FidelityGPT: Correcting Decompilation Distortions with Retrieval Augmented Generation

Zhiping Zhou
Tianjin University
zhou_zhiping@tju.edu.cn

Xiaohong Li
Tianjin University
xiaohongli@tju.edu.cn

Ruitao Feng*
Southern Cross University
ruitao.feng@scu.edu.au

Yao Zhang*
Tianjin University
zzyy@tju.edu.cn

Yuekang Li
University of New South Wales
yuekang.li@unsw.edu.au

Wenbu Feng
Tianjin University
ianx@tju.edu.cn

Yunqian Wang
Tianjin University
wangyq_0617@tju.edu.cn

Yuqing Li
Tianjin University
liyuqing0409@tju.edu.cn

***Abstract*—Decompilation is a crucial technique that converts machine code into a human-readable format, facilitating analysis and debugging in the absence of source code. However, this process is hindered by fidelity issues, which can significantly impair the readability and accuracy of the decompiled output. Existing approaches partially addressed these, such as variable renaming and structural simplification, but typically fail to provide adequate detection and correction, especially in complex but practical closed-source binary scenarios.**

To address this, we introduce *FidelityGPT*, a novel framework to improve the accuracy and readability of decompiled code by systematically detecting and correcting discrepancies between decompiled code and its original source. *FidelityGPT* defines distortion prompt templates tailored to closed-source environments and incorporates Retrieval-Augmented Generation (RAG) with a dynamic semantic intensity algorithm. The algorithm identifies distorted lines based on semantic intensity, retrieving similar code from a database. Additionally, a variable dependency algorithm is designed to overcome the limitations of long-context inputs by analyzing redundant variables through their dependencies and integrating redundant variable names into prompt context. These combined techniques establish *FidelityGPT* as the first framework capable of effectively addressing decompilation distortion issues in LLM-based decompilation optimization. We evaluated *FidelityGPT* on 620 function pairs from a binary similarity benchmark, achieving an average detection accuracy of 89% and a precision of 83%. Compared to the current state-of-the-art model, *DeGPT*, which achieved an average Fix Rate (FR) of 83% and an average Corrected Fix Rate (CFR) of 37%, *FidelityGPT* demonstrated superior performance. With an average FR of 94% and an average CFR of 64%, *FidelityGPT* significantly improves both accuracy and readability, underscoring its effectiveness in enhancing decompilation and its potential to drive advancements in reverse engineering.

* Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230989
www.ndss-symposium.org

I. Introduction

Decompilation is a critical technique that translates machine code (e.g., binary files) into human-readable languages [1], [2]. While decompiled code cannot be recompiled and executed, it significantly aids reverse engineers in comprehending, analyzing, and debugging programs when the source code is inaccessible. This makes decompilation indispensable in both software engineering [3] and cybersecurity [4], [5]. Despite its importance, decompilation faces significant challenges due to factors such as the loss or absence of symbolic information, complex control flows, and other related issues, often leading to discrepancies between the decompiled code and the original source. These discrepancies, known as *fidelity issues* (or *distortion issues*), can severely affect both the readability and the semantic integrity of the decompiled code [6]. As a result, decompiled code may suffer from issues such as meaningless variable names, type errors, redundant variables, incorrect return behavior, and the inclusion of compiler-specific functions. These problems not only hinder research but also pose a risk of misinterpreting the original code [7], [8].

To this end, a few attempts have been made, aiming at improving the accuracy of source code inference from decompiled binaries, with a particular focus on predicting variable names and other code elements. For instance, Banerjee et al. [9] introduced a method that employs masked language models, byte-pair encoding, and neural architectures to infer variable names in decompiled code. Similarly, HexT5 [10] leveraged a unified pre-training model with pseudo-code objectives, including code summarization and variable name recovery, while VARBERT [11] applied Transformer-based architectures to enhance variable name prediction in decompiled outputs. However, these machine learning-based approaches still have lots of limitations, particularly in that they can only target specific datasets and have limited generative capabilities. For example, these models often struggle to generate new code constructs or accurately infer variable names and types when encountering unfamiliar patterns or contexts. This limitation is

particularly pronounced when dealing with code that exhibits significant deviations from the patterns seen during training, reducing their effectiveness in real-world scenarios where code diversity is usually higher than expected.

In recent years, large language models (LLMs) have demonstrated outstanding performance in decompilation optimization. For instance, *DeGPT* [12] was the first to use LLMs for optimizing the output of decompiled code. It introduced a three-role mechanism to enhance decompilation results, achieving significant improvements in restoring variable names and reconstructing code structures, thereby greatly enhancing the readability of decompiled code. However, this approach primarily addresses isolated aspects of decompilation fidelity, such as variable name recovery and code restructuring, leaving substantial gaps in fully detecting and correcting discrepancies between the source code and the decompiled outputs.

Overall, current research has yet to develop extensive and automated methods for detecting and addressing distortion issues. Generally, three primary challenges persist in mitigating this issue:

C1: Impractical Description of Discrepancies. Existing research has primarily focused on identifying discrepancies in specific code elements, such as variables and types. However, these studies address only a small subset of the differences encountered during decompilation. While previous work [6] has provided a comprehensive taxonomy of these discrepancies, some types, such as missing variables and missing code, require reference to the source code for detection. This becomes impractical in closed-source environments where the source code is not directly available. Therefore, further investigation is necessary for closed-source scenarios.

C2: Handling Long Decompiled Code in LLMs. Decompiled functions often span hundreds of lines, posing a challenge for large language models (LLMs) due to token limitations and constrained attention mechanisms. Processing such long sequences in a single pass leads to performance degradation, increased computational costs, and loss of context for long-range variable dependencies. Efficiently managing long code sequences is critical for applying LLMs to large-scale decompilation tasks [13], [14].

C3: Inconsistent and Inaccurate Outputs of LLMs. One of the key challenges in optimizing decompilation using LLMs is ensuring accuracy and precision. LLMs often encounter issues such as semantic drift and hallucination during code interpretation, leading to inconsistent and unreliable results. Retrieval-Augmented Generation (RAG) can mitigate these problems by grounding LLM predictions with relevant external knowledge, but this requires a comprehensive database of distortion patterns and precise querying to ensure contextually accurate results.

In response to the challenges mentioned, we propose *FidelityGPT*, a novel framework designed to detect and correct distortions in decompiled code, especially in closed-source environments. This tool aims at optimizing decompiled output, providing reverse engineers with a more reliable foundation for analysis. Specifically, to address **C1**, we developed an extensive prompt template that systematically categorizes distortion types, aiding in the detection of decompilation issues in closed-source contexts(see III-D). To tackle **C2**, we employ a chunking strategy to split long decompiled code into smaller, manageable blocks, reducing performance degradation in LLMs. To preserve context across chunks, we implement a Variable Dependency Algorithm(see III-B) that extracts relationships between variables, ensuring accurate detection of distortions involving long-range dependencies.To solve **C3**, we leverage Retrieval-Augmented Generation (RAG) by constructing a decompilation distortion database containing annotated distortion patterns. We also apply a dynamic semantic intensity algorithm(see III-C) to identify potentially distorted code lines, enabling precise queries to the database. This enhances *FidelityGPT* 's ability to mitigate semantic drift and hallucination, improving the accuracy of distortion detection.

In this paper, we evaluate the performance of *FidelityGPT* across four key dimensions: distortion detection, distortion correction, algorithmic effectiveness (through ablation studies), and overall efficiency. First, in the distortion detection task, *FidelityGPT* demonstrates remarkable robustness, achieving an accuracy of 89% and a precision of 83%. These results highlight its strong capability in accurately identifying distortions. Second, for distortion correction, we introduce two metrics: Fix Rate and Corrected Fix Rate. The experimental results reveal that *FidelityGPT* achieves a Fix Rate of 94% for detected distortions and a Corrected Fix Rate of 64%, significantly outperforming baseline methods. Third, we assess the effectiveness of the dynamic semantic intensity algorithm, which achieves an optimal balance between token usage and runtime performance. In addition, our variable dependency algorithm substantially reduces false negatives related to redundant code. The ablation study shows that the dynamic semantic intensity retrieval algorithm extracts more meaningful code lines, improving *FidelityGPT*'s distortion detection performance, while the variable dependency algorithm achieves robust performance across different input code lengths. Finally, *FidelityGPT* exhibits exceptional efficiency, striking an excellent balance between token consumption and execution time, thus underscoring its practicality for real-world applications.

Our experiments on 620 function pairs from a binary similarity detection benchmark dataset [15] validate *FidelityGPT* 's efficacy in addressing decompilation fidelity issues. By integrating LangChain for Retrieval-Augmented Generation (RAG) and leveraging a distortion database built from 150 examples in Dramko's taxonomy [6], *FidelityGPT* pioneers the detection and correction of decompilation distortions, advancing reverse engineering in closed-source environments.

- We created the first large-scale decompilation distortion dataset, with 620 function pairs and over 40,000 lines of code, enabling robust training and evaluation for distortion detection in closed-source settings.
- We propose *FidelityGPT*, a novel framework that uses RAG to detect and correct distortions, supported by a Decompilation Distortion Database of annotated distortion

types and a Dynamic Semantic Intensity Algorithm to select semantically significant lines for efficient queries.
- We introduce the Variable Dependency Algorithm to preserve variable relationships across chunked decompiled code, addressing LLM context limitations and enhancing distortion detection accuracy.
- We evaluate *FidelityGPT* using Accuracy and Precision, and propose two new metrics, Fix Rate and Corrected Fix Rate, to measure distortion correction effectiveness, setting new standards for decompilation evaluation.

II. Background & Motivation

Before presenting the technical details of *FidelityGPT*, we first review the foundations of decompilation (II-A), discuss the RAG framework (II-B), and elaborate on our research motivations (II-C).

A. Decompilation

A decompiler, also known as a reverse compiler, is a tool that aims to reverse the compilation process. Given an executable program compiled from a high-level language, the decompiler seeks to generate a high-level language representation that approximates the functionality of the original program [16]. As software development and deployment have proliferated, decompilation has become increasingly critical in areas such as vulnerability discovery [17], [18], [19], [20], malware analysis [21], [22], [23], and the comprehension of closed-source software [24], [25]. By decompiling, analysts can gain insights into the logic and behavior of a program, even in the absence of its original source code, enabling tasks such as debugging, vulnerability patching, and malware analysis. However, the decompilation process faces significant challenges due to the loss of high-level information during compilation, including variable names, comments, and structural elements of the source code. Despite these obstacles, decompilation remains an indispensable tool in security research and program analysis due to its ability to uncover insights from executable programs.

B. Retrieval-Augmented Generation (RAG)

RAG is a framework that enhances generative models by integrating information retrieval. Before generating a response, the RAG model retrieves relevant documents or passages related to the input, which are then provided alongside the input to the generative model [26]. This approach enables the model to leverage external knowledge, improving the quality of the generated output. In contrast, traditional generative models, such as GPT, rely solely on the information encoded in the model's parameters for text generation. This limits the model's ability to generate accurate or up-to-date information, particularly when dealing with domain-specific queries or rapidly evolving topics [27]. RAG addresses this limitation by incorporating a retrieval mechanism that fetches relevant documents or knowledge from external corpora during the generation process. One of the primary advantages of RAG is its ability to dynamically incorporate external knowledge without the need to retrain the underlying model. This is especially beneficial when the model needs to respond to domain-specific queries [28] or handle real-time information [29]. However, RAG also faces challenges, such as effectively selecting the most relevant documents from the corpus and balancing the retrieved information with the model's inherent generative capabilities [30].

C. Motivation

In this section, we detail our research motivation, aiming to tackle the raised challenges.

1) Distortion Issues in a More Realistic Context: Decompilation is a critical tool for reverse engineers analyzing software without access to source code. However, decompiled outputs often diverge significantly from the original source, introducing structural and semantic distortions that hinder analysis. Prior work, such as Dramko et al.'s taxonomy [6], has advanced understanding of fidelity issues but primarily focuses on open-source contexts where source code is available for validation. In closed-source scenarios, challenges like missing variables or entire code segments(see Section A-A1) are often intractable, even with expert manual effort.

