

# An Empirical Study on the Effectiveness of Large Language Models for Binary Code Understanding

Xiuwei Shang · Zhenkan Fu · Shaoyin Cheng\* ·  
Guoqiang Chen\* · Gangyang Li · Li Hu ·  
Weiming Zhang · Nenghai Yu

Received: date / Accepted: date

**Abstract** Binary code analysis plays a pivotal role in the field of software security and is widely used in tasks such as software maintenance, malware detection, software vulnerability discovery, patch analysis, etc. However, unlike source code, reverse engineers face significant challenges in understanding binary code due to the lack of intuitive semantic information. Although traditional reverse tools can convert binary code into C-like pseudo code, the lack of code comments and symbolic information such as function names still makes code understanding difficult. In recent years, two groups of techniques have shown promising prospects: (1) Deep learning-based techniques have demonstrated competitive results in tasks related to binary code understanding, furthermore, (2) Large Language Models (LLMs) have been extensively pre-trained at the source-code level for tasks such as code understanding and generation. This has left participants wondering about the capabilities of LLMs in binary code understanding. To this end, this work proposes a benchmark to evaluate the effectiveness of LLMs in real-world reverse engineering scenarios, which covers two key binary code understanding tasks, i.e., function name recovery and binary code summarization. To more comprehensively evaluate, we include binaries with multiple target architectures as well as different optimization options. We gain valuable insights into the capabilities and limitations through extensive empirical studies of popular LLMs using our benchmark. Our evaluations reveal that existing LLMs can understand binary code to a certain extent, thereby improving the efficiency of binary code

\* Shaoyin Cheng and Guoqiang Chen are corresponding authors.

Xiuwei Shang, Zhenkan Fu, Gangyang Li and Li Hu  
University of Science and Technology of China, Hefei, China  
E-mail: {shangxw, ligangyang, pdxbshx}@mail.ustc.edu.cn, buildxcbpro@gmail.com

Guoqiang Chen  
QI-ANXIN Technology Research Institute, Beijing, China  
E-mail: guoqiangchen@qianxin.com

Shaoyin Cheng, Weiming Zhang and Nenghai Yu  
University of Science and Technology of China, Anhui Province Key Laboratory of Digital Security, Hefei, China  
E-mail: {sycheng, zhangwm, ynh}@ustc.edu.cn

analysis. Our results highlight the great potential of the LLMs in advancing the field of binary code understanding, and provide new directions for binary code analysis techniques.

**Keywords** Reverse Engineering · Binary Code Understanding · Program Comprehension · Large Language Models · Empirical Study

## 1 Introduction

In the field of software security, binary code analysis plays a foundational role in tasks such as reverse engineering (Canfora et al., 2011), software vulnerability detection (Giffin et al., 2004), digital forensics Naeem et al. (2022), and patch analysis Xu et al. (2017), with engineers constantly dealing with vast amounts of unknown binary files. However, unlike human-readable source code, binary code that has been compiled, optimized, and stripped of symbol information (Zhang et al., 2021) is like a "black box" devoid of semantic labels, where function names are simplified to placeholders, variable types are degraded to register operations, and code comments are completely absent. This semantic gap poses a huge challenge to reverse engineers in understanding binary code.

Although many decompilation tools, such as IDA Pro (Hex-RaysSA, 2024), Ghidra (NationalSecurityAgency, 2024) and BinaryNinja (Vector35, 2024), can heuristically convert binary code into C-like pseudo code, alleviating some of the difficulties, they still lack easy-to-understand semantics information, especially function names and code comments that play an important role in comprehending the code (Gellenbeck and Cook, 1991; Gao et al., 2021). Recently, borrowing ideas from Natural Language Processing (NLP), deep learning-based methods have been proposed for understanding binary code. In the task of function name recovery, NERO (David et al., 2020a), NFRE (Gao et al., 2021) and SymLM (Jin et al., 2022) utilized the disassembled assembly instruction sequence as neural models input to reassign descriptive names. NER (Chen et al., 2023b) utilized decompiled pseudo code with a higher abstraction level as input and achieves better performance.

