problem. The above problem can be alleviated by optimizing the implementation of DeGPT.

VI. Discussion

In Section V, we conduct thorough experiments to evaluate the efficacy of DeGPT in different aspects. In this section, we continue discussing another five topics about DeGPT. To facilitate the discussion, we randomly select 50 functions for each experiment from the non-stripped datasets for evaluation in Section V and name them with *D*. Our experiments during the discussion are based on the dataset *D*.

Influence of Compiler Optimization. In Section V-A, we construct the datasets setting the compiler optimization level with "O2". In this section, we compile the dataset *D* with various compiler optimization levels to explore how compiler optimizations can affect the effectiveness of DeGPT. In detail, we compile the dataset *D* with the compiler optimization levels from "O0" to "O3". After decompilation, we get the decompiler outputs in different compiler optimization levels and test them with DeGPT after filtering out the error outputs caused by the decompiler. The results are shown in Table VII. We observe that the different metrics have advantages and disadvantages as the compiler optimization level changes. The compiler optimizations did not lead to significant differences in the results, which reflects the stability of DeGPT facing compiler optimizations.

TABLE VII: Influence of Compiler Optimization Level (COL).

COL	MVR	ER	CR	NR
O0	36.5%	79.1%	99.5%	68.1%
O1	37.9%	79.6%	99.3%	67.8%
O2	35.4%	85.6%	97.8%	50.0%
O3	43.2%	71.9%	98.8%	63.3%

Exploration on GPT-4. GPT-4, as an advanced LLM, performs better than ChatGPT on many tasks [12]. Note that DeGPT is not limited to a specific LLM. Instead, it is designed to accommodate a variety of LLMs. The study of the kind of LLM that can better drive DeGPT is not the focus of our work. However, we conduct a small-scale experiment to explore the differences between ChatGPT and GPT-4 on our optimization tasks. We leverage ChatGPT (gpt-3.5-turbo) and GPT-4 (gpt-4) for optimization on the dataset *D*, respectively, and evaluate the results with the metrics introduced in Section V-A. The results are shown in Table VIII. Depending on the measure being compared, the GPT-4-based and ChatGPT-based DeGPT each perform better and worse. Overall, there is no overwhelming advantage between the two versions of DeGPT under our optimization tasks.

TABLE VIII: GPT-4 (gpt-4) vs ChatGPT (gpt-3.5-turbo).

Model	MVR	ER	CR	NR
ChatGPT	34.9%	72.2%	99.1%	75.0%
GPT-4	39.6%	84.7%	99.4%	73.6%

Applying to Other Decompilers. As a prototype, DeGPT is implemented based on Ghidra. However, DeGPT can seamlessly switch to other decompilers. This is because DeGPT optimizes the decompiled pseudo C code of the decompiler, and the decompiled pseudo code of most mainstream decompilers today including Ghidra, Hex-rays [13], Binary Ninja [3] conforms to the syntax of the C programming language. Therefore, when optimizing the decompiler output based on other decompilers, DeGPT does not require additional modifications. Moreover, to validate the effectiveness of DeGPT on different compilers, we utilized three decompilers, namely Ghidra, Hex-rays, and Binary Ninja, to decompile the binary generated from dataset *D* with compiler optimization level "O2". We then optimize the decompiler output using DeGPT and evaluate the results using performance metrics, as shown in Table IX. The results show that even the decompilers vary, DeGPT is effective with no significant performance decrease. DeGPT can apply to various decompilers.

TABLE IX: Apply DeGPT to various decompilers.

Decompiler	MVR	ER	CR	NR
Ghidra	26.1%	76.1%	99.5%	52.7%
Hex-rays	25.0%	83.7%	99.5%	51.3%
Binary Ninja	37.3%	89.4%	98.9%	49.3%

Impact of Removing Variables Involved in Invocations. One possible worry is that the removal of variables involved in invocations may harm the original function semantics of decompiler outputs. We analyze the optimization results of the evaluation in Section V-B and find that only 14.3% of removed redundant variables participate in the function calls. These variables play a limited role in the overall structure simplification optimization performance. We further analyze these cases in detail and conclude that the LLM tends to remove the invocation-related variables similarly, i.e., replace the redundant variables in invocations with their last assigned values, which is a conservative way and can hardly destroy the original function semantics. Figure 5 in the Appendix presents two examples of the removal. The removal of `local_2c`, `sVar1`, and `sVar2` does not harm the original function semantics of decompiler outputs. Considering the decompiler outputs are human-oriented, we also communicate with experienced reverse engineers and they also agree that the removal doesn't affect the original function semantics. Just in case, DeGPT also provides an option that allows the user to disable optimization of variables involved in function calls.