Given the inherent complexity of compiler-induced decompilation issues, which are often intractable without access to the original source code, our work shifts focus toward an application-driven perspective on fidelity issues. Rather than attempting to resolve all discrepancies, we prioritize those that are practically addressable in closed-source scenarios, ensuring actionable and relevant contributions to real-world decompilation.

Figure 1 provides a case study to illustrate decompilation fidelity issues. It is important to note that **these discrepancies are strictly based on a comparison between the decompiled code and the corresponding source code**. Panel (a) presents the source code, while panel (b) displays the corresponding decompiled code. The following differences are notable:

1) **Meaningless Parameter and Variable Names:** The source code features descriptive names that facilitate understanding, whereas the decompiled code often uses ambiguous names, complicating comprehension.
2) **Redundant Variables:** The source code uses two variables, `temp` and `i`, whereas the decompiled code includes an additional, redundant variable, `result` without adding any functional significance.
3) **Non-Inertial Dereferencing:** In the source code, array types are accessed directly. However, the decompiled code accesses arrays via pointers, with the representation of array members being obscure and difficult to interpret.
4) **Code Structure Changes:** The source code employs a `while` loop, whereas the decompiled code uses a `for` loop, indicating structural differences.
5) **Confusing Control Flow:** The source code uses a ternary operator for value retrieval, while the decompiled code replaces it with `if` statements, leading to a more complex and less intuitive control flow.

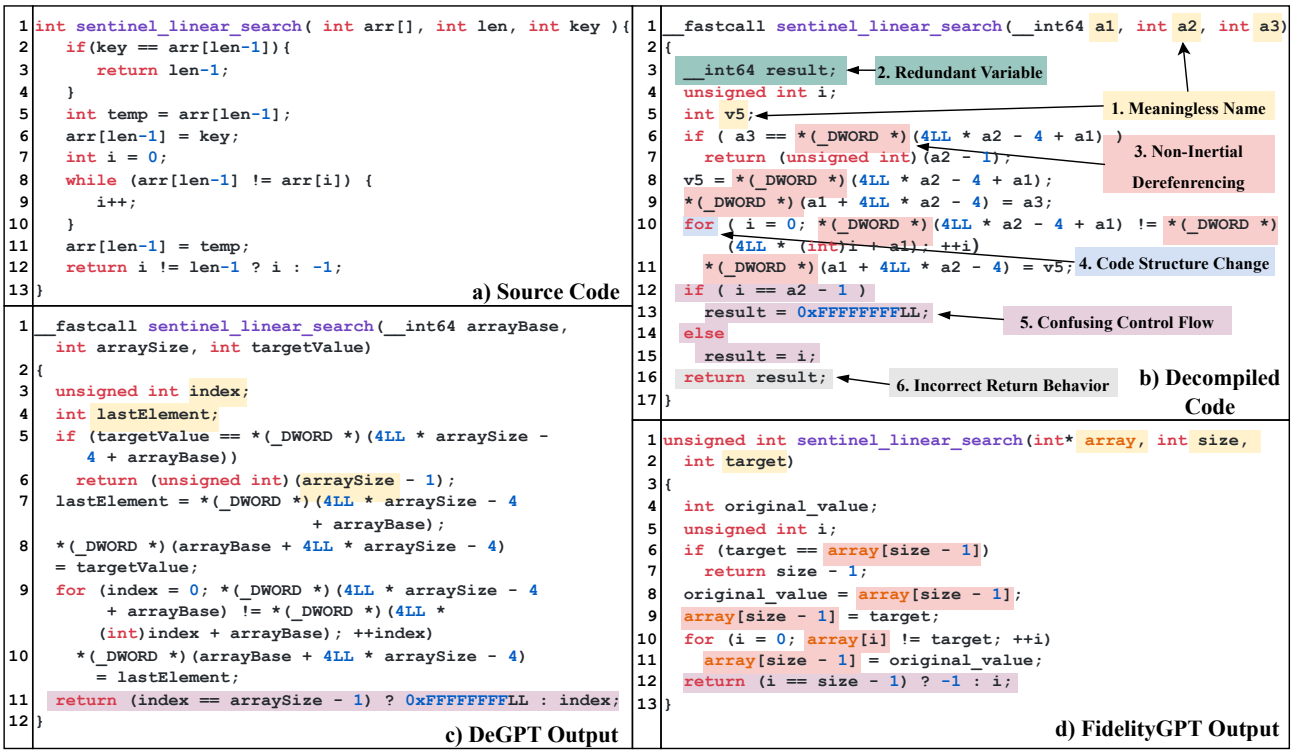

Fig. 1: Case Study of Decompilation Fidelity Issues and Comparison of *FidelityGPT* with DeGPT

6) **Incorrect Return Behavior:** The source code uses a ternary operator to determine return values, while the decompiled code returns values directly, creating inconsistencies.

These distortions impair readability and correctness, hindering security analysts' ability to identify vulnerabilities. For example, ambiguous names reduce comprehension, while pointer-based access may mask errors, risking missed threats. Prior approaches like DeGPT improve variable renaming but fail to address redundant variables and pointer issues, as shown in panel (c) of Figure 1. In contrast, panel (d) shows *FidelityGPT* 's output, leveraging a distortion database, variable dependency analysis, and semantic intensity scoring to enhance clarity and accuracy, significantly improving binary analysis for security professionals.

2) Overcoming Long-Range Variable Dependencies: Due to the inherent limitations of Large Language Models (LLMs), such as restricted input window sizes and constrained attention mechanisms, processing large volumes of decompiled code in a single pass significantly impairs performance in decompilation distortion detection tasks. The limited input window restricts the amount of code that can be analyzed simultaneously, while the attention mechanism struggles to maintain contextual connections across lengthy code sequences, leading to reduced distortion detection accuracy. As shown in Figure 2, the definition of `v23` at line 23 and its usage at line 501, separated by over 400 lines, may exceed the attention window, degrading detection performance. To address these limitations, we propose partitioning the decompiled code into smaller, manageable chunks. This chunking strategy reduces input complexity, allowing the LLM to operate within its input window and alleviating the burden on the attention mechanism, thereby enhancing analytical performance.

However, chunking risks splitting long-range variable dependencies across chunks, which could prevent the LLM from detecting distortions involving these variables. For instance, in Figure 2, the definition of `v23` at line 23 and its usage at line 501 may be split into different chunks, depriving the LLM of the context needed to identify variable redundancy. To overcome this, we propose extracting dependency relationships between variables in the decompiled code. By explicitly capturing these dependencies, we enable systematic analysis of variable redundancy across chunks, ensuring the LLM can accurately assess redundancy. This combined approach of code chunking and variable dependency extraction improves the precision and reliability of decompilation distortion detection, effectively mitigating the constraints of input window size and long-range dependencies.

3) Mitigating Semantic Drift and Hallucination in Distortion Detection via RAG: Large Language Models (LLMs) often produce inaccurate outputs in decompilation tasks due to *semantic drift*, where they misinterpret the context of code structures, and *hallucination*, where they generate erroneous elements. To address this, we employ Retrieval-Augmented

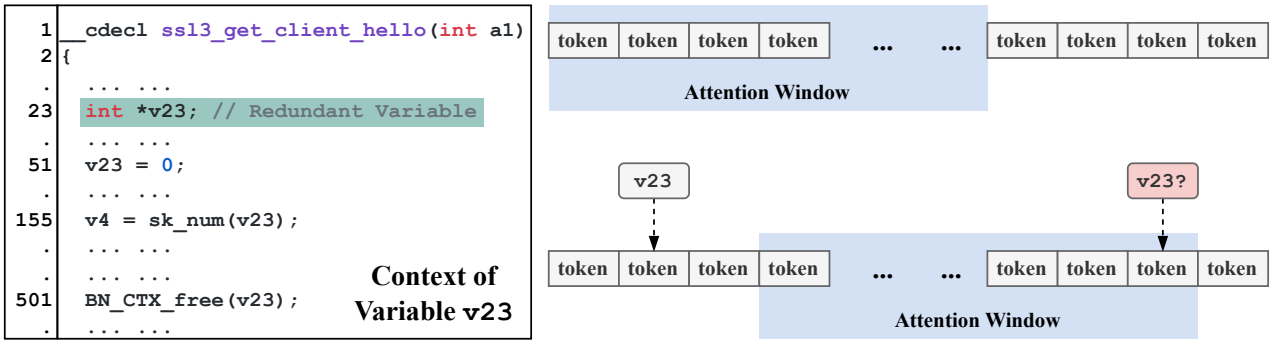

Fig. 2: Long-Range Variable Dependencies in Decompiled Code

Generation (RAG) to ground LLM predictions with contextually relevant knowledge, enhancing the accuracy of distortion detection. However, effective RAG deployment requires addressing two challenges: constructing a diverse distortion database and formulating precise input queries.

First, we curate a decompilation distortion database of distortion instances annotated with specific distortion types (e.g., redundant variables, control flow obfuscation), ensuring coverage of diverse decompilation scenarios. This diversity enables RAG to retrieve semantically aligned examples, improving the model's capacity to discern subtle distortion patterns. Second, formulating precise input queries is essential for targeted retrieval. We feed RAG with carefully selected code lines that exhibit potential distortion patterns. This approach ensures the retrieved knowledge directly informs the LLM's analysis, refining its ability to distinguish genuine distortions from hallucinated artifacts.

By addressing these challenges, *FidelityGPT* utilizes RAG to mitigate LLM hallucinations, ensuring reliable and accurate distortion detection in decompiled code.

III. Methodology

Our primary objective is to extensively detect and address discrepancies between decompiled code and source code within closed-source environments, with a particular emphasis on fidelity issues. We aim to fix decompiled code, making it closer to the original source code format, thereby aiding reverse engineers in analyzing binary files. In this section, we introduce the design of *FidelityGPT*, a framework built upon a large language model that leverages Retrieval-Augmented Generation (RAG) techniques to enhance the interpretability of decompiler outputs for security analysts.

We begin by providing an overview of the workflow, followed by discussing the challenges associated with accurately describing fidelity issues and the technical complexities inherent in utilizing LLMs and RAG. Finally, we detail the design of the prompt templates employed in the tasks of detection and correction.

A. Overview

As illustrated in Fig. 3, *FidelityGPT* operates through three main phases: **Preprocessing**, **Context Generation**, and **Detection and Correction of Distortions**, each designed to address the discrepancies between decompiled and source code.

Phase 1: Preprocessing. In this phase, the binary code is first decompiled into its corresponding functions. If any *decompiled function* exceeds a predefined line threshold (see IV-F2), it is segmented into smaller manageable blocks. A divide-and-conquer approach is applied, dividing the decompiled function into *variable names* and *code blocks*. This segmentation ensures that larger functions can be processed efficiently in later stages.

Phase 2: Context Generation. Two novel methods generate context for distortion detection. First, the **Variable Dependency Algorithm** III-B extracts dependency relationships for variables identified in preprocessing. These are fed into *Prompt Template 1* [31] to flag redundant variables via LLM analysis. Second, the **Dynamic Semantic Intensity Retrieval Algorithm** III-C selects semantically significant lines and queries a Decompilation Distortion Database using Retrieval-Augmented Generation (RAG) to retrieve relevant code. Redundant variables and retrieved code form *Prompt Template 2* for distortion detection in the next phase.

Phase 3: Detection and Correction of Distortions. In this final phase, predefined **distortion types** are combined with the *code blocks* and the *context* generated in the previous phase to form *Prompt Template 2*(see Fig. 4 panel(a)), which is used for distortion detection via the LLM. The output of this step is a decompiled function annotated with specific *distortion type identifiers* (e.g., '// I4'(see III-D)). The identified distortions are then passed into *Prompt Template 3*(see Fig. 4 panel(b)), which generates the corrected code annotated with '*// fixed*' markers, providing clear visibility of the modifications made to resolve the distortions.

B. Variable Dependency Algorithm

The attention mechanism of LLMs struggles to process long decompiled code in a single pass, often missing critical distortions such as redundant variables (see Section IV-F2).

