MiSum: Multi-modality Heterogeneous Code Graph Learning for Multi-intent Binary Code Summarization

KANGCHEN ZHU*, College of Computer Science, National University of Defense Technology, China
ZHILIANG TIAN†, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, China
SHANGWEN WANG*†, College of Computer Science, National University of Defense Technology, China
WEIGUO CHEN, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, China
ZIXUAN DONG, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, China
MINGYUE LENG, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, China
XIAOGUANG MAO*, College of Computer Science, National University of Defense Technology, China

The current landscape of binary code summarization predominantly revolves around the generation of a single summarization, limiting the scope of understanding and usability for reverse engineers. The existing approaches often fail to address the multifaceted needs of users, such as detailed insights into usage patterns, implementation nuances, and design rationale, as highlighted in the domain of source code summarization. Consequently, the necessity of multi-intent binary code summarization, an essential way to enhance the efficacy of reverse engineering processes, is underscored. To address this gap, our basic observation is that the two types of information essential for binary code summarization (i.e., the assembly code and pseudo code) can complement each other well. Specifically, the assembly code, characterized by its low-level nature, intricately delineates the execution logic, whereas the pseudo-code, operating at a higher level, retains valuable contextual information. Based on this insight, we propose MiSum, a novel multi-modality heterogeneous code graph alignment and learning method to integrate information from both assembly code and pseudo code. MiSum introduces a unified multi-modality heterogeneous code graph (MM-HCG) that achieves alignment between assembly code graph and pseudo code graph and carries low-level execution details and high-level structural information. To fuse the graph information, we propose multi-modality heterogeneous graph learning with heterogeneous mutual attention and message passing, which caters to important code blocks and discovers

*Kangchen Zhu, Shangwen Wang and Xiaoguang Mao are also with the State Key Laboratory of Complex and Critical Software Environment.

†Zhiliang Tian and Shangwen Wang are the corresponding authors.

Authors' Contact Information: Kangchen Zhu, College of Computer Science, National University of Defense Technology, Changsha, China, zhukangchen18@nudt.edu.cn; Zhiliang Tian, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha, China, tianzhiliang@nudt.edu.cn; Shangwen Wang, College of Computer Science, National University of Defense Technology, Changsha, China, wangshangwen13@nudt.edu.cn; Weiguo Chen, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha, China, chenweiguo@nudt.edu.cn; Zixuan Dong, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha, China, dongzixuan18@nudt.edu.cn; Mingyue Leng, National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha, China, lengmengyue23@nudt.edu.cn; Xiaoguang Mao, College of Computer Science, National University of Defense Technology, Changsha, China, xgmao@nudt.edu.cn.

ACM 2994-970X/2025/7-ARTFSE061
https://doi.org/10.1145/3715780

inter-dependencies between different forms of codes. We also propose an intent-aware summary generator with an intent-aware attention mechanism to produce customized summaries corresponding to multiple intents. Extensive experiments, including evaluations across various architectures and optimization levels, demonstrate that MiSUm outperforms state-of-the-art baselines in BLEU, METEOR, and ROUGE-L metrics. Human evaluations further validate its ability to effectively support reverse engineers in understanding diverse binary code intents, providing a significant advancement in the field of binary code analysis.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Binary code understanding, Multi-intent code summarization, Large language models, Multi-modality fusion, Reverse engineering

ACM Reference Format:
Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Weiguo Chen, Zixuan Dong, Mingyue Leng, and Xiaoguang Mao. 2025. MiSum: Multi-modality Heterogeneous Code Graph Learning for Multi-intent Binary Code Summarization. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE061 (July 2025), 23 pages. https://doi.org/10.1145/3715780

1 Introduction

The code summarization task (a.k.a. code comments generation) involves automatically generating precise, human-readable natural language descriptions of code, which can enhance understanding and facilitate maintenance [Al-Kaswan et al. 2023; Mu et al. 2023; Shang et al. 2024; Steidl et al. 2013; Zhang et al. 2022]. It is considered a critical way to facilitate program comprehension since developers usually forget or have no time to write detailed code summaries, thereby holding the potential to significantly boost software development and maintenance activities [Geng et al. 2022, 2023]. Currently, code summarization tasks can be categorized into two types: source code summarization and binary code summarization. (1) Source code summarization aims to automatically produce natural language descriptions of source code. (2) Binary code summarization generates summaries directly from binary code when source code is not accessible.

Recent studies reveal a growing trend in source code summarization that increasingly focuses on understanding the specific intents of developers [Geng et al. 2024; Mu et al. 2023]. The term "intents" refers to the perspectives from which developers aim to understand code in practical scenarios [Mu et al. 2023]. Specifically, Zhai et al. [2020] identified six common types of intents in real-world projects: *what*, *why*, *how-to-use*, *how-it-is-done*, *property*, and *others*, as shown in Table 1. A statistical analysis by Mu et al. [2023] found that approximately 67% of top-starred Java projects on GitHub contain code comments reflecting multiple intents. These findings suggest a need to move beyond traditional source code summarization (a one-to-one mapping between source code and its summary) and towards generating summaries that are customized according to developers' intents (a one-to-many mapping between source code and multi-intent summaries). This approach is known as multi-intent source code summarization [Geng et al. 2024]. To address this task, Mu et al. [2023] proposed an approach called DOME, which employs an attention mechanism to focus on different parts of the code for different intents. Geng et al. [2024] conducted an empirical study showing that Large Language Models (LLMs) can serve as few-shot summarizers for multi-intent source code using few-shot in-context learning. Similarly, in the field of binary code summarization, reverse engineers who specialize in analyzing binary files also need to understand the code from multiple perspectives. For instance, Ye et al. [2023] conducted a preliminary study involving 15 reverse engineers from both academia and industry, showing that they frequently analyze binary code with different intents, such as identifying the main function or understanding implementation details. In light of these findings, it is necessary to extend existing binary code summarization methods (one-to-one mapping) to consider multiple intents (one-to-many mapping). This task is referred to as **multi-intent binary code summarization** in this paper.

Table 1. The intent taxonomy of code comments [Geng et al. 2024; Zhai et al. 2020].

Category	Definition	Example
What	Describes the functionality of a method	"Checks if the tile units at the given coordinates are displayed on the screen"
Why	Explains the reason why a method is provided or the design rationale of the method	"Prepare to start making calls to the currently registered callbacks"
How-to-use	Describes the usage or the expected set-up of using a method	"Code executed before the intercepted method"
How-it-is-done	Describes the implementation details of a method	"Ends the current table, discards it and pops the top of the stack to be the new current table"
Property	Asserts properties of a method including pre-conditions or post-conditions of a method	"Returns true if the value is a string that matches a regex"
Others	Unspecified or ambiguous comments	"I am done with the model, free the resources"

To accomplish the task of multi-intent binary code summarization, we can adapt existing mainstream code summarization methods to this task. A common approach treats code summarization as a translation task, employing Neural Machine Translation (NMT) models [Stahlberg 2020] to convert code into natural language based on a specific intent. For binary code summarization, given the inherent difficulty in understanding binary code in its native form (as machine code or executable files), researchers typically use reverse engineering techniques to translate binary code into either assembly code or pseudo code, which is then converted into natural language. However, both assembly code and pseudo code have inherent limitations. Assembly code, being a low-level language, directly describes the underlying execution logic but lacks high-level structural and contextual information. Conversely, pseudo-code is closer to high-level languages and retains more context and logical structure, but it is often imprecise and lacks critical low-level details, such as specific memory operations and register manipulations. To address these limitations, we propose leveraging the complementary strengths of both assembly and pseudo code, enabling a more comprehensive understanding of binary code.

Based on these insights, we need to integrate both assembly code and pseudo code for multi-intent binary code summarization. Traditional fusion methods [Li et al. 2021; Ma et al. 2022; Tao et al. 2023] first use encoders to separately encode assembly code and pseudo code tokens and then fuse their feature representations. These methods require the models to accurately understand both types of code. Undergoing comprehensive training on extensive source code datasets, Large Language Models (LLMs) exhibit a robust proficiency in understanding code [Chen et al. 2021], prompting researchers to employ LLMs with prompt engineering [White et al. 2023] and in-context learning (ICL) [Min et al. 2022] for multi-intent source code summarization tasks [Geng et al. 2024]. However, applying prompt-based or ICL to fuse assembly code and pseudo code for multi-intent binary code summarization presents unique challenges. This approach requires LLMs to be pre-trained to understand both assembly and pseudo code. However, current LLMs are primarily trained on source code and have a limited understanding of assembly code, making it hard to effectively integrate assembly code and pseudo code to fully comprehend binary code. Therefore, developing a multi-modal fusion approach for multi-intent binary code summarization remains highly challenging.

In this paper, to better deal with the multi-intent binary code summarization task, we propose a multi-modality heterogeneous code graph alignment and learning method to leverage both assembly and pseudo code. Our proposed method, MiSUm, introduces a Multi-Modality Heterogeneous Code Graph (MM-HCG), which integrates assembly and pseudo code into a unified graph to capture both low-level execution details and high-level structural information. To better facilitate the integration of the two levels of graphs, we propose a heterogeneous graph alignment algorithm by a cross-form statement mapping, ensuring consistent and comprehensive representation across modalities. To fuse the information over the heterogeneous graph, we propose MM-HCG heterogeneous

graph learning with heterogeneous mutual attention and message passing, which attends more to important code blocks and helps to discover inter-dependencies and semantic relationships between different modalities (forms of code). Finally, we also construct an intent-aware summary generator with intent-aware attention to focus on the most relevant information for each intent and generate the corresponding summary.

