# Refining Decompiled C Code with Large Language Models

Wai Kin Wong
The Hong Kong University of Science
and Technology
Hong Kong SAR
wkwongal@cse.ust.hk

Huaijin Wang
The Hong Kong University of Science
and Technology
Hong Kong SAR
hwangdz@cse.ust.hk

Zongjie Li
The Hong Kong University of Science
and Technology
Hong Kong SAR
zligo@cse.ust.hk

Zhibo Liu*
The Hong Kong University of Science
and Technology
Hong Kong SAR
zliudc@cse.ust.hk

Shuai Wang*
The Hong Kong University of Science
and Technology
Hong Kong SAR
shuaiw@cse.ust.hk

Qiyi Tang
Tencent Security Keen Lab
Shanghai, China
dodgetang@tencent.com

Sen Nie
Tencent Security Keen Lab
Shanghai, China
snie@tencent.com

Shi Wu
Tencent Security Keen Lab
Shanghai, China
shiwu@tencent.com

## Abstract

A C decompiler converts an executable (the output from a C compiler) into source code. The recovered C source code, once recompiled, is expected to produce an executable with the same functionality as the original executable. With over twenty years of development, C decompilers have been widely used in production to support reverse engineering applications, including legacy software migration, security retrofitting, software comprehension, and to act as the first step in launching adversarial software exploitations. Despite the prosperous development of C decompilers, it is widely acknowledged that decompiler outputs are mainly used for human consumption, and are not suitable for automatic recompilation. Often, a substantial amount of *manual effort* is required to fix the decompiler outputs before they can be recompiled and executed properly.

This paper is motived by the recent success of large language models (LLMs) in comprehending dense corpus of natural language. To alleviate the tedious, costly and often error-prone manual effort in fixing decompiler outputs, we investigate the feasibility of using LLMs to augment decompiler outputs, thus delivering *recompilable decompilation*. Note that different from previous efforts that focus on augmenting decompiler outputs with higher readability (e.g., recovering type/variable names), we focus on augmenting decompiler outputs with *recompilability*, meaning to generate code that can be recompiled into an executable with the same functionality as the original executable.

We conduct a pilot study to characterize the obstacles in recompiling the outputs of the de facto commercial C decompiler — IDA-Pro. We then propose a two-step, hybrid approach to augmenting decompiler outputs with LLMs. In particular, we first launch a static, iterative augmenting step to fix the syntax errors in the decompiler outputs using LLMs to make it syntactically "recompilable." We then launch a dynamic, memory-error fixing step with LLMs to fix the memory errors only uncoverable at runtime. The final augmented decompiler outputs can be smoothly recovered by a C compiler, resulting in a recompiled executable with the same functionality as the original executable. We evaluate our approach on a set of popular C test cases, and show that our approach can deliver a high recompilation success rate to over 75% with moderate effort, whereas none of the IDA-Pro's original outputs can be recompiled. We conclude with a discussion on the limitations of our approach and promising future research directions.

## 1 Introduction

A C decompiler recovers C source code by analyzing and converting the low-level executable files. Given the pervasive use of C in the software industry, malware authors, and its unsafe nature, C decompilers have been widely used in software reverse engineering and security analysis tasks [25, 93]. For instance, C decompilers are often used to recover the source code of legacy software for security hardening purposes [37, 38].

To date, many mature C decompilers are available on the market, including commercial tools like IDA-Pro [49] whose licenses cost several thousands of US dollars, and free ones (e.g., Ghidra) actively maintained by the open-source community or the National Security Agency (NSA) [77, 79].

Despite the prosperous development and commercialization of C decompilers, it is widely acknowledged that decompiler outputs are mainly used for human consumption, and are not suitable for automatic *recompilation* [94, 95]. Software compilation is inherently a lossy process, with many high-level information, such as variable names, type information, and data structures, no longer exists in the binaries after compilation. Accordingly, decompilers are widely designed in a pragmatic and conservative manner, where the readability of the generated code is prioritized over its "recompilability" [69], meaning to generate code that can be recompiled into an executable with the same functionality as the original executable.

Despite the fact that decompiler outputs are not suitable for automatic recompilation, recent research has emphasized the importance of automatically recompiling the decompiler outputs. For

---

*Corresponding author.

instance, the end goal of various software cross architecture migration techniques is to recompile the decompiler outputs into a binary that can run on a different architecture. Also, to reuse legacy software, it is often necessary to recompile the decompiler outputs into a binary that can run on a different operating system. More importantly, in various security instrumentation and hardening tasks, it is often necessary to instrument the decompiler outputs with security checks and recompile them into a binary with the same functionality as the original binary [39]. Nevertheless, the progress of enabling "recompilable" decompiler outputs has been slow, with the limited work focusing on rule-based approaches [69] or manual effort [74].

To bridge the gap between outputs of de facto C decompilers and the demanding recompilation requirements, this research first conduct a pilot study to investigates the challenges faced by recompiling and executing the de facto C decompiler outputs. We summarize three key challenges by analyzing the outputs of the de facto commercial C decompiler — IDA-Pro.

Accordingly, we also analyze potential obstacles faced by the state-of-the-art large language models (LLMs), when being used in an "out-of-the-box" setting to refine the decompiler outputs and deliver recompilable decompilation.

To overcome the obstacles and offer a highly-efficient solution, we propose a two-step, hybrid framework named DecGPT to augmenting decompiler outputs with LLMs. First, we employ LLMs to statically fix the syntax errors in the decompiler outputs. This forms an iterative process, where we feed the augmented decompiler outputs to the C compiler, and then use the compiler error messages, if any, to guide the LLMs to re-augment the decompiler outputs in further iterations. Second, we launch a dynamic, memory-error fixing step with LLMs. When preparing this step, we compile the augmented decompiler outputs into an executable with address sanitizer (ASAN) [89] enabled. We then run the executable with a set of test cases, and use the ASAN error messages, if any, to guide the LLMs to fix the memory errors. Our experience shows that a large chunk of subtle defects can be exposed during runtime by ASAN, and therefore, the dynamic step effectively exposes the subtle defects that cannot be exposed by the static step. The final augmented decompiler output will be a piece of C code that can be smoothly recovered by a C compiler, resulting in a recompiled executable with the same functionality as the original executable.

We evaluate DecGPT on a subset of Code Contest dataset [62], consisting of 300 test cases, and demonstrate that DecGPT can significantly increase the recompilation success rate (from the baseline setting of 45% to 75%) with moderate effort. Further ablation study shows that both the static and dynamic steps are necessary to achieve the high recompilation success rate. We interpret that the The result indicate generative AI has promising capabilities in solving fundamental challenges inherent in reverse engineering. We also discuss several promising future research that can further enhance the quality of decompilation.

In sum, we make the following contributions:

- Conceptually, this study focuses on an important research direction of "recompilable decompilation," in response to the increasing demand of re-executing decompiler outputs in security and software re-engineering tasks. Accordingly, we

for the first time explore the feasibility of using LLMs to replace previous rule-based or manual efforts to augmenting decompiler outputs.

- Technically, we design a two-step, hybrid approach to deliberately unleashing the full potential of LLMs in augmenting decompiler outputs. Our approach employs LLMs to statically fix the syntax errors in the decompiler outputs, and also dynamically fix the memory errors only uncovered at runtime.

- Empirically, our evaluation over popular C test cases shows that the decompiled C code, once automatically augmented by LLMs, can be smoothly recovered by a C compiler, resulting in a recompiled executable with the same functionality as the original executable. We also discuss promising directions to further improve the quality of decompilation.