Future Work. Embedding the calling context [41, 58] of the functions effectively provides additional information. We believe that providing the calling context of the functions in the proper way can optimize the performance in variable renaming and appending comments since the semantic information can have an inter-procedural propagation. Optimizing the performance of MSSC is worth exploring. The complexity of functions may lead to an exponential increase in the number of execution paths, and then affect the analysis efficiency of MSSC. A promising mitigation approach is to partition the program by slicing [64] and perform step-by-step optimization. Besides, only the modified part is analyzed during program analysis.

VII. Related Work

Focusing on recovering the semantic information to optimize binary comprehension, previous studies propose several optimization methods. Some studies try to recover information, including function names, variable types, and data structures from the stripped binary. For example, Nero [27] leverages the encoder-decoder paradigm to predict the function names of the stripped binaries and tests multiple types of encoders, including LSTM-based encoder [34], transformer-based encoder [59], and GNN-based encoder. The results boost the authors to choose the GNN-based encoder and LSTM-based decoder. However, Nero is tested on a small scale, and the dataset's size can affect the prediction's efficacy. NFRE [29] designs the instruction-level control flow graph to save the program analysis cost and make the tool suitable for large-scale training. The prediction of function names is based on the graph embeddings of the aforementioned control flow graph and a model in encoder-decoder architecture (Bidirectional LSTM network as encoder and attentional LSTM network as the decoder). DEBIN [32] first lifts the assembly code to the intermediate language, then builds the dependency graph describing pairwise and factor relationships for the later Conditional Random Field (CRF) algorithm-based prediction. Besides function names, DEBIN also recovers the information of variables. SYMLM [41] refines Microtrace-Based Pretraining [51] and involves the calling context while calculating the embeddings of the functions for the next function name prediction. TIE [45] recovers the variable types based on solving constraints generated according to the variable usage. After lifting the binary to the intermediate language BIL, it recovers the variables based on the memory access patterns. It generates the type constraints, which are solved later for type recovery. OSPREY [68] defines deterministic reasoning rules for the hints collected by BDA [69], which serves for the following probabilistic inference to recover the variable types and data structures.

While the above studies pay more attention to the analysis of assembly code or intermediate language, some other studies better leverage the ability of decompilers and analyze the decompiler output for information recovery. For example, Jaffee et al. [40] leverage the statistical machine translation technique to rename the variables in decompiler output. They propose the

alignment method between source code and decompiler output to construct the training set. Besides the lexical information involved in the last work, DIRE [44] also uses structural information recovered by the decompiler to assist the variable renaming. It uses LSTM as the encoder to extract the lexical information and Gated-Graph Neural Network (GGNN) [56] as the encoder for the structural information. DIRTY [25] further leverages the data layout of variables provided by the decompiler with the transformer-based model to predict the variable types and names.

VIII. Conclusion

In this paper, we systematically explore how to use LLM to optimize the decompiler output. To release the potential power of LLM, we design a three-role mechanism. We also propose MSSC for function semantic checking to uphold the consistency of the optimized decompile output with the original decompiled output. We evaluate DeGPT and the results show that DeGPT can achieve 24.4% reduction in the cognitive burden of understanding the decompiler outputs and provide comments of which 62.9% can provide practical semantics to help the understanding of binaries. Our user surveys also show that the optimized code significantly improves the readability and comprehensibility of the decompiler output.

Acknowledgments

We want to thank our shepherd and reviewers for their insightful comments which highly improve our paper. Thanks to Yilin Li for his efforts in experiments. The authors are supported in part by NSFC (92270204, 62302497), Youth Innovation Promotion Association CAS, and a research grant from Huawei.