We conducted extensive experiments to evaluate the effectiveness of MISUM. Since training and evaluating MISUM require a large volume of labeled summary-intent data, we constructed a large-scale binary code summary dataset labeled with corresponding intents across six categories. Specifically, we classified intents based on the approach by Mu et al. [2023] on the dataset released by Ye et al. [2023], resulting in an intent-aware binary code summary dataset. We performed experiments across three different architectures (X86, X64, ARM) and three optimization levels (O1, O2, O3) to ensure a fair comparison with baselines. The results from both automatic metrics and human evaluations demonstrate the superiority of MISUM. In particular, human evaluations indicate that MISUM significantly meets the varying needs of reverse engineers in understanding binary functions. Our contributions are as follows:

- We introduce the novel task of multi-intent binary code summarization, offering a new perspective and research direction in the field.
- We create a comprehensive dataset specifically designed for multi-intent binary code summarization. This dataset includes a diverse range of binary code samples and their corresponding multi-intent summaries, facilitating robust evaluation and development in this area.
- We propose an innovative Multi-Modal Heterogeneous Code Graph (MM-HCG) alignment method, which effectively integrates information from both assembly code and pseudo code modalities. This method leverages the complementary strengths of each modality and improves the representation and understanding of binary code.
- We achieve state-of-the-art performance in multi-intent binary code summarization. Our extensive experiments demonstrate that our approach outperforms several strong baselines, including ChatGPT and GPT-4, on both automatic metrics and human evaluations.

2 Related work

2.1 Source Code Summarization

Code summarization involves automatically generating natural language descriptions (or comments) for code snippets [Shi et al. 2024; Steidl et al. 2013; Su and McMillan 2024; Zhang et al. 2022; Zhou and Liu 2024]. Research in this area dates back to 2010 when Haiduc et al. [2010] first applied automated text summarization techniques to source code. Until 2017, most methods focused on information retrieval (IR) [Haiduc et al. 2010; Moreno et al. 2013; WANG et al. 2015] and template-based approaches [McBurney and McMillan 2014; Sridhara et al. 2010, 2011]. With the advent of neural machine translation (NMT) [Stahlberg 2020] in natural language processing (NLP), many researchers adapted its encoder-decoder architecture for code summarization tasks [Gao et al. 2023; Hu et al. 2018; Zheng et al. 2019]. Recently, the field has seen a surge in research on large language models (LLMs) for code summarization [Shi et al. 2024; Su and McMillan 2024; Zhou and Liu 2024]. Fried et al. [2022] introduced an LLM called InCoder and conducted zero-shot training on the CodeXGLUE Python dataset [Lu et al. 2021], achieving notable results. However, fine-tuned smaller pre-trained language models (PLMs) like CodeT5 [Wang et al. 2021] still outperform this zero-shot setting. Ahmed and Devanbu [2022] explored few-shot prompting for adapting LLMs to code summarization, finding that it enables Codex [Chen et al. 2021] to significantly surpass fine-tuned smaller PLMs.

2.2 Binary Code Summarization

Binary code summarization aims to extract and generate concise summaries from binary code to aid in program understanding [Al-Kaswan et al. 2023; Jin et al. 2023; Shang et al. 2024; Taviss et al. 2024; Xiong et al. 2023; Ye et al. 2023]. BinT5 [Al-Kaswan et al. 2023], the pioneering model in this domain, extends the scope of source code pre-trained language models to binary code. It treats decompiled code as a specialized programming language, using fine-tuned CodeT5 [Wang et al. 2021] to capture its semantics and generate summaries. This innovation has paved the way for further research in binary code summarization. HexT5 [Xiong et al. 2023], an advanced pre-training model based on CodeT5, supports multi-task learning, including function name recovery and binary code summarization, demonstrating promising performance. CP-BCS [Ye et al. 2023] employs a bidirectional instruction-level control flow graph and pseudo code enriched with expert knowledge to learn comprehensive binary function execution behavior and logic. Jin et al. [2023] introduced the BinSum benchmark, featuring over 557,000 binary functions, and proposed a prompt-based method using ChatGPT/GPT-4 [Achiam et al. 2023]. Additionally, Shang et al. [2024] assessed the effectiveness of popular LLMs in binary code summarization, highlighting their potential to advance binary code understanding.

2.3 Code Summarization with Multiple Intents

Code summarization with multiple intents seeks to generate tailored code summaries based on specific user intents [Geng et al. 2024; Mu et al. 2023]. Mu et al. [2023] proposed a developer-intent-driven code comment generation approach called DOME, which generates comments aligned with specified intents using an attention mechanism focused on relevant code information. Geng et al. [2024] evaluated LLMs using the in-context learning [Dong et al. 2022] paradigm, showing that LLMs significantly outperform state-of-the-art supervised learning methods in generating multi-intent comments for source code. However, research on multi-intent summarization for binary code is sparse, representing a critical gap in the field. Understanding binary code in various contexts is as vital as it is for source code.

3 Problem Definition

The goal of the multi-intent binary code summarization task is to generate a summary that aligns with specific intents given a binary code snippet. Formally, given a binary code snippet x and a set of intent categories $E = \{e_1, e_2, \ldots, e_m\}$ where each e_i represents a particular intent category, the objective is to generate a textual summary $y = \{y_1, y_2, \ldots, y_n\}$ with a sequence of tokens y_i, which satisfies the intents specified by E. The summary should capture the key aspects of the code snippet x as per each intent in E.

4 Motivation Example

In existing literature, there are two main approaches for applying NMT models to translate binary code into natural language descriptions in the field of program repair. The first approach involves using reverse engineering to transform binary code into assembly code, which is then translated into natural language. The second method uses plugins to convert assembly code into pseudo code, which is subsequently translated. However, both assembly and pseudo code have inherent limitations: assembly code, being a low-level language, captures the execution logic but lacks high-level structural and contextual information. In contrast, pseudo-code is closer to high-level languages and retains more context and logical structure, but it is often imprecise and omits crucial low-level details, such as specific memory operations and register manipulations.

```
// Assembly Code:
   loop_start:                  next_node:
        LDRB R1, [R0, #15]          LDR R0, [R0]
        CMP R1, R2                  CMP R0, #0
        BNE next_node               BNE loop_start

        LDRB R3, [R0, #12]      MOV R0, #0
        CMP R3, R3              BX LR
        BGT next_node

        LDR R4, [R0, #8]
        CMP R4, R5
        BLT next_node

        MOV R0, R0
        BX LR
-------------------------------------
/** Summary:
* This code snippet performs a loop that checks
* conditions on memory values pointed by R0,
* using registers R1–R5 for comparisons, and
* branches to 'next_node' if not meet conditions,
* continuing until R0 points to a null address. */
```

Fig. 1. An example showing binary code summarization based on assembly code.

```
// Pseudo Code:
int **__fastcall find_matching_node(int ***a1,
        unsigned int a2, unsigned int a3, unsigned
        int a4) {
  int **v1;
  unsigned int v2, v3;

  for (v1 = *a1; v1; v1 = (int **)*v1) {
    v2 = *((unsigned __int8 *)v1 + 15);
    if (v2 == a2) {
      v3 = *((unsigned __int8 *)v1 + 12);
      if (v3 <= a3 && v1[2][10] >= a4)
        return v1;
    }
  }
  return 0;
}
-------------------------------------
/** Summary:
* The `find_matching_node` function searches
* through a linked list and returns the first node
* that meets specified criteria.
* If no such node is found, it returns `0`. */
```

Fig. 2. An example showing binary code summarization based on pseudo-code.

To motivate the need for a multi-intent binary code summarization approach, we observe that different types of information are useful for generating different types of summaries. Low-level details found in assembly code, such as memory operations, register manipulations, and instruction sequences, are particularly suitable for generating summaries that focus on "*how-to-use*" or "*how-it-is-done*" intents, which describe the precise operations and execution steps of the binary code. On the other hand, high-level information found in pseudo-code, such as the algorithmic flow and purpose of the code, is more suitable for generating summaries that align with "*what*," "*why*," or "*property*" intents, which provide an overview of the code's functionality, purpose, or characteristics. To illustrate the limitations of existing summarization techniques based on single-modality approaches, we present several examples that demonstrate how these methods fail to provide comprehensive binary code summaries. The summaries in Figures 1 and 2 are generated using GPT-4 following Shang et al. [2024], with prompts reflecting different intents: "what" for assembly code and "how-it-is-done" for pseudo code. The prompt templates are as follows: "*Imagine you are an experienced binary reverse engineer. Your task is to analyze the provided code and generate a brief comment that describes the function's overall functionality for assembly code (or implementation details for pseudo code). Here are the code snippets:* `Assembly Code` (`Pseudo Code`)."

Figure 1 illustrates an example of binary code summarization using assembly code. Assembly language is a low-level programming language where instructions consist of operands and opcodes that directly reflect the underlying machine instructions and execution logic. Therefore, summaries generated from assembly code are more likely to describe low-level operational details of the code. The assembly code snippet in Figure 1 performs a search operation to find a matching node in a linked list. The summary generated from this code specifically details the operations such as "*checks conditions on memory values pointed by R0*", "*using registers R1-R5 for comparisons*", and "*continuing until R0 points to a null address*". These elements of the summary focus on the specific interactions with registers and memory addresses, reflecting the fine-grained execution

logic inherent to assembly code. However, assembly code often struggles to provide a high-level, conceptual understanding of the function's overall purpose, making it less effective in satisfying the "what" intent. The focus on fine-grained details can obscure the broader functional goal of the code, leading to summaries that may lack clarity in conveying the code's overall behavior or intent.