**Artifact and Data.** We will release and maintain our artifact after the paper is officially published.

## 2 Background

To offer a self-contained paper, this section introduces the background of C decompilers and large language models (LLMs).
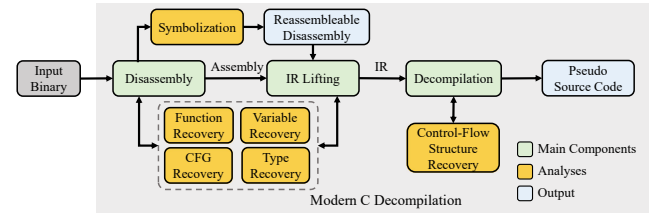
### 2.1 C Decompilation



**Figure 1: The workflow of modern C decompilers.**

Decompilation refers to the process of reconstructing pseudocode in a high-level programming language based on low-level assembly instructions extracted from binary executables. While the concept of decompilation has appeared since the mid 60s [47] for the reconstruction of Fortran source code, the foundation of modern decompilers was proposed in 1994 by Cifuentes [31, 32]. Nevertheless, despite decades of research, modern C decompilation techniques are still not perfect [69]. The difficulty of decompilation is rooted in the loss of information during the compilation process. High-level information that programmers define in source code, e.g., variables, types, and function prototypes, are discarded by compilers. Compiled binary code only operates low-level hardware resources like memory and registers. Recovering high-level abstracts from low-level assembly code is naturally undecidable [46]. To solve this problem, production-level decompilers like IDA Pro and Ghidra typically reconstruct pseudocode from input binary executable with the following steps.

**Step1: Disassembly.** First, the disassembler module of the decompiler translates the binary into assembly instructions by linearly or recursively scanning the text section of the binary [17, 81, 82]. During the disassembly process, function boundaries and prototypes are usually identified with control/data flow analyses [18, 90].

**Step2: IR Lifting.** The decompiler will then lift the assembly instructions into an intermediate representation (IR), which is deemed more analysis-friendly than assembly code since IR usually contains more high-level information like types and variables [16, 35]. Further analysis will be applied to the lifted IR for type inference and variable recovery [70]. While type information does not exist in binary code, decompilers have to rely on context to infer types.

**Step3: Code Generation.** Finally, the decompiler recovers high-level control flow structures like loops and branches and generates pseudo-source code as the output [22]. The decompiler generates control flow graphs (CFGs) based on lifted IR code. Then, with pre-defined control flow templates, the code generation module will match against the recovered CFGs and emit the pseudocode structures once a specific template is matched.

**Challenges.** As mentioned above, recovering high-level information is difficult. Specifically, since x86 assembly instructions are not inlined, and data may be embedded with code, it is non-trivial to distinguish data and code [95]. The disassembled code is thus usually not compilable and non-executable unless all symbols (e.g., code and data labels) are correctly identified. Also, due to complex compiler optimizations, e.g., different variables may share the same memory locations, and the limited scalability of data flow analysis on binary code, variable recovery and type inference is difficult [48, 83, 105]. Even state-of-the-art commercial decompilers tend to infer the wrong types [69], leading to mal-functional and not recompilable decompilation output. Besides, indirect jump instructions (e.g., `jmp rax`) are widely used by C compilers. Such indirect control flow can hardly be recovered statically, and decompilers may miss part of the CFG or the whole function [56]. Accordingly, the decompiled code may be broken and cannot be compiled into a functional binary. Due to the above challenges, modern decompilers do *not* guarantee functionality-preserving decompilation [23], and therefore, decompiled output usually cannot be used for automatic recompilation, let alone recompilation with the same functionality as the original executable.

**Recompilable Decompilation.** We present the following definition of recompilable decompilation.

> Given a binary executable $B$, a C decompiler $D$ offers recompilable decompilation if the decompiled output $O$ can be automatically recompiled by standard C compilers (e.g., `gcc`) into a functional binary $B'$ with the same functionality as $B$.

We clarify that recompilable decompilation is largely under-explored due to its high complexity. Instead, recent works have made promising results on another relevant pre-step, namely, re-assembleable disassembly. Uroboros [95] first proposed a heuristics-based method to recover symbols in binaries. Rambler [94] extends the heuristic rules on this basis of Uroboros. Followingly, Superset and probabilistic disassembly [19, 75] conservatively treat each address as a potential instruction start to make the disassembled code runnable. The latest solutions, Retrowrite [39] and Egal-ito [99], leverage the auxiliary information presented in position-independent code (PIC) to achieve recompilable disassembly. However, they cannot be applied to non-PIC code, and perfect reassembleable disassembly is still not solved for general executables.

Recompilable decompilation, on the other hand, greatly improves the user's ability to modify binary, that is, the user can directly insert high-level language code in the binary instead of writing assembly code and consequently can benefit a range of security-related applications, including binary instrumentation, binary hardening, and legacy software immigration. In the recompilable decompilation line of research, one recent work leveraging rule-based methods to fix the decompiler outputs to make them executable [69]; we present more details in Sec. 3. Similarly, [87] performs partial recompilation by rule based approach for automatic program repairing, and [74] recompiles binaries by manually fixing the decompilation output. As highlighted by Mantovani et al. [74], it takes 90 minutes to 8 hours for experienced analysts to fix compilation errors when compiling with the pseudocode, showing the difficulty of recompilable decompilation.

## 2.2 Large Language Model (LLM)

LLMs such as GPT-4 [80] and Llama2 [91] have emerged as powerful AI assistants, demonstrating remarkable capabilities across diverse tasks. The massive amounts of training data, as well as techniques such as Reinforcement Learning from Human Feedback (RLHF) [30] and Direct Preference Optimization (DPO) [86], have enabled these models to better understand and respond to human instructions, making it possible to interact with them more naturally. The interaction with LLMs typically begins with a prompt, which serves as the input to guide the model's response. Given such an input prompt, an LLM will output a probability distribution for the next token over its vocabulary. After a token is selected based on a decoding strategy like beam search or top-k sampling [50], the new token is appended to the prompt and used for the next token prediction until a special token indicates the end of the sequence.

The performance of LLMs benefits significantly from well-designed prompts that provide relevant context. It has been shown that task definitions alone [68] (also known as zero-shot learning) can often achieve satisfying results, and performance can be further improved by including more concrete examples in the prompt [97]. Prompt engineering techniques have been proposed to improve the performance of LLMs on specific tasks by carefully designing the input prompt. Among prompt engineering techniques, Chain-of-Thought (CoT) has been shown to elicit stronger reasoning from LLMs by asking the model to incorporate intermediate reasoning steps (rationales) when solving a problem [61, 63, 96, 98].

LLMs have manifested their potential and served as "domain experts" in a variety of applications, including code generation, code completion, and code summarization [33, 88]. In our work, we focus on leveraging LLMs to augment decompiler outputs with recompilability. We will present the technical details in Sec. 4.

## 3 Motivation

In this section, we discuss the research motivation behind this work from the perspectives of research demands, challenges, and technical feasibility.

### 3.1 Demanding in Producing "Recompilable" C Decompiler Outputs

The proposed approach is designed to augment the decompiler outputs of C programs to make them recompilable by standard C

compilers. The target, namely "recompilable" decompiler outputs, is highly demanding in practice. Overall, various real-world security applications and software re-engineering tasks not only rely on the decompiler outputs, but also require the automated recompilation of the decompiler outputs. For instance, in the context of vulnerability patching, security analysts expect to patch the decompiler outputs by inserting new and safe code snippets, and then recompile the patched code to generate a new executable for production usage. Nevertheless, the current decompiler outputs are not recompilable, and extensive manual effort is usually required to fix the outputs before they can be recompiled and reused.

We thus argue that the decompiler outputs, whose design goal is to serve human comprehension, are far from sufficient for many of today's production applications. This research strives to address the gap by augmenting the decompiler outputs with LLMs to make them recompilable.

From another perspective, the proposed approach particularly focuses on the augmentation of C decompiler outputs. The main reasons are twofold. First, C is among the most popular and widely used programming languages, and supporting the augmentation of C decompiler outputs can benefit a large number of real-world, production scenarios. Moreover, C is an unsafe language, and various security applications(e.g., vulnerability detection [24, 66], patching [58, 72, 87], legacy code hardening [39, 75, 99]), as well as malware analysis [92, 103], are particularly targeting C executables.

Second, C decompilations are well-known as a challenging problem in the research community, whereas recent advances in commercial decompilers like Ghidra and IDA-Pro have made it possible to generate decompiler outputs with high functionality accuracy [69]. We thus believe that this study is timely, addresses an emerging research problem, and can benefit a large number of real-world applications.

