DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code

WAI KIN WONG, Hong Kong University of Science and Technology, Hong Kong
DAOYUAN WU*, Hong Kong University of Science and Technology, Hong Kong
HUAIJIN WANG, Ohio State University, USA
ZONGJIE LI, Hong Kong University of Science and Technology, Hong Kong
ZHIBO LIU*, Hong Kong University of Science and Technology, Hong Kong
SHUAI WANG*, Hong Kong University of Science and Technology, Hong Kong
QIYI TANG, Tencent Security Keen Lab, China
SEN NIE, Tencent Security Keen Lab, China
SHI WU, Tencent Security Keen Lab, China

Decompilers are widely used in reverse engineering (RE) to convert compiled executables into human-readable pseudocode and support various security analysis tasks. Existing decompilers, such as IDA Pro and Ghidra, focus on enhancing the *readability* of decompiled code rather than its *recompilability*, which limits further programmatic use, such as for CodeQL-based vulnerability analysis that requires compilable versions of the decompiled code. Recent LLM-based approaches for enhancing decompilation results, while useful for human RE analysts, unfortunately also follow the same path.

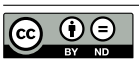
In this paper, we explore, for the first time, how off-the-shelf large language models (LLMs) can be used to enable *recompilable decompilation*—automatically correcting decompiler outputs into compilable versions. We first show that this is non-trivial through a pilot study examining existing rule-based and LLM-based approaches. Based on the lessons learned, we design DecLLM, an iterative LLM-based repair loop that utilizes both static recompilation and dynamic runtime feedback as oracles to iteratively fix decompiler outputs. We test DecLLM on popular C benchmarks and real-world binaries using two mainstream LLMs, GPT-3.5 and GPT-4, and show that off-the-shelf LLMs can achieve an upper bound of around 70% recompilation success rate, i.e., 70 out of 100 originally non-recompilable decompiler outputs are now recompilable. We also demonstrate the practical applicability of the recompilable code for CodeQL-based vulnerability analysis, which is impossible to perform directly on binaries. For the remaining 30% of hard cases, we further delve into their errors to gain insights for future improvements in decompilation-oriented LLM design.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**.

Additional Key Words and Phrases: Recompilable Decompilation, Reverse Engineering, Large Language Model

*Corresponding authors.

Authors' Contact Information: Wai Kin Wong, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, wkwongal@connect.ust.hk; Daoyuan Wu, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, daoyuan@cse.ust.hk; Huaijin Wang, Ohio State University, Ohio, USA, wang.18964@osu.edu; Zongjie Li, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, zligo@cse.ust.hk; Zhibo Liu, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, zliudc@cse.ust.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, 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.



ACM 2994-970X/2025/7-ARTISSTA081
https://doi.org/10.1145/3728958

ACM Reference Format:
Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA081 (July 2025), 24 pages. https://doi.org/10.1145/3728958

1 Introduction

In reverse engineering, decompilers play a vital role in helping analysts understand a binary. A decompiler recovers source code by analyzing and converting low-level executable files. Due to the pervasive use of C in the industry, its use by malware authors, and its unsafe nature, C decompilers are widely used in software reverse engineering [32, 52, 78, 80, 111, 115]. For example, C decompilers are often used to recover the source code of legacy software for security hardening [40, 41, 105, 106].

To date, many mature C decompilers are available on the market, including commercial tools like IDA Pro [54], whose licenses cost thousands of US dollars, and free ones (e.g., Ghidra) actively maintained by the open-source community or the National Security Agency (NSA) [84, 86]. Despite the prosperous development and commercialization of C decompilers, it is widely acknowledged that decompiler outputs are mainly intended for human consumption and are not suitable for automatic *recompilation* [112, 113]. Software compilation is inherently a lossy process, with much high-level information, such as variable names, type information, and data structures, no longer existing in the binaries after compilation. Accordingly, decompilers are often designed in a pragmatic and conservative manner, where the *readability* of the generated code is prioritized over its *recompilability* [76]—the decompiled code can be recompiled into an executable with the same functionality as the original.

Nevertheless, there is a growing emphasis on the importance of automatically recompiling decompiler outputs in recent research [76, 96]. For example, the end goal of various software cross-architecture migration techniques [23, 42, 46, 92] is to recompile the decompiler outputs into a binary that can run on a different architecture. Also, to reuse legacy software, recompiling the decompiler outputs into a binary that can run on a different operating system is often necessary. More importantly, various security instrumentation and hardening tasks benefit from instrumenting the decompiler outputs with security checks and recompiling them into a binary with the same functionality [43]. Nevertheless, progress in enabling "recompilable" decompiler outputs has been slow, with limited work focusing on manual effort [80] or rule-based approaches [76].

This work strives to address the gap by repairing decompiler outputs to make them recompilable. We believe this is a timely study that tackles an emerging research problem and can benefit many real-world applications, including enabling vulnerability analysis using CodeQL. However, this task is non-trivial, even with recent advances in LLMs [88]. We first conduct a pilot study in §3 that investigates the existing rule-based approach [76] and LLM-based approaches [55, 103] using the outputs of the leading commercial decompiler—IDA Pro 8.3. Our results show that manually defined heuristics are insufficient to address a wide range of compilation errors, and LLM-based approaches also suffer from issues like attention degradation, shortcuts, and hallucination. Despite these challenges, we observe potential in using LLMs to assist in recompilation, as the recompilability rate increased to 25% with the LLM version when provided with the entire program context, compared to 9% in the raw rule-based approach.

Further investigation of the pilot study results suggests that LLMs can repair decompiled code when provided with compilation error messages and expected outputs. This motivates us to supply both static recompiling and dynamic runtime feedback (i.e., compilation errors and mismatched runtime outputs) to LLMs to enhance recompilation repair. Such feedback should be strategically scheduled in an iterative loop, giving the LLM multiple chances to fix decompilation errors.

Based on this intuition, we design DecLLM, an iterative LLM-based repair loop that utilizes both static recompiling and dynamic runtime feedback as oracles to iteratively fix decompiler outputs. Specifically, for static repair, DecLLM feeds the decompiler outputs under repair to the C compiler, using any compiler error messages to guide LLMs in subsequent iterations. For dynamic repair, DecLLM aligns the expected outputs with those obtained from running test cases on the original executables and uses error messages from the address sanitizer (ASAN) [99] to guide LLMs in fixing subtle defects that the static step cannot expose.

We first evaluate DecLLM on 300 test cases from a popular C benchmark, the Code Contest dataset [67]. The results show that DecLLM can significantly increase the recompilation success rate, raising the rate from only 8% in rule-based recompilation, 10% in a recent readability-oriented LLM-based approach [55], 17.3% in a non-iterative LLM-based approach [103], to 74% for GPT-3.5 and 78.3% for GPT-4. Further evaluation shows that both the static and dynamic steps are necessary to achieve a high success rate.

We also investigate the possibilities of using DecLLM to recompile 108 real-world binaries from the Coreutils benchmark [1] on decompiler outputs. The results are slightly lower than those for the Code Contest dataset, but still achieve 48.1% for GPT-3.5 and 55.5% for GPT-4. We also demonstrate that while CodeQL cannot directly analyze binary or decompiled code, DecLLM revives its capability with recompilable C code, achieving promising vulnerability detection results.

Considering both the results from the Code Contest and Coreutils benchmarks, we conclude that with DecLLM, off-the-shelf LLMs can achieve an upper bound of recompilation success rate at an average of around 70%. We further analyze the remaining 30% of hard cases, delve into their errors, and obtain insights that could be useful for future decompilation-oriented LLM design.

Contributions. In sum, we make the following major contributions in this paper:

- Conceptually, we study the important topic of "recompilable decompilation" and explore the feasibility of using LLMs to replace the previous rule-based approach for repairing decompiler outputs; see our pilot study examining existing rule-based and LLM-based approaches in §3.
- Technically, we design an iterative LLM-based repair loop, DecLLM, to effectively harness the potential of LLMs in repairing decompiler outputs; see §4. Our approach combines static recompiling and dynamic runtime feedback, enabling LLMs to deliver recompilable decompilation.
- Our evaluation across different settings, datasets, and two decompilers (IDA Pro and Ghidra) shows highly encouraging results; see §5, §6, and §7. We also demonstrate DecLLM's applicability in downstream tasks, such as bridging executable vulnerability analysis with CodeQL.

2 Background

2.1 C Decompilation

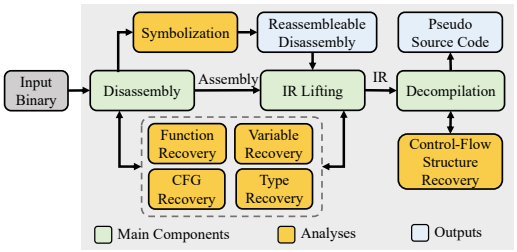


Fig. 1. The workflow of modern C decompilers.