Figure 2 illustrates an example of binary code summarization using pseudo code. Pseudo code is a high-level representation, offering a more abstract view compared to assembly language. It is designed to be easy to read and understand, reflecting the logic and flow of the code in a more human-readable form. The pseudo-code snippet in Figure 2 describes the function *find_matching_node*, which iterates through a linked list to find and return the first node that meets specific criteria, or 0 if no matching node is found. The summary provided for this pseudo code captures the high-level functionality of the algorithm, stating that it "*searches through a linked list*" and "*returns the first node that meets specified criteria*". It is well-aligned with the "what" intent, providing a clear and concise explanation of the function's purpose. However, pseudo code typically lacks the detailed operational insights found in assembly code, such as specific register usage or low-level memory operations, making it less effective at fulfilling the "how-it-is-done" intent. As a result, while pseudo code excels at summarizing a function's overall goal, it often misses critical implementation details that are necessary for a deeper understanding of the underlying execution logic.

Thus, each modality—assembly code and pseudo code—has inherent strengths and limitations when used independently for binary code summarization. Assembly code excels in providing detailed operational descriptions but falls short in summarizing the overall function ("what"), while pseudo code offers a clear high-level overview but lacks the depth to describe how the function is implemented ("how-it-is-done"). Combining both modalities can better address the diverse intents in multi-intent binary code summarization.

5 Method

5.1 Overview

Figure 3 illustrates the overview of MiSum, which comprises four core components: (1) **Multi-Modality Code Graph (MM-HCG) Constructor**, integrating pseudo code and assembly code into a unified heterogeneous graph structure and enabling the effective merging of information for improved semantic understanding; (2) **Multi-Modality Heterogeneous Code Graph Alignment and Integration**, aligning and integrating the heterogeneous graphs derived from different code modalities to ensure consistent and comprehensive representation; (3) **MM-HCG Heterogeneous Graph Learning**, learning to represent the aligned MM-HCG and fusion the semantic features of the code via the aligned graphs; (4) **Intent-Tailored Summary Generator**, utilizing an intent-aware attention mechanism to selectively summarize the most pertinent information, and generate summaries closely aligned with the user's specific intent.

5.2 Multi-Modality Code Graph Constructor

To better understand the information from the target binary code, we transfer the binary code into pseudo code and assembly code and then construct graphs to carry the information on pseudo code and assembly code, where models may discover the complex structure and unreachable codes, and execution behavior from the graphs.

5.2.1 Graph with Pseudo Codes. We construct an Augmented Abstract Syntax Tree (AAST) of pseudo code to obtain the structural and semantic information; and then transform the tree to a graph with pseudo codes. Specifically, in program analysis, several representations such as the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) are commonly used to capture the syntactic and semantic relationships within source code. To complement these

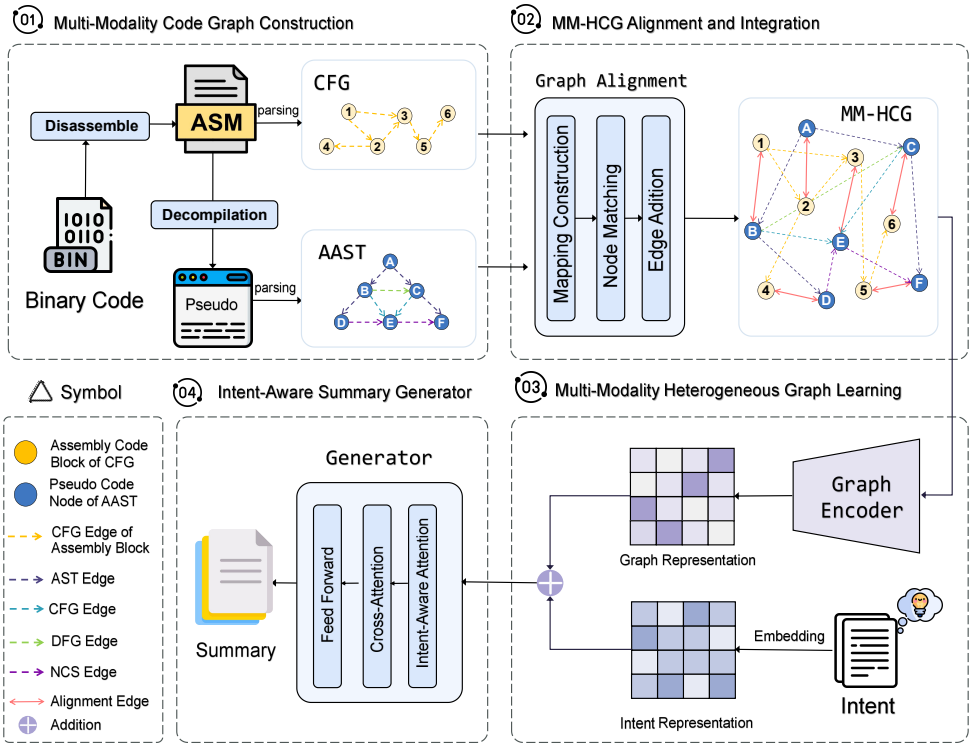

Fig. 3. Overview of MISUM. We first translate the binary code into assembly code (ASM) and pseudo code, converting each into a graph structure. Through the graph alignment and integration, we obtain the Multi-Modal Heterogeneous Code Graph (MM-HCG) and then conduct the graph learning. The Generator produces summaries awarded to specific intent.

traditional structures, we also incorporate the Natural Code Sequence (NCS), which provides a "human-readable" representation of code token relationships. We integrate the AST, CFG, DFG, and NCS into a unified framework known as the Augmented Abstract Syntax Tree (AAST). The AAST is defined as a graph $G^P = (V, E)$, where V represents the nodes from all four structures, and E represents the directed edges capturing syntactic, control flow, data flow, and sequential relationships. By combining these elements, the AAST offers a comprehensive view of a program's semantics and execution logic, facilitating more effective analysis and understanding of the code. [1]

5.2.2 Graph with Assembly Code. To understand the structure and execution behavior of assembly code, we extract the Control Flow Graph (CFG). A CFG is comprised of basic code blocks and their jump control flows among blocks. The nodes indicate basic blocks and the edges indicate jump control flows. CFGs are instrumental in optimizing the assembly code by enabling the identification of redundant or unreachable code blocks. Its capability to represent complex control logic succinctly makes CFGs effective in reverse engineering and automated analysis of assembly code.

5.3 Multi-Modality Heterogeneous Code Graph Alignment and Integration

To align and integrate the modality information from pseudo code and assembly code, we align the above two graphs (pseudo code graph G^P and assembly code graph G^A) by a cross-language

[1]We follow Zhou et al. [2019] to make use of the open-source code analysis platform for C/C++ based on code property graphs, *Joern* [Yamaguchi et al. 2014], to extract ASTs, CFGs, and DFGs for all code functions.

Algorithm 1: Graph Alignment and Integration for Heterogeneous Code Graph

Data: Pseudo Graph G^P, Assembly Code Graph G^A, Pseudo Code P_c, Assembly Code A_c, Address Information from Decompiler $\mathcal{M}$
Result: Heterogeneous Code Graph G^{PA}

Function `AlignGraphs`$(G^P, G^A, P_c, A_c, \mathcal{M})$**:**
  $G^{PA} \leftarrow$ InitializeEmptyGraph();
  `// Step 1: Construct mapping from address information`
  mapping $\leftarrow \varnothing$;
  for *each address pair* (p_i, a_j) *in* $\mathcal{M}$ **do**
    Add mapping from pseudo code statement p_i to assembly code instruction a_j;
  `// Step 2: Node matching`
  for *each node* s_i *in* G^P **do**
    for *each node* v_j *in* G^A **do**
      if s_i *matches* v_j *based on mapping* **then**
        Add edge between s_i and v_j in G^{PA};
      else
        Continue;
  return G^{PA};

statement mapping and construct a unified heterogeneous code graph $G^{PA} = \{G^P; G^A\}$ carrying both low-level execution details and high-level structural information.

Since the pseudo code is obtained by decompiling assembly code using IDA Pro's Hex-Rays plugin [Hex-Rays 2024], the decompilation provides address information that maps pseudo code statements to their corresponding assembly code instructions. This mapping facilitates the alignment of code blocks and statements between pseudo code and assembly code. Algorithm 1 encourages bridging the sub-graphs G^A and G^P and then composing multi-modality heterogeneous graph G^{PA} based on the assembly code-to-pseudo code mapping. The algorithm involves the following steps:

- **Mapping Construction:** Using address information provided by the decompiler, we construct a mapping from assembly code instructions to pseudo code statements. This mapping establishes the correspondence between nodes in the pseudo code graph and their counterparts in the assembly code graph.
- **Node Matching:** We iterate over nodes in both the pseudo code graph and the assembly code graph. Based on the constructed mapping, we determine the corresponding nodes by checking the matching relationships between pseudo code statements and assembly code instructions.
- **Heterogeneous Edge Addition:** After identifying matching nodes, we integrate the two graphs by adding edges between matched nodes. This process effectively unifies the pseudo code and assembly code graphs into a Multi-Modality Heterogeneous Code Graph (MM-HCG).