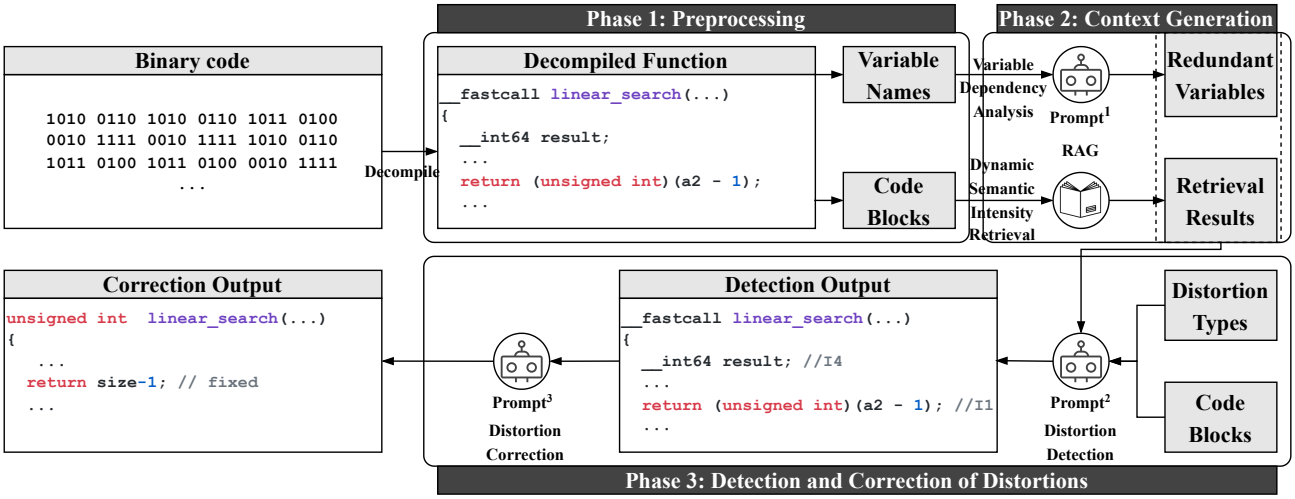

Fig. 3: Workflow of *FidelityGPT*

To address this, we adopt a divide-and-conquer approach, segmenting decompiled code exceeding a predefined threshold into smaller, manageable chunks. This chunking strategy preserves analysis consistency but introduces a challenge: variable definitions and their usages may be split across chunks, leading to false positives in redundant variable detection due to incomplete contextual information.

Our analysis indicates that the decompilation process frequently introduces redundant variables due to register reuse patterns. To systematically detect these variables, *Prompt Template 1* defines the following formal criteria for redundancy identification:

1) **Temporary Variables**: Variables used solely for short-term data storage. These are typically introduced during decompilation to hold transient values, such as register contents, and are not referenced beyond their immediate context.
2) **Intermediate Variables**: Variables employed in intermediate computational steps. These variables store results of temporary calculations, often generated by decompilation processes, and are only relevant within specific operations.
3) **Duplicate Variables**: Variables that replicate data already represented by other variables or constants. Such variables arise from redundant assignments or register reuse, unnecessarily duplicating information.
4) **Low-Usage Variables**: Variables referenced infrequently within the code. These variables, often used once or twice, do not contribute significantly to the program's logic and may indicate redundancy.
5) **Non-Significant Variables**: Variables lacking independent semantic importance. These variables do not convey unique information and are typically artifacts of decompilation without meaningful roles.
6) **Mergeable Variables**: Variables whose data can be logically combined with other statements. These variables store information that can be integrated into existing statements, reducing code complexity.

By integrating the **Variable Dependency Algorithm** with *Prompt Template 1*, we enable the LLM to identify redundant variables based on their dependency relationships and these criteria.

The **Variable Dependency Algorithm** (Algorithm 1) generates a mapping $\mathcal{M}$ of variables to their dependent statements in decompiled code, enabling detection of redundant variables across chunked code. It operates in two steps: constructing a Program Dependence Graph (PDG) and tracing variable dependencies.

First, the `VariableDependencyAnalysis` function builds the PDG by parsing the decompiled code C into variables (`var`) and statements (`lines`). It constructs a Control Flow Graph (CFG), computes control dependencies (CDG) and data dependencies (DDG), and combines them into the PDG to capture variable interactions.

Second, the TRACEVARIABLE procedure traces dependencies for each variable `var` by traversing the PDG. It collects statements from `lines` that depend on `var` into a list $\mathcal{D}$, using a set of visited nodes $\mathcal{V}$ to avoid redundant traversals, and recursively traces related variables. The mappings $\mathcal{M}$ store these dependencies.

These mappings are integrated into *Prompt Template 1* [31], which defines redundancy criteria, and fed to the LLM to detect redundant variables. This approach addresses chunking and LLM attention limitations, improving the accuracy and efficiency of redundant variable detection in decompiled code.

C. Dynamic Semantic Intensity Retrieval Algorithm

Directly feeding entire decompiled functions or code blocks into a Retrieval-Augmented Generation (RAG) system for similarity-based retrieval from the Decompilation Distortion Database is often inefficient. The high similarity among code segments frequently results in redundant retrievals, wasting

Algorithm 1: Variable Dependency Algorithm

Data: Decompiled code C
Result: Variable dependency mappings $\mathcal{M}$: a mapping of variables to their dependent statements

▷ $\mathcal{G}$: Program Dependence Graph (PDG), var: variable name, $lines$: list of code statements, $\mathcal{V}$: set of visited nodes, $\mathcal{D}$: list of dependent statements

$\mathcal{M} \leftarrow$ `VariableDependencyAnalysis(`C`)` **return** $\mathcal{M}$
Function `VariableDependencyAnalysis(`C`)`**:**
  `var` $\leftarrow$ parse C to identify all variables
  `lines` $\leftarrow$ split C into statements
  `cfg` $\leftarrow$ build CFG from statements
  `cdg` $\leftarrow$ compute control dependencies
  `ddg` $\leftarrow$ compute data dependencies
  `pdg` $\leftarrow$ `cdg` $\cup$ `ddg`
  $\mathcal{M} \leftarrow \{\}$ **for** $variable \in$ `var` **do**
    $\mathcal{V} \leftarrow \emptyset$, $\mathcal{D} \leftarrow [\,]$ **Procedure** TRACEVARIABLE(x)**:**
      for $predecessor \in$ `pdg.pred`(x) **do**
        if $predecessor \notin \mathcal{V}$ **then**
          $\mathcal{V} \leftarrow \mathcal{V} \cup \{predecessor\}$ **if** $predecessor \in \mathrm{dom}($`lines`$)$ **then**
            $\mathcal{D} \leftarrow \mathcal{D} \cup \{$`lines`$[predecessor]\}$ **for** $dep\_variable \in$ $var($`lines`$[predecessor])$ **do**
              TRACEVARIABLE($dep\_variable$)
    TRACEVARIABLE(var) $\mathcal{M}[var] \leftarrow \mathcal{D}$
  return $\mathcal{M}$

Algorithm 2: Dynamic Semantic Intensity Retrieval Algorithm

Data: Decompiled code C, Decompilation Distortion Database $\mathcal{D}$
Result: Top-k semantically significant lines

▷ $lines$: List of code lines from C to analyze, min_lines: Minimum number of lines to select, $base\_lines$: Base number of lines to select, $threshold$: Threshold for total lines to adjust selection, $step$: Step size for dynamic line selection, max_lines: Maximum number of lines to select, $\mathcal{D}$: Database with frequency data for syntax constructs.

Function `GenerateSemanticIntensityLines(`$lines$, min_lines, $base\_lines$, $threshold$, $step$, max_lines, $\mathcal{D}$`)`**:**
  $feature\_weights \leftarrow$ weights from frequency analysis of syntax constructs in $\mathcal{D}$; `// Compute weights for syntactic constructs`
  $intensities \leftarrow$ empty list
  for *each line in lines* **do**
    if *line contains constructs* {*assignment, addition, variable definition, return, loop, conditional, function call, type*} **then**
      $intensity \leftarrow$ sum of $feature\_weights$ for detected constructs
      $intensities.append((line, intensity))$
  Sort $intensities$ by intensity in descending order
  if $total\_lines \leq min\_lines$ **then**
    $k \leftarrow total\_lines$
  else
    $k \leftarrow \min(base\_lines + \lfloor \frac{total\_lines - threshold}{step} \rfloor, max\_lines)$
  $selected\_lines \leftarrow$ top k lines, prioritizing diverse construct types ; `// Select top-`k` lines with varied constructs`
  return $selected\_lines$

computational resources and hindering the reasoning capabilities of LLMs.

The **Dynamic Semantic Intensity Retrieval Algorithm** (Algorithm 2) selects the top-k semantically significant lines from decompiled code to enhance Retrieval-Augmented Generation (RAG) for distortion detection. It scores lines based on syntactic constructs and retrieves diverse, high-intensity lines to query the Decompilation Distortion Database $\mathcal{D}$, mitigating LLM semantic drift and hallucination.

The `GenerateSemanticIntensityLines` function assigns intensity scores to each line in $lines$ using $feature\_weights$ derived from frequency analysis of constructs (e.g., assignments, loops, function calls) in $\mathcal{D}$. Scores are stored in $intensities$ as line-intensity pairs and sorted descendingly. The number of selected lines, k, is set to $total\_lines$ if below min_lines, or $\min(base\_lines + \lfloor \frac{total\_lines - threshold}{step} \rfloor, max\_lines)$ otherwise, prioritizing diverse constructs.

These $selected_lines$ query $\mathcal{D}$ to retrieve similar code, guiding the LLM to detect distortions accurately while reducing computational overhead.

D. Distortion Types

DeGPT [12] is currently a leading framework for optimizing decompiler outputs. It excels at addressing variable renaming, simplifying code structures, and generating meaningful comments, significantly enhancing the efficiency of security analysts. However, the differences between source code and decompiled code go beyond variables and code structure. To comprehensively address these discrepancies, we adopt a predefined fidelity taxonomy [6]. As previously mentioned(see II-C), this taxonomy, originally proposed in the context of open-source environments, contrasts with the typical scenario in which reverse engineers perform decompilation without access to the source code. Therefore, we redefine decompilation fidelity issues in closed-source environments. In this context, we carefully selected and integrated specific fidelity types to enable the detection of source code versus decompiled code differences directly through analysis of the decompiled code. We have designed six classifications of fidelity issues, labeled I1 to I6, as detailed below.

- **I1: Non-inertial dereferencing.** Accessing structure members through pointers and arrays, or accessing array members using pointers.
- **I2: Character and string literal representation issues.** String literals being replaced with references or represented as integers.
- **I3: Control flow obfuscation.** Transformations involving while and for loops and destructuring ternary operators.
- **I4: Redundant code.** Includes redundant variables, meaningless parameter assignments, variable assignments within functions without return values, and redundant variables introduced by non-inertial dereferencing.
- **I5: Unexpected returns.** Function structures or return values that deviate from expected outcomes.
- **I6: Use of non-type symbols.** Employing symbols and

macros that do not match types in decompiled code, using symbols and user macros or function calls that do not match types in semantically equivalent decompiled code, and including compiler-specific functions.

In summary, this detailed characterization offers an extensive understanding of decompilation fidelity issues within closed-source environments. It establishes a solid foundation for the subsequent tasks of detection and correction.

E. Prompting for Distortion Detection and Correction

The final stage comprises two key tasks: **distortion detection** and **correction**. The detection task prompt template primarily defines the various distortion types. In contrast, the correction task prompt template outlines methods for correcting these distortions, ultimately leading to an optimized version of the decompiled code.

In constructing the **distortion detection prompt template**, we carefully considered several aspects to ensure it effectively supports distortion detection in decompiled code. The prompt, as shown in panel (a) of Fig. 4, consists of the following key components:

- **Role and Task Definition.** At the beginning of the template, we explicitly define the user as an "experienced reverse engineering expert." This sets the context for the model, preparing it to assume the role of a highly skilled professional.
- **Predefined Distortion Types.** To systematically identify distortions, we introduce six predefined distortion types labeled as **I1** through **I6**. These types serve as a reference framework for consistent classification during analysis, mitigating the risk of subjective bias.
- **Context.** To enhance the model's understanding and analysis of decompiled code, we integrated redundant variable names with the results retrieved from RAG. These results serve as the context for the language model, facilitating more accurate identification of distortions during decompilation.
- **Decompiled Code and Required Output.** The core section of the template involves the decompiled code itself. We specify the format for the model's output, requiring it to annotate the detected distortion types directly within the code, using the labels defined earlier (**I1–I6**). This ensures that the output is structured for ease of evaluation and analysis.