Decompilation [36, 37, 51] refers to reconstructing pseudocode in a high-level programming language based on low-level assembly instructions extracted from binaries. Despite decades of research, modern C decompilation techniques are still not perfect [76]. The difficulty of decompilation is rooted in the loss of information during compilation. High-level information that programmers define in source code, e.g., variables, types, and function prototypes, is 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 [50]. As illustrated in Fig. 1, decompilers typically solve this problem with the following steps.

Step 1: Disassembly. The disassembler module of the decompiler translates the binary into assembly instructions by scanning the text section of the binary [24, 90, 91]. During disassembly, function boundaries and prototypes are usually identified with control/data flow analyses [26, 101].

Step 2: 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 [23, 38]. Further analysis will be applied to the lifted IR for type inference and variable recovery [77]. While type information does not exist in binary code, decompilers have to rely on context to infer types.

Step 3: Code Generation. The decompiler generates control flow graphs (CFGs) based on lifted IR. 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 [30].

Challenges. As mentioned above, recovering high-level information is difficult. Specifically, since x86 assembly instructions are not aligned, and data may be embedded with code, it is non-trivial to distinguish data and code [113]. 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 [53, 94, 123]. Even state-of-the-art (SOTA) commercial decompilers tend to infer the wrong types [76], 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 [63]. 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 [31], and therefore, decompiled output usually cannot be used for automatic recompilation, let alone recompilation with the same functionality as the original executable.

Definition 1 (Recompilable Decompilation). *Let $\mathcal{B}$ be the set of all binary programs, $\mathcal{D}$ be the set of C decompilers, and $\mathcal{C}$ be the set of standard C compilers. A decompilation $D \in \mathcal{D}$ is recompilable if:*

$$\forall B \in \mathcal{B}, \exists O \in \mathcal{L}_C, \exists C \in \mathcal{C}, \exists B' \in \mathcal{B} : \tag{1}$$

$$O = D(B) \wedge B' = C(O) \wedge (\forall i \in \mathcal{I}, \forall s \in \Sigma : \delta_B(s, i) \cong \delta_{B'}(s, i)) \tag{2}$$

where $\mathcal{L}_C$ is the set of all valid C programs, $\mathcal{I}$ is the set of all possible inputs, Σ is the set of all possible program states, $\delta_B : \Sigma \times \mathcal{I} \to \Sigma$ is the state transition function for B, and $\cong$ denotes state equivalence.

We clarify that recompilable decompilation remains a largely under-explored area due to its inherent complexity. In contrast, recent advancements on reassembleable disassembly [27, 43, 81, 112–114] have shown promising results, enabling disassembled code to be reassembled into functional binaries. Recompilable decompilation, on the other hand, offers a significant advantage by allowing users to insert high-level language code directly, rather than writing assembly, largely simplifying the process of modifying and redeveloping binaries. While reassembleable disassembly has already demonstrated visible achievements in the field of binary security [43, 109, 114, 116], recompilation decompilation is poised to become the superior alternative by obtaining higher-quality reverse engineering results and, consequently, benefiting a wide range of security-related applications, including binary instrumentation, hardening, and legacy software migration. For instance, recompilable code can enable more efficient compiler-based instrumentation compared to dynamic binary instrumentation tools, which often incur considerable overhead [29, 79, 82, 83].

Furthermore, recompilable decompilation has the potential to bridge the gap between binary-level and source-level analysis. This opens up the exciting possibility of analyzing obscure binary code with well-established and powerful source code analyzers. By leveraging tools from traditional compilation chains, recompiled binaries can be transformed and optimized more effectively. These advancements hold the potential to significantly propel the field of binary analysis forward, paving the way to new extensions and applications that were previously unattainable.

In this area, recent work has used rule-based methods to fix decompiler outputs and make them executable [76] (see details in §3). [96] performs partial recompilation for binary patching, and [80] recompiles binaries by manually fixing decompiler outputs. As highlighted by Mantovani et al. [80], it takes experienced analysts 90 minutes to 8 hours to fix decompiler outputs, showing the difficulty of recompilable decompilation.

2.2 LLMs and Their Current Role in Decompilation

LLMs have recently demonstrated superior performance in various software engineering tasks [57, 58, 69, 70, 98, 122]. Typical LLMs, such as GPT-4 [87] and Llama2 [104], are pre-trained and fine-tuned on massive amounts of data. To interact with LLMs, users provide a *prompt* as input, and the model outputs a probability distribution over its vocabulary for the next token. This process continues until a special token indicates the end of the sequence, forming the *response* to the user.

With recent advances in LLMs for code synthesis and reasoning tasks [62, 71, 72, 98, 120], researchers have also adopted LLMs for the binary decompilation domain. The basic paradigm is to generate better, more semantically correct, and human-understandable decompiled code, without necessarily requiring it to be compilable. Notably, DeGPT [55] is a pioneering work in this category, utilizing off-the-shelf LLMs as multiple agents to refine decompiler outputs for better readability. Nova [59] further pre-trains a series of dedicated LLMs based on DeepSeek-Coder [49] for various binary tasks, including binary decompilation. Similarly, LLM4Decompile-Ref [103], fine-tunes DeepSeek Coder [49] to directly generate decompiler outputs with better re-executability. Besides these representative works, readers may refer to §8 for more details.

While existing LLM-based approaches are beneficial to human reverse engineering (RE) analysts, our pilot study in §3 will show that they are not yet ready to achieve recompilable decompilation. For example, both Nova and LLM4Decompile-Ref attempt to produce dedicated binary models to directly generate reliable decompilation results in a one-time manner. However, our findings in §3 indicate that an iterative approach, as we propose in §4, is necessary for recompilable decompilation. Given that off-the-shelf LLMs like GPT-4 and GPT-3.5 have better generic reasoning capabilities for reflective tasks, this work explores to what extent they can enable recompilable decompilation.

3 A Pilot Study

In this section, we conduct a pilot study to demonstrate that recompilable decompilation is a non-trivial task by examining existing rule-based (§3.1) and LLM-based (§3.2) approaches. We further summarize their pitfalls and, based on the lessons learned, explore a vanilla LLM approach in §3.3 to demonstrate the potential of using off-the-shelf LLMs for recompilable decompilation.

3.1 Limitations of the Rule-Based Approach

Existing efforts to enable automatic recompilation of decompiler outputs typically use the heuristic rule-based approach, as demonstrated in the studies conducted by Liu et al. [76] and Reiter et al. [96]. While decompilers usually support exporting output as source files, it is mostly impossible to directly recompile these files due to issues such as undefined symbols, as outlined in [76]. To facilitate the study, we extend the rules used by Liu et al. [76] to support the recompilation of multiple functions. (We outline our changes at [22].)

Dataset. We randomly selected 100 C/C++ samples from the Google Code Jam (GCJ) submissions collected between 2009 and 2020 as the dataset for the pilot study. We then investigated the root causes of failures when recompiling with (pseudo-)code derived from decompiler outputs. We stripped symbols from binaries and excluded binaries with trivial decompilation errors, such as stack pointer-related issues and failures in function symbolization.

Result. Eventually, only 9 out of the 100 samples could be recompiled. By checking the outputs of recompiled executables, we observed that all 9 samples are functionally equivalent to the original binaries. Nevertheless, the remaining 91 samples generated 2,677 compilation errors. Upon analyzing these errors through error message parsing and rule-based classification, we identified the following three root causes of compilation failures, with samples of each type of error available at [22]:

① *Specification Error*: This type of error usually occurs during the code generation phases of the decompiler. While the decompiler may correctly identify function calls, the decompiled code does not match API specifications. A typical error comes from calls to `stdin` and `stdout`. This can potentially be attributed to errors in function signature recovery [95].

② *Inference Error*: Some information crucial for humans to understand the program is discarded during compilation. Decompilers have to infer them from binaries. However, this process is error-prone; decompilers may fail to infer syntactical information for recompilation. For example, a `const int` array may be misinterpreted as `uint32`, triggering a "narrowing conversion" error.

③ *Error from Decompiler Templates*: As outlined in §2.1, decompiler output is generated based on control flow templates. Some register loading patterns can induce false positives in the template matching process, resulting in syntactical problems in the generated pseudocode.

> **Takeaway**: Manually defined heuristics are insufficient to address a wide range of compilation errors. To achieve full-scale recompilation, a system with the intelligence to identify and fix defects is required.

3.2 Challenges of Existing LLM-Based Approaches for Recompilable Decompilation

One challenge with using the rule-based approach in §3.1 is its lack of flexibility in synthesizing fixes for errors. Therefore, we assess the potential of recently proposed LLM-based approaches that are orthogonal to our work, notably DeGPT [55] and LLM4Decompile-Ref [103] (see §2.2), in enabling recompilable decompilation. To ensure a fair comparison with the rule-based approach, we evaluate the same dataset used in §3.1. This dataset includes programs with an average of 110.7 lines of code and 1,082.1 tokens (see §5.2 for details on dataset selection).

Result. DeGPT and LLM4Decompile-Ref automatically recompiled 5 and 15 of the 100 test samples, respectively. We analyze the failed samples to identify obstacles in these techniques.