5.4 MM-HCG Heterogeneous Graph Learning

To fusion the information over graphs, we propose MM-HCG heterogeneous graph learning with heterogeneous mutual attention and message passing. Specifically, it takes the multi-modality heterogeneous graph as input and produces representations that carry both structural and contextual information. It enhances the model's ability to understand complex dependencies and interactions within the code, ultimately leading to more accurate and nuanced code summarization.

5.4.1 Positional Encoder. The positional encoder serves as a foundational component for the graph learning process, enabling the model to capture the order of nodes in the heterogeneous graph. Given the importance of execution order, each node $v \in V$ is assigned a positional encoding based on its position in a depth-first traversal of the graph. It facilitates subsequent learning steps such as §5.4.2 heterogeneous mutual attention and §5.4.3 heterogeneous message passing.

5.4.2 Heterogeneous Mutual Attention. The Heterogeneous Mutual Attention is designed to effectively integrate and prioritize information from diverse types of nodes and edges within the MM-HCG. This component plays a crucial role in understanding how different elements of the pseudo code and assembly code contribute to the overall code semantics by dynamically adjusting the importance of information from neighboring nodes based on their contextual relevance. For each edge $e = (s, v)$ connecting nodes s and v, the attention mechanism evaluates the significance of the message from node s to node v. The attention score, which considers both node and edge types, determines how much influence the message from s should have in updating the representation of v. The attention score $A^{(k)}(s, e, v)$ is calculated as follows:

$$A^{(k)}(s, e, v) = \left(K^{(k)}(s) \cdot W^{A}_{\phi(e)} \cdot \left(Q^{(k)}(v) \right)^{T} \right) \cdot \frac{\mu_{\langle \tau(s), \phi(e), \tau(v) \rangle}}{\sqrt{d}}, \tag{1}$$

where $Q^{(k)}(v)$ and $K^{(k)}(s)$ are the query and key vectors derived from the node features (code block information) of v and s, respectively. These vectors are obtained through linear transformations that account for the node types (code structure information), $\text{Q-Linear}_{\tau(v)}$ and $\text{K-Linear}_{\tau(s)}$. The matrix $W^{A}_{\phi(e)}$ represents the edge type-specific weights, and $\mu_{\langle \tau(s), \phi(e), \tau(v) \rangle}$ is a trainable scaling factor for each meta relation triplet $\langle \tau(s), \phi(e), \tau(v) \rangle$. The normalization factor $\sqrt{d}$ ensures stability in the attention scores. After calculating the unnormalized attention scores, a softmax function is applied to normalize these scores. The attention heads are then concatenated to form the final heterogeneous attention representation.

5.4.3 Heterogeneous Message Passing. The Heterogeneous Message Passing aggregates information from neighboring nodes (context information) by accounting for both node and edge types. For each node v, messages are gathered from its neighbors $\mathcal{N}(v)$: $M^{(k)}(s, e, v) = \text{M-Linear}_{\tau(s)} \left(h_s^{(k-1)} \right) \cdot W^{M}_{\phi(e)}$. $\text{M-Linear}_{\tau(s)}$ projects the representation of node s into the message space, while $W^{M}_{\phi(e)}$ incorporates the influence of the edge type $\phi(e)$. The messages from multiple heads are concatenated to form a comprehensive heterogeneous message representation: $\text{Message} = \text{Concat}(M_1, \ldots, M_h)$. These messages are then weighted according to the attention scores computed in the Heterogeneous Mutual Attention Layer. The weighted aggregation of messages is expressed as:

$$a_v^{(k)} = \sum_{s \in \mathcal{N}(v)} \left(\text{Attention}^{(k)}(s, e, v) \cdot \text{Message}^{(k)}(s, e, v) \right) \tag{2}$$

Finally, the node representation is updated by combining the aggregated message with residual information from the previous layer: $h_v^{(k)} = \sigma \left(\text{C-Linear}_{\tau(v)} \left(a_v^{(k)} \right) \right) + h_v^{(k-1)}$.

The Heterogeneous Message Passing enhances node representations by integrating and updating information from neighboring nodes, taking into account their types and edge types.

5.5 Intent-Tailored Summary Generator

The Intent-Tailored Summary Generator aims to produce summaries that are closely aligned with user intentions by effectively integrating intent information with code representations. The generator employs two types of attention mechanisms: the Intent-Aware Attention Mechanism

(IAAM), which focuses on relevant parts of the encoded code representations based on the user's intent, and the Multi-Head Cross Attention, which refines the context by combining intent-aware information with the current decoder state. Summaries are then generated based on conditional probabilities, ensuring coherence and relevance to the user's objectives.

5.5.1 Intent-Aware Attention Mechanism (IAAM). The IAAM plays a crucial role in adjusting the attention distribution based on the specified intent. This mechanism ensures that the decoder focuses on the parts of the encoded code representations that are most relevant to the user's intent. For each decoder state h_i^{dec} at layer i, and the intent embedding z_{intent} (intent information), the IAAM computes attention scores to prioritize the most relevant encoder outputs, and then the scoring function combines the decoder state, encoder outputs, and intent embedding to produce:

$$\text{score}\left(h_i^{dec}, h_j^{enc}, z_{intent}\right) = \frac{\left(W_Q h_i^{dec}\right)\left(W_K h_j^{enc} + W_I z_{intent}\right)^T}{\sqrt{d_k}} \tag{3}$$

$$\alpha_j^{IAAM} = \frac{\exp\left(\text{score}\left(h_i^{dec}, h_j^{enc}, z_{intent}\right)\right)}{\sum_{j'} \exp\left(\text{score}\left(h_i^{dec}, h_{j'}^{enc}, z_{intent}\right)\right)} \tag{4}$$

The computed attention weights α_j^{IAAM} are then used to generate the intent-aware context vector c_i^{IAAM}($c_i^{IAAM} = \sum_j \alpha_j^{IAAM} h_j^{enc}$). This context vector c_i^{IAAM} represents the importance of different parts of the code with the user's intent, providing the decoder with the most relevant aspects.

5.5.2 Multi-Head Cross Attention with Gated Fusion. Following the IAAM, the Multi-Head Cross Attention refines the context further by integrating the intent-aware context c_i^{IAAM} with the current decoder state h_i^{dec}. This is achieved through cross-attention, where the context from the IAAM is used to enhance the decoder's current state: Cross-Attn$(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$. Here, $Q = W_Q h_i^{dec}$, $K = W_K c_i^{IAAM}$, and $V = W_V c_i^{IAAM}$. This operation results in a cross-attention context c_i^{cross}, which is then combined with the decoder state through a gated fusion mechanism:

$$h_{i+1}^{dec} = \sigma(W_g[c_i^{cross}; h_i^{dec}]) \odot c_i^{cross} + (1 - \sigma(W_g[c_i^{cross}; h_i^{dec}])) \odot h_i^{dec} \tag{5}$$

where σ denotes the sigmoid function, and W_g is a learnable weight matrix that controls how much of the cross-attention context is integrated with the decoder's current state.

5.5.3 Summary Generation through Conditional Probability. The generator outputs the summary by predicting the next token in the sequence based on the context provided by the aforementioned mechanisms. This process utilizes conditional probability to produce a coherent and contextually relevant summary. Given the decoder's state h_i^{dec} at a specific position in the sequence, the probability distribution over the vocabulary for the next token is computed using a softmax function: $P(w_i \mid h_i^{dec}) = \text{softmax}(W_{out} h_i^{dec})$. W_{out} is a learnable weight matrix, and w_i represents the possible tokens in the vocabulary. The decoder selects the token with the highest probability as the next token in the summary. This process is repeated iteratively to generate the complete summary.

In summary, the Intent-Tailored Summary Generator aligns code summaries with user intent through intent-aware and refined cross-attention mechanisms, ensuring that the generated summaries are both relevant and clear. The final summaries are produced using conditional probability, predicting each token based on the context from previous layers for a coherent output.

Table 2. The 50 binary projects and versions, chosen to reflect real-world reverse engineering needs, are commonly used in existing benchmarks for binary tasks.

Binary Projects	Version	Binary Projects	Version	Binary Projects	Version	Binary Projects	Version
a2ps	4.14	binutils	2.30	libiconv	1.15	libidn2	2.0.5
bool	0.2.2	ccd2cue	0.5	libmicrohttpd	0.9.59	libosip2	5.0.0
cflow	1.5	coreutils	8.29	libtasn1	4.13	libtool	2.4.6
cpio	2.12	cppi	1.18	libunistring	0.9.10	lightning	2.1.2
dap	3.10	datamash	1.3	macchanger	1.6.0	nettle	3.4
direvent	5.1	enscript	1.6.6	patch	2.7.6	plotutils	2.6
findutils	4.6.0	gawk	4.2.1	readline	7.0	recutils	1.7
gcal	4.1	gdbm	1.15	sed	4.5	sharutils	4.15.2
glpk	4.65	gmp	6.1.2	spell	1.1	tar	1.30
gnudos	1.11.4	grep	3.1	texinfo	6.5	time	1.9
gsasl	1.8.0	gsl	2.5	units	2.16	vmlinux	4.1.52
gsS	1.0.3	gzip	1.9	wdiff	1.2.2	which	2.21
hello	2.10	inetutils	1.9.4				

6 Experiment Design

6.1 Research Questions

To assess the effectiveness of MISUM, we propose to answer the following research questions:

- **RQ1: What is the performance of MISUM in binary code summarization?** This RQ aims to establish a new effectiveness baseline for binary code summarization.
- **RQ2: How do different architectures and optimization levels affect MISUM?** This RQ investigates the impact of various architectures and optimization levels on the results, providing insights into how these factors influence the effectiveness of MISUM.
- **RQ3: How do different intent categories affect the summarization results of MISUM?** This RQ analyzes the impact of various intent categories on the quality of the summaries generated by MISUM, identifying which types are summarized more effectively and which are less successful.
- **RQ4: How does each component contribute to the overall performance of MISUM?** This RQ involves an ablation study to measure the contribution of different components in MISUM, which could inspire future improvements.
- **RQ5: Can MISUM assist reverse engineers in understanding the different intents within binaries?** This RQ focuses on a user study aimed at evaluating the practical usefulness of MISUM in meeting the varied needs of reverse engineers when interpreting the intents of binary code.

