PseudoFix: Refactoring Distorted Structures in Decompiled C Pseudocode

Gangyang Li[1], Xiuwei Shang[1], Shaoyin Cheng[1,2,†], Junqi Zhang[1,2,†], Li Hu[1], Xu Zhu[1], Weiming Zhang[1,2], Nenghai Yu[1,2]
{ligangyang,shangxw,pdxbshx,zhuxu24}@mail.ustc.edu.cn
{sycheng,jqzh,zhangwm,ynh}@ustc.edu.cn
[1]University of Science and Technology of China, Hefei, China
[2]Anhui Province Key Laboratory of Digital Security, Hefei, China

***Abstract*—Decompilation can convert binary programs into clear C-style pseudocode, which is of great value in a wide range of security applications. Existing research primarily focuses on recovering symbolic information in pseudocode, such as function names, variable names, and data types, but neglecting structural information. We observe that even when symbolic information is fully preserved, severe and complex structure distortions remain in the pseudocode, greatly impairing code readability and comprehension. In this work, we first systematically investigate structure distortions in decompiled pseudocode, revealing their variation patterns through quantitative analysis. Using open coding, we derive a taxonomy comprising six top-level categories of structure distortions. Building upon this taxonomy, we propose PseudoFix, a novel framework that combines large language models (LLMs) with retrieval-based in-context learning. PseudoFix employs semantic retrieval to select the most relevant few-shot examples that provide structure distortion knowledge, and combines this with the well-structured coding patterns learned by LLMs from vast source code repositories, to efficiently refactor distorted pseudocode. Comprehensive evaluations demonstrate that PseudoFix significantly improves pseudocode readability, achieving up to a 34% reduction in Halstead Complexity Effort and a 105% increase in BLEU-4 score. Notably, it significantly outperforms state-of-the-art approaches in both temporary variable elimination and goto statement removal tasks. Additionally, human evaluations yield consistently positive feedback from users across readability, consistency, and reasonability.**

***Index Terms*—Decompilation, Refactor Structure Distortion, Taxonomy, In-context Learning, Large Language Models.**

I. Introduction

Reverse engineering analyzes software by working backward from its final form to understand its design, structure, and functionality, which plays a crucial role in vulnerability discovery [1]–[3], malware analysis [4]–[6], and code plagiarism detection [7]–[9]. Although source code offers the clearest view of program logic, commercial off-the-shelf (COTS) software is typically released as compiled binaries without source code, so reverse engineering often relies on decompilation to reconstruct and analyze functionality from these executables. Modern decompiler (e.g., IDA Pro [10], Ghidra [11]) convert machine code into C-style pseudocode, improving readability over raw machine code or assembly. However, due to the inherent absence of high-level abstractions, the pseudocode often lacks meaningful semantics information (e.g., function and variable names) and differs in structure from the original source code, limiting its overall comprehensibility.

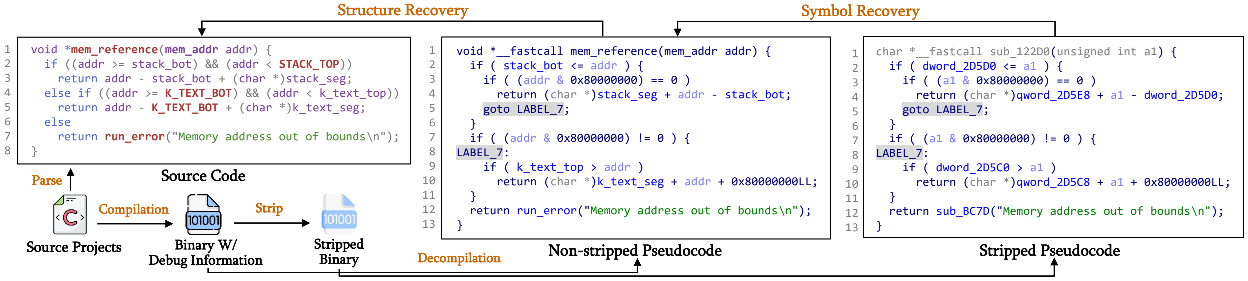
Research community has long focused on recovering symbolic information from decompiled pseudocode, such as function names [12]–[14], variable names [15]–[18], and type signatures [18]–[20]. In contrast, structural reconstruction have been largely overlooked. Notably, even with fully preserved symbolic information, significant structural discrepancies remain between decompiled pseudocode and the original source code, severely impairing readability. Figure 1 shows an example comparing source code and two decompilation outputs, one retaining symbols and the other stripped. Both maintain semantic equivalence but differ structurally. We are the first to term this phenomenon as **Structure Distortion**, emphasizing structural inconsistency beyond symbol loss.

Structure distortions stem from decompilers' imperfect recovery of the Control Flow Graph (CFG), which directly impacts the structural quality of decompiled code. During compilation, the hierarchical structure of source code is flattened into linear assembly instructions, causing irreversible loss of original structural information and making CFG recovery inherently complex. Modern decompilers predominantly rely on pattern matching binary code against predefined templates in a pattern library to reconstruct high-level control structures [21]–[23]. However, when patterns are unrecognized, decompilers fall back to using `goto` statements to approximate the original program logic. Although `goto` statements can accurately represent machine-level execution flow, they severely hinder code readability and is generally a last resort. As shown in Figure 1, successfully matched patterns yield structured constructs (e.g., if statements), while unmatched regions rely on `goto`, substantially degrading readability. Notably, we have examined all existing decompilers and observed varying degrees of structure distortions, highlighting this as a fundamental problem requiring urgent attention.

Preliminary Study. Before addressing this structure distortion problem, we first conduct a quantitative analysis to accurately assess the severity of distortions and identify the most affected structures. Based on this, we categorize the types of structure distortions to enable a divide-and-conquer approach. Specifi-

[†] Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00075



```c
void *mem_reference(mem_addr addr) {
  if ((addr >= stack_bot) && (addr < STACK_TOP))
    return addr - stack_bot + (char *)stack_seg;
  else if ((addr >= K_TEXT_BOT) && (addr < k_text_top))
    return addr - K_TEXT_BOT + (char *)k_text_seg;
  else
    return run_error("Memory address out of bounds\n");
}
```

```c
void *__fastcall mem_reference(mem_addr addr) {
  if ( stack_bot <= addr ) {
    if ( (addr & 0x80000000) == 0 )
      return (char *)stack_seg + addr - stack_bot;
    goto LABEL_7;
  }
  if ( (addr & 0x80000000) != 0 ) {
LABEL_7:
    if ( k_text_top > addr )
      return (char *)k_text_seg + addr + 0x80000000LL;
  }
  return run_error("Memory address out of bounds\n");
}
```

```c
char *__fastcall sub_122D0(unsigned int a1) {
  if ( dword_2D5D0 <= a1 ) {
    if ( (a1 & 0x80000000) == 0 )
      return (char *)qword_2D5E8 + a1 - dword_2D5D0;
    goto LABEL_7;
  }
  if ( (a1 & 0x80000000) != 0 ) {
LABEL_7:
    if ( dword_2D5C0 > a1 )
      return (char *)qword_2D5C8 + a1 + 0x80000000LL;
  }
  return sub_BC7D("Memory address out of bounds\n");
}
```

Fig. 1. Motivated Example. Comparison of Source code with Non-stripped Pseudocode and Stripped Pseudocode.

cally, we analyze outputs from two representative decompilers, commercial IDA Pro and open-source Ghidra, using open coding [24] combined with thematic analysis [25], identify six principal categories of structure distortions. Combining the insights gained from these analyses with the fact that Large Language Models (LLMs) have demonstrated deep code understanding and generation capabilities in a range of code-intensive tasks, we recognize the potential of using LLMs to address the problem of structure distortion. However, the lack of understanding of the concept of structure distortion has limited the application of LLMs in this field.

Our Methodology. To address this limitation, we propose PseudoFix, a retrieval-based in-context learning framework using LLMs for refactoring structure distortions in pseudocode. PseudoFix fully exploits the in-context learning capability of LLMs by providing few-shot examples that encapsulate structure distortion patterns, to identify and refactor such distortions in target pseudocode. Since the selection of few-shot examples significantly impacts the final performance, we retrieve pseudocode and source code pairs most semantically similar to the target pseudocode as examples. To support retrieval, we automatically annotate six types of structure distortions, constructing a dataset with 500 samples per distortion type. To handle the randomness of LLM outputs, we also design a twin-closure algorithm that performs semantic equivalence verification based on observed external program behaviors, ensuring each refactor is semantically consistent.

We comprehensively evaluate PseudoFix. On correctness, an average of 77.83% of the refactored pseudocode successfully restores the structure of the source code across all types of distortions. Regarding readability, the refactored pseudocode demonstrates a 34% reduction in Halstead Complexity Effort and a 105% improvement in BLEU-4 score compared to the original pseudocode. Comparative analysis of two distortion types reveals PseudoFix significantly outperforms the baseline, achieving state-of-the-art results. Finally, human evaluations provided positive feedback across three criteria.

Contributions. Our paper makes four key contributions.