① *Lack of Inter-procedural Context*: Both frameworks refine functions independently[1], potentially causing conflicts when combining the results, such as inconsistent global variable renaming. Furthermore, LLM4Decompile-Ref erroneously redefines globals as locals, causing functional deviations. While post-processing the result from DeGPT can mitigate some of these issues, we cannot prevent LLM4Decompile-Ref from transforming global declarations to locals, even when explicitly supplying declarations. This is likely due to the specialized fine-tuning [19].

② *Hallucination*: Hallucination in LLMs refers to generating non-factual content [56]. For example, LLM4Decompile-Ref might mistakenly convert raw pointers to nonexistent struct types

[1]LLM4Decompile-Ref's authors recommend refining in per-functions basis [20]. Our preliminary exploration reveals that when multiple functions are supplied, it randomly returns only one of them.

and misinterpret function calls as `printf` statements, treating all arguments as strings. Such misinterpretations lead to compilation errors or incorrect behavior.

③ *Attention Degradation*: LLM4Decompile-Ref might occasionally produce malformed pseudocode, such as repeating the same tokens until it reaches the output limit. We attribute this to the attention degradation problem, which occurs when LLMs processing long inputs [28, 61], regardless of LLM configurations or training methodologies [15, 16, 73].

3.3 Potential of Using Off-the-Shelf LLMs for Recompilation

The results in §3.2 show that per-function refinement cannot fully address challenges in rule-based recompilation. However, we observe that LLMs can flexibly transform decompiler outputs and overcome the limitations of rule-based methods. Motivated by this, we designed a zero-shot system [64] to assess the potential of vanilla LLMs when provided with the entire program context.

We design our system prompt by referring to open-source projects [10, 11] and the user guide from OpenAI [9]. To be specific, the prompt is designed with three considerations. First, we provide explicit requirements for the generated outputs to ensure the LLM does not generate outputs for other programming languages or use Windows C conventions. Second, we re-emphasize the repairing scope to the LLM to avoid neglecting the function entry point and resulting in compilation failures. Third, we carefully defined constraints to guide the LLM's responses to prevent LLM from only returning explanations or directly copying the "goto" syntax from decompiler outputs. Given the decompiler output, we leverage the system prompt below to query the LLM.

> "*Generate Linux compilable C/C++ code of the main and other functions in the supplied snippet without using goto, and fix any missing headers. Do not explain anything.*"

With above system prompt, we embed the decompilation outputs[2] and query the LLM (GPT-3.5) for a compilable version of the decompilation outputs.

Result. 25 out of the 100 samples were successfully recompiled using the zero-shot system. We also explored the effect of using different wordings in the prompt but found no significant differences in success rates, consistent with findings in [93]. An analysis of the failed samples shows that the system still faces attention degradation and hallucination issues, despite being provided with inter-procedural context. For example, due to attention degradation, the LLM "forgets" the instructions given in the system prompt and generates explanations or malformed pseudocode. Additionally, we encountered a new challenge:

④ *Repairing with Shortcuts*: We found that LLMs often abuse casting operators and cause memory corruption. It may also remove code fragments instead of fixing them, resulting in context loss and recompilation failures. This can be attributed to shortcut learning behavior [48].

> **Takeaway**: The studies in §3.2 and §3.3 illustrate the challenges of performing automatic recompilation using existing LLM-based approaches. Despite the hurdles, when provided with the entire program context, off-the-shelf LLMs resolved a significant fraction of specification errors and header inferences within a single query, increasing the recompilability rate to 25%. However, the vanilla LLM version is still insufficient for effective reasoning.

[2]We do not add information like headers or namespaces from source code.

4 DecLLM for Recompilable Decompilation

Based on the preliminary exploration conducted in §3, we see both potential and challenges in applying LLMs to assist in recompilation. In this section, we design a novel framework, DecLLM, to unleash the full potential of off-the-shelf LLMs in enabling automatic recompilable decompilation.

Targeted Decompiler. DecLLM is designed to repair C decompiler outputs with LLMs, making them recompilable by standard C compilers. We have clarified the design focus on C decompilers in §2.1. This study focuses on the de facto commercial C decompiler—IDA Pro (Hex-Rays) [54], an extensively used C/C++ decompiler that dominates both industrial and academic usage.

Targeted LLMs. In this study, we focus on exploring the potential of using off-the-shelf LLMs to assist in the recompilation process. Our primary focus is on GPT-3.5 [14], the most accessible LLM for its balance of cost and performance [102], as well as GPT-4 [87], the successor to GPT-3.5, which is considered the most powerful LLM available in the wild [75]. Both models can be easily accessed through their API interfaces and do not require specialized resources for deployment. We further discuss the model settings in §5.1.

4.1 Intuition and Overview

Intuition. While automatic recompilation is challenging, experienced reverse engineers can still achieve it manually [4, 31, 80]. In [80], compiler errors play a crucial role in guiding engineers, resembling a typical software debugging process [121]. This insight motivates us to design a framework that replicates this process, using both static and dynamic feedback (i.e., compilation errors and mismatched runtime outputs) to guide LLMs in fixing decompiler outputs. Similar to the typical software debugging process, we should not expect an LLM to fix all recompilation errors in one attempt; instead, we could strategically schedule static and dynamic feedback in an iterative loop, giving the LLM multiple opportunities to repair errors in the decompiler outputs.

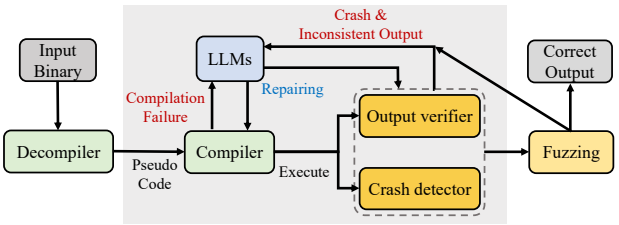


Fig. 2. The workflow of DecLLM.

Overview. Fig. 2 depicts the workflow of DecLLM. Given a binary, we first use a decompiler to extract the decompiled pseudocode for its functions. Then, we launch an LLM-based repair loop that leverages static recompiling and dynamic runtime feedback as oracles to iteratively fix decompiler outputs. Specifically, in §4.3, DecLLM launches a static repairing step to fix decompiler outputs and make them compilable by C compilers. After that, in §4.4, DecLLM launches a dynamic fixing step that not only aligns the expected outputs with those obtained from running a set of test cases on the original executables but also employs the error messages from the enabled ASAN to guide LLMs in fixing the memory errors. Finally, we use fuzzing to ensure consistency between the original and repaired binaries. Before diving into the details of these two components, we first introduce the prompt templates used by DecLLM in §4.2.

4.2 Prompt Design

Table 1 illustrates the prompt setting for DecLLM, corresponding to different scenarios during the recompilation process in §4.3 and §4.4. For example, the compilation error prompt is for instructing the LLM to fix compilation errors, the output error prompt is for functional errors, and the ASAN error prompt is for handling memory corruptions.

Additionally, we observed that the LLM often generates explanations for the functionalities of the given decompiler outputs, disregarding the instructions to generate a compilable version of

Table 1. The prompt templates used by DecLLM.

	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 reduce the number of intermediate variables. Only reply with the fixed source code. Do not explain anything and include 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}*"

the code during the pilot study in §3.3. Therefore, we provide hints for the expected output and specifically instruct the LLM to produce only the fixed code in the system prompt.

4.3 Static Repairing

Static repairing aims to fix the grammatical 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 compiler C (in our implementation, we use GCC). If C fails to compile o, we then launch the static repairing process as follows:

① **Initial Prompting.** With a decompiler output o, we launch the initial prompting step to emphasize the expected output from the LLM and instruct the LLM to reduce redundant variable assignments found in the decompiler outputs (illustrated as "System Prompt" in Table 1). This helps reduce the length of the LLM's outputs, avoiding input ballooning over iterations. With the prepared prompt P, we feed P to the LLM and generate a revised version of the decompiler output.

② **Processing Error Messages.** With an output o from the LLM, we compile it with the standard compiler C. If o fails to compile, we collect the compiler error message E during the compilation. Compiler error messages often contain information about the nature of the errors but also include irrelevant information such as line numbers and file paths. We remove this information from E. Following that, we feed the preprocessed error message E, retaining only the lines that consist of the errors, into the LLM to generate a new token sequence T.

③ **Repairing Decompiler's Output.** At this step, we tokenize o into a sequence of tokens T. Then, we prepare a prompt P (i.e., the System Prompt) to instruct the LLM to fix the token sequence T and generate a new token sequence T'.

④ **Iterative Repairing.** The fixed token sequence T' is then fed into the compiler C to recompile the repaired output o'. If C again fails to compile o', we first iterate over each function in o' and check if the function body of any function is being stripped; if so, we revert the changes. This aims to avoid the shortcut learning behavior observed in our pilot study while maintaining the integrity of the decompiler outputs throughout the iterative repair process. Otherwise, we repeat the above process until C successfully compiles o', or the number of iterations (denoted with N) exceeds a pre-defined threshold N_{Max} (which is set to 15 based on our preliminary experiments).