6.2 Dataset Construction

Source Code Selection. To reflect real-world reverse engineering needs, we selected 50 projects that are widely used in existing binary-related benchmarks, such as binary clone detection [Hemel et al. 2011; Yang et al. 2023] and binary function name prediction [Gao et al. 2021; Jin et al. 2022]. These projects have high credibility, excellent code quality, and maintenance standards, covering seven application domains, including cryptography, compression, networking, video, image processing, databases, and neural networks. Table 2 displays the list of 50 binary projects and their corresponding versions.

Compilation. We manually compiled the source code files into binary files using the GCC 7.3.0 compiler at three different optimization levels: `-O1`, `-O2`, and `-O3`, each representing progressively deeper levels of optimization. The `-O1` level applies basic optimizations that enhance code size and performance without significantly increasing compilation time. The `-O2` level introduces more advanced optimizations, such as loop unrolling and inlining. The `-O3` level includes all optimizations from `-O2` with additional techniques such as vectorization and function cloning. Each source code was compiled into nine different binary variants, corresponding to these three optimization levels across three computer architectures (x86, x64, ARM).

Stripping. We employed the strip command in Linux to remove the symbol tables from these binaries. To maintain consistency with actual stripped scenarios, we used the *strip -s* command to process the binary files. This stripping operation eliminates certain sections, including "*debug*," "*symtab*," and "*strtab*," among others. As a result, it removes symbol tables and all associated debugging information from the binary files, ensuring that the files reflect the conditions of a production environment where such data is typically stripped away.

Disassembling. We utilize *IDA Pro* [Hex-Rays 2024] to disassemble both the original and stripped binaries, generating their corresponding assembly code. The assembly code is then segmented at the function level. From the original binary, we extract tuples in the format of <*function_name*, *function_boundaries*>. In the stripped binary, while the *function_name* is replaced with a placeholder *sub_address*, the *function_boundaries* remain consistent regardless of whether the binary is stripped. For the stripped binary, we extract triplets in the format of <*sub_address, stripped assembly code, function_boundaries*>.

Multi-Intent Binary Summary Dataset Construction. To extract multi-intent summaries, we began by analyzing the source code using *srcML* [Maletic and Collard 2015], a tool that parses source files into an XML format, allowing us to identify and extract both single-line and multi-line summaries located above function declarations and definitions. This step resulted in a collection of <*function_name, summaries*> pairs. Next, we utilized the source code to identify the *function_boundaries*, which served as indices to match the function names with their corresponding functions in the stripped binary. This mapping enabled us to construct triplets in the format of <*function_name, stripped binary, summaries*>. To categorize the intents within the summaries, we employed the CodeBERT-base model [Feng et al. 2020], fine-tuned on a dataset manually labeled by the previous study [Mu et al. 2023]. This classifier achieved an accuracy of 92%, demonstrating its effectiveness on the intent classification task. We thus chose to use this model to classify the intents, allowing us to construct the final dataset in the format <*stripped assembly code, intent, summary*>. Subsequently, we implemented a two-step deduplication process. First, a hashing mechanism was used to generate unique identifiers for each data sample, effectively eliminating exact duplicates. Second, we performed an 8:1:1 random split to create training, validation, and test sets with no overlap. Table 3 presents the data statistics for the six different intents across three architectures.

Additionally, there is a concern that code clones would significantly influence the performances of the evaluated approaches. Regarding this, we argue that code duplication phenomenon is mitigated in the binary code domain. The binary code dataset uses <assembly code, summary> pairs instead of <source code, summary> pairs. When two similar source code samples are compiled into binary code, the complex compilation and disassembly processes significantly reduce the similarity between the resulting assembly codes. To illustrate, we analyzed 100 pairs of similar source code samples with an average cosine similarity of 0.92 (encoded by CodeLlama, where > 0.9 indicates high similarity [Yang et al. 2023]). After compilation and disassembly (ARM, O1), the similarity of the corresponding assembly code dropped to 0.61 (encoded by CLAP [Wang et al. 2024], where < 0.8 indicates dissimilarity [Yang et al. 2023]). This significant reduction demonstrates how compilation processes, such as instruction reordering and register allocation, alter the structure of the code, effectively mitigating code duplication in the binary code domain.

6.3 Baselines

6.3.1 DL-based Baselines. (1) **BinT5** [Al-Kaswan et al. 2023] is the first model focused on binary code summarization, extending the application scope of source code pre-trained language models. It treats decompiled code as a special programming language and uses fine-tuned CodeT5 to capture its semantics and generate summaries. (2) **HexT5** [Xiong et al. 2023] is a unified pre-training model also based on CodeT5, allowing for multi-task learning. It supports function name recovery, binary

Table 3. Dataset Statistics.

Dataset	ARM			X86			X64		
	Train	Valid	Test	Train	Valid	Test	Train	Valid	Test
What	455,481	37,524	39,764	439,294	34,659	36,505	433,749	36,296	37,345
Why	114,875	9,047	9,769	105,905	6,849	7,480	108,474	6,236	6,832
How-to-use	25,267	2,420	2,261	13,370	1,652	1,346	19,572	1,884	1,830
How-it-is-done	96,843	7,638	8,264	87,904	3,578	4,194	86,849	3,536	4,523
Property	88,166	7,159	7,683	81,577	4,117	4,740	81,476	4,075	4,932
Others	35,768	1,491	2,405	37,437	1,379	1,578	32,441	1,399	1,413
Total	816,400	65,279	70,146	765,487	52,234	55,843	762,561	53,426	56,875

Table 4. Detail information of Large Language Models. In the License column, "✓" denotes Open Source, while "×" denotes Closed Source.

Domain	Model	Size	Base Model	Training Corpus			Publisher	License
				Raw Size	#Tokens	#Instances		
Code LLMs	CodeGen2-7b-instruct	7B	CodeGen2	-	1.4T	-	Salesforce	✓
	WizardCoder-33b-V1.1	33B	Deepseek-Coder	-	-	-	WizardLM	✓
	Code Llama-7b-instruct-hf	7B	Llama-2-7b	4.4TB	525.0B	-	Meta AI	✓
	Code Llama-70b-instruct-hf	70B	Llama-2-70b	-	1.0T	-	Meta AI	✓
	DeepSeek-Coder-33b-instruct	33B	-	-	2.0T	-	DeepSeek-AI	✓
General LLMs	ChatGLM3-6B	6B	-	-	1.4T	-	THUDM	✓
	Vicuna-13b-v1.5	13B	Llama-2-13b	-	-	125.0K	L.Zheng et al.	✓
	Llama-3-13b-chat-hf	13B	-	-	2.0T	-	Meta AI	✓
	Llama-3-70b-chat-hf	70B	-	-	2.0T	-	Meta AI	✓
	Mistral-7B-Instruct-v0.2	7B	Mistral-7B	-	-	-	Mistral AI	✓
	ChatGPT	175B	-	-	-	-	OpenAI	×
	GPT4	1.7T	-	-	-	-	OpenAI	×

code summarization, and other downstream tasks, showing promising performance. (3) **CP-BCS** [Ye et al. 2023] utilizes a bidirectional instruction-level control flow graph and pseudo code to learn the comprehensive binary function execution behavior and logic semantics.

6.3.2 LLM-based Baselines. We follow Shang et al. [2024] to evaluate the effectiveness of the most popular large language models (LLMs) in multi-intent binary code summarization. Our evaluation was conducted in a zero-shot setting to assess the models' ability to generalize to this task without any fine-tuning or task-specific training. We extensively evaluated four code-domain LLMs (CodeGen [Nijkamp et al. 2022], WizardCoder [Luo et al. 2023], DeepSeek-Coder [Guo et al. 2024], Code Llama [Roziere et al. 2023]) and six general-domain LLMs (ChatGLM [GLM et al. 2024], Vicuna [Chiang et al. 2023], Llama [Touvron et al. 2023], Mistral [Jiang et al. 2023], ChatGPT [Ouyang et al. 2022], GPT4 [Achiam et al. 2023]). Detailed information about these Large Language Models can be found in Table 4.

6.4 Evaluation metrics

We use common metrics for evaluation: (1) **BLEU** [Papineni et al. 2002] measures the similarity between a generated and reference sentence using n-gram precision, commonly used in code comment generation. (2) **ROUGE** [Lin 2004], specifically ROUGE-L, evaluates text similarity based on overlapping units like n-grams, word pairs, and the longest common subsequence. (3) **METEOR** [Banerjee and Lavie 2005] assesses the quality of generated summaries by aligning them with reference summaries and obtaining similarity.