- **Taxonomy**: We formally define structure distortion in decompiled pseudocode for the first time, quantify its severity, and systematically categorize it into six types.
- **Method**: We propose PseudoFix, a framework that leverages the structural knowledge of programming conventions learned by LLMs from source code, and retrieval-based in-context learning to identify and refactor distorted pseudocode.
- **Dataset**: We release a dataset containing an average of 500 pseudocode–source code mapping pairs per distortion type, automatically annotated using LLMs, providing a solid data foundation for future research.
- **Insight**: We conduct a thorough evaluation and demonstrate the effectiveness of PseudoFix, which significantly improves the readability of decompiled pseudocode.

Artifacts. To facilitate future research, we have open-sourced both our PseudoFix and the structure distortion dataset.[1]

II. Background and Related Work

A. Decompilation and Decompiler

Decompilation is the inverse process of compilation, aiming to reconstruct the source code structure from compiled binary code. Conceptually, decompilation techniques, sharing symmetric requirements with compilation techniques, emerged almost concurrently and have accumulated profound theoretical foundations over nearly six decades of development. In practice, while compiler development has increasingly embraced open-source collaboration, decompilers have largely remained proprietary. The market is dominated by commercial tools such as IDA Pro [10], Binary Ninja [26], and JEB [27], whereas open-source alternatives like Ghidra [11] and RetDec [28] attract comparatively less attention.

From a technical perspective, decompilers employ a three-stage workflow. The frontend of a decompiler adapts to binary programs of different architectures and generates a unified intermediate representation (IR) (e.g., microcode in IDA or p-code in Ghidra). The middle-end performs structural analysis, optimizes the IR, and reconstructs the control flow and data flow. The backend generates the abstract syntax tree (AST) and synthesizes C-style pseudocode. Despite decades of research in decompilation, recent evaluation studies [21], [29], [30] highlight persistent challenges in correctness and readability, underscoring the need for further advancements in the field.

B. Large Language Models (LLMs)

Generally speaking, LLMs are based on the Transformer [31] architecture, which is developed for sequence transduction tasks. Following their success in Natural Language

[1] https://github.com/giles-one/PseudoFix

Processing (NLP), pretrained LLMs have been rapidly adapted to the source code domain, showing promising performance in code generation [32], code comprehension [33], and automated program repair [34]. Although decompiled pseudocode shares morphological similarities with source code, the absence of symbolic information and the presence of distorted structures makes this task distinct from source code processing.

When adapting LLMs to new tasks, two primary approaches are typically employed: fine-tuning and in-context learning. Fine-tuning often requires large amounts of high-quality labeled data and incurs prohibitive computational costs for modern models with tens or hundreds of billions of parameters. In contrast, in-context learning does not require updating the model parameters. In-context learning was popularized by GPT-3 [35] as an effective paradigm for task adaptation in LLMs. In this framework, a small number of demonstration examples (input-output pairs) are provided to LLMs before actual test case for prediction.

C. *Pseudocode Refactoring with LLMs*

Given the various defects present in decompilation and the powerful capabilities of LLMs, the community has been actively attempting to use LLMs for pseudocode refactoring. LLM4Decompile [36] takes the source code as its target, fine-tuned on five million C binary functions. It not only attempts to use LLMs to directly decompile machine code into pseudocode, but also to refine the outputs from Ghidra to solve readability issues and syntax errors. DeGPT [37] introduces a "three-role" mechanism that leverages an existing LLM, rather than training a new one, to refactor pseudocode, thereby improving its readability by recovering identifiers, adding comments, and removing redundant variables. DecLLM [38] focuses on the issue of recompiling decompiled pseudocode by recursively prompting an LLM to fix various errors during the compilation and execution process. While prior research has focused on restoring semantic information such as identifier names and comments, our work is the first to address structural deficiencies in pseudocode and propose a taxonomy of distortions. Furthermore, we utilize a retrieval-based in-context learning approach to unlock the refactoring capabilities of LLMs, enabling the effective identification and correction of these structural problems.

III. Preliminary Study

As shown in Figure 1, while we intuitively recognize the distorted pseudocode as challenging to understand, we cannot precisely identify the specific causes or measure the degree of distortion. Before diving into resolving these structure distortions, we first conduct an exploratory study on this issue from both quantitative and qualitative perspectives.

A. *Quantification of Structure Distortions*

Since decompiled pseudocode follows the syntax of the C language, we employ complexity metrics originally designed for source code to investigate the structural properties of pseudocode. It should be noted that our analysis focuses exclusively on structure complexity, while semantic complexities brought by symbols are not within our consideration.

TABLE I
Variation in Code Complexity. ↑ and ↓ Represent a Relative Increase and Decrease

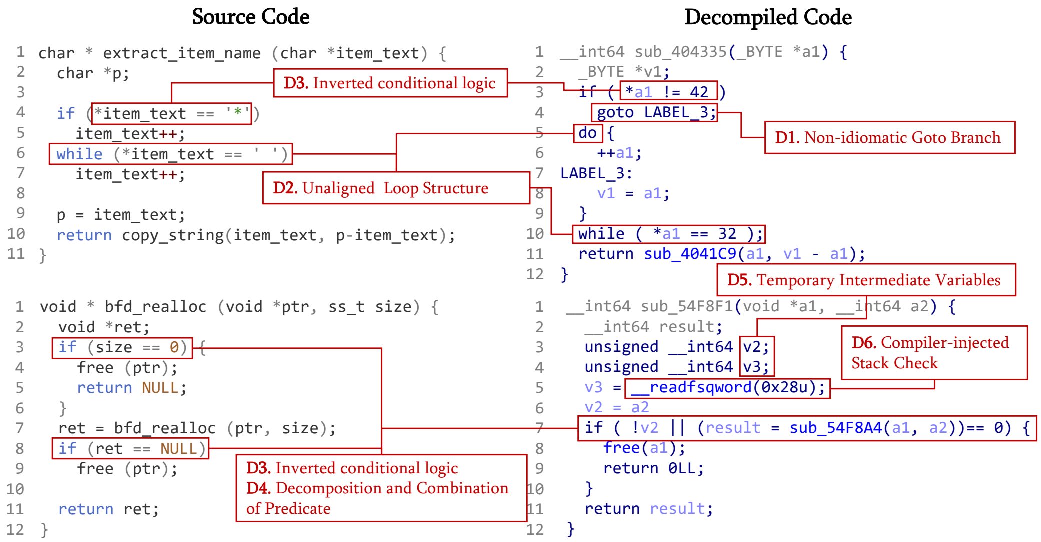
Code Form		LOC	ND	CC	Halstead Complexity		
					V($\times 10^2$)	D	E($\times 10^4$)
Source Code		34.4	4.0	8.3	14.1	20.6	5.5
IDA Pro Pseudo Code	O0	45.9(↑33%)	3.9(↓3%)	9.1(↑10%)	16.9(↑20%)	23.9(↑16%)	8.1(↑46%)
	O1	67.7(↑97%)	3.5(↓11%)	11.9(↑44%)	22.9(↑62%)	25.9(↑26%)	13.5(↑142%)
	O2	75.6(↑119%)	3.6(↓8%)	12.6(↑52%)	25.9(↑83%)	26.2(↑27%)	15.4(↑177%)
	O3	97.1(↑181%)	4.1(↑3%)	15.5(↑87%)	31.6(↑124%)	29.4(↑43%)	21.8(↑293%)
	Avg.	71.6(↑108%)	3.8(↓5%)	12.3(↑48%)	24.3(↑72%)	26.4(↑28%)	14.7(↑165%)
Ghidra Pseudo Code	O0	45.7(↑33%)	2.5(↓36%)	9.1(↑9%)	15.0(↑6%)	26.0(↑26%)	7.2(↑30%)
	O1	61.7(↑79%)	2.7(↓31%)	12.1(↑46%)	18.9(↑34%)	28.8(↑40%)	11.4(↑106%)
	O2	63.6(↑85%)	2.7(↓31%)	12.3(↑49%)	20.1(↑42%)	28.2(↑37%)	11.9(↑115%)
	O3	81.0(↑135%)	3.0(↓24%)	15.1(↑82%)	24.1(↑71%)	32.4(↑57%)	17.0(↑207%)
	Avg.	63.0(↑83%)	2.7(↓31%)	12.2(↑46%)	19.5(↑38%)	28.9(↑40%)	11.9(↑114%)

1) Metrics: We evaluate the degree of structure distortion using the following four metrics:

- **LOC (Lines of Code)** indicates the structural complexity through *code length*.
- **ND (Nesting Depth)** indicates the structural complexity through *hierarchical depth*.
- **CC (Cyclomatic Complexity)** quantifies the number of linearly independent paths by calculating `if` statements and conditional loops to indicate its complexity.
- **Halstead Complexity** is a composite metric with three measurable attributes. *V (Volume)* measures the volume of the code, where lower values typically indicate more concise code. *D (Difficulty)* assesses the cognitive complexity of understanding the code, with lower values suggesting easier comprehension. *E (Effort)* estimates the workload required to modify or review the code.

2) Datasets: We collect, compile, and process 51 GNU projects using BinKit [39]. These open-source projects have undergone extensive validation over time [40]–[42], ensuring high quality and credibility. To obtain pseudocode, we employ two widely used decompilers, IDA Pro and Ghidra, which represent commercial and open-source solutions respectively. Notably, for each project, we compile binaries exclusively for the x86_64 architecture with optimization levels ranging from O0 to O3. The x86_64 architecture is selected due to its widespread adoption. Furthermore, since decompiler front-ends transform machine code from different architectures into a unified IR, the architectural differences have minimal impact on the structure of the pseudocode.