The above static repairing aims to fix the grammatical and inference errors in the decompiler outputs and make the outputs compilable by standard C/C++ compilers. However, the outputs may still be functionally different from the original executables. 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, thus not satisfying our Definition 1. This discrepancy arises because static repairing focuses primarily on syntactical correctness rather than functional equivalence. To address this issue, we launch the following dynamic repairing.

4.4 Dynamic Repairing

Motivation and Design Consideration. Holistically, the dynamic repairing step aims to fix the functionality deviations in the decompiler outputs. While it appears straightforward to cross-compare the original executable and the recompiled executable (as we already covered recompileable code in §4.3), our tentative exploration shows that precisely pinpointing and fixing various functionality deviations in these two executables is challenging, if at all possible.

Nevertheless, our observation shows that various defects in the decompiler outputs can eventually result in memory errors in recompiled executables. Thus, we leverage ASAN to detect these memory errors by executing ASAN-instrumented recompiled binaries with test cases. When errors are uncovered, we prompt the LLM to generate fixes. To further ensure semantic consistency between the recompiled and the original binaries, we utilize coverage-guided fuzzing to synthesize extra test cases with high-quality shipped test cases as seeds. This helps us detect potential functionality deviations and ensure a high possible level of correctness of decompiled code.

Clarification. We clarify that our dynamic repairing should not be deemed as bug detection in binary code analysis. Binary code could contain various subtle errors, and bug detection is a well-known challenge in binary code analysis. We are *not* solving this hard problem. Instead, per our observation, errors in decompiled code, though altering the functionality, often manifest fewer variations and are not as "arbitrary" as those faced in binary code bug detection task. Locating and fixing errors using ASAN can practically and effectively shave a large number of errors in decompiled code. Moreover, unlike binary bug detection task, our context has a golden reference — the original executable. Using the original executable to uncover functionality deviations in the decompiler outputs further reduces the errors in the decompiler outputs.

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

⑤ **Collecting Memory Error Information.** We first collect the memory error information I when one of the sanitizer checks 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 the ASAN, and can be easily parsed for analysis using scripts.

⑥ **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'.

⑦ **Functionality Equivalence Testing.** If test case t does not result in an alarm, we will verify the output of E against the expected output. Specifically, given the same input t and initial state s, if E produces a different output from the original executable, this indicates a deviation in the program states between E and the original executable according to Equation 2. When such an inconsistency is detected, an inconsistency between E and the original executable is confirmed. Thus, 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; thus, our primary focus is fixing the above memory error. Overall, we repeat the above process from step ⑤ until E passes all the test cases or fails any test cases in T. Then, we extend the testing process with coverage-guided fuzz to check potential inconsistencies with synthesized test cases and use LLM to fix them when the budget is sufficient. Note that the purpose of fuzzing is to make the best efforts to find semantic inconsistencies that fail to detect in previous steps, instead of vulnerability detection.

5 Experimental Setup

DecLLM is primarily written in Python and C++, with about 2,504 lines of code, to support the decompilation outputs for IDA Pro 8.3. All experiments are conducted on a Ryzen 3970X 32-core server with 256GB of memory and an RTX A6000 GPU.

5.1 Model Setting

As mentioned in §4, our study aims to evaluate off-the-shelf LLMs for recompilation. Therefore, we deliberately refrain from conducting specific LLM hyperparameter tuning throughout the study, and interact with these models through the official OpenAI API. To prevent data leakage, we are using older versions of GPT-3.5 and GPT-4, specifically `gpt-3.5-turbo-0613` and `GPT-4-0314` respectively. We also take similar precautions in the selection of test cases. We also limited our selection of programs based on the maximum token length limit of our employed LLM model, which is 4,096 for both of the models. Notably, this limit includes both the input and output parts of the LLM. 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.

5.2 Datasets

We utilize 3 datasets in our evaluation: the Code Contest dataset [67], CGC binaries [6], and Coreutils-9.5 [1], where the GCJ dataset (which we use in §3) and the Code Contest dataset are popular and diverse C benchmarks for language model-oriented software engineering research [44, 118].

Specifically, for the Code Contest dataset, we selected programs with "AC" (Accepted) verdict in our evaluation. Thus, we can use the supplied test cases to validate the correctness of the recompiled binaries. To prevent data leakage, we only considered submissions made after knowledge cutoff for ChatGPT [12] (September 2021). Additionally, 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 context length and randomly selected 60 programs from each interval. We manually verified that no trivial decompilation errors are present in these 300 programs, and we report that the average lines of code and tokens are 112.9 and 1,110.4, respectively. We use this dataset to evaluate the effectiveness of DecLLM in handling the recompilation challenges from §3.

To better understand the challenges of recompiling real-world software, we evaluate Coreutils-9.5 [1] and CGC binaries [6]. Coreutils is a set of 108 unix utilities, and CGC is a set of diversified binaries designed to replicate real-world vulnerabilities in the Cyber Grand Challenge [5]. For CGC, after excluding binaries that fail to compile or validate functionality, we use a subset of 134 binaries for evaluation. We use all of the test cases supplied by these datasets for functionality checking, followed by fuzzing as outlined in §4.4. We compiled both datasets with default configuration settings without stripping symbols. This configuration reflects the practical scenario of re-engineering and vulnerability research in large-scale software, where debug symbols are usually provided by the vendor [124]. We used AFL++ [21] to fuzz the binaries. Depending on the complexity of the dataset, we follow previous fuzzing research [68, 74, 100, 110] to configure the time limit, with 0.5 CPU hours for Code Contest [68], 6 CPU hours for CGC [110], and 24 CPU hours for Coreutils [74, 100, 110].

5.3 Evaluation Setting

Since the recompilation process enabled by DecLLM is iterative, we use the length of the conversation chain (denoted as N) required to correctly recompile the decompiler's output as our different evaluation settings. Due to cost considerations, we evaluated with N values of 1, 5, 10, and 15. We deem it a success if the recompiled binaries pass all test cases without errors.

Table 2. Recompilation success rates for 3 settings with respective to different conversation length (N).

	DecRule	*LLM-baseline*	DecLLM-GPT3.5	DecLLM-GPT4
N = **1**	8%	32%	37%	40%
N = **5**	8%	39%	54.3%	67%
N = **10**	8%	42%	68%	73%
N = **15**	8%	45%	74%	78.3%

Table 3. Failure cases by root cause.

	DecLLM			*LLM-baseline*		
Root Causes	IO	FR	H	IO	FR	H
200 - 560	2	11	2	7	15	3
560 - 920	5	25	7	12	59	13
920 - 1280	9	78	11	16	243	19
1280 - 1640	13	235	19	37	320	25
1640 - 2048	52	347	24	93	476	31
Total	81	696	63	165	1113	91

6 Evaluation

In this section, §6.1 reports DecLLM's effectiveness in automatic recompilation, §6.2 demonstrates DecLLM's applications in recompiling real-world binaries and programmatic use of decompiled code for vulnerability detection, and §6.3 provides an error analysis of those hard cases.

6.1 RQ1: Effectiveness of DecLLM

In this RQ, we evaluate the effectiveness of DecLLM in addressing the challenges identified in §3. We benchmark DecLLM against four settings:

- DecRule: The rule-based approach evaluated in §3.1. To ensure a fair comparison with DecLLM, we append the header extracted from the source code. Note that DecLLM is expected to infer this information.
- *LLM-baseline*: An extended version of the approach in §3.3. We modified it to be iterative, using compilation and output errors, to assess its performance relative to DecLLM. In this baseline, the integrity check (④ in §4.3) and ASAN (§4.4) are disabled during recompilation.
- DeGPT [55]: A readability-oriented refinement framework evaluated in §3.2. Since it operates on a function-by-function basis, we exclude global variable renaming suggested by DeGPT to prevent recompilation failures due to duplicated variable names.
- LLM4Decompile-Ref [103]: An end-to-end LLM for re-executability-oriented refinement, which we evaluated in §3.2[3].

Result. The first three columns of Table 2 show the success rates for DecRule, *LLM-baseline*, and DecLLM concerning different conversation lengths (N). Overall, DecLLM successfully recompiles 74% of the binaries from the testing dataset, outperforming *LLM-baseline* by 29% and DecRule by 66%. This comparison highlights the effectiveness of DecLLM in addressing the challenges we identified in §3. Moreover, DecLLM is capable of handling decompiler outputs with long context compared to *LLM-baseline*. We interpret the results as follows:

Effectiveness of the Extended System Prompt. The results for $N = 1$ in Table 2 show the success rates of decompiler outputs being successfully recompiled with one repairing query. Compared to the results of *LLM-baseline*, it is evident that there is an immediate increase in the success rate from 32% to 37% when the system prompt is extended. In other words, by extending the system prompt, DecLLM effectively imposes more constraints on the LLM and helps the LLM to focus more on fixing decompiler outputs and align more closely with the desired output. This minimizes the performance degradation problems of LLMs when handling long inputs and contributes to improved performance without additional iterations.