Similarly, as shown in panel (b) of Fig. 4, the design of the **distortion correction prompt template** follows the same principles as the detection template. The major difference is in the output format. While the detection template focuses on identifying and labeling distortion types, the correction template requires the model to fix the distortions and mark the corrected lines with a **"//fixed"** annotation. This approach allows for a direct comparison between the detected distortions and their corresponding repairs, enabling a clear assessment of the effectiveness of the model's correction process.

IV. Evaluation

In this section, we evaluate our approach in order to answer the following research questions.

- **RQ1:** How effective is *FidelityGPT* in distortion detection?
- **RQ2:** How effective is *FidelityGPT* in distortion correction?
- **RQ3:** How does the impact of each component of *FidelityGPT* on its overall effectiveness?
- **RQ4:** How is the efficiency of *FidelityGPT*?
- **RQ5:** How robust and generalizable is *FidelityGPT* across settings?

A. Implementation

FidelityGPT utilizes the *GPT-4o* API with a temperature setting of *0.5*. The RAG component is implemented using *LangChain*, with the embedding model set to *text-embedding-ada-002*. The retrieval method is similarity-based, with $k = 1$.

To ensure effective RAG operation, we identify three necessary design conditions:

(1) Informative query selection. We apply the *Dynamic Semantic Intensity Retrieval Algorithm* to extract the most semantically intense lines from the input function—i.e., those most likely to exhibit distortion. These selected lines are used as targeted queries to the RAG database, improving retrieval relevance and downstream detection performance by reducing noise from less informative lines.

(2) Line-level similarity retrieval. Each selected line is embedded using `text-embedding-ada-002`, and the most semantically similar distorted line is retrieved from the database. This fine-grained, line-level matching ensures structural and contextual alignment with real-world distortion cases. Importantly, the retrieved lines are annotated with distortion types, providing actionable semantic cues for the LLM to infer distortion presence.

(3) Decompiled distortion database. We constructed a domain-specific distortion database by filtering and annotating distorted code lines from the public dataset introduced in [6]. All entries were deduplicated and labeled at the line level with fine-grained distortion types (*I1–I6*). To ensure diversity, the database includes 150 lines derived from `IDA Pro` and 91 additional lines from `Ghidra` decompilation outputs. This manually curated resource provides the retrieval foundation for distortion-aware reasoning and supports cross-tool generalizability.

Together, these components form a targeted and interpretable RAG pipeline that equips the LLM with reliable, context-rich priors grounded in realistic distortion cases.

B. Dataset & Setup

1) Dataset: To evaluate our approach for detecting distortions in decompiled code, we repurposed a subset of functions from established binary similarity detection datasets [15], including *Coreutils-ARM-32*, *Curl-MIPS-32*, *ImageMagick-ARM-32*, *OpenSSL-X86-32*, *Putty-X86-32*, and *SQLite-X86-32*. These benchmarks were selected for their quality, architectural diversity, and the availability of source–binary mappings. To enhance diversity, we also included the *CAlgorithm-X86-64*

```
### Begin of prompt
As an experienced reverse engineering expert, ...
### Define distortion types
I have predefined the following types of distortions...
I1: Non-conventional dereferencing...
... ...
### Context
First, consider the potentially redundant variables: ...
{Variable names}
Then, consider the retrieval results...
{Retrieval result}
### Decompiled code
__fastcall free_memory(_QWORD *a1)
{
  if ( a1 )
  ... ...
}
### Requirement
Output all decompiled code and add " // Distortion Type Number"...
```

a) Distortion Detection Prompt Template

```
### Begin of prompt
As an experienced reverse engineering expert, ...
### Define distortion types
I have predefined the following types of distortions...
I1: Non-conventional dereferencing...
... ...
### The decompiled code with distortion type labels
__fastcall free_memory(_QWORD *a1)
{
  if ( a1 )
  {
    if ( a1[2] ) // I1
    free_memory(a1[2]); // I1
    free(a1);
  ... ...
}
### Requirement
From the perspective of improving code readability and simplifying
the code, review all labeled code, fix the issues, and add
"//fixed" without further explanation.
```

Distortion Type

b) Distortion Correction Prompt Template

Fig. 4: Prompt Templates

repository [32], which contains community-written algorithms with varying coding styles and complexity.

From these sources, we randomly selected 55, 110, 60, 150, 55, 90, and 100 pairs of decompiled functions and their corresponding source code, based on the number of decompiled functions extracted from each project. All functions were decompiled using *IDA Pro 7.5* under the `-O0` optimization level to ensure maximal alignment with the source code. We then manually annotated the distortion types (*I1* to *I6*) for each function pair using a consistent labeling protocol, resulting in over 40,000 lines of annotated code.

To evaluate the generalization of *FidelityGPT* across compilers and optimization levels, we further constructed datasets using `-O1`, `-O2`, and `-O3` binaries (decompiled with *IDA Pro*) and a `-O0` dataset decompiled using *Ghidra*. All were annotated following the same protocol.

Please refer to Appendix A-C for detailed statistics and representativeness analysis of the evaluation subset.

2) Algorithm Configuration: The threshold for partitioning decompiled code blocks is set at *50* lines. The Dynamic Semantic Strength Retrieval Algorithm is designed to output between *5* and *10* lines of code, depending on the complexity and length of the input. The parameters used to control the output are as follows: *min_lines* = *5*, *base_lines* = *5*, *max_lines* = *10*, *threshold* = *5*, and *step* = *9*.

3) Metrics: In the distortion detection task, we primarily use accuracy (Acc) as the evaluation metric. Additionally, since false positives (FP) may impact downstream distortion correction tasks(see A-A3), we also include precision (Pr) as an additional metric. The specific calculation methods are as follows:

$$\text{Acc} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \quad \text{Pr} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

where TP (True Positive) refers to the number of correctly identified distortions. TN (True Negative) refers to the number of correctly identified non-distortions. FP (False Positive) refers to the number of non-distortions incorrectly identified as distortions. FN (False Negative) refers to the number of distortions that were missed or incorrectly identified as non-distortions.

In the distortion correction task, we evaluate *FidelityGPT* from the perspectives of alignment with the fidelity definitions, assessing from the perspectives of correctness and readability. We utilize two manually computed metrics, *Fix Rate (FR)* and *Corrected Fix Rate (CFR)*, similar to the evaluation of *DeGPT* [12].

FixRate (FR) measures the improvement in code semantics restoration and readability. It inspects all lines marked with "*//fixed*" to assess the effectiveness of distortion correction. If the identified distortion is rectified by comparing it with the decompiled code, the code is considered fixed.

$$\begin{aligned} \text{FR} &= \frac{\text{Fixed lines of code}}{\text{all annotations fixed lines of code}}, \\ \text{CFR} &= \frac{\text{Corrected fixed lines of code}}{\text{all annotations fixed lines of code}} \end{aligned} \tag{2}$$

Corrected Fix Rate (CFR) differs from FR in that it focuses solely on correctness issues. Since variable renaming and type changes alone can improve readability without altering code semantics, they are not considered as correctness fixes. Therefore, CFR emphasizes semantic corrections. The specific evaluation criteria are detailed in the tableI.

Assessing the correctness of automatic correction results is challenging. Consequently, we manually compute FR and CFR during the evaluation process to ensure reliability. Higher FR and CFR values indicate better correction performance.

C. Baselines

This section provides a brief explanation of the baseline methods used in our evaluation:

- ***Prompt*$_0$ (zero-shot):** The distortion correction prompt template excludes the definition of distortion types and distortion labels. Since it does not include distortion definitions, this method serves as a comparison for directly using the LLM for distortion correction.
- ***Prompt*$_{Def}$ (with definitions):** The distortion detection prompt template excludes contextual information. However,

TABLE I: Classification of Fixes for Variable and Code Corrections.

Fixed Content	Variable Renaming	Variable Type Modification	Variable Type Restoration	Numerical and Character	Code Structure	Non-Type Symbols
FR	✓	✓	✓	✓	✓	✓
CFR			✓	✓	✓	✓

it includes the definition of distortion types, making it applicable for both distortion detection and correction phases.

- ***Prompt*$_{Eg}$ (with examples):** The distortion detection prompt template excludes contextual information but includes three additional examples. This method also contains distortion type definitions, and uses them for comparison in both distortion detection and correction phases.
- ***DeGPT* [12]:** This method represents the current state-of-the-art (SOTA) in decompilation optimization, leveraging the GPT-4o API for its implementation. It focuses on three key tasks: variable renaming, code structure simplification, and comment generation. As our work does not involve comment generation, our comparison centers on variable renaming and code structure simplification to evaluate the optimization capabilities of DeGPT. Similarly, since it does not include distortion definitions, this method serves as a baseline for distortion correction.
- ***LLM4Decompile*** [33]: This method refines decompiled code from traditional decompilers like Ghidra to generate high-level source code, such as C, using a large language model fine-tuned on DeepSeek-Coder with approximately 40 billion tokens of assembly-to-C code pairs. As our work focuses on optimizing decompiled code output, we compare with LLM4Decompile-Ref.
- ***ReSym*** [34]: This approach combines large language models with program analysis to recover variable names and field accesses from stripped binary files. ReSym fine-tunes two models, VarDecoder and FieldDecoder, for predicting variable names and field accesses, respectively, and uses a Prolog-based algorithm to reduce uncertainty in LLM outputs. While effective in symbolic restoration, ReSym does not focus on optimizing the structural or functional aspects of decompiled code.

D. RQ1: How effective is FidelityGPT *in distortion detection?*

Since *DeGPT* does not include distortion detection functionality, we use *Prompt*$_{Def}$ and *Prompt*$_{Eg}$ as baselines. Table II summarizes the distortion detection results for various approaches, including *Prompt*$_{Def}$, *Prompt*$_{Eg}$, and *FidelityGPT*. The evaluation focuses on two key metrics: Accuracy and Precision. Results are provided for each individual dataset, as well as the overall average across all datasets.

As shown in Table II, in terms of accuracy, *FidelityGPT* achieved the highest average accuracy of 0.89, outperforming both *Prompt*$_{Def}$ and *Prompt*$_{Eg}$ across all datasets. *Prompt*$_{Eg}$ and *Prompt*$_{Def}$ methods performed similarly, with an average accuracy of 0.88. Regarding precision, *FidelityGPT* again demonstrated superior performance with an average precision of 0.83, significantly surpassing the *Prompt*$_{Def}$ (0.73) and the *Prompt*$_{Eg}$ (0.75). This suggests that *FidelityGPT* is more effective at minimizing false positives while maintaining high accuracy in detecting distortions. The *Prompt*$_{Eg}$ method showed higher average precision than *Prompt*$_{Def}$, this indicates that the inclusion of examples helps refine precision.

From the experimental results, it is evident that all methods achieved similar high accuracy due to the use of prompt templates based on systematically defined distortions. However, *FidelityGPT* demonstrated superior precision. The integration of RAG significantly reduced false positives, making *FidelityGPT* more reliable in accurately detecting distortions. The combination of efficient design and the precision of RAG highlights the distinct advantages of our tool.

Answer to RQ1: From the experimental results, it is evident that *FidelityGPT* achieved the highest accuracy and precision across all datasets. This demonstrates that *FidelityGPT* performs effectively in the distortion detection task. Furthermore, the RAG retrieval results as prompts yielded favorable outcomes, further enhancing the performance of *FidelityGPT*.

E. RQ2: How effective is FidelityGPT *in distortion correction?*

The distortion correction task builds upon the results of distortion detection. We evaluate the effectiveness of correction by examining code lines marked with the "*//fixed*" tag, which indicates that these lines have been successfully corrected. Lines classified as type I4 (redundant code) are removed during correction and therefore excluded from the statistics.