3) Results: We calculate the structural complexity of the source code as the baseline and compare it with pseudocode under different optimization levels, presenting the results in Table I. For the O3 optimization, we observe a maximum increase of 293% in the Halstead Complexity Effort metric for pseudocode decompiled by IDA Pro. However, Effort is a composite metric derived from V and D, which are calculated based on keywords (e.g., `if`, `for`, `while`, `return`) and



Source Code

```c
char * extract_item_name (char *item_text) {
  char *p;

  if (*item_text == '*')
    item_text++;
  while (*item_text == ' ')
    item_text++;

  p = item_text;
  return copy_string(item_text, p-item_text);
}
```

```c
void * bfd_realloc (void *ptr, ss_t size) {
  void *ret;
  if (size == 0) {
    free (ptr);
    return NULL;
  }
  ret = bfd_realloc (ptr, size);
  if (ret == NULL)
    free (ptr);

  return ret;
}
```

Decompiled Code

```c
__int64 sub_404335(_BYTE *a1) {
  _BYTE *v1;
  if ( *a1 != 42 )
    goto LABEL_3;
  do {
    ++a1;
LABEL_3:
    v1 = a1;
  }
  while ( *a1 == 32 );
  return sub_4041C9(a1, v1 - a1);
}
```

```c
__int64 sub_54F8F1(void *a1, __int64 a2) {
   __int64 result;
   unsigned __int64 v2;
   unsigned __int64 v3;
   v3 = __readfsqword(0x28u);
   v2 = a2
   if ( !v2 || (result = sub_54F8A4(a1, a2))== 0) {
     free(a1);
     return 0LL;
   }
   return result;
}
```

D3. Inverted conditional logic
D2. Unaligned Loop Structure
D3. Inverted conditional logic
D4. Decomposition and Combination of Predicate
D1. Non-idiomatic Goto Branch
D5. Temporary Intermediate Variables
D6. Compiler-injected Stack Check

Fig. 2. Taxonomy of Structure Distortions.

operators (e.g., &&, !=). This finding aligns the 87% structural complexity increase measured by Cyclomatic Complexity under O3 optimization. LOC also shows a similarly significant impact, increasing by 181% under O3 optimization.

The Nesting Depth exhibits both increases and decreases across different optimization levels. This suggests that it is a complex metric influenced by multiple factors, resulting in non-uniform trends. Furthermore, we observe similar variations across different decompilers, with Ghidra exhibiting a value up to six times higher than that of IDA Pro.

In summary, the increase in Halstead Complexity and Cyclomatic Complexity in the pseudocode intuitively suggests significant structure distortions occurring in the loops and branches. Meanwhile, the complex variations in Nesting Depth indicate structure changes involving the merging and splitting of branches to varying degrees. Additionally, the rise in LOC metric indicates structure changes in the newly added lines.

B. *Taxonomy of Structure Distortions*

Existing research has extensively focused on taxonomies of decompiler fault [21] and fidelity [30], while the structure distortions of pseudocode lack a comprehensive taxonomy and consequently remain unresolved. Therefore, in this section, we attempt to summarize the patterns of structure distortions and establish a systematic taxonomy. It should be clarified that we focus solely on structure distortions inherently introduced by decompilers, and exclude the impact of code obfuscation, which artificially introduces structure obfuscation.

1) Preparation: We employ an iterative open-coding [24] and thematic analysis [25] approach to establish the taxonomy of structure distortion issues in decompiled pseudocode. Specifically, we identify minimal samples exhibiting structure discrepancies between pseudocode and source code to construct a codebook. Subsequently, we inductively derive conceptual themes to generate a bottom-up taxonomic framework.

Three reverse engineering experts, each with over three years of professional experience, participated in this process. Using the dataset described previously, we randomly select 600 samples and divide them into working, testing, and validation sets in an 8:1:1 ratio.

2) Open Coding Execution: We conduct iterative coding following the four-step procedure outlined below.

Recording Discrepancies. Participants examine the working set and identify samples with structural discrepancies between pseudocode and source code based on their professional expertise. When multiple structural discrepancies are encountered in one sample, they are documented in a memo.

Building Codebook and Themes. Samples exhibiting structural discrepancies are deconstructed to establish initial concepts of structural differences, thereby generating a codebook. Core patterns and meaningful units that recurred in the data are inductively analyzed to formulate themes.

Testing. The constructed themes are tested on the test set. If any structure variations remain unexplained, steps 1 and 2 are repeated to incorporate additional themes. The coding process concludes when no new themes emerge.

Validation. We validate the classifications across different decompilers and participants using the validation set. The results show Krippendorff's Alpha coefficients of 0.77 and 0.81, respectively, indicating substantial inter-rater reliability.

3) Taxonomy: We establish a taxonomy consisting of six top-level categories, the results are demonstrated in Figure 2.

D1. Non-idiomatic Goto Branch. The goto statement represents an unconditional branch that arbitrarily alters control flow to a target label. In good programming practice, goto is discouraged as it can lead to the formation of spaghetti code. From a decompiler's perspective, goto is semantically equivalent to the jmp instruction in machine code and can thus be directly translated from machine instructions. Consequently, when pattern matching for higher-level constructs fails, goto serves as the last resort for synthesizing the final pseudocode.

D2. Unaligned Loop Structure. The C language features three loop constructs: `for`, `while`, and `do while`. Each loop type serves different purposes, with `do while` being the least frequently used as it guarantees at least one iteration. As shown in Figure 2, a concise `while` loop is decompiled into a `do while` construct with a distorted loop body. Additionally, D2 comprises several other subcategories, such as the `for` loops being erroneously converted into `while` loops.

D3. Inverted Conditional Logic. Compilers select machine instructions like `jz` (i.e., Jump if Zero) and `jnz` based on branch properties and optimization choices, while decompilers faithfully translate these jumps. This can result in logically inverted branches, while semantically equivalent, negatively impacting code indentation and subsequent structural analysis. We provide a standalone example in Figure 3 that exclusively exhibits this issue. The code's nesting depth triples from 1 to 3 and its branch count doubles from 3 to 6, an effect that grows exponentially with source code complexity.

D4. Decomposition and Combination of Predicates. Decompilers typically decide whether to combine predicates based on the distance in the machine code. Due to compilation optimizations, predicates and branches in the source code often undergo transformations, which can lead to incorrect predicate decomposition and combination by the decompiler.

D5. Temporary Intermediate Variables. Local variables are typically stored in stack frames or registers and move between memory locations and registers for computation. In this process, intermediate computational values may be incorrectly identified as distinct variables by decompilers.

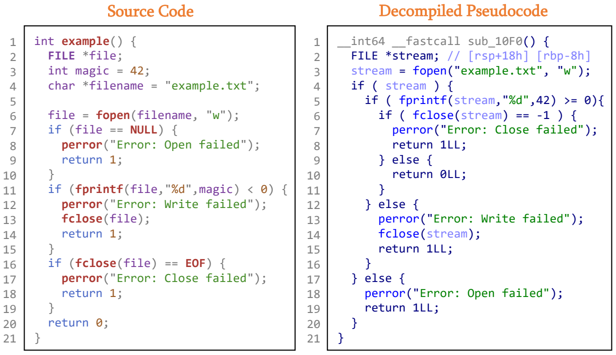
D6. Compiler-injected Stack Check. The stack canary implemented via `__readfsqword(0x28)` is automatically inserted by the compiler for overflow detection and has no corresponding representation in the source code.

During the open coding process, we frequently encounter instances of D1, D2, and D4, which contributed to the significant increase in Cyclomatic Complexity and Halstead Complexity observed in the quantitative analysis presented in the previous section. In D4, decomposition and combination together leads to complex variations in Nesting Depth. Increased combinatorial distortion occurs, ultimately resulting in an overall reduction of Nesting Depth. Additionally, the impacts of D5 and D6 directly led to an increase in lines of code.

C. Insights and Opportunities

After completing the taxonomy of structure distortions, we interview the three participating reverse engineering experts to discuss the identification and refactoring of such distortions. Our discussions yield several critical insights.

First, human experts emphasize ① the importance of experience. The accumulated expertise in good coding practices from daily learning enables us to identify various structure distortions, while reverse engineering experience allows us to understand the underlying principles behind these distortions, thereby recovering their original structure. Second, human experts also indicate that ② a given structural distortion typically exhibits similar patterns. Additionally, human experts also ③ reject the rule-based approaches to address this issue, as the structure distortions in pseudocode inherently stem from imperfect pattern matching in decompilers.