## 3.2 Challenges in Recompilation with Outputs from Decompiler

Following the introduction in Sec. 2.1, we note that prior rule-based work suffers from various limitations. For example, Liu et al. [69] enable recompilable decompilation when grammar and types are restricted, yet they can hardly be applied to general cases. Reiter et al. [87] circumvented this limitation through partial recompilation combined with binary rewriting, achieving state-of-the-art results in software hardening. However, this approach does not retain a complete copy of recompilable source code. To understand difficulties in rule-based automatic recompilation, we randomly selected 100 C/C++ submissions from the Google Code Jam (GCJ) [6] dataset collected between 2009 and 2020 and investigated the root causes of failure when recompiling with (pseudo-)source code derived from decompiler outputs.[1] While decompiler usually supports exporting output as source files, it is mostly impossible to directly recompile these files due to issues such as undefined symbols, as outlined in [69]. In order to avoid these issues, the aforementioned study restricted the number of functions existing in the input program to

one. However, this assumption does not apply to real-world programs, where it is common for programmers to define multiple functions for solving challenges. Therefore, we expanded upon the rules used in [69] and outlined as follows :

① In our attempts to recompile the decompilation outputs, we found that the pseudocode consists of undefined ELF binary-specific symbols, such as "_gmon_start_" and "__cxa_finalize". These functions are added during compilation to support the runtime.

② The output from decompilers usually contains security checks like stack canary checks and array size checks added during compilation. We removed such checks from the decompiled pseudocode, as they do not impact the program logic or data structure needed for recompilation.

③ Fixing minor errors that affect compilation, such as removing the "fastcall" from the function declaration and fixing the function type of "main."

④ Fixing the namespace and header based on source code.

**Result.** After applying the aforementioned rules, only 9 out of the 100 samples can be recompiled. By executing the recompiled executables and checking the outputs with the test inputs shipped by the GCJ dataset, we see that all the 9 samples are functionally equivalent to the original binaries. Nevertheless, the remaining 91 samples generated a total of 2677 compilation errors. Upon analyzing these errors, we summarize the following three root causes of compilation failures:

① Specification error: This type of error usually happens in the code generation phases of the decompiler. While the decompilers' signature system may correctly identify function calls, the final decompiled code does not match API specifications. A typical error comes from the call towards `stdin` and `stdout`. We observe that decompilers usually break a function call toward these IO interfaces into several function calls toward other undefined operators, resulting in undefined function errors when compiling the decompiled code. Another similar example is the function call towards "`std::ios_base::sync_with_stdio.`" While it only consumes a single boolean type variable, the pseudocode generated from decompilers consists of other auxiliary variables.

② Inference error: During compilation, some information crucial for humans to understand the program is discarded. Decompilers have to infer such information according to the assembly instructions. However, such a process is error-prone; decompilers may fail to infer syntactical information that is necessary for recompilation. Fig. 3 illustrates an example of the type inference error.

As shown in Fig. 3a, the actual type of the array is `const int`. However, it is inferred as `uint32` by the decompiler (shown in Fig. 3b). As a result, due to the presence of negative numbers in the array, it triggered the "narrowing conversion" when compiled with default compilation options. Another common error originates from the inference of array size. As shown in Fig. 4a, the array size has been hardcoded to 20. Moreover, we can see the array size cannot be inferred by the decompiler (see Fig. 4b line 1) and triggered the "storage size of isn't known" error. As the array access index `i` is determined by the user input "N" (`dword_202288` of the decompiled output), the decompiler fails to infer the size of the array with range analysis. Hence, the array size has been left blank in the decompilation outputs.

---

[1]To mimic real-world scenarios where executables are stripped before releasing, we strip symbols from binaries used in this work. We also excluded binaries with trivial decompilation errors, which usually are stack pointer related issues.

```
1   mov rax , rsp
2   mov [ rbp + var_F8 ], rax
3   lea rax , [ rbp + var_94 ]
4   mov rsi , rax
5   ...
6   call std :: cin
7   ...
8   lea rsi , [ rax -1]
9   mov [ rbp + var_60 ], rsi
10  ...
```

**Figure 2: Assembly instruction.**

```
1   const int fx [] = {-1, 0, 0, 1};
2   const int fy [] = {0, -1, 1, 0};
```

**(a) Source code.**

```
1   uint32 dword_2F60 [4] = {-1, 0, 0, 1};
2   uint32 dword_2F70 [4] = {0, -1, 1, 0};
```

**(b) Decompiler's output.**

**Figure 3: Example of type inference error.**

③ Error from decompiler templates. As outlined in Sec. 2.1, decompilers' output is generated based on control flow templates. We observe that some register loading patterns can induce false positives in the template matching process, resulting in syntactical problems in the generated pseudocode. We illustrate an example in Fig. 5, where the decompiler generates lines (3, 6) that do not exist in the source code in Fig. 5a. By examining the corresponding assembly instructions (Fig. 2), we identify that the decompiler incorrectly interprets certain stack variable movements (i.e., mov [rbp+OFFSET], <REGISTER> in lines 2-9 in Fig. 2) as array accesses. Consequently, lines (3, 6) are erroneously generated in Fig. 5b, leading to invalid conversion errors when recompiling with the decompiler's output.

Our manual study shows that specification errors are the most common causes of recompilation failure, which account for 80.1% of the total errors. This can be attributed to the extensive usage of stdin and stdout in programming contests. Furthermore, although decompilers incorporate signature databases (e.g., IDA-Pro's Lumina database [15]) to identify function calls, they often rely on rule-based approaches for the generation of the decompiled code. However, the presence of diverse combinations of compilers, optimization passes, and operating systems in real-world scenarios pose a significant challenge in creating universally-applicable rules that are completely error-free. It is worth noting that the errors originating from decompilation templates account for less than 1% of the total errors, highlighting the efforts invested by decompiler developers in striving for near-perfection in their rules.

## 3.3 Feasibility and Challenge of using LLMs to Deliver Recompilable Decompilation

*3.3.1 Feasibility of LLMs.* To understand challenges of using LLMs for recompilation, again we performed an empirical study with 100 randomly sampled programs from the GCJ dataset. We reported that, the average lines of code and tokens of the dataset is 110.7

```
1   int v [20];
2   ...
3   cin >> E >> R >> N;
4   ...
5   for (int i = 0; i < N; i ++){
6       cin >> v[i];
7       ...
8   }
```

**(a) Source code snippets.**

```
1   int dword_2022A0 [];
2   ...
3   cin >> dword_202280 >>
4       dword_202284 >> dword_202288;
5   ...
6   for (j = 0; j < dword_202288; ++j){
7       cin >> dword_2022A0[j];
8       ...
9   }
```

**(b) Decompiler's output with minor cleanup.**

**Figure 4: Example of array size inference error.**

```
1   std::cin >> cases;
2   for (int x = 0; x < cases ; x ++){
3       int n;
4       std::cin >> n;
5       int m[n][n];
6       int vestigium = 0;
7   }
```

**(a) Source code snippets.**

```
1   cin >> v24 ;
2   for ( i = 0; i < v24 ; ++ i ){
3       v23[1] = v23;
4       cin >> v25;
5       v38 = v25 - 1LL;
6       v23[6] = v25;
7   }
```

**(b) Decompiler's output with minor cleanup.**

**Figure 5: Example of decompiler template error.**

and 1082.1 respectively. See Sec. 5.2 for dataset selection criteria and Sec. 4.1 for LLMs selection.