To provide a comprehensive evaluation, we compare *FidelityGPT* with two categories of baselines:

- **Commercial off-the-shelf LLMs:** These include *Prompt*$_{Def}$, *Prompt*$_{Eg}$, *Prompt*$_0$, and *DeGPT*. They are general-purpose models typically used in zero-shot or prompt-based settings, without adaptation to the challenges of decompilation or binary analysis.
- **Domain-specific fine-tuned LLMs:** These include *LLM4Decompile* and *ReSym*, which have been specifically fine-tuned or developed to address binary analysis and code recovery, making them more suited for tasks involving the restoration of decompiled code to source code form.

a) Results with commercial off-the-shelf LLMs: As shown in Table III, *FidelityGPT* achieves the highest average fix rate (FR = 0.94) and corrected fix rate (CFR = 0.64) across all datasets. In contrast, *Prompt*$_{Eg}$ and *Prompt*$_{Def}$ show competitive but slightly lower performance, while *DeGPT* and *Prompt*$_0$ exhibit clear limitations when handling more complex distortions. These results highlight the importance of incorporating distortion-type detection to guide LLM-based correction effectively.

TABLE II: Performance Comparison of Different Approaches on Distortion Detection across Various Datasets.

Approach	ImageMagick		curl		putty		CAlgorithm		coreutils		OpenSSL		SQLite		Average	
Metrics	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**
***Prompt*$_{Def}$**	0.89	0.66	0.90	0.85	0.88	0.78	0.88	0.66	0.82	0.67	0.87	0.62	0.88	0.86	0.87	0.73
***Prompt*$_{Eg}$**	0.91	0.74	0.88	0.87	0.86	0.75	0.88	0.69	0.84	0.67	0.88	0.64	0.88	0.87	0.88	0.75
FidelityGPT	0.91	0.83	0.90	0.88	0.89	0.85	0.90	0.71	0.85	0.79	0.89	0.81	0.88	0.92	0.89	0.83

TABLE III: Performance Comparison of Different Approaches on Distortion Correction across Various Datasets with FR and CFR Metrics.

Approach	ImageMagick		curl		putty		CAlgorithm		coreutils		OpenSSL		SQLite		Average	
Metrics	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**
***Prompt*$_0$**	0.8	0.13	0.8	0.17	0.73	0.17	0.8	0.21	0.77	0.18	0.77	0.17	0.6	0.15	0.75	0.17
DeGPT	0.8	0.29	0.85	0.3	0.87	0.4	0.93	0.48	0.85	0.39	0.8	0.38	0.73	0.33	0.83	0.37
***Prompt*$_{Def}$**	0.87	0.62	0.92	0.58	0.94	0.43	0.96	0.71	0.92	0.55	0.9	0.53	0.93	0.6	0.92	0.57
***Prompt*$_{Eg}$**	0.89	0.56	0.91	0.53	0.91	0.44	0.96	0.64	0.91	0.49	0.91	0.54	0.9	0.58	0.91	0.54
FidelityGPT	0.92	0.67	0.96	0.61	0.95	0.61	0.96	0.76	0.92	0.63	0.92	0.61	0.96	0.6	0.94	0.64

b) Results with domain-specific fine-tuned LLMs: Table IV presents the results on Ghidra-decompiled code. While both *LLM4Decompile* and *ReSym* demonstrate notable correction capabilities (with an average CFR of 0.44–0.45), their performance is consistently lower than that of *FidelityGPT*. Specifically, *LLM4Decompile*, despite being fine-tuned, struggles with longer functions due to generation instability, whereas *ReSym*, although effective in variable renaming, suffers from limited semantic coverage since it does not account for control-flow or syntactic distortions.

Answer to RQ2: From the experimental results, it is evident that *FidelityGPT* outperforms both (i) commercial off-the-shelf LLM approaches, which lack the capability to understand compiler-induced distortions, and (ii) domain-specific fine-tuned LLMs, which remain prone to instability or partial coverage. By explicitly modeling distortion types within a unified detection-correction pipeline, *FidelityGPT* strikes a robust balance between generality and practical performance.

F. RQ3: How does the impact of each component of FidelityGPT *on its overall effectiveness?*

In this section, we evaluate the effectiveness of the individual components of *FidelityGPT*, specifically the **Retrieval-Augmented Generation**, the **Variable Dependency Algorithm**, and the **Dynamic Semantic Intensity Retrieval Algorithm**. To achieve this, we conducted ablation studies to assess the performance and efficiency of each component across different levels of function complexity and datasets.

For the RAG component, we demonstrate its impact on the overall effectiveness of *FidelityGPT*. For the Variable Dependency Algorithm, we analyzed decompiled functions by sampling at regular 10-line intervals. This strategy enabled us to systematically examine the algorithm's performance as function complexity increased.

For the Dynamic Semantic Intensity Retrieval Algorithm, we introduced two baseline approaches for comparison: randomly selecting code lines and retrieving all code lines. These comparisons provide insights into how well the proposed algorithm balances efficiency and effectiveness compared to more simplistic selection methods.

1) Retrieval-Augmented Generation: In the distortion detection experiment(see IV-D), we validated the effectiveness of RAG. The experimental results presented with RAG included the overall performance of *FidelityGPT*. Without RAG, the results would be identical to those obtained using *Prompt*$_{Eg}$ and *Prompt*$_{Def}$.

2) Variable Dependency Algorithm: To evaluate the efficiency of the variable dependency algorithm across different ranges of decompiled function lines, we conducted a detailed analysis focusing on functions with at least 30 lines. As shown in Table V, the recall of redundant variables and the average processing time are reported for different line intervals. Notably, for functions exceeding 90 lines, the recall drops to zero, indicating that the algorithm fails to identify redundant variables in highly complex cases.

For functions with more than 30 lines, the recall starts at 0.31 and gradually decreases as the function length increases, eventually reaching 0 for functions with over 80 lines. This indicates a significant drop in performance as function complexity increases. In terms of processing time, functions with over 30 lines take 11.1 seconds on average, and the time slightly decreases to 6 seconds for functions with over 80 lines.

To assess the algorithm's efficiency, we aim for a high recall with lower processing time. We define **recall** as the proportion of correctly identified distorted lines among all true distorted lines in a function. Based on this, we introduce a correlation factor f, defined as $f = \frac{\text{Recall}}{\text{Time}}$. A larger f value signifies better algorithmic performance. As depicted in Fig. 5, the highest f value is achieved at 50 lines, indicating that 50 lines are the optimal balance between efficiency and accuracy when processing decompiled functions. Therefore, we consider functions with more than 50 lines as **long functions**, which require chunk-based processing.

3) Dynamic Semantic Intensity Retrieval Algorithm: To evaluate the efficiency of the Dynamic Semantic Intensity Retrieval Algorithm, we adopted two baseline methods: randomly selecting six code lines and retrieving all code lines. We compared the performance of these three methods in terms of processing time, average number of query tokens, accuracy of distortion detection, and precision.

TABLE IV: Correction Performance Comparison of Domain-specific Fine-tuned LLMs.

Approach	ImageMagick		curl		putty		CAlgorithm		coreutils		OpenSSL		SQLite		Average	
Metrics	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**
LLM4Decompile	0.47	0.27	0.75	0.58	0.77	0.62	0.91	0.56	0.80	0.33	0.90	0.46	0.31	0.27	**0.70**	**0.44**
ReSym	0.81	0.46	0.85	0.43	0.91	0.43	0.89	0.49	0.87	0.51	0.79	0.41	0.74	0.41	**0.84**	**0.45**

TABLE V: Performance of Redundant Variable Recall and Time Across Decompiled Function Lines

Lines	**30**	**40**	**50**	**60**	**70**	**80**
Recall	0.31	0.23	0.24	0.18	0.06	0.00
Time (s)	11.1	9.2	7.4	7.1	6.5	6.0

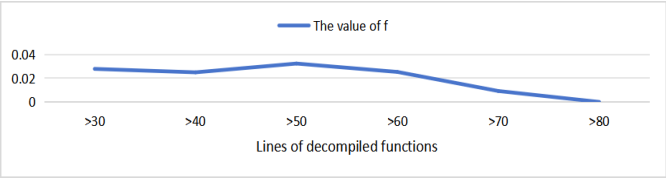

Fig. 5: The value of f across different decompiled function line ranges

Table VI indicates that the Dynamic Semantic Intensity Retrieval Algorithm has a slightly higher processing time and token count than the random selection method but is significantly more efficient than the method that retrieves all code lines. Specifically, *FidelityGPT* achieved a processing time of 4.51 seconds and a token count of 1,502.23, demonstrating its effectiveness in balancing efficiency and performance. In contrast, the random selection method had a slightly shorter processing time of 4.42 seconds and a token count of 1,402.08, but performed worse in terms of accuracy (0.81) and precision (0.84). This is mainly because the random selection of code lines does not guarantee the inclusion of the most representative code segments.

On the other hand, the all code lines retrieval method, while achieving relatively high accuracy (0.86) and precision (0.86), resulted in a significantly longer processing time (7.94 s) and a higher token count (1,641.55). This method's drawback lies in its inclusion of too many irrelevant or distorted code lines, adversely affecting the reasoning capabilities of LLMs and reducing overall efficiency.

The Dynamic Semantic Intensity Retrieval Algorithm achieved the highest accuracy (0.91) and precision (0.88). This suggests that by dynamically selecting code lines based on the semantic intensity and size of the input function, the method effectively filters out irrelevant or detrimental information, enhancing distortion detection performance. This strategy ensures the quality of the model's reasoning while avoiding unnecessary computational overhead, thereby optimizing retrieval efficiency and maintaining high performance.

TABLE VI: Performance of Dynamic Semantic Intensity Retrieval Algorithm

Approach	**Performance**		**Metrics**	
	Time (s)	**Tokens**	**Accuracy**	**Precision**
FidelityGPT	4.51	1,502.23	0.91	0.88
Random code	4.42	1,402.08	0.81	0.84
All code	7.94	1,641.55	0.86	0.86

TABLE VII: Comparison of Average Token Count and Processing Time across Different Approaches for Analyzing Decompiled Functions

Approach	**Tokens**	**Time (s)**
$Prompt_0$	1,424.3	3.2
$Prompt_{Def}$	2,836.4	5.66
$Prompt_{Eg}$	4,154.2	6.08
FidelityGPT	2,982.5	8.99
DeGPT	4,966.8	11.09

Answer to RQ3: The ablation study demonstrates that RAG plays a crucial role within *FidelityGPT*. The Dynamic Semantic Intensity Retrieval Algorithm effectively selects the decompiled code lines with the highest semantic intensity, striking a balance between real-world performance and token usage. The Variable Dependency Algorithm shows significant advantages in handling redundant code, and using 50 lines as the threshold offers a reasonable trade-off between recall and processing time.

G. RQ4: How is the efficiency of FidelityGPT?

Table VII presents the average token count and processing time required to analyze a decompiled function. Firstly, $Prompt_0$ shows the smallest number of tokens and the shortest processing time, as expected. It uses only basic prompts and directly outputs the results. $Prompt_{Def}$ follows it with efficiency slightly reduced due to the inclusion of additional fidelity-related prompts and the dual tasks of distortion detection and correction. Its token count is nearly doubled compared to $Prompt_0$'s. $Prompt_{Eg}$ exhibits a significant increase in both token count and processing time, as it involves three examples. This additional context contributes to a higher token count and longer processing duration. *FidelityGPT* achieves a balanced performance, with a modest increase in token count compared to $Prompt_{Def}$. Since it includes only a small set of similar distortion codes retrieved from the distortion database as context, the token count does not increase excessively compared to $Prompt_{Def}$'s. *DeGPT* has the highest token count and processing time, which aligns with its method. It involves three roles and requires multiple checks to generate the final result, leading to relatively lower efficiency.

Answer to RQ4: The experimental results demonstrate that *FidelityGPT* achieves favorable outcomes in terms of both token usage and processing time. This suggests that *FidelityGPT* strikes an effective balance between efficiency and performance.

H. RQ5: How robust and generalizable is FidelityGPT *across compilers, decompilers, and LLM backends?*

To assess whether *FidelityGPT* maintains stable performance under different compilation, decompilation, and model settings, we further evaluate its robustness and generalizability.

1) Evaluation Across Compiler Optimization Levels: We examine the impact of compiler optimization levels (`-O1`, `-O2`, `-O3`) on distortion detection and correction. As shown in Table IX, *FidelityGPT* achieves consistently high performance across all three settings. Detection accuracy remains stable at around 0.89–0.90, with precision gradually decreasing from 0.78 at `-O1` to 0.70 at `-O3`. For correction, the fix rate (FR) stays above 0.96 across all levels, while the corrected fix rate (CFR) shows a slight decline from 0.60 to 0.59 as optimizations become more aggressive. These results indicate that, although aggressive compiler optimizations introduce additional distortions (particularly reducing precision), *FidelityGPT* maintains strong robustness without significant overall degradation. Note that the performance under `-O0` was reported separately in Table II and Table III, where similarly high accuracy and fix rates were achieved.