Effectiveness of Static Repairing. We examine the intermediate cases in which DecLLM and *LLM-baseline* failed to recompile to evaluate the contribution of static repairing. We categorize failures into three types from §3.2 and §3.3: *illegal outputs* (IO), *fixing by removal* (FR), and *hallucinations*

[3]We also explored extending this model to be iterative, but the LLM returned an empty output, likely due to specialized training as acknowledged by the authors [19].

(H). Table 3 shows the distribution of these failures across different context lengths. The results show that DecLLM can effectively handle these issues, particularly for the fixing by removal errors, where static repairing prevents a total of 696 instances. Notably, in *LLM-baseline*, none of these samples could be repaired after the LLM removed functions from the input.

Besides, as discussed in §3.2, illegal outputs often stem from degraded instruction-following capabilities, resulting in generating redundant explanations and fragmenting the pseudocode. Table 3 shows that the number of illegal outputs in DecLLM is lower than that of *LLM-baseline*, demonstrating the repairing ability of DecLLM. We observe that some fragmented functions can be reintroduced back to the input by DecLLM, allowing for repair in the next iteration.

Effectiveness of Dynamic Repairing. Dynamic repairing addresses functionality deviations that cannot be resolved through static repairing alone. Specifically, hallucinations likely introduce stealthy functionality deviations, which DecLLM leverages dynamic information to capture.

Regarding hallucinations, dynamic repairing handles them by providing the expected output or ASAN error message (if it crashes) for the LLM to fix the error. Compared with *LLM-baseline*, DecLLM reduces hallucinations by 31% ($(91-63)/91 \approx 31\%$) as illustrated in Table 3. The extra context information derived from ASAN errors guides the LLM to precisely fix the errors and avoid introducing new bugs to the input under repair.

Types of the Fixed Errors. Using compilation errors from DecRule as a reference, we analyze the types of errors DecLLM can fix. We find that DecLLM fixes 98% of specification errors within two conversation rounds regardless of whether the test case can be recompiled. Unlike traditional approaches that use heavy-weight analyses like symbolic execution to infer function prototypes [35, 107, 108], LLMs can fix these errors by replacing the code block, which is reasonable because typical APIs and function prototypes are widely represented in the training sets or can be inferred from the compilation errors submitted to the LLMs.

DecLLM can fix 60.1% of inference errors and all identified decompiler template errors. In one successful case, a variable's integer type is incorrectly inferred as unsigned, causing compilation errors when used with the `max` function. DecLLM fixed this in two iterations by correctly identifying and replacing the type. Similar situations also occur for LLMs to fix decompiler template errors, which usually result in type conversion failure during compilation.

Effectiveness of fuzzing. We report that three functional inconsistencies are being asserted by cases generated from fuzzing. These cases resulted from an erroneously inferred buffer, and the recompiled binary did not crash when the original binaries did. Moreover, none of these cases can be fixed by DecLLM; we further discuss this in §6.3.

Comparison with DeGPT. We evaluate DeGPT by running it three times and deem it a success if any run successfully recompiles the binary and passes all the test cases. DeGPT successfully recompiles 30 binaries (10%), which includes all cases handled by DecRule and 3 additional cases. Most modifications by DeGPT involve comments and variable renaming, with some code simplification. For the 3 additional binaries, all of them have minor specification errors. Therefore, when the LLM attempts to simplify the decompiler outputs, it inadvertently fixes these errors. However, we observed instances where DeGPT oversimplified code by replacing it with comments. Its root cause is likely due to the inaccuracy of the symbolic reasoning engines. This highlights the need for a two-step paradigm for recompilation, using the test suite to validate functional correctness. We clarify that DeGPT is intended as a lightweight framework to improve readability without compilation. As such, the result is within our expectations.

Comparison with LLM4Decompile-Ref. Due to resource constraints, we evaluate the 6.7b-v2 version of LLM4Decompile-Ref. We run the evaluation three times on the testing dataset, and the outputs are identical for the same input, so we do not repeat it further. LLM4Decompile-Ref successfully recompiled 52 out of the 300 binaries (17.3%). More than half of failed cases were due

to compilation errors resulting from inconsistencies when each function was refined separately without feedback. Also, hallucination-related crashes or functional inconsistencies occurred in 27% of failure cases. These results highlight the importance of DecLLM's iterative design, which provides additional context for the LLM to fix errors exposed by the compiler or during runtime.

Effect of Compiler Optimization. Decompile-eval [103] is a benchmark comprising 656 binaries, compiled from 164 distinct programs at 4 optimization levels. We utilize the published dataset for evaluation. DecLLM successfully fixes 45% of the functions with a single query ($N = 1$), and a total of 67.4% up to $N = 15$. Table 4 presents the success rate for each optimization level together with the reference result reported in the original paper. By reviewing cases that DecLLM fails to fix, we find undefined constant variables, with 78, 99, 91, and 129 instances for O0, O1, O2, and O3 test cases, respectively, which inversely correlates with the success rate. Even with the feedback mechanism, DecLLM can only fix 23 of them. This shows that recompilation decompilation is upper-bounded by the decompiler output's quality. For remaining cases, it correlates to code patterns from decompiler. For example, the inlined `fabs` has multiple undefined variables. While it fails to handle by DecLLM, LLM4Decompile-Ref can fix it. We acknowledge this as a limitation of off-the-shelf LLMs, which are not trained specifically to handle decompiler outputs.

Table 4. Effect of Compiler Optimization.

	O0	O1	O2	O3	AVG
DecLLM	80.49%	65.24%	65.5%	58.5%	67.43%
LLM4Decompile	80.49%	58.54%	59.76%	57.93%	64.18%

Extension with Other LLMs. To compare the performance of DecLLM with different LLMs, we evaluate its effectiveness when replacing GPT-3.5 with the more powerful GPT-4 model. As illustrated in Table 2 and Table 7 (available in the subsequent §7), both GPT-3.5 and GPT-4 exhibit similar performance in terms of success rate, with GPT-4 showing only a 4.3% enhancement over GPT-3.5. We further investigate GPT-4's performance in terms of conversation length. Comparing the average conversation length for GPT-3.5 and GPT-4, as shown in Table 2, we see that the average conversation length is reduced with GPT-4 for all intervals. This illustrates how the DecLLM design successfully offsets the limitations of the underlying LLM.

> **Answer to RQ1**: DecLLM significantly increases the recompilation success rate, raising it from 8% with rule-based recompilation and 10% with a recent readability-oriented LLM-based approach to 74% with GPT-3.5 and 78.3% with GPT-4.

6.2 RQ2: Real-World Applications

We consider two real-world applications of DecLLM: recompiling real-world binaries and programmatic use of decompiled code for vulnerability detection.

6.2.1 Recompiling Real-World Binaries. Decompiler outputs of real-world software can be too long for DecLLM. We configure it to process in a function-by-function manner, eventually forming the recompilable code. Specifically, we extract error messages related to undefined functions and types, and use an LLM to infer their dependencies. Next, we modify the response criteria by removing the "main" keyword and the instruction to fix missing headers from the system prompt. Similar to DecLLM, we query the LLM 15 times to fix each function[4].

During our evaluation, we notice the likelihood of the LLM replying with explanations or code fragments increases when we modify the system prompt. We add an integrity check to prevent LLM fixes from removing more than 50% of the code. If this occurs, we revert the changes.

[4]To handle functions with an input length exceeding the upper limit of DecLLM, we embed the backward slice into the LLM and request a JSON output instead.

Table 5. Statistics for Recompiling Real-World Binaries with DecLLM.

	Coreutils-GPT-3.5	Coreutils-GPT-4	CGC-GPT-3.5	CGC-GPT-4
Fully functional binaries	52/108	60/108	70/134	79/134
Malfunction binaries	3/108	3/108	2/134	2/134
Crashing binaries	14/108	13/108	11/134	14/134
Overall	63.9% (69/108)	70.4% (76/108)	61.9% (83/134)	70.9% (95/134)

Result. We present our results in Table 5. First, we note that none of the binaries in these two dataset recompile with DecRule and the inferred dependencies. This is because DecRule cannot handle the errors present in the decompiler outputs. Both DeGPT and LLM4Decompile-ref also fail to recompile any binaries. DeGPT's changes only add comments to the decompiled code, which does not address compilation errors. LLM4Decompile-ref, meanwhile, produces conflicting content that cannot be compiled. Using GPT-3.5 and GPT-4 as the LLM models, DecLLM can recompile 69 and 76 binaries for Coreutils and 83 and 95 for CGC-binaries, respectively. As expected, GPT-4 demonstrated better performance on this task. Overall, DecLLM can fix 52.8% and 58.1% of the compilation errors for GPT-3.5 and GPT-4, respectively. The average lengths of conversation sequences for GPT-3.5 and GPT-4 to repair a function are 4.29 and 3.79, and the numbers of illegal outputs reverted by DecLLM are 746 and 607, respectively. From these results, we can see that GPT-4 has a slightly better instruction-following capability than GPT-3.5.