With the profound abilities of LLMs, security researchers have been adopting LLMs to their reverse engineering pipelines [3, 7, 9, 102]. Current LLMs aided reverse engineering pipelines often adopts a bottoms-up approach to analyzing binary functions, which has achieve promising results in various tasks like annotating the decompiler's outputs, predicting function names, and so on. However, this approach may overlook dependencies across functions, which is necessary for LLMs to fix inference error by interpret code-level semantics. Therefore, we include the complete outputs

from decompiler in each query to the LLM. We interacted with ChatGPT in this way until $N$ iterations or successfully recompiling the test cases. $N$ is configured as 15 per our prelminary study, and we present other configurations at this step in Sec. 5.1.[2]

Notably, previous works have demonstrated that the provided context can significantly influence the output generated by LLMs. Methods such as few-shot learning [97] and Chain-of-Thought (CoT) [98] have been proposed to improve performance on complex reasoning tasks. However, these techniques are not directly applicable to our recompilation task, since they typically require specialized context samples to be supplied, which is difficult to obtain for general recompilation scenarios. Therefore, to maintain reproducibility while preserving effectiveness, inspired by prior works on using LLMs for automated program repair (APR) [100] we designed a conversational approach with a carefully designed system prompt (shown in Table 1) to guide the model to correct the decompiled outputs iteratively. The zero-shot method avoids reliance on specific training samples, while the system prompt is crafted to focus the model on the recompilation task. This allows our approach to be applied in a generalizable manner across different decompilation outputs.

To be specific, the system prompt is designed with the three considerations. First, we provide explicit requirements for the generated outputs to ensure GPT do not generate outputs for another programming languages nor using Windows C convention, for example using `wmain` as program's entry point. Second, we re-emphasize the scope to GPT, to avoid the model neglecting the function entry point (i.e. `main`) and results in compilation failures. Third, we clearly stated the dos and don'ts for recompilation to the model. This avoid the model to only return explanations or directly copy the "goto" syntax from the decompiler outputs, without understanding the semantics. For user prompt, we formatted the decompiled pseudocode with inline code formatting in Markdown. This is a well-known prompt engineering technique [1, 4] to indicate our pseudocode from the inputted prompts and providing cues to the expected outputs from the model.

For handling compilation errors and output errors, we use the same system prompt with extra messages appended before the problematic code, as outlined in Table 1.

*3.3.2 Challenges of LLMs.* With aforementioned settings, 42 out of the 100 samples can recompile successfully. To understand the remaining obstacles, we manually analyzed all failed samples and identified the three root causes of failure:

**Limited Context Length.** Despite explicit instructions to Chat-GPT, it occasionally "forget" the instructions given by our input prompts. This can manifest in providing explanations of functionalities or returning mal-formed pseudocode, etc. Overall, context length is a common weaknesses of LLMs in solving long inputs [20, 54]. It frequently happens regardless of the configurations or the training methodologies of the model [10, 11, 67].

**Repairing with Shortcuts.** Shortcut learning behavior [44] refers to a phenomenon where a decision performs well on benchmarks but fails to generalize or address the underlying problems. This issue has been observed across various learning-related tasks [29, 34, 84].

---

[2]As a fair setup, we do not obtain information like headers or namespace from source code. Instead, we use LLM to infer this.

```
1        v14 &= byte_2022A0[2048 * m + n];
```

**(a) ChatGPT's output.**

```
1        v14 |= byte_2022A0[2048 * m + n];
```

**(b) Decompiler's output.**

**Figure 6: Example of hallucination error.**

Upon analyzing failure samples, we have identified similar issues in ChatGPT. Specifically, when encountering type-related errors, ChatGPT tends to utilize casting operators like "static_cast" or "reinterpret_cast" to wrap statements. While this approach may appear reasonable initially, it often results in memory corruption during testing. Additionally, ChatGPT may attempt to fix compilation errors by removing related code fragments or functions. While this may seem like a reasonable action to eliminate compilation errors, it leads to the loss of contextual information in the decompiler outputs and ultimately results in recompilation failure. This behavior mirrors a similar issue observed in the field of Automated Program Repair (APR) [85], where plausible patches generated by APR tools overfit towards specific test suites used for patch synthesis, failing to address the root cause of defects. These observations highlight the tendency of LLMs, similar to APR, to generate patches that are tailored specifically to the provided conditions, potentially limiting the generalizability and robustness of the recompilation process.

**Hallucination.** Typically, hallucination refers to LLMs generating content that deviates from real-world facts learned during training [51]. Although factuality may not be the primary objective of generated code, similar phoenomenon is observed in the decompilation process, wherein critical semantics of the code are wrongly altered by LLMs. Specifically, when a long sequence of code tokens are fed to the LLM, the model may return the results that slightly change the original code tokens. An interesting example of such hallucination is shown in Fig. 6, where the bitwise operator &= has been replaced with |= after two conversations with LLMs. Since the variable v14 is used as a status flag in the following conditional statement, this minor defect results in a significant change in the data flow, leading to wrong answer returns from the recompiled program.

## 3.4 Distinction with Previous Works Augmenting Decompiled Code

Below discussing the technical details, we first clarify the following key differences between our work and previous works in related fields. In short, our focus is to using LLMs to fix the decompilation errors such that the decompiled code can be smoothly recompiled by standard C/C++ compilers, whose output is functionally equivalent to the original binaries. In other words, we do not aim to "beautify" the decompiled code, but to make it automated recompilable. We have clarified the demanding of recompilation in Sec. 3.1.

In contrast, another line of research aims at augmenting decompiled code to make it more readable. Often, such works do not directly improve the result correctness, but try to recover various high-level program information and enhance the readability for
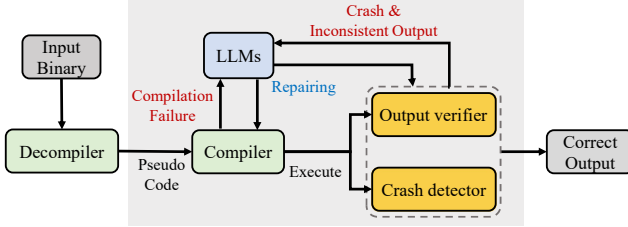
**Table 1: Prompt templates designed for our preliminary studies.**

| | Prompt |
|---|---|
| System Prompt | *"Generate linux compilable C/C++ code of the main and other functions in the supplied snippet without using goto, fix any missing headers. Do not explain anything."* |
| Compilation Error | *"Please fix the following compilation errors in the source code: {compiler_error} {pseudocode}"* |
| Output Error | *"The expected output of the program for input: {expected_input} is {expected_output}, but we got {wrong_output}. Please fix the issue in the source code: {pseudocode}"* |

humans, e.g., an anti-malware expert can read the enhanced decompiled code more easily if the variable names are precisely recovered and very meaningly.

To this end, DIRE [57] and DIRTY [27] try to improve the readability of decompiled code by assigning decompiled variables meaningful names and types with NLP models. Nero [36], on the other hand, generates function names for decompiled code. Similarly, NFRE [43] and SymLM [53] try to predict function names for stripped binaries. Such works do not improve the result correctness but ease humans and LLM to comprehend. Moreover, Debin [48] and CATI [26] predict debugging information from binaries without decompilation. A recent work, LmPA [102] combines program analysis with LLM to predict names of decompiled functions. As aforementioned, this paper is *orthogonal* with the above works, as we study the automatic recompilation of decompiled code into functional output, which is rarely studied in the literature except those few papers listed in the **Recompilable Decompilation** paragraph of Sec. 2.



**Figure 7: The workflow of DecGPT.**

## 4 DecGPT: Enabling Automated Recompilable Decompilation with LLMs

In line with the preliminary studies presented in Sec. 3, we see the promising technical potential of LLMs in assisting the recompilation process. LLMs behave as "human experts" to handle the tedious, costly and error-prone manual effort in fixing decompiler outputs. Given that said, Sec. 3.3 has illustrated that such LLM-based approach is by no means trivial. In this section, we present our approach, DecGPT, that offers a two-step, hybrid pipeline to augment decompiler outputs with LLMs.

Fig. 7 depicts the workflow of DecGPT. We first launch a static, iterative augmenting step to fix decompiler outputs and make them compilable by standard C compilers (details in Sec. 4.2). Then, we launch a dynamic, memory-error fixing step to fix the memory errors detected during runtime (see Sec. 4.3). The final augmented

decompiler outputs can be smoothly recovered by C compilers to generate a semantics correct executable. Before we dive into the details of our approach, we first introduce the application scopes and other setups below.