This small performance gap can be attributed to two factors. First, our definition of six distortion types and the construction of a comprehensive decompiled distortion database ensure that both unoptimized distortions (`-O0`) and compiler-induced transformations (`-O1`, `-O2`, `-O3`) are well covered. Combined with RAG-based retrieval, *FidelityGPT* consistently supplies the LLM with semantically similar distortion exemplars, making it less sensitive to optimization levels. Second, this finding is consistent with prior observations [35] that LLMs are overly reliant on pattern matching rather than deep logical reasoning for complex code. As long as surface distortion patterns are retrievable from the database, *FidelityGPT* can effectively guide recovery, thus narrowing the performance differences across optimization levels.

2) Generalizability Across Decompilers and Model Backends: To test the portability of our approach across different toolchains, we vary both the decompiler (IDA Pro vs. Ghidra) and the underlying LLM (GPT-4o vs. DeepSeek-chat). The results in Table VIII show that *FidelityGPT* achieves stable performance across all settings. *FidelityGPT* (GPT-4o, Ghidra, T=0) yields only marginally lower accuracy and CFR compared with the IDA-based setup, confirming its **decompiler-agnostic** capability. Meanwhile, *FidelityGPT* (DeepSeek-chat, IDA, T=0) delivers comparable detection quality and even higher CFR (0.72 on average), validating the **model-agnostic** property of our design.

Answer to RQ5: *FidelityGPT* demonstrates strong robustness and generalizability across compiler optimization levels, decompilers, and LLM backends. This robustness is primarily enabled by the distortion database and RAG-based guidance, which provide LLMs with representative distortion exemplars and mitigate their reliance on deeper logical reasoning. As a result, our method can be reliably integrated into diverse reverse engineering pipelines with minimal performance degradation.

V. Discussion

A. Manual Evaluation

Manual evaluation was conducted by seven team members and served as a critical component of our dataset construction and verification pipeline. Specifically, it involved aligning source and decompiled code, identifying and labeling distortions, and validating the corresponding repair outcomes. This annotation process formed the basis of our 620-pair ground truth dataset used for evaluation. The participants included two researchers, two PhD students, and three master's students with relevant expertise in software engineering and binary analysis. They were divided into two groups: one member performed the initial annotation, while the other conducted independent verification. Discrepancies were resolved through discussion or literature reference to ensure consensus and annotation reliability.

B. Threats to Validity

Despite our efforts to ensure robustness, several threats to validity remain.

LLM Randomness. Large language models inherently exhibit non-deterministic behavior: identical inputs can yield different outputs across runs. This variability may affect the reproducibility and stability of our results. To mitigate this, we conducted multiple trials and reported average performance. Moreover, we provide results generated at zero temperature, which significantly reduces randomness and enhances reproducibility. Nonetheless, some degree of stochasticity remains intrinsic to current LLMs and should be considered when interpreting outcomes.

Potential Data Leakage. Another concern is corpus leakage—where pretraining data may overlap with source code used in our experiments. While we observed inconsistent outputs even for repeated inputs, indicating the absence of direct memorization, this cannot fully rule out indirect exposure. We repeated all experiments multiple times to average out such effects, but further analysis is needed to fully assess this risk.

C. User Study

The user study was designed to assess the effectiveness of our repair method in assisting real-world reverse engineering tasks. We recruited 15 participants outside the development team, all with prior experience in binary analysis or software reverse engineering. Each participant was asked to compare raw decompiled code and code repaired by our method, and then evaluate them based on readability, correctness, and ease

TABLE VIII: Detection and correction performance of *FidelityGPT* across different decompiler and model settings.

Setting	Metric	ImageMagick		curl		putty		CAlgorithm		coreutils		OpenSSL		SQLite		Avg
FidelityGPT (GPT-4o, IDA, T=0)	Acc / Pr	0.91	0.78	0.90	0.79	0.88	0.84	0.91	0.76	0.86	0.76	0.89	0.78	0.88	0.89	**0.89 / 0.80**
	FR / CFR	0.97	0.68	0.97	0.72	0.97	0.62	0.98	0.65	0.97	0.65	0.97	0.72	0.96	0.62	**0.97 / 0.67**
FidelityGPT (GPT-4o, Ghidra, T=0)	Acc / Pr	0.89	0.71	0.89	0.75	0.85	0.81	0.90	0.79	0.91	0.71	0.88	0.73	0.87	0.78	**0.88 / 0.75**
	FR / CFR	0.94	0.65	0.92	0.68	0.89	0.63	0.91	0.75	0.96	0.56	0.92	0.57	0.93	0.62	**0.92 / 0.64**
FidelityGPT (DeepSeek-chat, IDA, T=0)	Acc / Pr	0.91	0.70	0.89	0.73	0.91	0.78	0.90	0.74	0.89	0.73	0.91	0.73	0.87	0.76	**0.90 / 0.74**
	FR / CFR	0.96	0.79	0.96	0.77	0.97	0.52	0.95	0.75	0.95	0.71	0.96	0.78	0.99	0.69	**0.96 / 0.72**

TABLE IX: Detection and correction performance of *FidelityGPT* under different compiler optimization levels.

Approach	-O1		-O2		-O3	
Detection Metrics	**Acc**	**Pr**	**Acc**	**Pr**	**Acc**	**Pr**
FidelityGPT (Detection)	0.90	0.78	0.89	0.75	0.89	0.70
Correction Metrics	**FR**	**CFR**	**FR**	**CFR**	**FR**	**CFR**
FidelityGPT (Correction)	0.96	0.60	0.97	0.61	0.96	0.59

of semantic recovery. The study was conducted in a controlled setting with standardized tasks, and detailed instructions were provided to minimize bias. Full procedures and aggregated results are provided in Appendix A-B.

D. Limitations and Future Work

Scalability to Complex Functions. *FidelityGPT* handles long functions differently in the detection and correction phases. For detection, functions exceeding 50 lines are divided into smaller chunks, allowing the model to maintain accuracy on large inputs. In contrast, correction operates on the entire function to preserve variable naming consistency and global semantics—segmenting at this stage would disrupt cross-chunk dependencies and reduce fix quality. As function size increases beyond the LLM's effective reasoning scope, correction performance may degrade due to limited long-range dependency modeling. We note this as a general limitation of current LLM-based repair systems. Future improvements in model architecture or techniques like hierarchical generation may help enhance scalability.

Coverage of Distortion Types. Our distortion database is manually built on top of functions sampled from the `Dramko` dataset. While this offers solid coverage of commonly encountered distortions, it may fall short in representing rarer or more complex semantic anomalies. In such cases, distortions must be manually identified and incorporated by experts. We aim to automate this discovery process in future work by combining program differencing techniques with anomaly detection methods, thereby enabling more complete and scalable construction of distortion benchmarks.

Downstream Tasks. Our current research focuses on automated detection and correction of decompilation distortions. Future studies will investigate the impact of these distortions on downstream tasks, such as binary similarity detection, to enhance the reliability of analyses dependent on decompiled code.

Semantic Consistency Verification. Although *FidelityGPT* corrects decompilation distortions, the repaired code often cannot be recompiled and executed to verify semantic consistency due to missing type information or incomplete constructs. Future work will develop methods to generate recompilable code, such as inferring types and reconstructing function signatures, and validate functionality through execution, improving the reliability of semantic equivalence assessments.

VI. Related Work

A. Fidelity in Decompilation

Decompilation presents key challenges such as variable name recovery, type inference, and code structure reconstruction. While machine learning (ML) and large language models (LLMs) have advanced in these areas, limitations remain. ML techniques have been applied to enhance decompilation outputs, particularly in variable name recovery. The *DIRE* model [36] uses probabilistic methods to leverage both lexical and structural information for this task. However, it struggles with generalization. To address this, *VARBERT* [11], based on BERT, employs transfer learning to improve variable name prediction, setting new benchmarks. LLMs have made significant strides in decompilation. *LLM4Decompile* [33] introduces a range of models trained specifically for decompilation tasks. *ReSym* [34] combines LLMs with program analysis to recover variable names and types, improving overall accuracy.

B. Prompt Engineering

Prompt engineering focuses on designing prompts to enhance LLM performance. Recent work emphasizes prompt design, contextual prompting, and task-specific tuning. Early studies show that prompt phrasing affects LLM output. Webson's [37] explores the effectiveness of prompt design, revealing that models often perform well even with irrelevant prompts, raising questions about prompt understanding. Cao's [38] investigates how prompt templates impact debugging performance in ChatGPT. Contextual prompting incorporates task-specific context. Wei's [39] improves zero-shot learning by finetuning LLMs using instruction templates, leading to better performance on unseen tasks.

C. Retrieval-Augmented Generation (RAG)

RAG enhances LLMs by retrieving external knowledge, which improves consistency and reduces hallucinations in generated content. RAG stabilizes output by incorporating external knowledge, ensuring consistent results across similar inputs [40], [41]. By grounding outputs in verified information, RAG reduces the likelihood of hallucinations, leading to more accurate responses [42], [43].

VII. Conclusion

In this study, we addressed decompilation distortions in the closed-source context using large language models (LLMs) and the Retrieval-Augmented Generation (RAG) technique. Our framework, *FidelityGPT*, effectively detected and corrected six types of distortions, achieving 89% accuracy, 83% precision, a 94% fix rate, and a 64% correction rate. These results significantly enhance decompiled code readability and accuracy, offering a valuable tool for reverse engineering.

VIII. Data Availability

Our source code and dataset are available at [31].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Key Program, Grant No. 62332005).

References

[1] Hex-Rays, "Hex-rays," https://hex-rays.com, 2025, accessed: 2025-08-28.

[2] Ghidra, "Ghidra," https://ghidra-sre.org/, 2025, accessed: 2025-08-28.

[3] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse {Engineers'} processes," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1875–1892.

[4] X. Liu, Y. Wu, Q. Yu, S. Song, Y. Liu, Q. Zhou, and J. Zhuge, "Pg-vulnet: Detect supply chain vulnerabilities in iot devices using pseudo-code and graphs," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 205–215.

[5] T. Ye, L. Wu, T. Ma, X. Zhang, Y. Du, P. Liu, S. Ji, and W. Wang, "Cp-bcs: Binary code summarization guided by control flow graph and pseudo code," *arXiv preprint arXiv:2310.16853*, 2023.

[6] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. Le Goues, "A taxonomy of c decompiler fidelity issues," in *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[7] K. Burk, F. Pagani, C. Kruegel, and G. Vigna, "Decomperson: How humans decompile and what we can learn from it," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2765–2782.

[8] 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.

[9] P. Banerjee, K. K. Pal, F. Wang, and C. Baral, "Variable name recovery in decompiled binary code using constrained masked language modeling," *arXiv preprint arXiv:2103.12801*, 2021.

[10] J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, "Hext5: Unified pre-training for stripped binary code information inference," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 774–786.

[11] 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 Computer Society, 2024, pp. 152–152.

[12] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," in *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, vol. 267622140, 2024.

[13] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," *arXiv preprint arXiv:2408.11324*, 2024.

[14] Y. Lu, X. Zhou, W. He, J. Zhao, T. Ji, T. Gui, Q. Zhang, and X. Huang, "Longheads: Multi-head attention is secretly a long context processor," *arXiv preprint arXiv:2402.10685*, 2024.

[15] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[16] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[17] A. Mantovani, L. Compagna, Y. Shoshitaishvili, and D. Balzarotti, "The convergence of source code and binary vulnerability discovery–a case study," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 602–615.

[18] A. H. A. Chukkol, S. Luo, K. Sharif, Y. Haruna, and M. M. Abdullahi, "Vulcatch: Enhancing binary vulnerability detection through codet5 decompilation and kan advanced feature extraction," *arXiv preprint arXiv:2408.07181*, 2024.

[19] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, "Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation," *arXiv preprint arXiv:2202.12336*, 2022.