References

[1] "alpaca," https://crfm.stanford.edu/2023/03/13/alpaca.html, 2023.
[2] "audioflux," https://github.com/libAudioFlux/audioFlux, 2023.
[3] "Binary ninja," https://binary.ninja, 2023.
[4] "Chatgpt," https://openai.com/blog/chatgpt, 2023.
[5] "cinspector," https://github.com/PeiweiHu/cinspector, 2023.
[6] "Codeql," https://codeql.github.com/, 2023.
[7] "Coreutils," https://www.gnu.org/software/coreutils/, 2023.
[8] "Dirt," https://cmu-itl.s3.amazonaws.com/dirty/dirt.tar.gz, 2023.
[9] "dirty," https://github.com/CMUSTRUDEL/DIRTY, 2023.
[10] "effort," https://verifysoft.com/en_halstead_metrics.html, 2023.
[11] "Ghidra," https://ghidra-sre.org/, 2023.
[12] "Gpt-4," https://openai.com/research/gpt-4, 2023.
[13] "hex-rays," https://hex-rays.com, 2023.
[14] "Leetcode," https://leetcode.com/, 2023.
[15] "llama," https://github.com/facebookresearch/llama, 2023.
[16] "llvm," https://llvm.org/, 2023.
[17] "Mirai," https://github.com/jgamblin/Mirai-Source-Code, 2023.
[18] "side effect," https://en.wikipedia.org/wiki/Side_effect_(computer_science), 2023.
[19] "tree-sitter," https://tree-sitter.github.io/tree-sitter/, 2023.
[20] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," IEEE access, vol. 7, pp. 86 121–86 144, 2019.
[21] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida et al., "Binrec: dynamic binary lifting and recompilation," in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.
[22] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," IEEE Access, vol. 8, pp. 6249–6271, 2020.
[23] A. Borji, "A categorical archive of chatgpt failures," arXiv preprint arXiv:2302.03494, 2023.
[24] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in Proceedings of the 38th Annual Computer Security Applications Conference, 2022, pp. 508–518.
[25] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 4327–4343.
[26] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in 2005 IEEE symposium on security and privacy (S&P'05). IEEE, 2005, pp. 32–46.
[27] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," Proceedings of the ACM on Programming Languages, vol. 4, no. OOPSLA, pp. 1–28, 2020.
[28] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020, pp. 1497–1511.
[29] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 607–619.
[30] M. H. Halstead, Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., 1977.
[31] I. U. Haq and J. Caballero, "A survey of binary code similarity," ACM Computing Surveys (CSUR), vol. 54, no. 3, pp. 1–38, 2021.
[32] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1667–1680.
[33] S. Heelan and A. Gianni, "Augmenting vulnerability analysis of binary code," in Proceedings of the 28th Annual Computer Security Applications Conference, 2012, pp. 199–208.
[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.
[35] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "{AURC}: Detecting errors in program code and doc-

umentation," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 1415–1432.
[36] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in Proceedings of the 26th conference on program comprehension, 2018, pp. 200–210.
[37] ——, "Deep code comment generation with hybrid lexical and syntactical information," Empirical Software Engineering, vol. 25, pp. 2179–2217, 2020.
[38] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, 2017, pp. 88–98.
[39] N. Idika and A. P. Mathur, "A survey of malware detection techniques," Purdue University, vol. 48, no. 2, pp. 32–46, 2007.
[40] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 20–30.
[41] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022, pp. 1631–1645.
[42] M. Karpinska and M. Iyyer, "Large language models effectively leverage document-level context for literary translation, but critical errors persist," arXiv preprint arXiv:2304.03245, 2023.
[43] V. Kovah, "Finding new bluetooth low energy exploits via reverse engineering multiple vendors' firmwares," BlackHat USA 2020, 2020.
[44] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 628–639.
[45] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.
[46] K. L. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in Proceedings of the 28th USENIX Conference on Security Symposium, 2019.
[47] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection," in Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, 2020, pp. 1837–1852.
[48] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in 30th USENIX Security Symposium, 2021.
[49] P. M. Newton, "Chatgpt performance on mcq-based exams," 2023.
[50] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021, pp. 833–851.
[51] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," arXiv preprint arXiv:2012.08680, 2020.
[52] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: An fmri study," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 524–536.
[53] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp. 697–710.
[54] B. C. Pierce, Types and programming languages. MIT press, 2002.
[55] P. P. Ray, "Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," Internet of Things and Cyber-Physical Systems, 2023.
[56] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," IEEE transactions on neural networks, vol. 20, no. 1, pp. 61–80, 2008.
[57] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," arXiv preprint arXiv:2301.08653, 2023.
[58] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 525–534.
[59] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.
[60] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, "Survey on malware detection methods," in Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09), 2009, pp. 74–79.
[61] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in NDSS, 2017.
[62] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution." in NDSS, 2009.
[63] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," arXiv preprint arXiv:2201.11903, 2022.
[64] M. Weiser, "Program slicing," IEEE Transactions on software engineering, no. 4, pp. 352–357, 1984.
[65] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 133–147.
[66] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2015, pp. 380–389.
[67] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in Proceedings of the 29th ACM SIGSOFT International

Symposium on Software Testing and Analysis, 2020, pp. 376–387.
[68] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: recovery of variable and data structure via probabilistic analysis for stripped binary," in 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021, pp. 813–832.
[69] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pp. 1–31, 2019.