### 4.1 Application Scopes and Setup

**Decompiler Selection.** The designed DecGPT aims to augment C/C++ decompiler outputs with LLMs to make it re-compilable by standard C/C++ compilers. We have clarified the design focus on C/C++ decompilers in Sec. 3.1. Given that said, our technical pipeline is not limited to C/C++ decompilers. We believe that our approach can be easily extended to decompilers for different programming languages; we present further discussions in Sec. 7. In this study, we focus on the de facto commercial C decompiler — IDA-Pro [49]. IDA-Pro is an extensively used C/C++ decompiler that domains both industrial and academic usages.

**LLM Selection and Setup.** In this study, we investigate the potential of LLMs as a black box for assisting the recompilation process. We specifically focus on GPT-3.5 [8], a readily accessible LLM model, and interact with it automatically through its API, "gpt-3.5-turbo-0613". Similar to its predecessor, GPT-3 [20], GPT-3.5 undergoes pre-training on a large corpus of web data and subsequent fine-tuning using Reinforcement Learning from Human Feedback (RLHF)[30], as outlined in Sec. 2.2. This enables the model to generate responses to user queries.

Although GPT-4 [80], the successor of GPT-3.5, is considered to be the most powerful LLM available in the wild, GPT-3.5 still offers several practical advantages. First, in terms of the cost of querying the model, our observation shows that GPT-3.5 is about 20 times more cost-effective based on current pricing structures [13]. Second, GPT-3.5 provides at least 2 times faster response [14] than GPT-4. Therefore, GPT-3.5 is a more suitable and cost-effective LLM for our study. Given that said, our technical pipeline is not limited to a certain LLM, and it is possible to use other LLM models as replacements. We present further discussions in Sec. 7.

### 4.2 DecGPT— Static Augmenting

The static augmenting step aims to fix the grammatical errors and inference errors in the decompiler outputs and make the outputs compilable by standard C/C++ compilers. Given a piece of decompiler output *o*, we first compile the output with a standard compiler *C* (in our implementation, we use the standard GCC compiler). If *C* fails to compile *o*, we then launch the following static augmenting step.

**Table 2: Designed prompt templates for DECGPT.**

|  | Prompt |
|---|---|
| System Prompt | *"Generate linux compilable C++ code of the main and other functions in the supplied snippet without using goto, fix any missing headers and reducing the number of intermediate variable. Only reply the fixed source code. Do not explain anything and include any extra instructions, only print the fixed source code."* |
| Compilation Error | *"Please fix the following compilation errors in the source code: {compiler_error} {pseudocode}"* |
| Output Error | *"The expected output of the program for input: {expected_input} is {expected_output}, but we got {wrong_output}. Please fix the issue in the source code: {pseudocode}"* |
| ASAN Error | *"Please fix the {type_of_memory_corruption} triggered in {statement}: {pseudocode}"* |

① **Initial Prompting.** The static augmenting step is an iterative process. With default decompiler outputs, we apply preprocessing rules ① to ③ as documented in Sec. 3.3.2. This removes unnecessary tokens from the pseudocode and helps prevent hallucination originated from these. Expanded upon the settings in Sec. 3.3.1, we instruct the model to reduce redundant variable assignments found in the decompiler outputs (as illustrated in Table 2). By doing this, we are striving to reduce output length and also the chance of hallucination if the source snippet is required for further repairing with LLM. With the above steps, we prepare an initial prompt for LLM to fix the inputs.

② **Post Processing.** Although we specified the desired output format in our prompts, our testing has revealed that ChatGPT sometimes produces partially mal-formed outputs that include code explanations. Using these outputs directly for recompilation often leads to compilation errors. To address this issue, we create a set of rules to cleanup the output $o$. After processing, we re-compile $o$ with the standard C/C++ compiler.

③ **Processing Error Message.** If $o$ fails to be compiled, we collect the compiler error messages $E$ during the compilation. Then, we first preprocess $E$ to remove the irrelevant information, e.g., the line numbers and file names, with scripts. We then feed the preprocessed error messages $E$ with certain lines that consist of the errors into the LLM to generate a new token sequence $T$.

④ **Repairing Decompiled Output $o$.** At this step, we then tokenize the decompiler output $o$ into a sequence of tokens $T$. Then, we prepare a prompt $P$ to be fed into the LLM. The prompt $P$ instructs the LLM to fix the token sequence $T$ and generate a new token sequence $T'$.

⑤ **Iterative Augmenting.** The fixed token sequence $T'$ is then fed into the compiler $C$ to re-compile the repaired output $o'$. If $C$ again fails to compile $o'$, we firstly iterate over each function in $o'$ and check if certain function's function body is being stripped; if so, we revert the changes. We then repeat the above process from ① until $C$ successfully compiles $o'$, or the number of iterations exceeds a pre-defined threshold $N$. In our current implementation, we set $N$ to 15 based on our preliminary experiments.

The above static augmenting step aims to fix the grammatical errors and inference errors in the decompiler outputs and make the outputs compilable by standard C/C++ compilers. However, the outputs may still contain various memory errors and functional errors. In other words, even if the output of this static repairing phase is "re-compilable", its induced executable may still crash or produce obviously incorrect results. To address this issue, we launch the following dynamic repairing step.

## 4.3 DECGPT— Dynamic Repairing

**Motivation and Design Consideration.** The dynamic repairing step aims to fix the functional errors in the decompiler outputs. While this appears to be a straightforward and demanding task, our tentative exploration shows that precisely pinpointing and fixing various functionality errors in the decompiler outputs is a very challenging task, whose technical solutions are still in their infancy. Note that performing automated bug detection in C code is very challenging, let alone we are dealing with decompiler outputs. Similarly, automatically "patching" the detected bugs in C code is also a very challenging task. While the community has made some encouraging progress on LLM-based bug detection and patching, our preliminary study shows that these approaches are not directly applicable to our problem.

Thus, this research takes a pragmatic approach. Instead of somehow instructing the LLMs to search and fix arbitrary functional errors in the decompiler outputs, we focus on fixing the memory errors in the decompiler outputs. Memory errors are particularly common in C programs, and are often the root cause of functional errors. Moreover, the C/C++ community has developed full-fledged tools to detect and fix memory errors, with the help of memory address sanitizers (ASAN) [89]. As a result, we target on leveraging the existing tools to detecting memory errors in the decompiler outputs, and then instructing the LLMs to fix the detected memory errors. We aim to present such meaningful benchmarking results to the community, and leave exploring technical solutions to fix arbitrary functional errors in the decompiler outputs as future work.

**Approach.** At this step, we profile the compiled executable $E$ with a set of test cases $T$. Moreover, we configure the C compiler to inject address sanitizers into $E$ during the compilation stage. This way, whenever $E$ contains subtle memory errors, the address sanitizers shall faithfully detect and report them during the runtime profiling phase. With the memory detection results, we then launch the following dynamic repairing step.

⑥ **Collecting Memory Error Information.** We first collect the memory error information $I$ when one of the address sanitizer injected in $E$ alarms on the program input $t$. The memory error

information $I$ includes the address, the erroneous instruction, the register values in the context, and the stack trace. This information is well-formed by address sanitizers, and can be easily parsed for analysis using scripts.

⑤ **Repairing Defects.** We then prepare a prompt $P$ to be fed into the LLM. The prompt $P$ instructs the LLM to fix the memory error information $I$ originated in the test case $t$. The LLM then generates a new token sequence $T'$.

⑤ **Testing Functionality Equivalence.** If test case $t$ does not result in an alarm, we will verify the output of $E$ against the expected output. If the output is correct, we proceed to the next test case. If the output is incorrect, we prepare a prompt $P$ and instruct the LLM to fix the functional defects *at our best effort*; again, we believe directly fixing functional defects is a very challenging task, and our main focus is on the above memory error fixing. Overall, we then repeat the above process from step ⑥ until $E$ passes all the test cases or fails any test cases in $T$, and instruct the LLM to fix the identified defects.

## 5 Evaluation Setup

As previous discussed in Sec. 4.1, we implement DECGPT for supporting the decompilation outputs for IDA-Pro [49], a commercial and well tested decompiler. DECGPT is primarily written in Python and C++, with about 2504 lines of code. All experiment are conducted on a Ryzen 3970X 32-core server with 256GB memory. Below, we introduce the evaluation setup.