Besides, the function name is only part of the semantic completion and is not enough to represent the complete behavioral logic of the code (Sridhara et al., 2010). If a natural language description can be generated for the binary code, it will greatly save the reverse engineer's analysis time. BinT5 (Al-Kaswan et al., 2023) is the first pre-trained generative model designed specifically for binary code summarization, which is based on the source code model CodeT5 (Wang et al., 2021) and fine-tuned on binaries. Subsequently, as a unified multi-task pre-training model, HexT5 (Xiong et al., 2023) can perform multiple downstream tasks such as code summarization and function name recovery. However, the expert methods mentioned above generally perform poorly when faced with unseen code samples, and their generalization capabilities still need to be improved.

Recently, breakthroughs in Large Language Models (LLMs) have brought new opportunities in this field. General LLMs, such as Llama (Touvron et al., 2023b), ChatGPT (Ouyang et al., 2022), etc., have been widely demonstrated for their capabilities

in natural language understanding and generation. Furthermore, LLMs in the programming language domain (e.g., CodeLlama (Roziere et al., 2023) and WizardCoder (Luo et al., 2023)) have shown notable ability in program analysis tasks, like fixing security vulnerabilities (Wu et al., 2023), test cases auto-generation (Zhang et al., 2023). These developments demonstrate the potential of LLMs to handle complex and structured information flows that are particularly important for understanding binary code. More strikingly, the few-shot learning property of LLMs enables them to quickly adapt to new domains via prompt engineering Dai et al. (2022). This capability offers new possibilities for binary code analysis: Can LLMs bridge the representation gap between source code and binary code, and directly infer function semantics from decompiled pseudo code? Does the generated semantic information exhibit accuracy and interpretability comparable to that of professional reverse engineers? These questions have yet to be systematically answered in existing research.

In this paper, instead of developing a new technique, we investigate and compare the capabilities of various LLMs in understanding binary code. By harnessing the advanced semantic modeling and reasoning power of LLMs, we seek to explore the extent to which these models are able to understand binary code, a task that is traditionally handled by skilled human engineers (David et al., 2020b). To facilitate our evaluation, we design an automated approach to construct an evaluation benchmark dataset, which includes aligned source code, natural language summaries, and decompiled pseudo code. We then contrast the capabilities of LLMs on two binary code understanding tasks, namely: (1) function name recovery, and (2) binary code summarization. We extensively evaluated eight code domain LLMs (CodeGen (Nijkamp et al., 2023), WizardCoder (Luo et al., 2023), DeepSeek-Coder (Guo et al., 2024), CodeLlama (Roziere et al., 2023) et.al.), eight general domain LLMs (ChatGLM (Zeng et al., 2022), Vicuna (Zheng et al., 2023), Llama (Touvron et al., 2023b), Mistral (Jiang et al., 2024), ChatGPT (Ouyang et al., 2022) et.al.), and four deep learning-based expert models (SymLM (Jin et al., 2022), NER (Chen et al., 2023b), BinT5 (Al-Kaswan et al., 2023), HexT5 (Xiong et al., 2023)). Additionally, we explore the impact of injecting domain knowledge by fine-tuning LLMs on specific tasks. Furthermore, we conduct case studies in the context of virus analysis to showcase the performance of the LLMs in understanding binary code in real-world scenarios.

Our findings demonstrate that LLMs exhibit excellent potential in advancing automated binary code understanding. We call for more research in this area to further enhance the capabilities of LLMs to play a more critical role in the complex task of binary code analysis.

**Contributions.** In summary, the primary contributions of our work are as follows:

- We design an automated method to construct a benchmark dataset to evaluate the capabilities of binary code understanding and release it to facilitate further research <sup>1</sup>.
- We conduct a thorough empirical study that evaluates the capabilities of eight code domain LLMs, eight general domain LLMs, and four DL-based methods on binary code understanding. Our primary focus lies on two fundamental tasks: function name recovery and binary code summarization.

---

<sup>1</sup> <https://github.com/Sxxxw/BinaryLLMs-Eval>

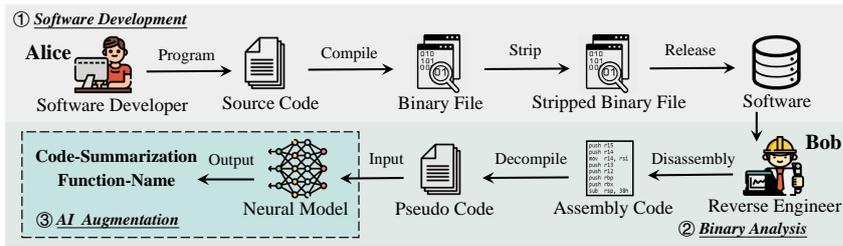


Fig. 1: Application background of binary code understanding.