APPENDIX

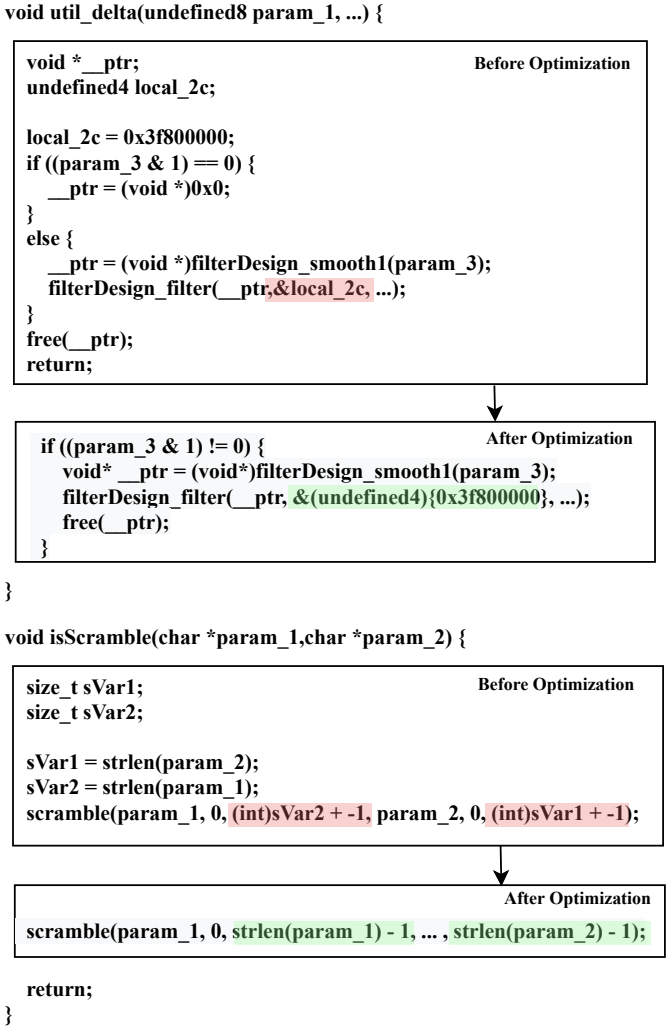

Fig. 5: Examples of removing redundant variables involved in invocations.

A - CASE OF MSSC FAILURE

As shown in Figure 6, in the original code before optimization, the assignment is done using a pointer to a hexadecimal sequence. In the optimized code, the assignment is done directly using a string literal corresponding to the hexadecimal sequence. MSSC fails to process this case.

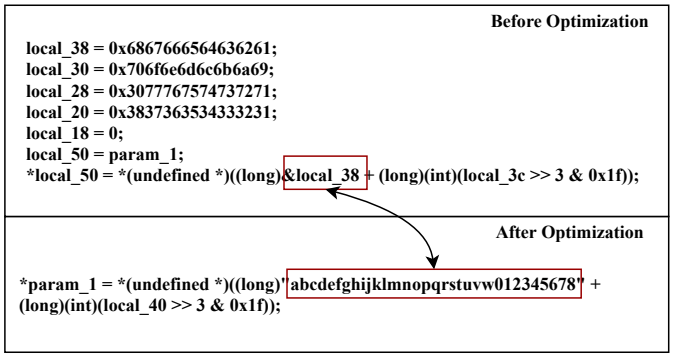

Fig. 6: LLM converts the hexadecimal sequence to string literal, which MSSC fails to analyze.

B - IMPACT OF DIFFERENT ORDERS

In Section III-C, we introduce the principles for sorting different optimizations. This section presents two experiments to show how different orders impact the optimizations. In the first experiment, we investigate the impact of performing semantic-related optimization before structure optimization by renaming variables before simplifying the structure. Specifically, we randomly select 50 functions from the non-stripped dataset and first rename the variables, followed by structure simplification. The result shows that after structure simplification, 60 out of the 313 renamed variables are removed, indicating that 19.2% of the variable renaming wastes API tokens. Note that not only variable renaming, the removal of newly appended semantic information can also happen in appending comments. Thus, putting structure simplification in prior can avoid the aforementioned removal. In the second experiment, we again randomly selected 50 functions from the non-stripped dataset to compare the differences between directly renaming variables and renaming variables after adding comments. The result shows that the MVR for direct renaming is 22.1%, while the MVR for renaming after adding comments is 28.9%. These two experiments demonstrate the rationality of our optimization prioritization. By putting structure simplification in prior, DeGPT prevents the semantic information from being removed by structure simplification, causing API token wasting. By putting appending comments ahead of variable renaming, newly appended comments boost the variable renaming.