### 5.1 Model Setting

We limited our selection of programs based on the maximum token length limit of our meployed GPT-3.5 model, which is 4096 for the GPT version "gpt-3.5-turbo-0613". Notably, this limit includes both the input and output parts of the program. As such, to ensure that the model had enough capacity to generate outputs, we only selected programs where the total number of tokens in the decompiled pseudocode and the system prompt was less than half of the input token length limit. This is because the length of the recompiled code is roughly equivalent to the length of the input decompiled pseudocode, and our system prompt had fewer tokens. By doing this, we were able to generate complete recompiled programs and avoid any errors that may have been caused by truncation, which could have affected the experimental results.

### 5.2 Datasets

In our evaluation, we use the Code Contest dataset [62], which is a diverse collection of submissions obtained from five online judge platforms. Each problem in the dataset consists of multiple test cases used to validate the correctness of the submissions, with the "AC" (Accepted) verdict indicating that all test cases were passed successfully. In addition, we specifically selected submissions with AC verdicts for our evaluation as it indicates that the corresponding code submissions have successfully passed all the provided test cases. This enables us to assess the correctness of the recompiled binaries based on the supplied test cases, and ensure the reliability of the evaluation results.

To address the uncertainty about the source code in the training set of ChatGPT, we took precautions in our evaluation. We only considered submissions made after September 2021, which is the

knowledge cutoff of ChatGPT [12]. Also, context length plays a crucial role in the performance of LLMs. To account for this, We divided the dataset into five equal intervals based on the context length and randomly selected 60 programs from each interval. This gave us a dataset of 300 programs. We manually verified that there is no trivial decompilation errors presence in the decompilation outputs, and we reported that the average lines of code and tokens is 112.9 and 1110.4, respectively.

### 5.3 Evaluation Metrics

Since the recompilation process enabled by DECGPT is an iterative process, we use the length of the conversation chain (denoted as $C$) required to correctly recompile decompiler's output as our primary evaluation metric. Due to the cost, we evaluated with $C = 1, 5, 10$ and 15. We deem a success if the recompiled binaries passed all test cases, without triggering errors.

## 6 Evaluation

**Research Questions.** Our evaluation aims to answer the following research questions:

**RQ1**: How effective is DECGPT in solving the challenges of rule-based recompilation?

**RQ2**: How effective is DECGPT in alleviating the challenges of LLM-based recompilation?

**RQ3**: How much does the iterative design contribute to the performance of DECGPT.

**Table 3: Recompilation success rate for 3 different tools.**

|  | DECGPT | DECRULE | *llm − baseline* |
|---|---|---|---|
| **Success Rate** $C = 1$ | 37% | 8% | 32% |
| **Success Rate** $C = 5$ | 55% | 8% | 39% |
| **Success Rate** $C = 10$ | 69% | 8% | 42% |
| **Success Rate** $C = 15$ | 75% | 8% | 45% |

### 6.1 RQ1: Comparison with Rule-Based Recompilation

To evaluate the effectiveness of DECGPT in dealing with the challenges of rule-based recompilation, we compared it with the settings we used in Sec. 3.2; to ease presentation, we denote the setting used in Sec. 3.2 as DECRULE. To ensure a fair comparison with DECGPT, we added the header and namespace extracted from the original source code to the decompilation results. Note that in the case of DECGPT, these information is expected to be inferred by LLM.

As shown in Table 3, DECRULE can successfully recompile 8% of binaries from the testing dataset, while DECGPT can recompile 75% of them. This comparison highlights the effectiveness of DECGPT in addressing the challenges associated with rule-based recompilation.

To understand the types of errors that DECGPT is capable of fixing, we use the compilation errors generated when using DECRULE as a reference and manually examine the conversation chain between the LLM and DECGPT. Our analysis reveals that about 98% of the specification errors are being fixed by DECGPT within two conversation rounds, regardless of whether the test case can be recompiled or not. We believe the findings are reasonable. To generically fix specification errors, heavy-weighted binary analyses

like symbolic execution are usually employed to infer the function prototype [28]. In contrast, LLMs can fix these errors relatively easily by replacing the syntactically incorrect code block with the correct one. This is possible because typical APIs and function prototypes are widely represented in the training sets used for training the LLMs or can be inferred from the compilation errors submitted to the LLMs.

With our design, DECGPT can fix 62.5% of inference error and all of the identified decompiler template errors. By leveraging decompiled pseudocode as input and using compilation errors as feedback, DECGPT iteratively replace erroneous segments within the decompiler outputs. In one of the successful recompiled cases, we found that the initially decompiled output contains a type inference error, where a variable's integer type is incorrectly inferred as an unsigned integer type. Subsequently, this variable and another correctly inferred variable are used to invoke the max function. While this results in compilation errors, we find that with two iterations of conversation, DECGPT can successfully fix this issue by precisely replacing the unsigned integer type to the integer type. Similar situation also happens for LLM to fix the decompiler template errors, which usually results in type conversion failure during compilation.

However, DECGPT may not be able to fix all of the inference errors. Out of the remaining 37.5% of samples, we found that 70.2% of them come from the Standard Template Library (STL) functions. STL is an important part of the C++ programming paradigm and offers reusable containers and algorithms. Note that the identification of STL functions is generally considered difficult [5], in the sense that decompiled code of the same STL function with different types can be significantly different. In our study, we observed that the presence of STL function calls tends to increase the context length of the decompiler outputs from 2.8 times to 16.1 times. This poses a challenge for LLMs, as the performance of LLMs is highly dependent on the context length [67]. However, we also view this as a challenge from decompilations, where the decompiled pseudocode may not accurately represent the used STL functions, thereby hindering LLMs in fixing syntactical errors.

The remaining errors all come from ill-inferred buffer sizes in different contexts. While buffer size can be indirectly fixed through ASAN error messages, lacking of accurate buffer size information still posing challenges for LLMs to successfully recompile the binaries. We consider this as a challenge for both LLMs and the decompiler, due to the fact that the performance of LLMs highly depends on the quality of its inputs (i.e., decompiler outputs).

**Table 4: Number of success cases in terms of varying context lengths across different settings.**

|             | DECGPT | $llm - baseline$ | DECGPT Zero |
|-------------|--------|------------------|-------------|
| **200 - 560**   | 57 | 48 | 58 |
| **560 - 920**   | 54 | 45 | 50 |
| **920 - 1280**  | 48 | 24 | 10 |
| **1280 - 1640** | 39 | 9  | 5  |
| **1640 - 2048** | 27 | 6  | 2  |

## 6.2 RQ2: Comparison with LLM-Based Recompilation

We now evaluate DECGPT to understand its effectiveness in addressing the challenges of LLM-based recompilation that we identify in Sec. 3.3.2 (referred to as $llm - baseline$). Compare to $llm - baseline$, DECGPT has made advancements in several aspects. First, DECGPT extends the system prompt for enhancing the overall stability of the recompilation process. Second, a structural level sanity check is added to the source code returned by LLM, and ensure the input to the model in next iteration includes all the necessary information for recompiling a binary. Third, we compile the binaries with ASAN enabled. This allows us to extract the line of statement that triggers the memory corruption when binary crashes. We then use this information as part of the prompt for LLM to generate a version of the output that fixes the problems. Tables 3 and 4 show the success rate and the number of successful cases per interval for these two settings. Overall, DECGPT outperforming $llm - baseline$ by 30% in terms of success rate in recompiling the same set of binaries; it is seen as more capable in handling decompiler outputs with long context. We interpret the results as follows.

**Effectiveness of the Extended System Prompt.** To evaluate the effect of the extended system prompt in DECGPT, we refer to the result of $C = 1$ in Table 3. In this case, the success rate is measured on binaries successful recompiled without extra repairing query. Compare to the result of $llm - baseline$, it is easy to observe an immediate increase in the success rate from 32% to 37%, when the system prompt is extended. In other words, we interpret that by extending the system prompt, DECGPT effectively imposes more constraints on the LLM and helps the LLM to concentrate on fixing decompiler outputs and aligns more closely with the desired output. These contribute to the improved performance without additional iterations.