Excluding dependencies errors handled by DecLLM, the remaining errors can be categorized into struct-related errors, type inference errors, and undefined variables. The major source of errors of both dataset is struct-related errors, which account for 58.2% of the reported error messages when we attempt to compile the output from the decompiler. To illustrate the effectiveness of DecLLM in handling these errors, we present one of the compilation errors as follows:

```
1  // Original Decompiler Outputs
2  ...
3  emit_ancillary_info::infomap *map_prog;
4  emit_ancillary_info::infomap infomap[7];
5  ...
6  infomap[0].program = "[";
7  infomap[0].node = "test␣invocation";
8  ...
```

```
1  // Fix by DecLLM
2  struct infomap {
3    const char *program;
4    const char *node;
5  } infomap[7];
6  struct infomap *map_prog;
7  infomap[0].program = "[";
8  ...
```

We can see that lines 3-4 of the original decompiler outputs are syntactically incorrect. By utilizing the error message with the usage example of the struct from line 6, DecLLM successfully fixes the error by first inferring that the infomap struct has two members (i.e., "program" and "node") based on the value assignment below. Next, it corrects the syntactical issue by replacing the incorrect line with the correct one (refer to lines 2 to 6 of the fixed version). This demonstrates the powerful debugging and reasoning capabilities of off-the-shelf LLMs.

DecLLM can correct 79.1% and 81.3% of the memory corruption errors for GPT-3.5 and GPT-4, respectively. Manual analysis revealed all memory corruptions originate from the decompiler. Apart from the decompiler template error we discussed in §3, memory corruption also stems from false-positive analysis results from the decompiler [7], leading to null pointer dereference. By supplying the ASAN error message alongside the decompiler outputs, we observed DecLLM replacing &dword_0 with 0 to prevent triggering such runtime errors that originate from decompiler.

Compared to §6.1, where a significant portion of memory corruption was caused by the LLM, there are no such cases in the real-world binaries. One reason we interpret is that there is only a single function within each prompt provided to the LLM. Hence, there is less noise compared to §6.1, where the LLM had the full code snippets to consider. Nevertheless, this approach will prohibit the inference of undefined global variables and fixing of type inference errors.

6.2.2 Vulnerability Detection. DecLLM can bridge source-based vulnerability analysis tools like CodeQL to detect vulnerabilities in binaries. Following the approach from [68], we assess DecLLM's effectiveness with CodeQL's official queries in detecting 12 CWEs (details in [22]) from the Juliet Test Suite [85]. Specifically, we sample 50 test cases per CWE variant, with half containing vulnerabilities. We report that the precision, recall, and F1 scores of official queries over the source code are 0.99, 0.33, and 0.50, respectively.

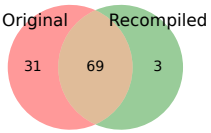


Fig. 3. Count of vulnerabilities detected by CodeQL.

To evaluate the effectiveness of DecLLM, we compile these 600 samples into binaries with GCC (with the default compilation/optimization configurations), strip those binaries, and then decompile them with DecLLM to get their recompilable code[5]. DecLLM only fails to produce recompilable code of 13/600 samples. We attribute these cases to incorrect analysis. We then run CodeQL on the outputs of DecLLM; we report the precision, recall, and F1 score of the CWE detection results as 0.70, 0.24, and 0.36, respectively. Compared to the reference results, CodeQL generates 22 false positives (FP) and 31 false negatives (FN). Notably, the test cases of CWE-190 account for all the FPs and 20 of the FNs, with the remaining FNs originating from CWE-134.

As illustrated in Fig. 3, among 100 vulnerabilities initially detected in source code, 69 of them are successfully identified when analyzing recompiled code. For the remaining 31 vulnerabilities, we further investigated the possibility of customizing existing queries to better suit recompiled code. Our investigation revealed that the 31 FNs stemmed from two queries: "ArithmeticWithExtremeValues" and "UncontrolledFormatString". In the case of "ArithmeticWithExtremeValues", the key issue is the query's reliance on source-level artifacts, such as macro names (e.g., `INT_MAX`) [13]. By modifying these queries to match literal values instead, we are able to capture these previously missed vulnerabilities. For "UncontrolledFormatString", the query's failure potentially stemmed from a mismatch between IDA Pro's code generation patterns and heuristics in CodeQL [18, 68], which hindered proper propagation of `printf` calls through the call tree.

By following the fixing procedure outlined by CodeQL developers [18], we successfully detected *all* these FNs. Despite the necessity of manual interventions, the required query adaptations are minimal and aligned with common practices in query-based static analysis, where queries are often customized for specific codebases. The results suggest that recompiled code does not fundamentally undermine CodeQL's vulnerability detection capabilities. In fact, with customized queries, CodeQL's analysis of recompiled code can yield results consistent with those obtained from source code analysis. Furthermore, CodeQL detected three additional vulnerabilities missed when analyzing source code, primarily due to macro expansion, where the vulnerable code was expanded and subsequently captured by CodeQL. This highlight the unique opportunity presented by analyzing decompiled code, which we plan to explore in future work.

Overall, our evaluation demonstrates DecLLM's ability to bridge the gap between source-level analyzers and binary code, unlocking the potential of source-based tools for binary analysis.

Answer to RQ2: For real-world binaries, the recompilation results are slightly lower than those in RQ1, but DecLLM still achieves an average of 62.8% with GPT-3.5 and 70.7% with GPT-4. Furthermore, we demonstrate that recompilable code enables the effective application of a source-level vulnerability detector, CodeQL, to binary code, yielding promising results.

[5]For vulnerable samples, we verify that the recompiled code can trigger the corresponding CWEs.

6.3 RQ3: Error Analysis of Hard Cases

Despite DecLLM's ability to recompile a portion of binaries from both the Code Contest dataset (§6.1) and Coreutils (§6.2), our evaluation shows that 21.7% to 36.1% of binaries cannot be fixed by DecLLM automatically. To better understand these hard cases, we conduct a manual study on these unrecompilable binaries and summarize our findings.

Variable Recovery. Since decompiler outputs are the primary input for DecLLM, any defects from the decompiler affect the performance of DecLLM, particularly in the Coreutils dataset. Out of the 32 binaries that DecLLM cannot recompile, 10 of them have problems with distinguishing between addresses and data within the decompiler outputs. Five binaries from CGC fail to recompile due to the same cause. For example, the decompiler misinterprets the "-c" string in the argument of the `execl` function as an address. Thus, a non-existing variable "loc_632D" is referenced (632D is the little-endian of "-c"), leading to a compilation error. Also, all unfixable malfunctions and crashing binaries in §6.2 are attributed to this problem as well. The wrongly inferred variable content leaves the binaries in an invalid internal state. After calling some system APIs like "getpwuid", it returns an invalid pointer, leading to crashes within the main function.

Ill-inferred Buffer Size. As we observe in §6.1, while DecLLM can fix 60.1% of inference errors, we see that the remaining errors stem from ill-inferred buffer sizes in different contexts. Although buffer size can be indirectly fixed through ASAN error messages, the lack of accurate buffer size information still poses challenges for LLMs in repairing decompiler outputs. We consider this a challenge for both LLMs and decompilers, given that the performance of LLMs highly depends on the quality of inputs (i.e., decompiler outputs).

Insufficient or Ambiguous Information Consumed by LLMs. Insufficient information poses a challenge for any LLM-based reasoning system. As discussed in §4.4, dynamic repairing relies on a set of valid test cases to identify different input-output relations and trigger crashes. Therefore, DecLLM cannot fix functionality deviations that are not covered by the available test cases.

Ambiguous compiler errors also challenge LLMs. For example, "*expected ';' before*" is raised due to the omitted "struct" keyword before "sigaction". Although it is easy for programmers to fix [2], LLMs struggle to synthesize fixes from error messages alone. As a result, DecLLM could only fix 11% of these cases. To conclude, these two issues are beyond the capabilities of off-the-shelf LLMs, and we will leave the investigation of resolving these challenges as future work.

Attention Degradation. As mentioned in §6.1, DecLLM may not be able to fix all inference errors. Out of the remaining 40% of samples, 69.9% come from the Standard Template Library (STL) functions. Since calls to STL functions introduce their implementations into the binary, it increases the size of the decompiler's outputs from 2.8 to 16.1 times. As LLMs performance is highly dependent on context length [73], long decompiled outputs pose a challenge. However, automatic identification of STL functions is challenging [3, 8]; a possible solution is adopting Retrieval-augmented Generation (RAG) [66] techniques to recognize STL functions and replace this implementation with the identified calls. We leave this as a future enhancement for DecLLM.

Poor Instruction Following. In §6.1, we see that both GPT-3.5 and GPT-4 settings result in over half of the triggered memory errors that cannot be automatically fixed by DecLLM. We manually investigate these remaining memory errors by examining the intermediate conversation between the LLM and DecLLM. We then categorize the causes into two types. The major cause (81.8% for GPT-3.5 and 78.5% for GPT-4) is the problematic usage of type casting operators (e.g., "static_cast" and "reinterpret_cast") when the LLM attempted to fix type conversion errors. Despite our efforts to mitigate this, the LLMs persist in using them as a fixing strategy, which may be due to shortcut learning behavior mentioned in §3.2 and poor instruction following. The remaining errors are associated with inferring incorrect buffer sizes, which are hard to fix by LLM.