[20] Y. Wang, P. Jia, X. Peng, C. Huang, and J. Liu, "Binvuldet: Detecting vulnerability in binary program via decompiled pseudo code and bilstm-attention," *Computers & Security*, vol. 125, p. 103023, 2023.

[21] O. Mirzaei, R. Vasilenko, E. Kirda, L. Lu, and A. Kharraz, "Scrutinizer: Detecting code reuse in malware via decompilation and machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*. Springer, 2021, pp. 130–150.

[22] I. Almomani, M. Ahmed, and W. El-Shafai, "Android malware analysis in a nutshell," *Plos one*, vol. 17, no. 7, p. e0270647, 2022.

[23] N. Mauthe, U. Kargén, and N. Shahmehri, "A large-scale empirical study of android app decompilation," in *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 400–410.

[24] X. Wang, Z. Yuan, Y. Xiao, L. Wang, Y. Yao, H. Chen, and W. Huo, "Decompilation based deep binary-source function matching," in *International Conference on Science of Cyber Security*. Springer, 2023, pp. 244–260.

[25] D. Pizzolotto and K. Inoue, "Bincc: Scalable function similarity detection in multiple cross-architectural binaries," *IEEE Access*, vol. 10, pp. 124 491–124 506, 2022.

[26] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," *arXiv preprint arXiv:2402.19473*, 2024.

[27] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6491–6501.

[28] S. Wang, J. L. S. Song, J. Cheng, Y. Fu, P. Guo, K. Fang, Y. Zhu, and Z. Dou, "Domainrag: A chinese benchmark for evaluating domain-specific retrieval-augmented generation," *arXiv preprint arXiv:2406.05654*, 2024.

[29] W. Su, Y. Tang, Q. Ai, Z. Wu, and Y. Liu, "Dragin: Dynamic retrieval augmented generation based on the real-time information needs of large language models," *arXiv preprint arXiv:2403.10081*, 2024.

[30] S. Zeng, J. Zhang, P. He, Y. Xing, Y. Liu, H. Xu, J. Ren, S. Wang, D. Yin, Y. Chang *et al.*, "The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag)," *arXiv preprint arXiv:2402.16893*, 2024.

[31] FidelityGPT, "Fidelitygpt website," https://github.com/ZhouZhiping045/FidelityGPT, accessed: 2025-08-28.

[32] The Algorithms, "C - all algorithms implemented in c," https://github.com/TheAlgorithms/C, 2025, accessed: 2025-08-28.

[33] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *arXiv preprint arXiv:2403.05286*, 2024.

[34] 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," 2024.

[35] Y. Li, P. Branco, A. M. Hoole, M. Marwah, H. M. Koduvely, G.-V. Jourdan, and S. Jou, "Sv-trusteval-c: Evaluating structure and semantic reasoning in large language models for source code vulnerability analysis," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3014–3032.

[36] L. Dramko, J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, G. Neubig, B. Vasilescu, and C. Le Goues, "Dire and its data: Neural decompiled variable renamings with respect to software class," *ACM Transactions*

on Software Engineering and Methodology, vol. 32, no. 2, pp. 1–34, 2023.
[37] A. Webson and E. Pavlick, "Do prompt-based models really understand the meaning of their prompts?" *arXiv preprint arXiv:2109.01247*, 2021.
[38] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *arXiv preprint arXiv:2304.08191*, 2023.
[39] 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.
[40] F. Cuconasu, G. Trappolini, F. Siciliano, S. Filice, C. Campagnano, Y. Maarek, N. Tonellotto, and F. Silvestri, "The power of noise: Redefining retrieval for rag systems," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2024, pp. 719–729.
[41] Q. Guo, X. Li, X. Xie, S. Liu, Z. Tang, R. Feng, J. Wang, J. Ge, and L. Bu, "Ft2ra: A fine-tuning-inspired approach to retrieval-augmented code completion," *arXiv preprint arXiv:2404.01554*, 2024.
[42] Y. Wu, J. Zhu, S. Xu, K. Shum, C. Niu, R. Zhong, J. Song, and T. Zhang, "Ragtruth: A hallucination corpus for developing trustworthy retrieval-augmented language models," *arXiv preprint arXiv:2401.00396*, 2023.
[43] J. Li, Y. Yuan, and Z. Zhang, "Enhancing llm factual accuracy with rag to counter hallucinations: A case study on domain-specific queries in private knowledge-bases," *arXiv preprint arXiv:2403.10446*, 2024.

APPENDIX A
STUDY AND DATASET OVERVIEW

A. Case Studies

To empirically validate the challenges in decompilation and the limitations of existing approaches, we present three concrete case studies analyzing common distortions in closed-source binary decompilation. These studies demonstrate how decompiler outputs diverge from original source code, the impact of retrieval-augmented generation (RAG) in distortion detection, and the cascading effects of false positives/negatives during correction. Through these analyses, we highlight why current taxonomies and manual inspection methods fail to address practical decompilation scenarios where source references are unavailable.

1) Limitations of Existing Taxonomy in Closed-Source Scenarios: To illustrate these challenges and the limitations of existing taxonomy, we present Figure 6, which showcases specific examples of decompilation distortions prevalent in closed-source scenarios. These examples, rooted in concepts from Dramko et al.'s taxonomy [6], include **unaligned code**, **decomposition of composite variables**, and **expanded symbols**. The figure juxtaposes the **source code** (the original, unavailable in closed-source settings), the **decompiled code** (the decompiler's output), and the **LLM output** (results from attempting correction with a large language model). These examples demonstrate why Dramko et al.'s taxonomy, while insightful, falls short in addressing practical decompilation issues when source code is absent.

1) **Unaligned code** (Panel a): During compilation, unused code segments may be optimized out, leading to misalignment between the decompiled output and the original source. Without access to the source code, reconstructing these missing sections is challenging. As shown in the **LLM output** column, attempts to regenerate the code using an LLM often introduce semantic errors, deviating from the intended functionality.
2) **Decomposition of composite variables** and **expanded symbols** (Panel b): When functions directly utilize data structures, decompilers may split composite variables into individual elements, while user-defined macros are expanded into constants. Without the source code, accurately reconstructing the original variable composition or macro definitions is nearly impossible. The **LLM output** illustrates how such corrections often yield inaccurate or incomplete results, exceeding the capabilities of manual inspection.

2) Impact of RAG Retrieval on Distortion Detection: In this section, we discuss the impact of the RAG retrieval results on *FidelityGPT*. As depicted in Fig. 7, panel (a) shows the input decompiled code, while panel b illustrates the retrieval results from the decompilation distortion database, which are used as contextual prompts for the large language model. From panel (b), we can observe that the first retrieved line corresponds to a redundant variable, which is similar to line three of the decompiled code. However, the variable *i* is not actually redundant, and the output does not result in a false positive. The second retrieved line is similar to the fifth line of the decompiled code and is correctly labeled as *I1* in the output. The third, fourth, and fifth retrieved lines do not have any particularly similar counterparts in the decompiled code, and as input prompts, they do not affect the distortion detection results. Thanks to the well-defined prompt templates and the large language model's understanding of code, the output results, as shown in panel (c), demonstrate that the decompiled code detection is accurate.

3) Effects of Detection Errors on Correction Outcomes: In this section, we conduct a case study to examine the impact of false negatives and false positives during the distortion detection phase on the distortion correction phase, as shown in figure 8. Panel (a) presents the source code, while panels (b) and (d) depict the decompiled code with correct labels and the decompiled code with false negatives and false positives, respectively. Panel (c) shows the output for the decompiled code with correct labels. When compared to the source code, the correction phase effectively addresses the distortion issues, and the result closely resembles the original source code.

In panel (d), the third and twelfth lines are false positives, while the fifth, seventh, and eighth lines are false negatives. The correction results are shown in panel (e). For false negatives, the corrected code retains its original structure. However, for false positives, the code semantics may be altered, and incorrect "fixed" labels may be introduced, as seen in line 13.

It is important to note that in the source code shown in panel (a), the "out" variable on the third line is a static array. Due to compiler optimizations, the specific type could not be recovered during the decompilation process. As mentioned earlier (see II-C1), in the absence of source code references, this issue goes beyond the scope of code optimization. Therefore, this underscores why we place greater emphasis on false positives during the detection phase.

These case studies collectively reveal three critical insights:

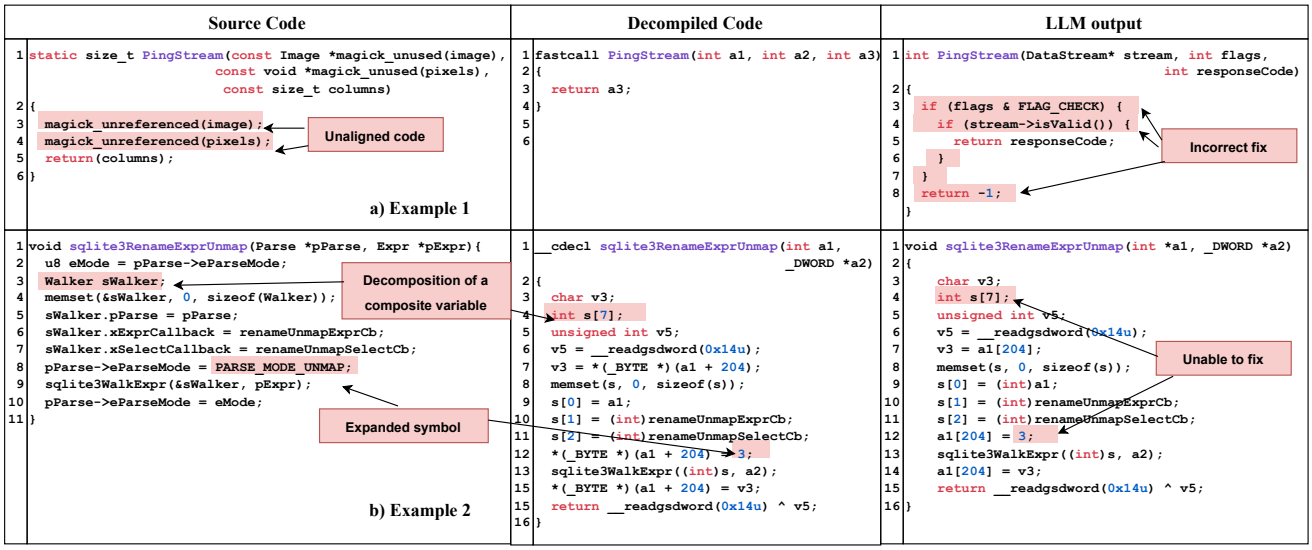

Fig. 6: Examples of Taxonomy-Defined Decompilation Distortions Intractable in Closed-Source Environments