**Effectiveness of Static Augmenting.** DECGPT outperforms $llm - baseline$ when recompiling outputs with long context lengths. As shown in Table 4, DECGPT successfully re-compiles 4.3 times and 4.5 times more decompiler outputs than that of $llm - baseline$ for context lengths between 1280 to 1640 and 1640 to 2048, respectively. This suggests that DECGPT can effectively alleviate the challenges of LLM-based recompilation for longer context lengths.

This notable difference in performance is attributed to the synergy effect of extended system prompt and static augmenting. When examining the success cases within the two aforementioned context length intervals, we found that DECGPT is capable of detecting and reverting a total of 194 instances where LLM recklessly strips function bodies during the fixing process. DECGPT effectively preserves the integrity of the decompiler outputs by avoiding these undesirable modifications, and ensures that necessary context can pass to the next iteration of fixing. As a result, DECGPT demonstrates better performance than $llm - baseline$ in recompilation with long context inputs.

**Effectiveness of Dynamic Augmenting.** The design of dynamic augmenting is to address functional incorrectness that cannot be resolved through static augmenting alone. To evaluate the success of fixes provided by the dynamic augmenting of DECGPT, we manually reviewed 50 cases where DECGPT successfully fixed a runtime

errors, and we categorized these fixes into three categories. The two major categories are output formatting errors and minor algorithmic errors, which accounted for 48% and 42% of the sampled cases, respectively. Output formatting errors typically involve issues related to the structure or formatting of the output, like missing a newline after each output. As for minor algorithmic errors, they typically include small mistakes such as a reversed plus/minus sign or reversed program logic. These problems can be fixed by LLMs when the correct output is provided as part of the user prompt. The remaining 10% of the fixed cases belongs to memory corruption issues that stem from inference errors in the decompiler outputs.

**Limitations of Dynamic Augmenting.** An important part of the dynamic augmenting is the use of ASAN to detect memory corruption issues. Out of the 55 ASAN instances triggered during runtime, DecGPT is able to fix 40% of them. We believe the results as reasonable and generally encouraging, showing the potential of incorporating ASAN (whose outputs are documents of severe memory errors in a well-structured format) with LLM to augment the decompiled outputs.

We further investigate the cause of the unfixable ASAN errors by manually examine the intermediate conversation between LLM and DecGPT. We categorized the cause of unfixable ASAN into two types. The major cause of unfixable ASAN errors belongs to the usage of type casting operators (like "static_cast" and "reinterpret_cast") when LLM attempted to fix type conversion errors. It contributes to 81.8% of the unfixable ASAN errors. Despite attempts to explicitly disallow the usage of these type casting operators in the system prompt or user prompt, LLM continued to employ them as the fixing strategy. This behavior may be attributed to the shortcut learning behavior discussed in Sec. 3.3.2. However, the underlying reasons and appropriate mitigations for this behavior are left as future work, as they are beyond the scope of the current paper. The remaining 18.2% of unfixable ASAN errors are caused by errors associated with hallucination, like wrong buffer size being infered by LLM, as we discussed in Sec. 6.1.

## 6.3 RQ3: Ablation Study

To understand the contribution of the iterative and two-phase design to the performance of DecGPT, we run DecGPT without either the static augmenting or dynamic augmenting mentioned in Sec. 4, and we denote this setting as DecGPT Zero. Similar to DecGPT, we use this setting to query LLM 15 times, and we consider the recompilation a success if any of the outputs returned from LLM passes all of the test cases (using the same criteria as DecGPT).

The results are shown in Table 4. Generally, DecGPT has clear advantage over DecGPT Zero, especially for handling decompiler outputs with long context. For the context length between 200 to 560 and 560 to 920, we observe a comparable performance between DecGPT and DecGPT Zero. The result is reasonable, given that the average number of iterations requires for DecGPT to fix the decompiler output in these two intervals are 1.24 and 2.13, respectively. With such a small number of iterations, the performance of DecGPT and DecGPT Zero are similar.

However, for the remaining intervals, DecGPT Zero shows a notably lower success rate than the iterative design. This is reasonable: with an increase in context length, more functions are

present in the decompiler outputs, which implies that more syntactical errors may be present in the decompiler outputs as well. Another technical obstacle for LLMs in handling long context is the performance degradation [67] and information forgetting problems [11]. With the iterative design, DecGPT can fix the errors in a step-by-step manner by providing sufficient context for LLMs to resolve existing syntactical errors present in decompiler outputs or errors introduced by LLMs in previous iterations. In contrast, DecGPT Zero only has one chance to fix the errors, and as a result, DecGPT Zero is more likely to suffer from the obstacles mentioned above. These results suggest that the iterative design is a key factor in the performance of DecGPT.

## 7 Discussion

**Extension to Other Decompilers.** DecGPT is evaluated to augment the outputs of IDA Pro, the de facto commercial C/C++ decompiler that is extensively used in industry and research. Nevertheless, we believe the proposed approach is independent of the underlying C/C++ decompiler. As illustrated in Sec. 4, in the static augmenting stage, we leverage error messages from the compiler to refine the decompilation output. Consequently, DecGPT requires the decompiler to output pseudo source code that is close to the source code consumed by compilers. To our observation, this prerequisite is consistent with the concept of software decompilation, and is generally satisfied by mainstream decompilers that are available on the market. Besides, given that the incredible power of LLM is not limited to the output of any specific decompiler, we envision that LLM is able to read and understand the decompilation output of other mainstream C/C++ decompilers, and also decompilers of other programming languages. DecGPT, therefore, is expected to be extended to different decompilers without showing a significant difference in effectiveness.

**Validity of Using The Current LLM.** One potential threat to the validity of our study is whether DecGPT can be applied to other LLMs. To ensure generality, we deliberately refrain from conducting specific hyperparameter tuning in our research. Instead, we evaluate DecGPT using the default configuration of GPT-3.5. This eases the reproducibility of our work and enhances the generalizability of our findings. Additionally, we conduct a comprehensive evaluation of each component of DecGPT in Sec. 6 to assess their effectiveness. Based on this analysis, it should be accurate to conclude that DecGPT is not limited to specific LLMs. Furthermore, we perform our empirical study (in Sec. 3) and evaluation (in Sec. 6) on two separate datasets, both of which demonstrate comparable performance. This indicates that the performance of DecGPT is not restricted to a specific dataset.

## 8 Related Work

We have reviewed the general workflow of software decompilation and recent progress in decompilation, particularly on recompilation, in Sec. 2. Below, we review other research works that are relevant to this paper.

**Automatic Program Repair (APR).** The goal of APR is to fix defeats in software automatically. Generate-and-validate is a typical approach of APR by synthesizing patches base on the fault localization results. A patch will be consider as plausible patch if it passes

all of the test case and fix the issue. In short, patches are usually synthesized by three main approaches, namely search-based approach [45, 71], template-based approach [55] and semantic-based approach [64, 72]. By leveraging LLMs's capabilities in suggesting code snippets base on user's prompts [59], recent study [41, 52, 100, 101] has demonstrated encouraging results in applying LLMs to APR. In short, by carefully expressing localized defective code fragment or failure as prompts, it is shown that LLMs generates potential patches through conversational prompting. In contrast, this study employs LLMs in a hybrid pipeline to fix decompiler outputs, a well-known challenging problem in reverse engineering that has many distinct characteristics from APR of source code.

**Dynamic Binary Rewriting.** Binary rewriting refers to the process of changing the semantics of a program executable file without access to its source code. Most of its applications are in the field of software security, such as defeating code reuse attack, software obfuscation, and software watermarking. While we mainly discuss static binary rewriting in Sec. 2, dynamic rewriting techniques have also developed significantly in the past few decades. Dynamic rewriting techniques are performed during the runtime, often on the basis of system-level support like Intel Pin [73], DynamoRIO [21], and Valgrind [76]. Those tools intercept program execution at the runtime with an external or grafted process. Dynamic binary rewriting advantages itself by enabling modification of all code without restrictions suffered by static methods [39, 40]. However, it incurs a high overhead because the cost of rewriting occurs at runtime. In contrast, this work's focus lies in static binary rewriting, which is ahead of program execution and is more efficient. The success of our proposed approach shall further extend the capability of static binary rewriting methods, i.e., enabling smoothly instrumenting a piece of binary code with modest cost as how source code is easily modified.