Source Code

```
int example() {
  FILE *file;
  int magic = 42;
  char *filename = "example.txt";

  file = fopen(filename, "w");
  if (file == NULL) {
    perror("Error: Open failed");
    return 1;
  }
  if (fprintf(file,"%d",magic) < 0) {
    perror("Error: Write failed");
    fclose(file);
    return 1;
  }
  if (fclose(file) == EOF) {
    perror("Error: Close failed");
    return 1;
  }
  return 0;
}
```

Decompiled Pseudocode

```
__int64 __fastcall sub_10F0() {
  FILE *stream; // [rsp+18h] [rbp-8h]
  stream = fopen("example.txt", "w");
  if ( stream ) {
    if ( fprintf(stream,"%d",42) >= 0){
      if ( fclose(stream) == -1 ) {
        perror("Error: Close failed");
        return 1LL;
      } else {
        return 0LL;
      }
    } else {
      perror("Error: Write failed");
      fclose(stream);
      return 1LL;
    }
  } else {
    perror("Error: Open failed");
    return 1LL;
  }
}
```

Fig. 3. An Illustrative Example of Inverted Conditional Logic.

Base on the aforementioned insights and recent advancements related to decompiled pseudocode [40], [43], [44], we have opted to employ LLMs for identifying and refactoring structure distortions. During the training process, LLMs have acquired sufficient source code knowledge, thereby obtaining well-structured programming experience akin to human developers. Moreover, LLMs have demonstrated superior capabilities in code comprehension and generation compared to human engineers. However, in specialized domains such as reverse engineering, LLMs exhibit suboptimal performance due to existing knowledge gaps. Given the inherent generalization capabilities of LLMs, we have chosen to adopt an in-context learning strategy, where structure distortion patterns is provided to the model through few-shot demonstrations.

IV. Methodology

The preliminary study quantifies structure distortions and establishes a systematic taxonomy, offering insights into leveraging LLMs. Building upon these findings, in this section, we propose PseudoFix, a retrieval-based in-context learning framework to identify and refactor distorted pseudocode.

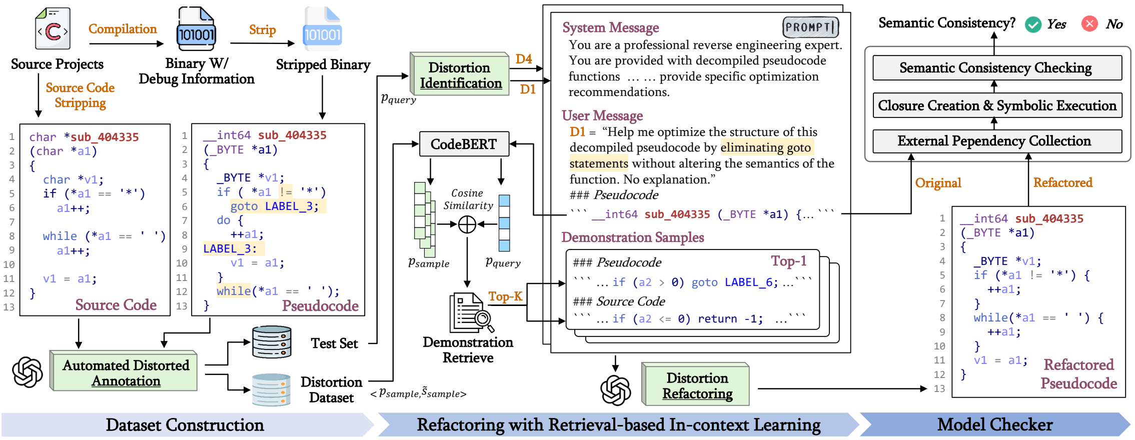
A. Problem Formulation

We define structure distortion as the discrepancy Δ between decompiled pseudocode P and its corresponding source code S, where the two representations are semantically equivalent but exhibit structural inconsistencies as Equation 1. It is categorized into six types $\mathcal{D}_1, ..., \mathcal{D}_6$ according to the magnitude of structural discrepancy δ.

$$\Delta(S, P) \iff (S \equiv_{\text{sem}} P) \land (S \not\equiv_{\text{struct}} P) \tag{1}$$

We formulate the structure distortion refactoring as an optimization problem, aiming to minimize the structural discrepancy between P' and S as Equation 2, where P' represents the optimized result of P.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{6} \delta_i(S, P') \\ \text{subject to} \quad & P' = \text{Refactor}(P), \\ & \Delta(S, P) \text{ holds.} \end{aligned} \tag{2}$$



```
char *sub_404335
(char *a1)
{
  char *v1;
  if (*a1 == '*')
    a1++;

  while (*a1 == ' ')
    a1++;

  v1 = a1;
}
```

```
__int64 sub_404335
(_BYTE *a1)
{
  _BYTE *v1;
  if ( *a1 != '*')
    goto LABEL_3;
  do {
    ++a1;
LABEL_3:
    v1 = a1;
  }
  while(*a1 == ' ');
}
```

System Message
You are a professional reverse engineering expert. You are provided with decompiled pseudocode functions … … provide specific optimization recommendations.

User Message
D1 = "Help me optimize the structure of this decompiled pseudocode by eliminating goto statements without altering the semantics of the function. No explanation."
Pseudocode
``` __int64 sub_404335 (_BYTE *a1) {… ```

Demonstration Samples
Pseudocode
``` … if (a2 > 0) goto LABEL_6; …```
Source Code
``` … if (a2 <= 0) return -1; …```

```
__int64 sub_404335
(_BYTE *a1)
{
  _BYTE *v1;
  if (*a1 != '*') {
    ++a1;
  }
  while(*a1 == ' ') {
    ++a1;
  }
  v1 = a1;
}
```

Fig. 4. Overview of PseudoFix.

We illustrate the overview of the PseudoFix in Figure 4. Given an input decompiled pseudocode p_{query}, we prompt the LLMs to perform structure distortion $\mathcal{D}_i$ identification and refactoring using the prompt template as Equation 3, where $\mathcal{TI}$ represents the task instruction that specifies the intent, while $\mathcal{FE}$ denotes few-shot examples consisting of a series of input-output demonstrations that serve as empirical reverse engineering knowledge. A well-crafted Prompt should guide the LLMs to output $\{\texttt{Yes}, \texttt{No}\}$ for the structure distortion identification, and generate p' that minimizes $\delta_i(s, p')$.

$$\text{Prompt} = \{\mathcal{TI}_{\mathcal{D}_i} + \mathcal{FE} + p_{\text{query}}\} \qquad (3)$$

B. Dataset Construction

1) Automated Distorted Example Annotation: Since real-world datasets often exhibit complex patterns and contain multiple types of distortions, we propose an automated distortion annotation method based on few-shot learning. Initially, we manually annotate a small number of samples for each type of distortion, ensuring that these samples capture the essential characteristics of the distortion and contain only one type of distortion while remaining representative. These manually annotated samples are referred to as *golden examples*. They are incorporated as part of the prompt input to the model, guiding it to determine whether the query sample $(p_{\text{query}}, s_{\text{query}})$ exhibits the specified type of distortion. Notably, the manual annotation effort is negligible, and due to the powerful in-context learning capabilities of LLMs, in our work, only three golden examples per distortion category are required.

We evaluate each distortion type as a separate binary classification task instead of performing multi-class classification iteratively. This strategy addresses the model's limited context length, which restricts the number of samples available for all distortion types simultaneously. By concentrating on one distortion type at a time and constraining responses to $\{\texttt{Yes}, \texttt{No}\}$ answers, we effectively reduce the risk of hallucination.

2) Source Code Stripping and Pseudocode Indexing: Our constructed distortion dataset contains sample pairs (p, s) with an overlooked issue of symbolic discrepancy between the source code s and its corresponding pseudocode p. Source code s contains semantically meaningful variable and function names, while pseudocode p, reconstructed from stripped binary files, uses generic identifiers (e.g., `v1`, `sub_B240`) that cannot be mapped back to s. Our work focuses on structure distortion rather than symbolic discrepancies, and the latter may impair model comprehension. To eliminate the symbolic information in the source code s while preserving only structural features, we propose a novel source code stripping technique. By utilizing binary files with debug information as an intermediate medium, we indirectly align variables between the source code s and the stripped pseudocode p. Specifically, we replace variable names of s with generic identifiers of p, extending this to other symbols. The stripped output, denoted $\tilde{s}$, retains only structural features.

We construct this distortion database to provide retrievable structure distortion $(p, \tilde{s})$ pairs for a given query p_{query}, where the retrieval process uses pseudocode as key. Accordingly, we enable pseudocode indexing with two objectives: (1) mapping p to `Yes` for the structure distortion identification task, and (2) mapping p to $\tilde{s}$ for the structure distortion refactoring task.

C. Refactoring with Retrieval-based In-context Learning

1) In-context Learning with LLMs: As shown in Figure 4, we implement an in-context learning approach where the LLM receives task descriptions of structure distortion along with reference examples (i.e. the context). This enables the model to identify and refactor structure distortions without parameter update training. This essentially constitutes a prompting-based meta-learning method, where the model rapidly adapts to novel domain tasks through forward computation rather than gradient descent. Following the standard in-context learning prompt paradigm, we distinguish between zero-shot learning (no examples) [45]–[47] and few-shot learning (multiple examples). The few-shot examples serve dual purposes: (1) establishing output format standards, and (2) offering structure distortion patterns similar to the target pseudocode to assist the model in identifying and refactoring distorted pseudocode. Consequently, the selection of few-shot examples is critical and ultimately determines the model's performance.

2) Retrieval-based Few-shot Selection: We select few-shot examples by retrieving the Top-K most relevant samples from the structure distortion dataset that match the given pseudocode p_{query} using information retrieval techniques [48]. Traditional frequency-based retrieval methods (e.g., BM-25 [49]) rank text similarity at the lexical level. However, since the symbolic information in pseudocode is stripped away, identifiers such as variable and function names are reconstructed based on predefined templates. Consequently, different pseudocode segments often contain numerous same tokens, rendering such frequency-based approaches unsuitable.

Beyond frequency, we employ a semantics-based approach to retrieve the most relevant examples. Specifically, the distortion database stores $(p_{\text{sample}}, \tilde{s}_{\text{sample}})$ pairs. When we receive a pseudocode query p_{query}, we transform it and each p_{sample} in the dataset into embedding vectors in semantic space and rank their relevance by computing cosine similarity. When the distorted sample is selected, p_{sample} and $\tilde{s}_{\text{sample}}$ will be integrated into our prompt as few-shot examples. For this purpose, we employ a pre-trained CodeBERT [50] as our embedding model. Then we fine-tune this model on 1.4 million pseudocode functions collected from the Arch User Repository [51] to enhance its domain adaptation capabilities for our task. The stripping of symbolic information causes the semantics of pseudocode to manifest primarily in its structure. Therefore semantic similarity detection can provide valuable references for identifying and refactoring distorted structures.

3) Prompt Builder: Our framework fundamentally comprises two key tasks: distortion identification and distortion refactoring. For each input pseudocode p_{query}, our method iteratively examines all potential distortion types. Upon detecting a specific distortion, PseudoFix performs the corresponding refactoring. This identify-then-refactor paradigm is intentionally designed to enhance the interpretability of our approach.

Prompt Design for Distortion Identification. We first employ a role-playing prompt method [52] in the task instruction $\mathcal{TI}$ to position the model as a "reverse engineering expert". Subsequently, we explicitly describe the distortion pattern to precisely convey the task requirements. The output format is constrained to {`Yes`, `No`}, and we prohibit model explanations to minimize unnecessary consumption. Furthermore, we retrieve the most relevant examples to p_{query} as context to guide the model toward generating the desired results.

Prompt Design for Distortion Refactoring. We follow a similar template to prompt the model for distortion refactoring, as shown in Figure 4. It is worth noting that during the refactoring process in $\mathcal{TI}$, we strictly require that the semantics of p_{query} must not be altered, as equivalence serves as the fundamental requirement for our pseudocode refactoring. Furthermore, unlike distortion identification, the retrieved examples consist of $(p, \tilde{s})$ pairs, which provide the model with relevant distortion refactoring patterns as context.

Algorithm 1: Twin-Closure Equivalence Verification