- Our findings provide valuable insights into the capabilities and limitations of LLMs for understanding binary code. We thoroughly discuss the outcomes of our evaluations and offer suggestions for future research directions, aiming to propel advancements in this domain.

**Extended Version.** This paper is an extended version of our work published in the *40th International Conference on Software Maintenance and Evolution* (Shang et al., 2024). Specifically, we extend the previous work in the following aspects:

- The original study only targeted the x64 architecture and used the default optimization level of each project, without exploring the impact of different target architectures and optimization levels on binary code understanding. In extending this paper, we first substantially extend the evaluation analysis by assessing the effectiveness of LLMs in understanding binary code across four target architectures (x86, x64, ARM, MIPS) and four optimization options (O0, O1, O2, O3), respectively. (Corresponding to Section 4.1 and 4.2)
- We expand the scope of the experiment by incorporating additional LLMs for in-depth analysis, when investigating the key factors affecting the performance of LLMs in binary code understanding, as well as the impact of fine-tuning on performance. (Corresponding to Section 4.3 and 4.4)
- We conduct additional case studies to further demonstrate the practical role of LLMs in assisting reverse engineers in understanding binary code in real-world scenarios. (Corresponding to Section 4.5)
- We extend the scope of our discussion, particularly focusing on future work and limitations. (Corresponding to Section 5)
- We have also updated the related work with additional studies published in this research area, providing more detailed explanations of evaluation design, experimental metrics, etc.

**Structure of the Paper.** The rest of this paper is organized as follows: Section 2 summarizes the research background and related work, and explains our motivation. Detailed evaluation design and results analysis are presented in Section 3 and Section 4, respectively. Subsequently, the discussion is thoroughly studied from multiple aspects in Section 5. Finally, Section 6 concludes this paper.

## 2 Background and Motivation

### 2.1 Binary Code Understanding

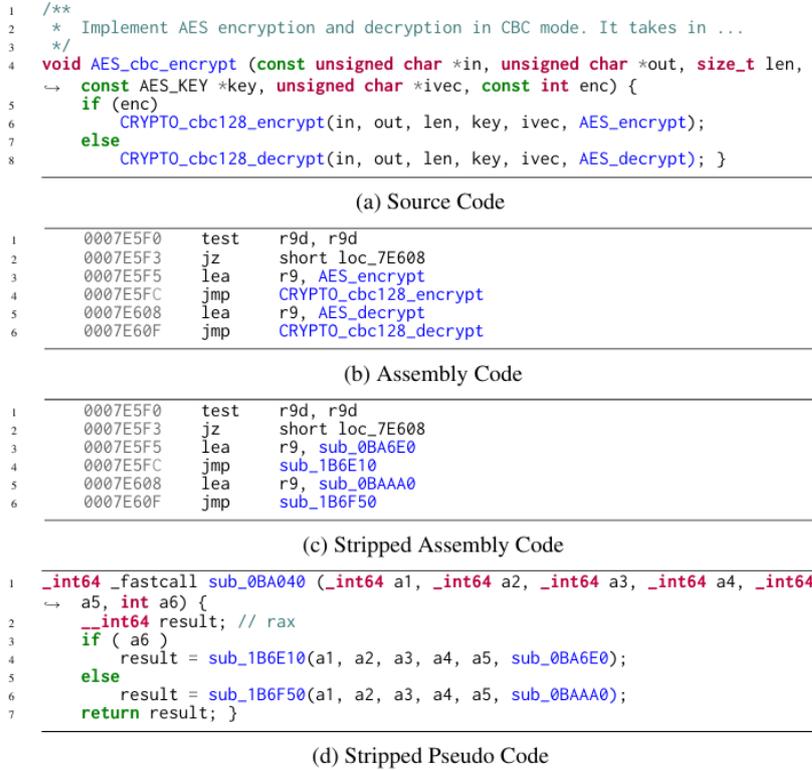


Fig. 2: Example of source, assembly, stripped assembly and stripped pseudo code snippet.