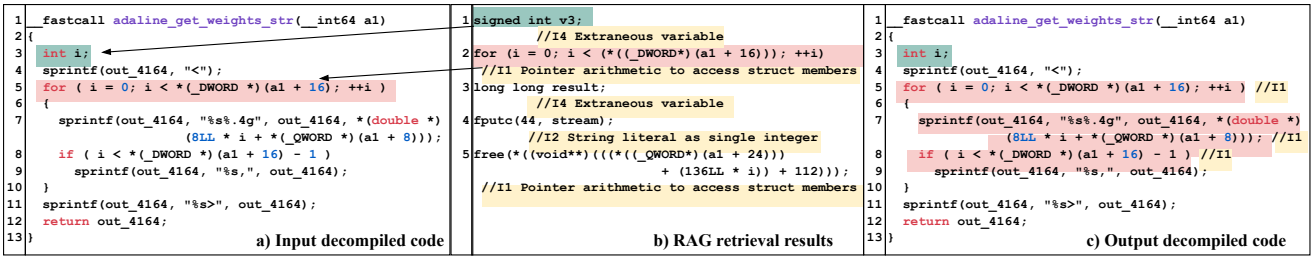

Fig. 7: Impact of RAG Retrieval on Distortion Detection

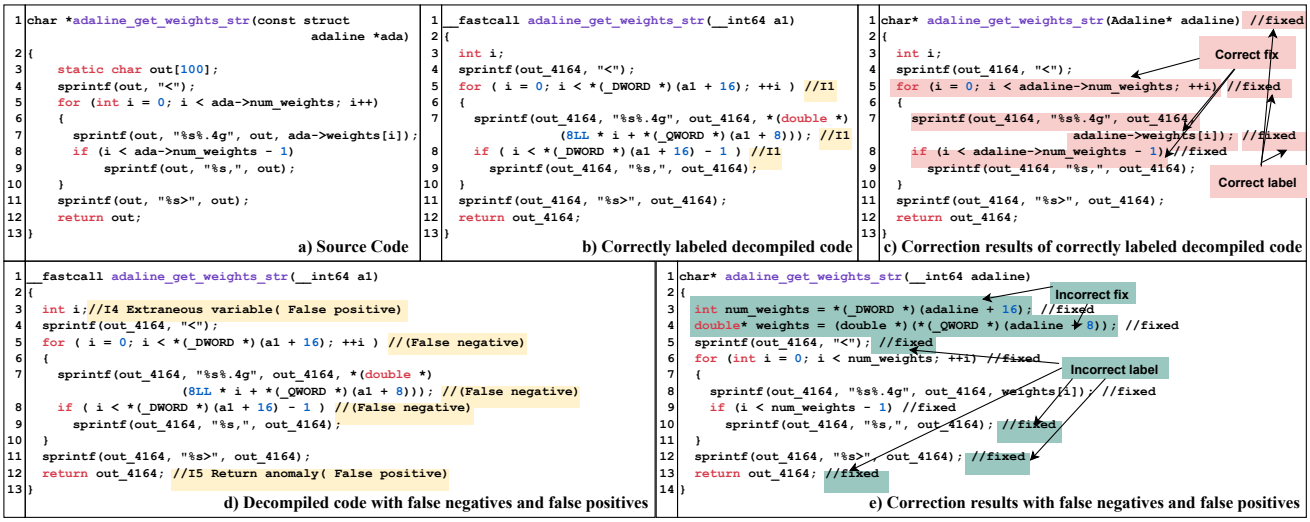

Fig. 8: Effects of Detection Errors on Correction Outcomes

TABLE X: Comparison of statistics between evaluation subset and full dataset

Dataset	Lines per Function				Tokens per Function				Large Functions	
	Max	Min	Median	Mean	Max	Min	Median	Mean	>50 lines	>1024 tokens
Evaluation Subset (620 functions)	**503**	**4**	**20**	**30.1**	**4105**	**10**	**139**	**212.9**	**75 (12.1%)**	**12 (1.9%)**
Full Dataset (46,941 functions)	**3342**	**3**	**10.0**	**29.5**	**12538**	**8**	**33.0**	**110.1**	**5463 (11.6%)**	**675 (1.4%)**

1) Source absence is irreplaceable: Even with LLMs, reconstructing optimized-out code or composite variables remains error-prone without source references.
2) Context matters: Retrieval-augmented detection improves accuracy but depends on the semantic alignment of prompts and decompiled code.
3) False positives are high-risk: False positives during detection propagate irreversible semantic errors in correction, whereas false negatives merely preserve decompiler output.

B. Supplementary User Study Details

This appendix provides additional details for the user study evaluating *FidelityGPT* 's repaired decompiled code outputs, as discussed in Section V-C. The survey was distributed to participants with varying expertise in reverse engineering, who rated the repaired code's readability, conciseness, accuracy, and semantic fidelity on a scale from 1 (Strongly Disagree) to 10 (Strongly Agree). Table XI presents the survey instrument.

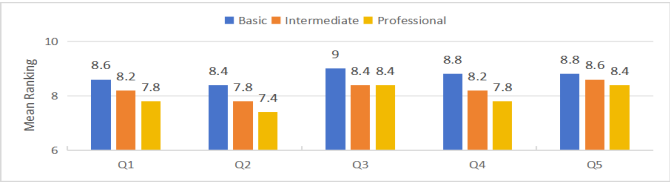

Fig. 9: Mean user survey ratings for *FidelityGPT* 's repaired decompiler output

Figure 9 shows mean survey ratings for *FidelityGPT* 's repaired outputs across 15 participants: 5 Basic (<1 year experience), 5 Intermediate (1–3 years), and 5 Professional (>3 years), with the survey instrument in Table XI.

C. Dataset Statistics and Representativeness

To validate the representativeness of the 620 annotated function pairs, we compared their structural properties with those of the full decompiled function dataset (46,941 functions in total). As shown in Table X, the evaluation subset exhibits slightly larger average function length (30.1 vs. 29.5 lines) and token count (212.9 vs. 110.1), indicating higher complexity.

While the subset's maximum function length (503 lines) and token count (4105 tokens) are lower than the dataset's global extremes (3342 lines, 12,538 tokens), these outliers are extremely rare—only 0.6% and 0.1% of functions exceed these respective thresholds. Therefore, the subset already covers over 99% of practical real-world function cases.

These statistics confirm that the evaluation subset is structurally diverse and statistically representative.

TABLE XI: User Survey Instrument for Evaluating *FidelityGPT* 's Repaired Decompiler Output

Introduction The survey evaluates the effectiveness of *FidelityGPT*, a framework for repairing decompiled code outputs. Participants reviewed example code pairs and rated agreement with statements on a scale: 1 (Strongly Disagree) to 10 (Strongly Agree).
Participant Expertise Participants selected one: - Basic: Limited experience in reading and understanding decompiled code. - Intermediate: Moderate experience in working with decompiled code. - Professional: Extensive experience in reverse engineering and decompiled code analysis.
Questions 1. The repaired decompiler output is easier to read and understand, particularly in terms of variable names, types, and dereferencing (e.g., accessing structure members through pointers or arrays), compared to the original output. Rating Scale: [1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10] 2. The repaired decompiler output reduces redundant code (e.g., unnecessary variables, meaningless assignments) and improves code conciseness and clarity, enhancing overall readability. Rating Scale: [1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10] 3. The repaired decompiler output is more accurate in representing the expected program structure (e.g., handling unexpected returns or type mismatches) compared to the original output. Rating Scale: [1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10] 4. The repaired decompiler output better preserves the intended functionality and semantics of the original source code compared to the original decompiler output. Rating Scale: [1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10] 5. Overall, the repaired decompiler output is more helpful in understanding and potentially debugging the decompiled code compared to the original decompiler output. Rating Scale: [1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10]
Additional Feedback Participants were invited to provide optional comments or suggestions regarding the repaired decompiler output.

APPENDIX B
ARTIFACT APPENDIX

This artifact appendix provides a roadmap for setting up and evaluating the FidelityGPT artifact, as presented in the paper, "FidelityGPT: Correcting Decompilation Distortions with Retrieval Augmented Generation." The artifact supports the detection and correction of distortions in decompiled functions using large language models (GPT-4o). It includes datasets, scripts, and configuration files to reproduce the paper's results.

A. Description & Requirements

This section details the hardware, software, and dataset requirements necessary to recreate the experimental setup for the FidelityGPT artifact.

1) How to access: The artifact is hosted at https://doi.org/10.5281/zenodo.17070171.

2) Hardware dependencies:

- **CPU**: Intel Core i7-12700H or equivalent.
- **RAM**: 32 GB.
- **GPU**: NVIDIA RTX 3070Ti (8 GB VRAM) or equivalent.
- **Storage**: 2 GB free disk space.

3) Software dependencies:

- **Operating System**: Windows 11.
- **Python**: Version 3.9.
- **Dependencies**: Langchain 0.1.2, IDA Pro 7.5, and API libraries (e.g., `openai`) listed in `requirements.txt`.
- **APIs**: GPT-4o (requires API keys).

4) Benchmarks: The artifact includes the following datasets and scripts:

- **Datasets**:
  - **Ground truth functions**: Stored in `Ground truth/*.txt`, with functions separated by `/////`.
  - **Decompiled functions**: Stored in `Dataset/*.txt`, with functions separated by `/////`.
  - **Decompilation distortion database**: `fidelity_new.c` (for IDA Pro) and `fidelity_ghidra.c` (for Ghidra).
- **Scripts**:
  - `FidelityGPT.py`: Detects distortions in decompiled functions.
  - `Correction.py`: Corrects distortions in decompiled functions.
  - `PromptTemplate.py`: Provides all prompt templates, including those for variable dependency analysis, distortion detection, distortion correction, and baseline methods.
  - `pattern_matcher.py`: Implements the Dynamic Semantic Intensity Retrieval Algorithm, generating semantic strength weights from the decompilation distortion database, calculating semantic strength for input code lines, and extracting top-k lines for Retrieval-Augmented Generation (RAG).
  - `variabledependcy.py`: Implements the Variable Dependency Algorithm, generating Program Dependence Graphs (PDGs) and extracting variable dependencies to determine redundancy.
  - `Evaluation/Evaluation.py`: Evaluates detection results.

B. Artifact Installation & Configuration

To prepare the environment for evaluating the artifact:

1) **Set up Python 3.9 virtual environment**:

```
python -m venv venv
venv\Scripts\activate
```

2) **Install dependencies**:

```
pip install -r requirements.txt
```

C. Experiment Workflow

The FidelityGPT artifact supports two experiments to validate the paper's claims:

- **Detection Phase**: Uses `FidelityGPT.py` to detect distortions in decompiled functions, labeling them with distortion types (I1 to I6).
- **Correction Phase**: Uses `Correction.py` to correct distortions, producing functions labeled with `//fix`.
- **Evaluation Phase**: Uses `Evaluation/Evaluation.py` to evaluate detection results against ground truth data.

The workflow involves configuring `config.ini`, running detection and correction scripts, and evaluating results against ground truth data.

D. Major Claims

We have two major claims:

- (C1): FidelityGPT effectively detects distortions in decompiled functions, achieving high accuracy and precision. This is proven by experiment (E1), with results reported in Table II of the paper.
- (C2): FidelityGPT effectively corrects distortions in decompiled functions, as measured by Fix Rate (FR) and Correct Fix Rate (CFR). This is proven by experiment (E2), with results reported in Table III of the paper.

E. Configuration

Before running the system, update the configuration file `config.ini`:

```
[LLM]
model = gpt-4o
temperature = 0
api_key = sk-XXXX
api_base = XXXX

[PATHS]
input_dir = Dataset_4_AE
output_dir = Dataset_4_AE_output
knowledge_base = fidelity_new.c
```

- Input functions: `.txt` files, each with functions separated by `/////`.
- Distortion database: `fidelity_new.c` (IDA Pro) or `fidelity_ghidra.c` (Ghidra).

F. Evaluation

This section provides operational steps to validate the artifact's functionality and reproduce the paper's results. The experiments (E1 and E2) correspond to the major claims (C1 and C2).

1) Experiment (E1): Detection Evaluation: [Preparation]

- Create a dedicated folder for test functions and copy the inputs:

```
mkdir Dataset_4_AE
cp testdata/*.txt Dataset_4_AE/
# alternatively:
cp Dataset/*.txt Dataset_4_AE/
```

[Execution]

- Run the detection script:

```
python FidelityGPT.py
```

- Input: Files from `Dataset_4_AE/`.
- Output: Results saved in `Dataset_4_AE_output/`, where each function is labeled with distortion types (`I1--I6`) and separated by `/////`.
- For functions longer than 50 lines, the system applies chunk-based detection with a 5-line overlap. After detection:
  - Manually merge chunked functions.
  - Remove overlapping duplicate lines.
  - Preserve the `/////` separator between functions.
- Ensure line alignment before evaluation: each line in `model_output.txt` must correspond exactly to the same function segment in `ground_truth.txt`. In practice, we place both files side-by-side (e.g., in Excel) to verify alignment.
- Once aligned, run the evaluation script:

```
python Evaluation/Evaluation.py
```

[Results]

- The evaluation script generates metrics (accuracy, precision, etc.).
- Expected results: Metrics should match those reported in Table II of the paper.

2) Experiment (E2): Correction Evaluation: [Execution]

- Run the correction script:

```
python Correction.py
```

- Input: Aligned functions from the detection phase.
- Output: Corrected functions saved to `Dataset_4_AE_output/`, with `//fix` annotations.

[Results]

- Compare corrected outputs against `Ground truth/`.
- Evaluate using Fix Rate (FR) and Correct Fix Rate (CFR), following the definitions in Table I of the paper.
- Note: Manual assessment is required for correction evaluation, as discussed in Section V of the paper.
- Expected results: FR and CFR values should match those reported in Table III of the paper.