```
Input: Original program $P$, Optimized program $P'$
Output: Boolean value indicating semantic equivalence
seed $\leftarrow 42$;
$\mathcal{C} \leftarrow \emptyset, \mathcal{G} \leftarrow \emptyset$;
// Dependency collection
foreach node $v \in AST(P \cup P')$ do
    if $v.is\_external\_call()$ then
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{v\}$ ;      // add external calls
    end
    if $v.is\_out\_of\_scope()$ then
        $\mathcal{G} \leftarrow \mathcal{G} \cup \{v\}$ ;   // add external globals
    end
end
HOOK($\mathcal{C}$);
randomize($\mathcal{G}$, seed);
// Twin closure with shared environment
$p, q \leftarrow$ createClosure($\mathcal{C}, \mathcal{G}, P, P'$);
symbolicExec($p, q$, randomInput(seed));
// equivalence check
return returnEqu($p, q$) $\wedge$ globalsEqu($p, q$) $\wedge$ callSeqEqu($q, p$)
```

D. Model Checker

In our work, the fundamental theory underlying the refactoring of structure distortions is to gradually optimize the pseudocode p_{query} while preserving semantic consistency, such that it approximates the ideal source code structure $\tilde{s}_{\text{query}}$. It is essential that each LLM output remains semantic equivalence with the input pseudocode. Next, we consider the semantics of the pseudocode function. From an internal perspective, the semantics primarily manifest in the function's implementation logic, including control flow, branching, and looping. From an external perspective, the semantics are reflected in the parameters and return values when the function is called, as well as its impact on the global states, such as modifications to global variables or calls to external functions.

Given that we optimize the internal structure of functions during distortion refactoring, we define the semantics of the function as its behavior on external states. Therefore, verifying the semantic consistency of functions equates to assessing the consistency of their external behaviors. We propose a novel **Twin Closure** method to verify the semantic equivalence of functions before and after refactoring, as illustrated in Algorithm 1. The algorithm takes two pseudocode functions P and P' as input and returns a boolean value indicating whether they are semantically equivalent. It consists of three steps.

External Dependency Collection. We traverse the ASTs of both functions to collect calls to external functions and references to external resources (e.g., global variables). We hook these external calls so that during symbolic execution, the invoked functions can record call details and control return values. Additionally, we initialize a symbol table for external resources with randomized values.

Closure Creation and Symbolic Execution. We construct closures by treating external resources as free variables, ensuring that both closures share the same environment. Due to the symmetric form of the two functions, we refer to them as twin closures, denoted as p and q. Based on the function signatures, we randomly generate input parameters and perform symbolic execution, during which calls to external functions and accesses to external resources are logged.

Semantic Consistency Checking. We evaluate whether the

TABLE II
CORRECTNESS OF INDIVIDUAL COMPONENTS.

Components	**DeepSeek-V3**			**GPT-4**		
	Precision	***Recall***	***F1-score***	***Precision***	***Recall***	***F1-score***
Annotation	89.61	80.65	84.90	93.28	83.35	88.04
Identification	82.75	78.37	80.50	83.86	79.78	81.77
Model Checker	91.79	82.12	86.69	91.79	82.12	86.69

TABLE III
CORRECTNESS OF DISTORTED STRUCTURE REFACTORING.

Structure Distortions	**DeepSeek-V3**				**GPT-4**			
	0-shot	***1-shot***	***5-shot***	***10-shot***	***0-shot***	***1-shot***	***5-shot***	***10-shot***
D1	87.33	90.00	96.67	**99.33**	91.33	95.33	97.33	98.67
D2	52.45	55.94	65.03	**73.43**	55.24	57.34	66.43	70.63
D3	61.98	63.64	71.90	79.34	66.94	67.77	73.55	**83.47**
D4	44.76	46.15	48.95	54.55	46.85	47.55	53.15	**62.94**
D5	31.39	35.77	44.53	46.72	29.93	33.58	45.26	**52.55**
D6	90.00	92.50	96.25	**98.75**	91.25	95.00	97.50	**98.75**

twin closures exhibit identical external behaviors by comparing their return values, global variable states, and sequences of external function call details. The final result determines whether the two functions are semantically equivalent.

Symbolic execution-based approaches inevitably encounter the path explosion problem. Inspired by latest work, we transform loop blocks into branch blocks [53] and merge multiple paths [54] to optimize our method and mitigate this problem. Furthermore, for excessively long execution paths, we implement both a maximum instruction execution length constraint and a timeout mechanism.

V. EVALUATION

We conduct experiments to evaluate the effectiveness of PseudoFix by addressing the following research questions:

- **RQ1**: To what extent can PseudoFix maintain correctness during distorted pseudocode refactoring?
- **RQ2**: To what extent does PseudoFix improve readability after distorted pseudocode refactoring?
- **RQ3**: How does PseudoFix perform compared to state-of-the-art methods?
- **RQ4**: How is the feedback from human developers on the refactored pseudocode by PseudoFix?

A. Experimental Setup

Large Language Models. We select DeepSeek-V3 [55] and GPT-4 [56] as the foundation models for our framework. GPT-4, as a member of the popular ChatGPT [57] series, demonstrates exceptional performance, while DeepSeek-V3 serves as a representative open-source model whose results can be fully reproduced. These two models are accessed via a unified API interface, with the temperature parameter set to 1.0. During the retrieval process, we employ the CodeBERT [58] model for embedding generation. We fine-tune the model on 1.4 million pseudocode functions for 3 epochs, utilizing a batch size of 16 and the AdamW optimizer with a learning rate of 2e-5. To account for uncertainty of LLM output, we repeated each experiment three times, yielding a p-value of 0.8602 (> 0.05), which indicates no significant difference and confirms the stability of our results.

Dataset. To validate correctness, We employ the same dataset as described in Section III-A2, which consists of GNU projects. The pseudocode functions are obtained through decompilation using IDA Pro 9.0 and Ghidra 11.1.2. We manually annotate 200 samples for each type of distortion (totaling 1,200 samples) as ground truth. To validate readability, we also randomly select 600 functions from four real-world malware families (Rootkit, Trojan horse, Backdoor, Worm), which are decompiled and used as the malware dataset. It should be noted that these datasets are independently processed by us, and are proprietary rather than downloaded directly from the internet. To minimize the risk of data leakage, we further strip the symbol tables, rendering the data completely devoid of symbolic features and thus difficult to identify.

Platform. All experiments are conducted on a server running Ubuntu 22.04, equipped with an Intel Xeon Gold 6330 CPU, 500GB RAM, 8TB disk storage, and 8 NVIDIA RTX A6000 GPUs for embedded generation and retrieval operations.

B. RQ1: Correctness Evaluation

We define correctness as the structure fidelity to the source code, indicating that the results identified or refactored by our method should be consistent with the original source code. The evaluation of PseudoFix's correctness is conducted in two aspects: the correctness of individual key components, and the correctness of refactored pseudocode.

1) Correctness of Individual Components: Considering that we evaluate the correctness of each component, we uniformly mix different types of distorted data for experimentation, with results presented in Table II. Both the annotation and identification of distortion type are automated processes driven by LLMs, which employ the in-context learning paradigm. But the annotation process generally achieves higher average Precision, Recall, and F1-score than distortion identification by 5.46%. This is because the input for the annotation task consists of pseudocode and source code pair (p_{query}, $\tilde{s}_{\text{query}}$), while the identification task operates on individual pseudocode function p_{query} thus the complexity is higher. And for different metrics, both tasks demonstrate higher Precision than Recall, indicating a lower false positive rate and fewer misclassified positive samples. This suggests that the automatically annotated dataset maintains high quality. Regarding model performance, GPT-4 consistently outperforms DeepSeek-V3, with an average improvement of 2.21%, and achieves a peak Precision of 93.28%.