**Large Language Model for Code** In recent years, LLMs have made significant strides in enhancing various code-related tasks. Commercial models such as Github Copilot [2] and OpenAI GPT-4 [80] have demonstrated impressive performance, while several open-source models have been developed using large-scale code corpora.

Incoder [42], for instance, employs a causal masking training objective to excel in code infilling and synthesis, while CodeGen [78] is a pre-trained model for multi-turn program synthesis with over 16B parameters. The BigCode Project has also contributed to the development of StarCoder [60], an open-source model with 15.5B parameters. More recently, Code-Llama [88], a code-specialized version of Llama 2, was created by further training on code-specific datasets and sampling more data from the same dataset for longer context. Apart from LLMs that focus on source-code level tasks, models have been developed for low-level code. For example, CodeCMR [104] and IRGEN [65] are pre-trained models designed for low-level code on various code-related tasks. Additionally, researchers [33] have attempted to train a 7B-parameter model from scratch on low-level code to optimize LLVM assembly for code size. Overall, these LLMs have demonstrated impressive performance in various code-related tasks, with potential for further advancements in the field.

## 9 Conclusion

This research is motivated by the high demand of delivering recompilable decompilation in security and software re-engineering tasks. Moreover, we are also motivated by the recent major success of LLMs in comprehending dense corpus of both natural language text and programs. To bridge the gap between outputs of de facto commercial C/C++ decompilers and the demanding recompilation requirements, we propose a two-step, hybrid framework named DecGPT to augmenting decompiler outputs with LLMs. Evaluations show that DecGPT can significantly increase the recompilation success rate with moderate effort. We conclude with a discussion on promising future research directions on top of DecGPT.

## References

[1] [n.d.]. ChatGPT Prompt Engineering for Developers. https://www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers/.
[2] [n.d.]. Copilot. https://github.com/features/copilot.
[3] [n.d.]. DAILA. https://github.com/mahaloz/DAILA/.
[4] [n.d.]. Examples and guides for using the OpenAI API. https://github.com/openai/openai-cookbook/.
[5] [n.d.]. Find the C++ STL functions in a binary. https://reverseengineering.stackexchange.com/a/3890.
[6] [n.d.]. gcj-dataset. https://github.com/Jur1cek/gcj-dataset.
[7] [n.d.]. Gepetto. https://github.com/JusticeRage/Gepetto/.
[8] [n.d.]. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.
[9] [n.d.]. gpt-wpre. https://github.com/moyix/gpt-wpre/.
[10] [n.d.]. How Long Can Open-Source LLMs Truly Promise on Context Length? https://lmsys.org/blog/2023-06-29-longchat/.
[11] [n.d.]. LLM forgetting part of my prompt with too much data. https://community.openai.com/t/244698.
[12] [n.d.]. OpenAI Models overview. https://platform.openai.com/docs/models/overview.
[13] [n.d.]. OpenAI Pricing. https://openai.com/pricing.
[14] [n.d.]. Please be kind and helpful in conversations! GPT-3.5 and GPT-4 API response time measurements. https://community.openai.com/t/237394.
[15] 2023. IDA-Pro Lumina. https://www.hex-rays.com/lumina/.
[16] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[17] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An {In-Depth} Analysis of Disassembly on {Full-Scale} x86/x64 Binaries. In *25th USENIX security symposium (USENIX security 16)*. 583–600.
[18] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code *(USENIX Security)*.
[19] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.
[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[21] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003*. IEEE, 265–275.
[22] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 353–368.
[23] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *31st USENIX Security Symposium (USENIX Security 22)*. 2765–2782.
[24] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[25] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.
[26] Ligeng Chen, Zhongling He, and Bing Mao. 2020. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 88–98.

[27] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.

[28] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 677–693.

[29] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[30] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[31] Cristina Cifuentes. 1994. *Reverse compilation techniques*. Queensland University of Technology, Brisbane.

[32] Cristina Cifuentes and K. John Gough. 1995. Decompilation of Binary Programs. *Softw. Pract. Exper.* 25, 7 (July 1995), 811–829.

[33] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large Language Models for Compiler Optimization. *arXiv preprint arXiv:2309.07062* (2023).

[34] Nikolay Dagaev, Brett D Roads, Xiaoliang Luo, Daniel N Barry, Kaustubh R Patil, and Bradley C Love. 2023. A too-good-to-be-true prior to reduce shortcut reliance. *Pattern Recognition Letters* 166 (2023), 164–171.

[35] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. 2020. Scalable Validation for Binary Lifters.

[36] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[37] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS*.

[38] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *PLDI*.

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

[40] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 151–163.

[41] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *ICSE*.

[42] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).

[43] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. 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*. 607–619.

[44] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. 2020. Shortcut learning in deep neural networks. *Nature Machine Intelligence* 2, 11 (2020), 665–673.

[45] Claire Le Goues, Thanhvu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.

[46] Keerthana Gurushankar and Pulkit Grover. 2021. A minimal intervention definition of reverse engineering a neural circuit. *arXiv preprint arXiv:2110.00889* (2021).

[47] Mark I Halpern. 1965. Machine independence: its technology and economics. *Commun. ACM* 8, 12 (1965), 782–785.

[48] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *CCS '18*.

[49] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.

[50] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[51] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. (2023).

[52] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *International conference on software engineering (ICSE)*.

[53] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. 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*. 1631–1645.

[54] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[55] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[56] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level.. In *NDSS*.

[57] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *ASE*.

[58] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*.

[59] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *ICSE*.

[60] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[61] Shiyang Li, Jianshu Chen, Yelong Shen, Zhiyu Chen, Xinlu Zhang, Zekun Li, Hong Wang, Jing Qian, Baolin Peng, Yi Mao, et al. 2022. Explanations from large language models make small reasoners better. *arXiv preprint arXiv:2210.06726* (2022).

[62] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[63] Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. 2022. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336* (2022).

[64] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[65] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*. 2253–2265.

[66] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[67] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[68] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* (2023).

[69] Zhibo Liu and Shuai Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020).

[70] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1100–1119.

[71] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[72] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[73] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[74] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery–A Case Study *(AsiaCCS)*.

[75] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1187–1198.

[76] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[77] Lily Hay Newman. 2019. The NSA makes Ghidra, a powerful cybersecurity tool, open source. https://www.wired.com/story/nsa-ghidra-open-source-tool/.

[78] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model

for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[79] National Security Agency (NSA). 2018. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.

[80] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023).

[81] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. 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, 833–851.

[82] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. 2022. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX Security 22)*. 2479–2495.

[83] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702.

[84] Mohammad Pezeshki, Oumar Kaba, Yoshua Bengio, Aaron C Courville, Doina Precup, and Guillaume Lajoie. 2021. Gradient starvation: A learning proclivity in neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 1256–1272.

[85] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015).

[86] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290* (2023).

[87] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doup'e, Ruoyu Wang, and Stephanie Forrest. 2022. Automatically Mitigating Vulnerabilities in x86 Binary Programs via Partially Recompilable Decompilation. *ArXiv* abs/2202.12336 (2022).

[88] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[89] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker *(USENIX ATC'12)*. 28–28.

[90] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX security symposium (USENIX Security 15)*. 611–626.

[91] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[92] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147.

[93] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-scale Empirical Study on Mobile App Obfuscation. In *ICSE*.

[94] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[95] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. 627–642.

[96] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. 2022. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747* (2022).

[97] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[98] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[99] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[100] Chun Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *ArXiv* abs/2301.13246 (2023).

[101] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[102] Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2023. LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis. *arXiv preprint arXiv:2306.02546* (2023).

[103] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 52–63.

[104] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[105] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832.