Other Issues. In addition to the aforementioned issues, there is a tendency for the LLM to generate illegal outputs when we do not provide it with the full context of the decompiled pseudocode. We interpret that providing the full decompiled pseudocode will hint to the LLM to generate the source code, not the explanation. Instruction fine-tuning is a potential solution to this problem and the shortcut learning behavior we reported in §3.2. It modifies self-attention heads to encourage response alignment [117], making the LLM specific to tasks.

Answer to RQ3: For the remaining hard cases beyond the capabilities of off-the-shelf LLMs, we delve into their errors and obtain insights that could guide future decompilation-oriented LLM design and fine-tuning.

6.4 RQ4: Cost of DecLLM

We evaluated the cost of using DecLLM with GPT-3.5, considering both the time required for repairing and fuzzing and the financial cost associated with token consumption. Table 6 reports the results across three datasets. While repairing decompiled code with LLMs is generally efficient, static repairing is notably more time-consuming due to multiple rounds of LLM interactions. Besides, we used tiktoken [17] to estimate the financial cost associated with token consumption. Table 6 indicates that fixing a complex binary (e.g., Coreutils binary) costs notably more, suggesting that decompiled code of complex binaries tends to have more errors to fix.

Table 6. The cost of static and dynamic repairing with DecLLM.

Dataset	Static Repair Time	Dynamic Repair Time	Financial Cost
Code Contest	60.04s	17.09s	$0.04
Coreutils	420.20s	93.30s	$0.18
CGC	77.26s	39.34s	$0.04

Additionally, DecLLM ensures a higher level of confidence in the recompiled binary's correctness by incorporating a fuzzing mechanism to detect behavioral inconsistencies that humans might miss. As explained in §4.4, the fuzzing process in DecLLM aims to detect behavioral deviations introduced by decompilation or LLMs. Such deviations are easier to detect than bugs stemming from programming defects. Thus, the fuzzing process in DecLLM will not consume as much time as traditional fuzzing research. Moreover, the workload of fuzzing can be parallelized and amortized across CPU cores to save time.

In the experiments, we adopted a conservative fuzzing strategy continued beyond coverage saturation. Specifically, the fuzzing time for the Code Contest, Coreutils, and CGC dataset is 1, 45, and 12 minutes, respectively. Since we deployed the fuzzing process on a machine with 64 cores, the CPU hours used for these three datasets are about 1, 48, and 12 hours. Nevertheless, the fuzzing time is adjustable and can be largely reduced in practice. Recent advances in fuzzing research can also be employed to speed up further.

Overall, DecLLM achieves a reasonable processing time and incurs minimal financial cost. As the first automated solution for recompilable decompilation, DecLLM shows significant advantages in reliability and efficiency compared to prior attempts that fix decompiled code into recompilable forms [80] manually, which can take hours to days of skilled human efforts.

Answer to RQ4: DecLLM shows low financial costs with reasonable time costs, whereas the cost of fuzzing is adjustable and can be further optimized through advanced fuzzing techniques. This makes DecLLM highly practical as the only automated recompilable decompilation solution. Additionally, as LLMs continuously evolve for better performance and lower prices, we expect its financial cost to decrease significantly in the future.

Table 7. Success cases across different context lengths. GPT3.5-IDA is from §6.1. Config-① denotes supplying line numbers and code to LLM during static repair, and Config-② denotes only supplying line numbers.

Context Length	GPT3.5-IDA	GPT3.5-Ghidra	GPT4-IDA	Config-①	Config-②
200-560	57/60	56/60	59/60	58/60	55/60
560-920	53/60	52/60	56/60	54/60	42/60
920-1280	48/60	49/60	50/60	49/60	39/60
1280-1640	37/60	38/60	40/60	36/60	21/60
1640-2048	27/60	25/60	30/60	26/60	13/60
Overall	74% (222/300)	73.3% (220/300)	78.3% (235/300)	74.3% (223/300)	56.6% (170/300)

7 Discussion

In line with our presented evaluation and application results, we now discuss the following several aspects and mitigate potential concerns.

Extension to Other Decompilers. In §3 and §6, we perform studies on recompilation and benchmark DecLLM with decompilation outputs of IDA Pro, the de facto commercial C/C++ decompiler. However, the proposed approach is independent of the underlying decompilers. To examine DecLLM's generalizability, we extend DecLLM to support an open-source C decompiler, Ghidra 10.4 [86]. We repeat the experiment in §6 and denote this setting as GPT3.5-Ghidra. The results are shown in Table 7. Compared with using IDA Pro, our evaluation shows that DecLLM achieves a comparable performance with Ghidra, with merely a 0.7% difference in recompilation success rate, illustrating that DecLLM is not tightly coupled to a specific decompiler. Moreover, given that the powerful LLM is not limited to a specific programming language, we envision that the workflow of DecLLM can also work on software compiled from other programming languages.

Extension to Other LLMs and Datasets. Another potential extension of DecLLM is to leverage other LLMs. To ensure generality, we deliberately refrain from conducting specific LLM hyperparameter tuning. Such an LLM-agnostic setting largely improves our work's reproducibility and enhances our findings' generalizability. We also replicated our evaluation in §6 with GPT-4. As anticipated, we observe a consistent improvement across all settings (see Table 7), indicating that DecLLM is not limited to a specific LLM. We also performed our empirical study (in §3) and evaluation (in §6) on two separate datasets, demonstrating comparable performance. The results indicate that the performance of DecLLM is not restricted to a specific dataset.

Validity in CodeQL Results. In §6.2.2, we have presented an important application of DecLLM in vulnerability detection. We view the results as highly promising, as DecLLM can bridge the gap between binary-level analysis and source-level analysis. However, one may be concerned about the study's validity because the decompiler or DecLLM may eliminate certain vulnerabilities before analysis, as DecLLM is designed to fix memory errors found by ASAN, and decompilers (which are essentially rule-based) to some extent may "smooth" the code. Yet, in practice, experiment in §6.2.2 shows that the detection results in the fixed code are consistent with that of the original source code. In practice, we recommend users to check errors reported by ASAN when using DecLLM as well as findings reported by CodeQL to ensure a comprehensive vulnerability understanding.

Functional Equivalence. DecLLM utilizes official test suite to assert functional inequivalence between the original and recompiled executables. It is important to note that checking equivalence between original and recompiled executables is inherently undecidable, as per Rice's theorem [97]. While DecLLM focuses on automatic recompilation, developing practical equivalence verification would be a valuable enhancement for DecLLM.

Postprocessing of Compiler Error Messages. We investigated the impact of including line numbers in compiler error messages during static repair. As shown in Table 7, Config-① (including line numbers and erroneous code) performs similarly to our default setting, while Config-② (only line numbers) shows significant performance degradation, especially for longer contexts. This

suggests that concrete code examples are more valuable than line numbers for LLMs to fix errors, aligning with known LLM limitations in counting tasks [25].

8 Related Work

Besides reviewing software decompilation and recompilation techniques in §2. We review other related work below.

Dynamic Binary Rewriting. Binary rewriting modifies executables without source code access. While we mainly discuss static binary rewriting in §2, dynamic rewriting techniques have developed significantly over the past few decades. Dynamic rewriting is performed during runtime, often based on system-level support like Intel Pin [79], DynamoRIO [29], and Valgrind [83]. Dynamic binary rewriting enables binary code modification without the restrictions faced by static methods [43, 45]. However, it incurs high overhead due to runtime rewriting. In contrast, this work focuses on more efficient static binary rewriting. Our approach further extends the capability of static binary rewriting methods, enabling smooth instrumentation of binary code at a modest cost.

Decompilation Readability. This work focuses on the automatic recompilation of decompiled code, emphasizing its functionality and correctness. Another important and orthogonal research area in decompilation is improving readability by assigning meaningful names and types to decompiled variables using NLP models [34, 65, 89], generating function names for decompiled code [39], and predicting function names for stripped binaries [47, 60]. Such works do not improve the correctness of decompiled code but facilitate human comprehension. Moreover, Debin [53] and CATI [33] predict debugging information from binaries. Recent works, like LmPA [119] and DeGPT [55], combine program analysis with LLM to enhance the readability of decompiler outputs.

9 Conclusion

This paper explored enabling recompilable decompilation with off-the-shelf LLMs for the first time. We demonstrated that this is a non-trivial task through a pilot study of existing rule-based and LLM-based approaches. We designed DecLLM, an iterative LLM-based repair loop that combines static recompilation and dynamic runtime feedback to fix decompiler outputs. Evaluation on popular C benchmarks and real-world binaries showed that DecLLM can achieve up to a 70% recompilation success rate. The recompilable code can be used for CodeQL-based vulnerability analysis, demonstrating largely consistent results with the ground-truth source code.