The Model Checker relies on program analysis rather than being LLM-driven, making its performance independent of the chosen LLMs. Inspired by the CLIP [59], we construct positive and negative examples to evaluate our Twin-Closure method. As shown in Table II, our method demonstrates high Precision by leveraging the fact that semantically different

TABLE IV
VARIATIONS IN CODE COMPLEXITY AND CORRELATION. ↑AND ↓REPRESENT A RELATIVE INCREASE AND DECREASE.

Dataset	Code Form		LOC	ND	CC	Halstead Complexity			BLEU-4	METEOR	Rouge-L	Semantic Similarity
						V($\times 10^2$)	D	E($\times 10^4$)				
GNU	Source Code		35.8	3.4	5.7	11.1	19.8	2.4	/	/	/	/
	Raw PseudoCode		44.2	2.9	8.2	14.3	22.1	3.4	13.5	35.8	17.9	92.1
	DeepSeek-V3	0-shot	42.8(↓3%)	**3.0**(↑2%)	8.1(↓2%)	13.8(↓4%)	21.8(↓1%)	3.2(↓5%)	16.2(↑19%)	38.3(↑7%)	22.9(↑28%)	95.8(↑4%)
		1-shot	42.3(↓5%)	3.1(↑4%)	8.0(↓3%)	13.4(↓6%)	21.7(↓2%)	3.2(↓6%)	18.5(↑36%)	40.4(↑13%)	23.5(↑32%)	96.4(↑5%)
		5-shot	39.1(↓12%)	3.1(↑7%)	7.2(↓13%)	12.7(↓11%)	21.1(↓4%)	2.9(↓13%)	23.0(↑70%)	45.1(↑26%)	27.4(↑54%)	98.6(↑7%)
		10-shot	38.4(↓13%)	3.2(↑11%)	6.6(↓20%)	**12.3**(↓14%)	**20.8**(↓6%)	**2.7**(↓19%)	26.8(↑98%)	**48.0**(↑34%)	33.8(↑89%)	99.2(↑8%)
	GPT-4	0-shot	43.3(↓2%)	**3.0**(↑3%)	7.9(↓4%)	13.8(↓3%)	21.8(↓1%)	3.2(↓6%)	18.7(↑39%)	39.1(↑9%)	24.6(↑38%)	95.9(↑4%)
		1-shot	42.6(↓4%)	3.1(↑4%)	7.6(↓8%)	13.5(↓5%)	21.8(↓1%)	3.1(↓9%)	20.6(↑52%)	41.5(↑16%)	27.9(↑56%)	97.3(↑6%)
		5-shot	38.1(↓14%)	3.2(↑8%)	7.1(↓14%)	12.9(↓10%)	21.3(↓4%)	2.9(↓16%)	23.8(↑76%)	44.4(↑24%)	30.2(↑69%)	98.7(↑7%)
		10-shot	**36.6**(↓17%)	3.3(↑13%)	**6.3**(↓23%)	12.5(↓13%)	20.9(↓5%)	**2.7**(↓22%)	**27.8**(↑105%)	47.7(↑33%)	**35.6**(↑99%)	**99.4**(↑9%)
Malware	Raw PseudoCode		48.3	3.2	9.5	23.1	23.3	5.4	/	/	/	/
	DeepSeek-V3	0-shot	44.9(↓7%)	**3.2**(↑1%)	9.0(↓6%)	20.6(↓11%)	22.0(↓5%)	5.0(↓8%)	/	/	/	/
		1-shot	43.4(↓10%)	**3.2**(↑1%)	8.7(↓9%)	19.4(↓16%)	22.0(↓5%)	4.9(↓10%)	/	/	/	/
		5-shot	37.0(↓23%)	3.3(↑3%)	8.1(↓15%)	17.6(↓24%)	20.8(↓11%)	4.5(↓17%)	/	/	/	/
		10-shot	34.7(↓28%)	3.4(↑6%)	**7.6**(↓21%)	16.1(↓30%)	20.1(↓14%)	3.8(↓31%)	/	/	/	/
	GPT-4	0-shot	45.2(↓6%)	**3.2**(↑1%)	9.1(↓5%)	19.9(↓14%)	22.8(↓2%)	5.3(↓3%)	/	/	/	/
		1-shot	43.4(↓10%)	**3.2**(↑1%)	8.9(↓7%)	18.6(↓19%)	22.7(↓3%)	4.9(↓9%)	/	/	/	/
		5-shot	38.5(↓20%)	3.3(↑3%)	8.3(↓13%)	17.2(↓25%)	21.2(↓9%)	4.3(↓21%)	/	/	/	/
		10-shot	**31.8**(↓34%)	3.4(↑5%)	7.8(↓18%)	**15.7**(↓32%)	**19.9**(↓14%)	**3.6**(↓34%)	/	/	/	/

samples exhibit distinct external dependencies. This enables effective detection of variations in external function calls, allowing accurate verification of semantic equivalence.

2) *Correctness of Distorted Structure Refactoring:* We investigate the impact of varying numbers of shots as context under the guidance of two LLMs on the correctness of pseudocode refactoring for different types of structure distortions, with the results presented in Table III. Overall, different types of distortions exhibit significant variance in correctness performance. Under the 10-shot setting with DeepSeek-V3, the recall value for D1-type distortions reaches 99.33%, whereas for D5-type distortions, it drops to merely 46.72%. This suggests that the difficulty of refactoring varies substantially across distortion types. D6 represents compiler-injected security checks, which exhibit low coupling with other code elements in the pseudocode functions, enabling our method to effectively eliminate such distortions. Notably, although PseudoFix performs relatively poorly in correctness in the D4 and D5 tasks, this does not prevent it from playing a substantial role in improving readability of pseudocode.

Answer to RQ1: We achieve mean component correctness precisions of 91.45% (Annotation), 83.31% (Identification), and 91.79% (Model Checker), while refactoring correctness recall averaged 77.83% across all types with GPT-4 10-shot, peaking at 99.33% for D1.

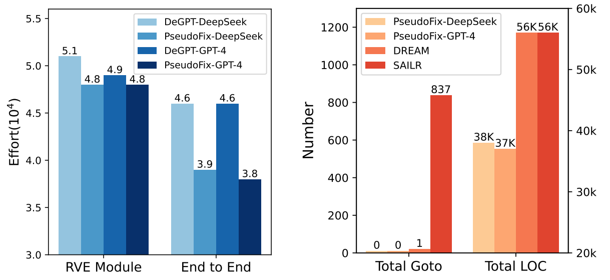
C. RQ2: Readability Evaluation

To evaluate the readability of the refactored code by PseudoFix, we add the malware dataset to simulate real-world malware analysis scenarios. In addition to the complexity metrics outlined in Section III-A3, we employ BLUE-4, METEOR, Rouge-L, and Semantic Similarity as evaluation metrics for the GNU dataset to assess the correlation between the LLM-generated code (i.e. $p_{\text{refactored}}$) and the original source code (i.e. $\tilde{s}_{\text{query}}$). Especially for Semantic Similarity, we use the pre-trained CodeBERT model to get embeddings for the pseudocode and source code, and then compute the cosine similarity between these embeddings to observe their semantic similarity. We investigate four distinct few-shot scenarios, with the experimental results presented in Table IV.

First, we analyze the performance across different code complexity metrics. Overall, the refactored code demonstrates reductions in all metrics except for ND. Nesting Depth serves as a unique metric, as evidenced by the quantitative experiment shown in Table I, in which we observe that distorted pseudocode exhibits a lower value compared to the original source code. Consequently, our refactored pseudocode improves the Nesting Depth, bringing it closer to the level of its original value. In particular, under the 10-shot setting with GPT-4, we reduce Cyclomatic Complexity by 32% and Halstead Complexity Effort by 34% for the malware dataset, significantly easing comprehension challenges in malware analysis. And for GNU dataset, under the same 10-shot GPT-4 setting, the refactored pseudocode exhibited 90.4% similarity to the source code in the LOC metric. Notably, while the average function size in both datasets was under 50 lines, for GNU functions exceeding 100 lines, the Halstead Complexity Effort of refactored pseudocode from Deepseek-V3 and GPT-4 dropped by only 9.3% and 7.7% respectively, demonstrating our method's stability and effectiveness on larger programs.

Next, for the remaining four correlation metrics, we also observe a gradual improvement in the relevance between the refactored pseudocode and the source code. Among them, BLEU-4 is the most stringent metric, with the original pseudocode exhibiting the lowest correlation. However, we achieve a significant relative improvement of 105% after refactoring. For the Semantic Similarity, the scores for all settings surpassed 95.8%, reaching a peak of 99.4% in the GPT-4 10-shot. This highlights our method's effectiveness in not only correcting distortions but also restoring semantic integrity.

Finally, we observe greater optimization potential on the



(a) Compare with DeGPT.

(b) Compare with Decompilers

Fig. 5. A comparison with state-of-the-art approaches, including LLM-driven DeGPT and decompilers enhanced by static analysis for structural reinforcement, DREAM and SAILR.

malware dataset compared to the well-maintained GNU dataset, which we attribute to the typically poorer code structure of malware samples. This finding further demonstrates that PseudoFix can deliver more substantial improvements in real-world scenarios.

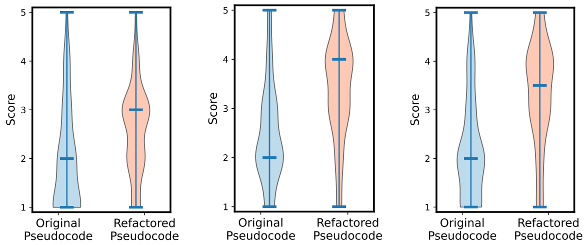
> **Answer to RQ2**: For the GNU and Malware datasets, the refactored pseudocode significantly reduced the Halstead Complexity Effort metric by 22% and 34% under a GPT-4 10-shot setting. Furthermore, PsudoFix also effectively improves structural relevance and semantic integrity.

D. RQ3: Comparative Analysis

Compare with DeGPT [37]. DeGPT primarily leverages LLMs to enhance semantic information in pseudocode, such as variable renaming and comment insertion. Additionally, it optimizes code structure via redundant variable elimination (RVE module), a task analogous to addressing D5-type distortions in our framework. Following the original DeGPT paper, we focus solely on the Effort metric of Halstead Complexity, comparing the RVE module individually against PseudoFix's refactoring of only D5-type distortions. Additionally, we conduct end-to-end comparisons between DeGPT and PseudoFix, with the results presented in Figure 5(a).

In the single RVE module experiment, our method achieves a greater reduction in Effort caused by temporary variables compared to DeGPT, thanks to richer contextual experience. In the end-to-end experiment, while DeGPT's variable renaming and comment insertion show limited effectiveness, our method, based on a scientific taxonomy, precisely identifies and effectively eliminates various structure distortions, leading to a significant reduction in code complexity.