6.5 Implementation details

Our model is implemented with the framework PyTorch [2]. All the experiments are performed on a server with 8 NVIDIA A100 GPUs. The batch size is set to 256, and the Adam optimizer is used with

[2]https://pytorch.org/

Table 5. Results on the dataset for ARM architecture with O1 optimization level for comparison of DL-based, Code LLMs and General LLMs baselines on multi-intent binary code summarization. To simplify notation, we use B to represent *BLEU*, M to represent *METEOR*, and R to represent *Rouge-L*. The best-performing methods in each intent domain are highlighted with a blue background, while the overall best methods across all domains are highlighted with a red background and marked in **bold**.

Intent / Method	What			Why			How-to-use			How-it-is-done			Property			Others			Average		
	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R
BinT5	0.04	2.12	4.83	0.03	2.09	4.81	0.05	2.13	4.86	0.04	2.06	4.79	0.03	2.08	4.78	0.03	2.04	4.68	0.04	2.09	4.79
HexT5	0.09	6.32	8.53	0.07	6.24	8.46	0.11	6.45	8.64	0.08	6.28	8.49	0.10	6.32	8.55	0.08	6.29	8.48	0.09	6.32	8.53
CP-BCS	5.83	26.19	20.67	3.47	10.06	15.42	3.06	10.81	15.84	3.61	10.64	15.39	3.48	10.73	15.43	3.41	10.04	15.48	3.81	13.08	16.37
CodeGen	3.87	23.61	18.95	3.64	23.15	18.64	3.46	22.57	17.65	3.51	22.64	18.64	3.56	22.96	18.57	3.41	22.67	17.65	3.58	22.93	18.35
Code Llama-7B	4.65	22.20	20.66	4.21	21.80	20.13	4.32	21.82	20.13	4.12	21.58	19.68	4.05	21.65	20.61	4.63	21.98	20.12	4.33	21.84	20.22
WizardCoder	4.74	23.43	20.34	4.65	23.13	20.16	4.68	23.12	20.84	4.59	23.31	19.86	4.65	23.17	19.89	4.65	23.23	19.93	4.66	23.23	20.17
DeepSeekCoder	5.22	24.10	20.84	5.17	23.68	20.64	5.13	23.80	20.64	5.16	23.68	20.46	5.02	23.85	20.16	4.86	23.74	20.62	5.09	23.81	20.56
Code Llama-70B	4.64	23.75	20.82	4.28	23.46	20.47	4.35	23.46	20.14	4.35	23.46	20.49	4.67	23.14	20.68	4.35	23.48	20.63	4.44	23.46	20.54
ChatGLM3	4.29	26.37	20.66	4.25	26.29	20.13	4.15	26.06	19.43	4.17	26.24	19.24	4.16	26.03	20.17	4.13	26.01	18.98	4.19	26.17	19.77
Mistral	5.98	24.37	23.55	5.81	24.23	23.45	5.61	24.15	23.42	5.61	24.37	23.05	5.61	23.89	22.86	5.24	24.16	23.42	5.64	24.20	23.29
Llama3-13B	6.16	24.47	22.51	5.79	24.17	22.34	6.09	24.38	22.16	6.01	24.32	22.15	6.02	23.97	22.23	5.86	24.23	21.86	5.99	24.26	22.21
Vicuna	4.90	21.20	22.48	4.52	20.84	22.13	4.52	20.84	22.13	4.86	20.96	22.37	4.68	20.86	21.86	4.68	20.48	22.01	4.69	20.86	22.16
Llama3-70B	5.51	26.26	21.51	5.16	26.13	21.41	5.16	25.89	21.34	5.12	26.05	20.98	5.46	26.04	21.45	5.25	26.13	21.46	5.28	26.08	21.36
ChatGPT	6.89	27.51	22.74	4.81	27.46	22.67	6.48	27.46	22.35	6.21	27.11	22.32	6.82	27.17	22.63	6.12	27.23	22.32	6.22	27.32	22.51
GPT4	7.37	28.13	23.76	7.14	27.81	23.48	7.13	28.01	23.37	7.28	27.84	23.24	7.19	27.87	23.68	7.15	27.93	23.48	7.21	27.93	23.50
MiSum (Ours)	**10.56**	**33.05**	**28.91**	**9.87**	**32.63**	**28.17**	**8.96**	**32.36**	**28.48**	**10.16**	**32.76**	**28.43**	**10.44**	**31.86**	**28.61**	**10.29**	**32.43**	**28.76**	**10.05**	**32.52**	**28.56**

an initial learning rate of 10^{-4}. The training process will terminate after 100 epochs or will stop early if the performance on the validation set does not improve for 10 epochs. The number of blocks N in both the encoder and decoder is 8, with the number of heads h in the multi-head attention mechanism set to 16. The dimension of the word embedding vectors is 512. We leverage greedy search during validation and beam search during model inference, setting the beam width to 4. Additionally, for the LLM-based baselines, we follow the prompt engineering approach proposed by Geng et al. [2024]. For example, the prompt used in our implementation when the intent is "*what*" is: "You are an expert C/C++ programmer and reverse engineer, please describe the functionality of the method. <Pseudo Code> <Assembly Code>."

7 Evaluation

7.1 RQ1: What is the performance of MiSum in binary code summarization?

Compared to DL-based baselines. We compare our approach with three pre-trained models fine-tuned on binary comprehension tasks. Due to space constraints, we present results obtained on the ARM architecture with the O1 optimization level as Table 5 shown, while results from other settings are available in RQ2, where similar trends can be observed. For these DL-based expert models, BinT5 [Al-Kaswan et al. 2023] achieves average scores of 0.04, 2.09, and 4.79 on BLEU-4, METEOR, and ROUGE-L across the six intent categories, respectively. HexT5 [Xiong et al. 2023] shows slight improvements, but its performance remains significantly lower. Interestingly, CP-BCS [Ye et al. 2023] achieves a comparable level with other LLMs on the "what" intent, but shows a marked decline on the other five intents. This is mainly because CP-BCS is trained to learn a one-to-one mapping of <binary, code functionality>, making it well-suited for describing the functionality of a method but unable to meet other intents.

Compared to LLM-based baselines. We compare our approach with the most popular LLMs, including Code LLMs, which specialize in code-related tasks, and General LLMs, which are designed for more general tasks, as Table 5 shown. For Code LLMs, we compared models of various sizes, including 7B, 33B, and 70B. The results indicate that DeepSeekCoder-33B outperforms other Code LLMs in BLEU-4, METEOR, and ROUGE-L, with average scores of 5.09, 23.81, and 20.56, respectively. For General LLMs, GPT-4 demonstrates a strong capability for binary understanding and summary generation, surpassing all other LLMs, including both Code LLMs and General LLMs. However, due to limited training on binary data, GPT-4 still lacks domain-specific knowledge and has room for

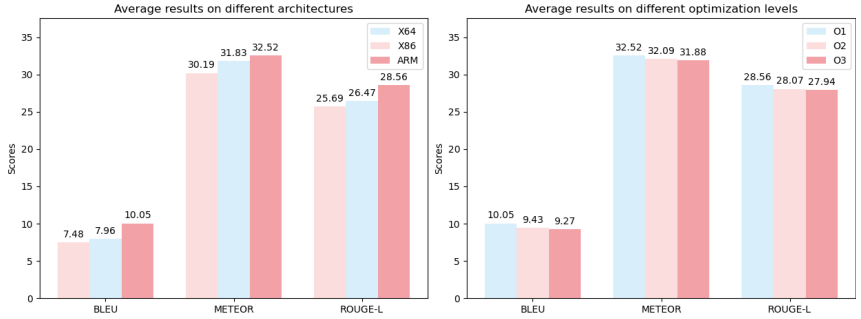

Fig. 4. Comparison of average results for different architectures and optimization levels.

improvement. Our method, MiSUm, effectively integrates pseudo-code and assembly code into a heterogeneous graph to fully capture the semantic information of binary code and is further trained with user intents. As a result, MiSUm outperforms all other baselines, including DL-based models, Code LLMs, and General LLMs, achieving average scores of 10.05, 32.52, and 28.56, respectively. Our paired t-tests confirmed that MiSUm significantly outperforms all baselines (p-values < 0.01).

> **Answering RQ1:** For each intent category, MiSUm outperforms the state-of-the-art baselines in terms of three metrics, including DL-based models, Code LLMs, and General LLMs. MiSUm improves the performance of BLEU, METEOR, ROUGE-L by 39.4%↑, 16.4%↑, 21.5%↑.

7.2 RQ2: How do different architectures and optimization levels affect MiSUm?

Analysis of Different Architectures. We evaluated MiSUm across three architectures: ARM, X86, and X64, as illustrated in the left of Figure 4. On average, MiSUm achieved 26.3% and 34.4% higher BLEU performance on ARM compared to X86 and X64, respectively, with notable gains in both METEOR and ROUGE-L metrics. These improvements can be attributed to ARM's simpler and more adaptable Reduced Instruction Set Computing (RISC) architecture. In contrast, the X86 and X64 architectures are based on Complex Instruction Set Computing (CISC), which involves a larger number of operation codes and registers to handle intricate mathematical operations, thus complicating the interpretation of their assembly code.