10 Data Availability

We maintain a website at [22] hosting the research artifacts.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. The HKUST authors were supported in part by a NSFC/RGC JRS grant under the contract N_HKUST605/23 and a CCF-Tencent Open Research Fund.

References

[1] 2003. Coreutils. https://www.gnu.org/software/coreutils/.
[2] 2013. sigaction both struct and function. https://stackoverflow.com/questions/14213270.
[3] 2014. Find the C++ STL functions in a binary. https://reverseengineering.stackexchange.com/a/3890.
[4] 2014. Starcraft Reverse Engineered to run on ARM. https://news.ycombinator.com/item?id=7372414.
[5] 2016. DARPA. Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge/.
[6] 2017. cgc-challenge-corpus. https://github.com/lungetech/cgc-challenge-corpus.
[7] 2019. IDA - ObjC - Meaning of _BYTE and dword_0? https://reverseengineering.stackexchange.com/questions/22708.

[8] 2022. AN EXHAUSTIVELY ANALYZED IDB FOR COMLOOK. https://www.msreverseengineering.com/blog/2022/1/25/an-exhaustively-analyzed-idb-for-comlook.
[9] 2022. Examples and guides for using the OpenAI API. https://github.com/openai/openai-cookbook/.
[10] 2022. Gepetto. https://github.com/JusticeRage/Gepetto/.
[11] 2022. gpt-wpre. https://github.com/moyix/gpt-wpre/.
[12] 2022. OpenAI Models overview. https://platform.openai.com/docs/models/overview.
[13] 2023. ArithmeticWithExtremeValues for CWE-190. https://github.com/github/codeql/blob/main/cpp/ql/src/Security/CWE/CWE-190/ArithmeticWithExtremeValues.ql.
[14] 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.
[15] 2023. How Long Can Open-Source LLMs Truly Promise on Context Length? https://lmsys.org/blog/2023-06-29-longchat/.
[16] 2023. LLM forgetting part of my prompt with too much data. https://community.openai.com/t/244698.
[17] 2023. tiktoken. https://github.com/openai/tiktoken.
[18] 2023. UseAfterFree.ql miss case 00. https://github.com/github/codeql/issues/13896.
[19] 2024. Empty output in slightly more complex prompt. https://github.com/albertan017/LLM4Decompile/issues/15.
[20] 2024. Question about ghidra's decompile.py file. https://github.com/albertan017/LLM4Decompile/issues/31.
[21] 2025. AFLplusplus. https://github.com/AFLplusplus/AFLplusplus.
[22] 2025. Artifact. https://sites.google.com/view/decllm.
[23] 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.
[24] 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.
[25] Thomas Ball, Shuo Chen, and Cormac Herley. 2024. Can We Count on LLMs? The Fixed-Effect Fallacy and Claims of GPT-4 Capabilities. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=qt4d0EGZsK
[26] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code *(USENIX Security)*.
[27] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.
[28] 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. *NeurIPS* (2020).
[29] 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.
[30] 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.
[31] 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.
[32] 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 *FSE*.
[33] 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.
[34] 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 *USENIX Security*.
[35] 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.
[36] Cristina Cifuentes. 1994. *Reverse compilation techniques*. Queensland University of Technology, Brisbane.
[37] Cristina Cifuentes and K. John Gough. 1995. Decompilation of Binary Programs. *Softw. Pract. Exper.* 25, 7 (July 1995).
[38] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. 2020. Scalable Validation for Binary Lifters.
[39] 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.
[40] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS*.
[41] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *PLDI*.
[42] Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. 2024. Polynima: Practical Hybrid Recompilation for Multithreaded Binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems*.

[43] 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.
[44] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.
[45] 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.
[46] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*.
[47] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *ISSTA*.
[48] 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).
[49] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
[50] Keerthana Gurushankar and Pulkit Grover. 2021. A minimal intervention definition of reverse engineering a neural circuit. *arXiv preprint arXiv:2110.00889* (2021).
[51] Mark I Halpern. 1965. Machine independence: its technology and economics. *Commun. ACM* 8, 12 (1965), 782–785.
[52] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. 2023. QueryX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[53] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *CCS '18*.
[54] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.
[55] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. (2024).
[56] 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).
[57] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and explaining large language model-based code generation: A causality-centric approach. *arXiv preprint arXiv:2310.06680* (2023).
[58] Zhenlan Ji, Daoyuan Wu, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2024. Testing and Understanding Erroneous Planning in LLM Agents through Synthesized User Inputs. *arXiv preprint arXiv:2404.17833* (2024).
[59] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning. *arXiv:2311.13721* (2024).
[60] 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 *CCS*.
[61] 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).
[62] Ping Fan Ke and Ka Chung Ng. 2025. Human-AI Synergy in Survey Development: Implications from Large Language Models in Business and Research. *ACM Transactions on Management Information Systems* (2025).
[63] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level.. In *NDSS*.
[64] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
[65] 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*.
[66] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[67] 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 (2022).
[68] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Transactions on Dependable and Secure Computing* (2024).
[69] 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 *ICSE*.
[70] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *ICSE*.

[71] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Daoyuan Wu, Shuai Wang, Cuiyun Gao, and Yang Liu. 2024. Split and Merge: Aligning Position Biases in LLM-based Evaluators. In *EMNLP*.
[72] Zongjie Li, Daoyuan Wu, Shuai Wang, and Zhendong Su. 2024. API-guided Dataset Synthesis to Finetune Large Code Models. *arXiv preprint arXiv:2408.08343* (2024).
[73] 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).
[74] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. 2023. VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device. In *ASE*.
[75] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. In *NDSS*.
[76] 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).
[77] 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.
[78] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. 2023. Decompiling x86 deep neural network executables. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7357–7374.
[79] 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.
[80] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery–A Case Study *(AsiaCCS)*.
[81] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *ICSE*.
[82] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *USENIX Security 21*.
[83] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.
[84] Lily Hay Newman. 2019. The NSA makes Ghidra, a powerful cybersecurity tool, open source. https://www.wired.com/story/nsa-ghidra-open-source-tool/.
[85] nist. 2021. Juliet Test Suite for C/C++. https://samate.nist.gov/SRD/testsuite.php.
[86] National Security Agency (NSA). 2018. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.
[87] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023).
[88] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
[89] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. 2024. len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *IEEE S&P*.
[90] 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 *IEEE S&P*.
[91] 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.
[92] Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt, and Michael Franz. 2024. What you trace is what you get: dynamic stack-layout recovery for binary recompilation. In *ASPLOS*.
[93] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[94] 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 *FSE'21*. 690–702.
[95] Yi Qian, Ligeng Chen, Yuyang Wang, and Bing Mao. 2022. Nimbus: Toward speed up function signature recovery via input resizing and multi-task learning. In *IEEE QRS*.
[96] 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:2202.12336* (2022).
[97] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.
[98] 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:2308.12950* (2023).

[99] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker *(USENIX ATC'12)*. 28–28.
[100] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. Fox: Coverage-guided fuzzing as online stochastic control. In *CCS*.
[101] 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.
[102] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *ICSE*.
[103] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. *arXiv preprint arXiv:2403.05286* (2024).
[104] 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).
[105] Huaijin Wang, Zhibo Liu, Yanbo Dai, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. Preserving Privacy in Software Composition Analysis: A Study of Technical Solutions and Enhancements . In *ICSE*.
[106] Huaijin Wang, Zhibo Liu, Shuai Wang, Ying Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2024. Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis. In *IEEE EuroS&P '24*.
[107] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Transactions on Software Engineering and Methodology* (2023).
[108] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Enhancing dnn-based binary code function search with low-cost equivalence checking. *IEEE TSE* (2022).
[109] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2020. Generating effective software obfuscation sequences with reinforcement learning. *IEEE Transactions on Dependable and Secure Computing* (2020).
[110] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID 2019*. 1–15.
[111] 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*.
[112] 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*.
[113] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. 627–642.
[114] 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 *ASPLOS*.
[115] Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. Binaug: Enhancing binary similarity analysis with low-cost input repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.
[116] Wai Kin Wong, Huaijin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving Deep Neural Networks-Based Binary Code Matching with Adversarial Programs. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.
[117] Xuansheng Wu, Wenlin Yao, Jianshu Chen, Xiaoman Pan, Xiaoyang Wang, Ninghao Liu, and Dong Yu. 2024. From Language Modeling to Instruction Following: Understanding the Behavior Shift in LLMs after Instruction Tuning. *NAACL* (2024).
[118] Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is dpo superior to ppo for llm alignment? a comprehensive study. *arXiv preprint arXiv:2404.10719* (2024).
[119] 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).
[120] Xiangzhong Yu, Wai Kin Wong, and Shuai Wang. 2024. EMI Testing of Large Language Model (LLM) Compilers. In *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*.
[121] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc.
[122] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *34rd USENIX Security Symposium (USENIX Security 25)*.
[123] 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 *IEEE S&P*.
[124] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *ASPLOS*.

Received 2025-02-27; accepted 2025-03-31