Compare with SAILR [60] and DREAM [61]. Both DREAM and SAILR focus on addressing the structured issue in decompilers (i.e. goto statements in pseudocode), which corresponds to the D1-type structure distortion in our taxonomy. Although DREAM significantly reduces goto statements by duplicating conditional branches, it inevitably increases code complexity and deviates from the structure of the original source code. SAILR strikes a balance by proposing a compiler-aware deoptimization technique that eliminates goto statements while preventing an excessive surge in code complexity.



(a) Readability

(b) Consistency

(c) Reasonability

Fig. 6. Human Evaluation under Three Criteria.

Following their setup, we evaluate the structuring performance (i.e., the total number of goto statements) and code complexity (i.e., total LOC) on their dataset (i.e. O2-optimized coreutils), as shown in Figure 5(b). After excluding functions that SAILR and DREAM failed to decompile, we collect a total of 1,310 functions. Our approach not only eliminates all goto statements but also reduces the length of the pseudocode to just 66.1% of that produced by state-of-the-art structure-enhanced decompilers. And we evaluate the average execution time per function in a single-core, non-concurrent environment: PseudoFix-DeepSeek(19.7s), PseudoFix-GPT4(20.3s), SAILR(12.2s), and DREAM(13.6s). This demonstrates that our use of LLMs to optimize pseudocode, despite an average overhead of 7.1s, strongly achieves a groundbreaking breakthrough, pointing in a new direction.

> **Answer to RQ3**: PseudoFix significantly outperforms DeGPT on the D5-type, DREAM and SAILR on the D1-type, achieving state-of-the-art refactoring performance.

E. RQ4: Human Evaluation

We conduct a human evaluation to assess the feedback of refactored pseudocode by PseudoFix. For participants, we recruit 3 senior reverse engineers with over two years of professional experience, 4 intermediate-level members from Caputer the Flag (CTF) teams specializing in reverse engineering and binary exploitation, and 3 junior CS undergraduates with no background in reverse engineering. For the data, we randomly sample 20 instances from each distortion category (totaling 120 samples). Each participant is asked to evaluate the given code or code pairs from the following three perspectives and rate them: 1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent. Notably, the participants in this experiment did not overlap with those in our preliminary study, to ensure the credibility and validity of our results.

- **Readability**: Is the structure of the test sample clear and easy to understand?
- **Consistency**: Does the test sample exhibit structural similarity to the reference sample?
- **Reasonability**: Is the structure of the test sample logically sound and well-designed?

We ensure that all participants begin their evaluations simultaneously and maintain no communication throughout

TABLE V
COMPARISON OF STRIPPED AND UNSTRIPPED SOURCE CODE IN TERMS OF CODE COMPLEXITY AND CORRELATION

Dataset	**Code Form**		**LOC**	**ND**	**CC**	**Halstead Complexity**			**BLEU-4**	**METEOR**	**Rouge-L**	**Semantic Similarity**
						V($\times 10^2$)	**D**	**E($\times 10^4$)**				
GNU	Source Code		35.8	3.4	5.7	11.1	19.8	2.4	/	/	/	/
	Raw PseudoCode		44.2	2.9	8.2	14.3	22.1	3.4	13.5	35.8	17.9	95.1
	DeepSeek-V3 (Unstripped Source Code)	0-shot	42.8 (↓3%)	3.0 (↑2%)	8.1 (↓2%)	13.8 (↓4%)	21.8 (↓1%)	3.2 (↓5%)	16.2 (↑19%)	38.3 (↑7%)	22.9 (↑28%)	95.8 (↑4%)
		1-shot	42.6 (↓4%)	3.0 (↑4%)	8.0 (↓2%)	13.6 (↓5%)	21.7 (↓2%)	3.2 (↓6%)	17.3 (↑28%)	38.9 (↑9%)	23.0 (↑29%)	95.8 (↑4%)
		5-shot	41.7 (↓6%)	3.1 (↑6%)	7.5 (↓9%)	13.3 (↓7%)	21.3 (↓4%)	3.1 (↓9%)	20.4 (↑50%)	40.5 (↑13%)	27.3 (↑53%)	96.9 (↑5%)
		10-shot	40.9 (↓7%)	3.1 (↑8%)	7.1 (↓14%)	13.2 (↓8%)	21.1 (↓5%)	3.0 (↓11%)	22.6 (↑67%)	45.0 (↑25%)	29.5 (↑65%)	97.2 (↑6%)
	DeepSeek-V3 (Stripped Source Code)	0-shot	42.8 (↓3%)	3.0 (↑2%)	8.1 (↓2%)	13.8 (↓4%)	21.8 (↓1%)	3.2 (↓5%)	16.2 (↑19%)	38.3 (↑7%)	22.9 (↑28%)	95.8 (↑4%)
		1-shot	42.3 (↓5%)	3.1 (↑4%)	8.0 (↓3%)	13.4 (↓6%)	21.7 (↓2%)	3.2 (↓6%)	18.5 (↑36%)	40.4 (↑13%)	23.5 (↑32%)	96.4 (↑5%)
		5-shot	39.1 (↓12%)	3.1 (↑7%)	7.2 (↓13%)	12.7 (↓11%)	21.1 (↓4%)	2.9 (↓13%)	23.0 (↑70%)	45.1 (↑26%)	27.4 (↑54%)	98.6 (↑7%)
		10-shot	**38.4** (↓13%)	**3.2** (↑11%)	**6.6** (↓20%)	**12.3** (↓14%)	**20.8** (↓6%)	**2.7** (↓19%)	**26.8** (↑98%)	**48.0** (↑34%)	**33.8** (↑89%)	**99.2** (↑8%)

the experiment. The results are presented in Figure 6. In terms of readability, both the original pseudocode and the refactored pseudocode exhibit low scores, which is attributed to the inherent limitations of decompiled pseudocode lacking symbolic information. Despite being limited by the upper limit of scores, the refactored pseudocode achieves an average readability score of 2.57, representing a 31.8% improvement over the original pseudocode's score of 1.95. Regarding structural consistency, the optimized pseudocode attains a score of 3.82 in alignment with the source code, marking a 79.3% enhancement compared to the original. This robust improvement conclusively demonstrates PseudoFix's efficacy in reconstructing the overall pseudocode structure. As for reasonability, the optimized pseudocode outperforms the original by an average margin of 1.47 points, a benefit derived from the pre-training of LLMs on extensive high-quality code repositories, enabling it to generate more sound and logically coherent code.

Answer to RQ4: PseudoFix is rated favorably by human experts in Readability, Consistency, and Reasonability, outscoring the original pseudocode by an average of 0.62, 1.69, and 1.47 points, respectively.

VI. DISCUSSION

Scalability of the Taxonomy. In this paper, we perform open-coding to establish a taxonomy for structure distortions. Although our taxonomy is currently built upon C/C++ programs, essentially, both our taxonomy and PseudoFix are data-driven methodologies that can be readily updated with minimal representative samples to accommodate future unknown variations across different platforms, architectures, programming languages, or compiler and decompiler implementations.

Generality of the PseudoFix. Building upon our taxonomy, we introduce PseudoFix, a retrieval-based in-context learning method for structure refactoring. Currently, we only use binary classification for distortion identification, but this can be extended to a more efficient multi-classification task, which is significant for reducing iteration time. While our attempts prove it's not feasible due to limitations in current model context length and hallucination, we believe this change remains promising as future model capabilities improve. Furthermore, in our method, each few-shot example is a $(p, \tilde{s})$ pair, where $\tilde{s}$ represents the source code's structure with its symbols stripped. This stripping is crucial for establishing a clear mapping with p to the correction pattern to guide the model effectively. We validate this approach using the DeepSeek-V3 model on the GNU dataset. As shown in Table V, on average, retaining original identifiers yielded only 76.3% of the performance of the stripped method. We hypothesize that original identifiers interfere with the model's ability to map structure distortions to the intended structure corrections.

Limitations and Future Directions. Although well-defined procedures exist, our taxonomy system is initially implemented manually. Modern LLMs exhibit advanced comprehension of human instructions. To enable fine-grained structure distortion taxonomy, we propose developing an automated agent capable of open coding execution. Furthermore, during our few-shot retrieval process, semantic retrieval is performed solely on the sequential representation of code. However, code inherently possesses graph-structured representations. To provide LLMs with more effective experiences, graph-based retrieval methods are worth exploring in future work.

VII. CONCLUSION

This work presents a comprehensive analysis of structure distortions in decompiled C pseudocode and establishes their systematic taxonomy. Leveraging recent advances in LLMs for decompiler optimization, we introduce PseudoFix, a novel retrieval-based in-context learning framework with LLMs that automatically annotates, identifies, and refactors distorted pseudocode. We also present the Twin-Closure algorithm, which verifies semantic equivalence by analyzing function's external behavior to preserve consistency during refactoring. Our evaluation shows PseudoFix's effectiveness in correctness and readability, showing superiority over state-of-the-art approaches through comparative analysis and human studies.

ACKNOWLEDGMENT

This work was supported in part by the Natural Science Foundation of China (Grant Nos. U20B2047, 62072421, 62002334, 62102386, and 62121002), the Postdoctoral Fellowship Program of CPSF (Grant No. GZC20252180), and the Anhui Provincial Natural Science Foundation (Grant No. 2508085QF213)