Analysis of Different Optimization Levels. We assessed our method under various optimization levels, including O1, O2, and O3, as depicted in the right of Figure 4. Overall, MiSUm demonstrates superior performance under the O1 optimization level compared to O2 and O3. Our observations of assembly code generated under different optimization levels reveal that O2 and O3 apply a range of advanced optimization techniques, such as vectorization instructions and loop unrolling, to enhance execution speed. However, these techniques result in more complex assembly code. On the other hand, O1 employs relatively straightforward strategies, such as register allocation and basic block reordering, which avoid producing excessively complex assembly code. Additionally, the pseudo-code derived from O1-optimized assembly code tends to be more accurate and closer to the original source code compared to O2 and O3. Consequently, the pseudo-code generated under O1 provides richer and more precise semantic information, enabling MiSUm to achieve better performance across all three metrics.

> **Answering RQ2:** The performance of MiSUm is affected by both architecture and optimization levels. The best results are on the ARM architecture and at the O1 optimization level compared to other settings.

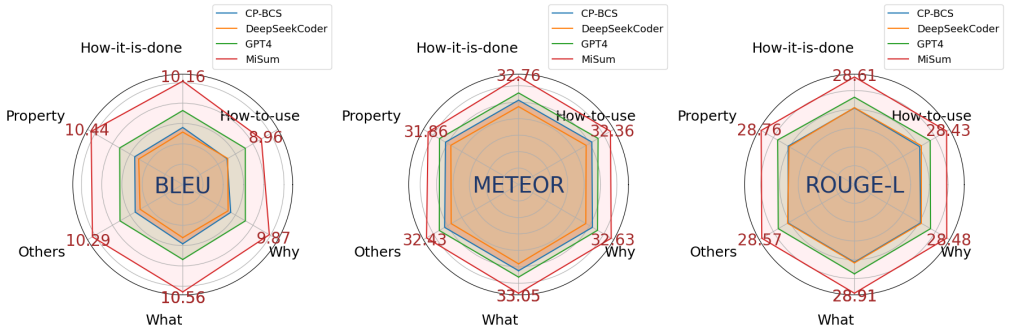

Fig. 5. Results of MiSum on six categories of intents.

7.3 RQ3: How do different intent categories affect the summarization results of MiSum?

To analyze the impact of different intent categories on summary generation, we selected the best-performing models from three domains in RQ1: CP-BCS (DL-based), DeepSeekCoder (Code LLMs), GPT-4 (General LLMs), and our proposed MiSum. Figure 5 presents the results of these models across six intents on three metrics. Overall, MiSum outperforms other state-of-the-art models across all six intents. This superior performance is attributed to MiSum's ability to integrate the semantic information of both pseudo-code (which captures high-level code abstraction) and assembly code (which retains detailed information). While CP-BCS also considers both types of information, it lacks an effective fusion strategy, resulting in suboptimal complementary effects. In contrast, MiSum employs a heterogeneous graph to blend pseudo-code and assembly code, enabling a more comprehensive understanding of the code.

When examining performance by intent category, MiSum achieves the highest scores across all three metrics for the "What" intent, which aligns with the results seen in other baselines. This suggests that models are generally more adept at describing the primary functionality of code snippets. However, while baselines show a marked decline in performance for other intents (except "What"), MiSum consistently excels across all intents. This is primarily due to its effective fusion of pseudo-code and assembly code semantics using a heterogeneous graph, allowing the model to fully capture the nuances of binary code. In contrast, other baselines primarily focus on the textual information of code while overlooking the importance of structural information and effective information integration, thereby limiting their ability to comprehensively understand binary code.

> **Answering RQ3:** While all models demonstrate strong performance in the "What" intent, other models struggle with intents beyond "What." In contrast, MiSum excels across all intent categories, effectively integrating both pseudo-code and assembly code to deliver superior results.

7.4 RQ4: How does each component contribute to the overall performance of MiSum?

To evaluate the performance of different components within the MiSum framework, we conducted a series of ablation studies, as shown in Table 6. For the model components, these studies involve comparing three modified versions of the MiSum system, each with a critical element removed: (1) the removal of the Multi-Modality Heterogeneous Code Graph (MM-HCG), denoted as - *MM-HCG*; (2) the exclusion of the MM-HCG Encoder, indicated as - $Enc_{MM\text{-}HCG}$; (3) the removal of the Intent-Aware Attention Mechanism (IAAM), marked as - *IAAM*. Regarding the code modalities, we performed analyses by removing either the assembly code or the pseudo-code, denoted as - *Assembly Code* and - *Pseudo Code*, respectively. To simplify the demonstration, we focused on a specific dataset with a fixed architecture (ARM) and optimization level (O1).

Table 6. Ablation study results on the dataset for ARM architecture with O1 optimization level.

Model	What			Why			How-to-use			How-it-is-done			Property			Others			Average		
	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R	B	M	R
MISUM	10.56	33.05	28.91	9.87	32.63	28.17	8.96	32.36	28.48	10.16	32.76	28.43	10.44	31.86	28.61	10.29	32.43	28.76	10.05	32.52	28.56
- MM-HCG	5.42	24.63	21.04	4.69	25.31	21.06	4.72	23.77	20.65	5.27	23.41	21.15	5.38	22.69	20.84	4.38	23.16	20.82	4.98	23.83	20.93
- $Enc_{MM\text{-}HCG}$	5.21	24.23	21.64	4.87	25.18	21.75	4.68	23.65	20.48	5.16	23.21	21.06	5.24	22.41	20.47	4.25	22.86	20.47	4.90	23.59	20.98
- IAAM	5.86	25.16	22.79	4.23	25.06	21.41	4.32	23.22	20.17	5.01	22.89	20.96	5.06	21.86	20.19	4.03	21.30	20.11	4.75	23.25	20.94
- Assembly Code	6.74	26.12	23.63	5.45	26.63	22.81	5.17	24.35	21.83	5.92	23.84	21.78	5.63	22.81	20.86	4.48	23.67	21.26	5.57	24.57	22.03
- Pseudo Code	5.36	23.85	20.93	4.06	24.84	20.76	4.28	23.62	20.13	4.93	22.75	20.64	4.96	21.54	19.62	3.67	21.52	19.88	4.54	23.02	20.33

From the perspective of model components, we replaced MM-HCG with the textual representations of pseudo-code and assembly code, resulting in a notable decline in summary quality across all intents. Subsequently, we substituted the MM-HCG-specific graph encoder with a traditional graph attention network (GAT) encoder [Veličković et al. 2017], which also led to a significant negative impact on performance, demonstrating the effectiveness of the MM-HCG Encoder. Additionally, when the Intent-Aware Attention Mechanism (IAAM) was removed from the decoder, we observed a substantial decline in performance across all intents except "what," indicating that IAAM is crucial for generating intent-specific summaries.

Regarding the code modalities, we first removed assembly code and retained only pseudo-code, which resulted in a noticeable performance drop. When pseudo-code was removed and only assembly code was retained, the decline in performance (BLEU: 44.88%↓, METEOR: 29.21%↓, ROUGE-L: 28.82%↓) was greater than when only pseudo-code was retained (BLEU: 44.58%↓, METEOR: 24.45%↓, ROUGE-L: 22.86%↓). This suggests that pseudo-code is more effective than assembly code in supporting multi-intent binary summarization tasks.

Answering RQ4: The ablation studies reveal that removing any component—whether the Multi-Modality Heterogeneous Code Graph (MM-HCG), the MM-HCG Encoder, or the Intent-Aware Attention Mechanism (IAAM)—significantly degrades performance, emphasizing the critical role of each in maintaining high-quality summaries. Additionally, omitting pseudo-code leads to the most substantial performance decline, highlighting its importance in multi-intent summarization.

7.5 RQ5: Can MISUM assist reverse engineers in understanding the different intents within binaries?

We conducted a human evaluation to assess the quality of summaries generated by MISUM in terms of Similarity, Fluency, and Appropriateness. For this evaluation, we recruited 10 participants, consisting of 5 PhD students and 5 reverse engineers, each with 1-3 years of experience in software engineering and reverse engineering. To ensure a manageable evaluation process, we divided the 2,400 generated summaries into 10 groups, with each group containing 240 summaries covering all 6 different intents. Each summary was randomly selected to avoid any bias across groups, and the groups were designed to ensure that no summary was duplicated across groups. Each participant was assigned to review 2 different groups, leading to each summary being reviewed by 2 participants. For the evaluation, we used the best-performing models from three domains as baselines: CP-BCS [Ye et al. 2023] for DL-based domain models, DeepSeekCoder [Guo et al. 2024] for Code LLMs, and GPT-4 [Achiam et al. 2023] for General LLMs. Participants were asked to evaluate the summaries based on the following criteria:

- **Similarity:** How similar is the generated summary to the ground truth?
- **Fluency:** Is the generated summary syntactically correct and fluent?
- **Appropriateness:** Does the generated summary align with the intended meaning?

Participants rated each summary on a scale from 1 to 5 for each metric, with higher scores indicating better quality. To ensure consistency, the number and length of assembly code instructions were kept uniform across all groups. Figure 6 presents the results of the human evaluation. MISUM outperforms all other models across all three metrics, demonstrating consistently high performance,

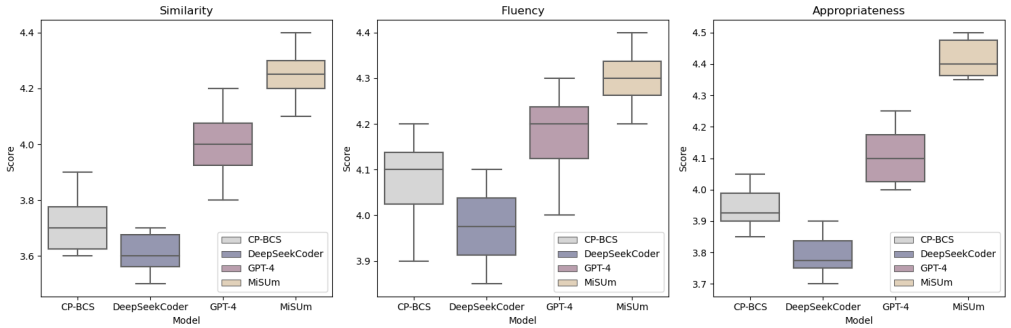

Fig. 6. Results of Human evaluation.

as evidenced by the boxplots. In Similarity, it achieves a median score of 4.2, with a range from 4.0 to 4.4 and an upper quartile (Q3) of 4.3, highlighting its strong consistency and ability to generate highly relevant content. The absence of outliers further underscores its reliability in maintaining contextual accuracy. In Fluency, MiSUm exhibits a similar pattern, with a median of 4.3 and a range from 4.2 to 4.4, reflecting its consistent ability to produce fluent and natural language. The upper quartile (Q3) at 4.35 reinforces the model's linguistic smoothness with minimal variation. Finally, in Appropriateness, MiSUm scores a perfect 4.4, with a median of 4.4 and a narrow range from 4.3 to 4.4, indicating its exceptional suitability for the tasks. This narrow range and high consistency across all metrics suggest that MiSUm excels not only in generating contextually accurate content but also in producing fluent and appropriate responses, making it the standout model in this evaluation.

In contrast, GPT-4, DeepSeekCoder, and CP-BCS perform lower than MiSUm, particularly in Similarity and Fluency. GPT-4 shows consistent performance, with a median of 4.0 in Similarity, but it slightly lags behind MiSUm in terms of relevance. DeepSeekCoder exhibits greater variability, with a wider range in Similarity (median of 3.8) and less consistency in Appropriateness, indicating that its outputs are less reliably suitable for the given tasks. CP-BCS shows the weakest performance overall, with narrow ranges in both Similarity (median of 3.7) and Fluency (median of 4.1), suggesting limited variability and weaker overall output quality. These models demonstrate considerable room for improvement in generating content that is more relevant, fluent, and consistent.

> **Answering RQ5:** Human evaluations show that MiSUm surpasses all other baselines in similarity, fluency, and appropriateness, demonstrating its effectiveness in generating summaries that align well with user intents and enhance understanding of binary code.

8 Discussion

8.1 Threats To Validity

External threats. Our study faces several external threats. First, the current evaluation relies solely on C/C++ datasets due to the limited availability of open-source binary code summarization datasets, which introduces a potential selection bias. Although our proposed multi-modality heterogeneous fusion and alignment method has been demonstrated to be effective on these datasets, its performance on other programming languages remains unverified. Notably, while our framework is initially implemented for C/C++ binaries, it is designed with the potential to be extended to other languages by incorporating appropriate decompilers and disassemblers. However, the applicability and reliability of such extensions are yet to be confirmed. To address these threats, future work will systematically evaluate other languages to further validate the framework's generalizability. Secondly, there is a potential selection bias in baseline models. Recent years have seen a surge in

the development of large models, including both code-specific and general-purpose ones [Achiam et al. 2023; Touvron et al. 2023]. Our study may have a selection bias by considering only a subset of these models as baselines. However, the selected models are widely recognized and have demonstrated state-of-the-art performance on code-related tasks [Shang et al. 2024]. To provide a more comprehensive evaluation, we plan to include additional models in future studies.

Internal threats. One internal threat comes from the potential biases in human evaluation, that is, the results can be influenced by the participants' programming experience and their understanding of the evaluation metrics. Nonetheless, we have implemented several strategies to mitigate such influences. Specifically, the inclusion of both PhD students and professional reverse engineers with diverse experience levels ensures a comprehensive evaluation spectrum. Reviewing each code-summary pair with two participants mitigates individual biases and enhances the reliability of the assessment. The structured grouping and random shuffling of summaries promote fair comparisons and minimize potential order effects, contributing to the validity and robustness of the results. Another internal threat is the influence of evaluation metrics on our study's results and conclusions. To mitigate the bias from metric selection, we employed three different metrics that evaluate the quality of generated code summaries from multiple perspectives. This multi-metric approach provides a more robust and comprehensive evaluation of our method's performance.

8.2 The Potential of Involving Additional Modality Information

Our study explores the combination of different modality information for multi-intent binary code summarization. A natural question would be whether we could involve more modality information, such as the LLVM IR, for further performance improvement. To explore this, we conducted an additional experiment with two setups on the ARM architecture and at the O1 optimization level: (1) We first used IR as the sole input, which led to a 14.2% performance drop due to IR's lack of explicit structure and context. (2) We integrated IR with pseudo-code and assembly. Specifically, we used the LLM-Compiler-7B model [Cummins et al. 2024] to extract IR features and concatenated these with pseudo-code and assembly features, resulting in a 3.9% performance improvement, highlighting the valuable semantic information provided by IR. Encouraged by these results, we plan to explore more advanced approaches for incorporating IR effectively in future work.

9 Conclusion

In this study, we introduce the task of multi-intent binary code summarization, which significantly enhances the efficacy of reverse engineering processes. To address this challenge, we propose MISUM, an innovative approach that utilizes Multi-modal Heterogeneous Code Graph (MM-HCG) alignment to integrate semantic information from both assembly and pseudo code. Our Summary Generator employs an intent-aware attention mechanism to produce customized summaries tailored to various intents. Extensive experiments and human evaluations show that MISUM outperforms leading baselines, paving the way for future advancements in software reverse engineering.

10 Data Availability

All the code and data in this study are publicly available at https://github.com/Kobe-Zed/MiSum.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62402506), and the Research Foundation from NUDT (Grant No. ZK24-05).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)* 2, 3 (2023), 6.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524* (2024).

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 607–619.

Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

Mingyang Geng, Shangwen Wang, Dezun Dong, Shanzhi Gu, Fang Peng, Weijian Ruan, and Xiangke Liao. 2022. Fine-Grained Code-Comment Semantic Interaction Analysis. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 585–596. https://doi.org/10.1145/3524610.3527887

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. 2023. Interpretation-based code summarization. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 113–124.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. *arXiv preprint arXiv:2406.12793* (2024).

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working conference on reverse engineering*. IEEE, 35–44.

Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72.

Hex-Rays. 2024. IDA Pro. https://hex-rays.com/ida-pro/. 2021.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601* (2023).

Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1631–1645.

Xingzheng Li, Bingwen Feng, Guofeng Li, Tong Li, and Mingjin He. 2021. A vulnerability detection system based on fusion of assembly code and source code. *Security and Communication Networks* 2021, 1 (2021), 9997641.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

Zheng Ma, Yuexiu Gao, Lei Lyu, and Chen Lyu. 2022. MMF3: neural code summarization based on multi-modal fine-grained feature fusion. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 171–182.

Jonathan I Maletic and Michael L Collard. 2015. Exploration, analysis, and manipulation of source code using srcML. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 951–952.

Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837* (2022).

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International conference on program comprehension (ICPC)*. IEEE, 23–32.

Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-intent driven code comment generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 768–780.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

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

Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2024. How far have we gone in stripped binary code understanding using large language models. *arXiv preprint arXiv:2404.09836* (2024).

Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al. 2024. Natural Language Outlines for Code: Literate Programming in the LLM Era. *arXiv preprint arXiv:2408.04820* (2024).

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*. 43–52.

Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. 101–110.

Felix Stahlberg. 2020. Neural machine translation: A review. *Journal of Artificial Intelligence Research* 69 (2020), 343–418.

Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*. Ieee, 83–92.

Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for source code summarization. *Automated Software Engineering* 31, 1 (2024), 22.

Wenxin Tao, Xiaohong Su, Jiayuan Wan, Hongwei Wei, and Weining Zheng. 2023. Vulnerability detection through cross-modal feature enhancement and fusion. *Computers & Security* 132 (2023), 103341.

Scarlett Taviss, Steven HH Ding, Mohammad Zulkernine, Philippe Charland, and Sudipta Acharya. 2024. Asm2Seq: Explainable Assembly Code Functional Summary Generation for Reverse Engineering and Vulnerability Analysis. *Digital Threats: Research and Practice* 5, 1 (2024), 1–25.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 503–515.

Jinshui WANG, Xingsi XUE, and Wei WENG. 2015. Source code summarization technology based on syntactic analysis. *Journal of Computer Applications* 35, 7 (2015), 1999.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. 2023. HexT5: Unified Pre-Training for Stripped Binary Code Information Inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 774–786.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*. IEEE, 590–604.

Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–40.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. 2023. CP-BCS: Binary Code Summarization Guided by Control Flow Graph and Pseudo Code. *arXiv preprint arXiv:2310.16853* (2023).

Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 1359–1371.

Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. 2022. A survey of automatic source code summarization. *Symmetry* 14, 3 (2022), 471.

Wenhao Zheng, Hongyu Zhou, Ming Li, and Jianxin Wu. 2019. CodeAttention: translating source code to comments by exploiting the code constructs. *Frontiers of Computer Science* 13 (2019), 565–578.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

Zhaolei Zhou and Lei Liu. 2024. MMCS: A Code Summarization Approach Based on Multi-Modal Feature Enhancement. In *2024 4th International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. IEEE, 389–393.

Received 2024-09-13; accepted 2025-01-14