References

[1] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, “Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 896–899.

[2] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang *et al.*, “Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 413–430.

[3] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, “Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search.” in *NDSS*, 2023.

[4] M. Cao, S. Badihi, K. Ahmed, P. Xiong, and J. Rubin, “On benign features in malware detection,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1234–1238.

[5] J. Garcia, M. Hammad, and S. Malek, “Lightweight, obfuscation-resilient detection and family identification of android malware,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, pp. 1–29, 2018.

[6] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitraș, “When malware changed its mind: An empirical study of variable program behaviors in the real world,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3487–3504.

[7] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.

[8] J.-H. Ji, G. Woo, and H.-G. Cho, “A plagiarism detection technique for java program using bytecode analysis,” in *2008 third international conference on convergence and hybrid information technology*, vol. 1. IEEE, 2008, pp. 1092–1098.

[9] Y. Hu, Y. Zhang, J. Li, and D. Gu, “Binary code clone detection across architectures and compiling configurations,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 88–98.

[10] Hex-Rays SA, “IDA Pro,” 2025. [Online]. Available: https://hex-rays.com/ida-pro

[11] NationalSecurityAgency, “Ghidra,” 2025. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[12] L. Jiang, X. Jin, and Z. Lin, “Beyond classification: Inferring function names in stripped binaries via domain adapted llms.”

[13] Z. Sha, H. Wang, Z. Gao, H. Shu, B. Zhang, Z. Wang, and C. Zhang, “llasm: Naming functions in binaries by fusing encoder-only and decoder-only llms,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–22, 2025.

[14] H. Gao, S. Cheng, Y. Xue, and W. Zhang, “A lightweight framework for function name reassignment based on large-scale stripped binaries,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.

[15] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, “Unleashing the power of generative model in recovering variable names from stripped binary,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[16] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, “Dire: A neural approach to decompiled identifier naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[17] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. Shoshitaishvili *et al.*, “” len or index or count, anything but v1”: Predicting variable names in decompilation output with transfer learning,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4069–4087.

[18] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, “Augmenting decompiler output with learned variable names and types,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.

[19] G. Li, X. Shang, S. Cheng, J. Zhang, L. Hu, X. Zhu, W. Zhang, and N. Yu, “Beyond the edge of function: Unraveling the patterns of type recovery in binary code,” *arXiv preprint arXiv:2503.07243*, 2025.

[20] Z. Song, Y. Zhou, S. Dong, K. Zhang, and K. Zhang, “Typefsl: Type prediction from binaries via inter-procedural data-flow analysis and few-shot learning,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1269–1281.

[21] Z. Liu and S. Wang, “How far we have come: Testing decompilation correctness of c decompilers,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 475–487.

[22] I. Guilfanov, “Decompilers and beyond,” *Black Hat USA*, vol. 9, p. 46, 2008.

[23] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, “Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring,” in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.

[24] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[25] V. Clarke and V. Braun, “Thematic analysis,” *The journal of positive psychology*, vol. 12, no. 3, pp. 297–298, 2017.

[26] V. 35, “Binary ninja: A reverse engineering platform,” 2025. [Online]. Available: https://binary.ninja/

[27] P. Software, “Jeb pro: Advanced decompiler and debugger,” 2023, commercial tool, Version 5.0. [Online]. Available: https://www.pnfsoftware.com/

[28] Avast, “Retdec: Retargetable decompiler,” 2023, open-source project. [Online]. Available: https://github.com/avast/retdec

[29] Y. Cao, R. Zhang, R. Liang, and K. Chen, “Evaluating the effectiveness of decompilers,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 491–502.

[30] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. Le Goues, “A taxonomy of c decompiler fidelity issues,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 379–396.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[32] J. Li, C. Tao, J. Li, G. Li, Z. Jin, H. Zhang, Z. Fang, and F. Liu, “Large language model-aware in-context learning for code generation,” *ACM Transactions on Software Engineering and Methodology*, 2023.

[33] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, “Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[34] C. S. Xia and L. Zhang, “Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[35] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[36] H. Tan, Q. Luo, J. Li, and Y. Zhang, “Llm4decompile: Decompiling binary code with large language models,” 2024.

[37] P. Hu, R. Liang, and K. Chen, “Degpt: Optimizing decompiler output with llm,” in *Proceedings 2024 Network and Distributed System Security Symposium*, vol. 267622140, 2024.

[38] W. K. Wong, D. Wu, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, “Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1841–1864, 2025.

[39] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, “Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned,” *IEEE Transactions on Software Engineering*, pp. 1–23, 2022.

[40] X. Shang, G. Chen, S. Cheng, S. Guo, Y. Zhang, W. Zhang, and N. Yu, “Foc: Figure out the cryptographic functions in stripped binaries with llms,” *ACM Transactions on Software Engineering and Methodology*, 2024.

[41] X. Shang, Z. Fu, S. Cheng, G. Chen, G. Li, L. Hu, W. Zhang, and N. Yu, “An empirical study on the effectiveness of large language models for binary code understanding,” *arXiv preprint arXiv:2504.21803*, 2025.

[42] X. Jin, J. Larson, W. Yang, and Z. Lin, “Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models,” *arXiv preprint arXiv:2312.09601*, 2023.

[43] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, “Resym: Harnessing llms to recover variable and data structure symbols from stripped

binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[44] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang, and N. Yu, "How far have we gone in binary code understanding using large language models," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 1–12.

[45] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *International conference on machine learning*. PMLR, 2022, pp. 9118–9147.

[46] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[47] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[48] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463, no. 1999.

[49] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[50] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[51] archlinux, "Arch User Repository," 2025. [Online]. Available: https://aur.archlinux.org/

[52] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, "Unleashing the potential of prompt engineering in large language models: a comprehensive review," *arXiv preprint arXiv:2310.14735*, 2023.

[53] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "{AURC}: Detecting errors in program code and documentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1415–1432.

[54] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[55] deepseek, "deepseek-v3," 2025. [Online]. Available: https://api-docs.deepseek.com/news/news250325

[56] openai, "GPT-4," 2025. [Online]. Available: https://openai.com/index/gpt-4/

[57] ——, "ChatGPT," 2025. [Online]. Available: https://openai.com/research/index/release/

[58] microsoft, "CodeBERT," 2025. [Online]. Available: https://huggingface.co/microsoft/codebert-base

[59] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *International conference on machine learning*. PmLR, 2021, pp. 8748–8763.

[60] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy {SAILR}! there is no need to {DREAM} of c: A {Compiler-Aware} structuring algorithm for binary decompilation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 361–378.

[61] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.