As shown in Figure 1, consider a scenario where a software developer, Alice, who writes a program in a high-level language like C/C++. Her code, written in a human-readable format (Figure 2a), must be translated into a form the computer can execute. This is where the compilation process comes into play, turning Alice’s source code into binary code, which is a series of 1s and 0s that the machine can understand and execute.

Once compiled, Bob, a reverse engineer, wants to understand how Alice’s program works. He uses a disassembler to convert the binary code back into an assembly code sequence (Figure 2b), which is more readable than binary but still quite low-level and contains a large number of machine instructions. By doing this, Bob can get some basic structure of the program. However, Alice has used the “strip” command to remove symbol information from the original binary code in order to reduce file size and protect intellectual property. This makes Bob’s job more difficult because he is

now missing critical information such as function names (replaced with meaningless placeholders such as `sub_0BA6E0`) and code comments (Figure 2c).

Finally, Bob uses a decompiler in an attempt to reconstruct the high-level logic of the program. The decompiler generates pseudo code (Figure 2d), an approximation of what Alice’s original source code might have looked like. However, due to the complexity of the decompilation process and the lack of symbolic information in the stripped binary, the pseudo code may not exactly match Alice’s original code, making it still difficult for Bob to understand the function and behavior of the program.

At this point, Bob attempts to use advanced natural language processing (NLP) techniques, such as LLMs or deep learning models, which are adept at identifying patterns and inferring logical structures. Bob leverages these techniques to **predict function names** and **generate natural language summaries** of the code’s functionality. This process can be formalized as:

$$\mathcal{N}, \mathcal{S} = f(\mathcal{F}, \mathcal{B}) \quad (1)$$

where  $\mathcal{F}$  is a stripped decompiled function in pseudo code form in binary file  $\mathcal{B}$ , which is fed into LLMs denoted as  $f$ . The objective is to generate a meaningful function name denoted as  $\mathcal{N}$ , and a natural language description denoted as  $\mathcal{S}$  of this function.

Through this process, Bob combined the analytical power of AI with his reverse engineering skills to bridge the gaps left by the stripped binaries and gain a deeper understanding of Alice’s original programming intent. Specifically, by recovering the function names and summarizing the code functions, Bob was able to quickly infer the role of each function, and then understand the design logic of the entire program in a relatively short period of time, which greatly improved the efficiency of reverse analysis.

## 2.2 Related Works

### 2.2.1 Function Name Recovery

The task of binary function name recovery aims to reassign descriptive function names to functions in binary files that have been stripped of symbolic information. During compilation and release, debugging symbols such as function names, variable names, and type information are often stripped to minimize file size, enhance security, and obscure implementation details. However, their absence complicates reverse engineering, security analysis, malware detection, and vulnerability discovery. Function name recovery helps researchers quickly extract critical function semantics and enhance the understanding of binary program behavior.

The research community has extensively explored the task of function name recovery. Initially, signature matching techniques (Zaremski and Wing, 1995) were applied to restore library function names. However, their adaptability to broader contexts posed challenges. As a result, probabilistic prediction approaches gained traction. A notable example is Debin (He et al., 2018), which leverages a conditional random field (CRF) model to infer debugging information.

In recent years, the rapid progress of artificial intelligence technology has led to the widespread adoption of neural network-based methods for function name recovery. Among them, NERO (David et al., 2020a) uses augmented control flow graphs, combined with the neural model of the encoder-decoder paradigm, to effectively capture the behavioral characteristics of functions and provide a new method for recovering binary function names. NFRE (Gao et al., 2021) proposes a lightweight framework for function name recovery that utilizes the sequential and structural information of assembly instructions. The efficiency and scalability of the framework provide the possibility to process large-scale binary files. Based on NFRE, SymLM (Jin et al., 2022) further considers the calling context to help the model understand function semantics, and leverages advanced pre-training models (Pei et al., 2020) for instruction embedding. XFL (Patrick-Evans et al., 2023) leverages feature engineering to extract constants and control flow, employs an aggregation strategy to integrate global and contextual embeddings. And utilizes PfastreXML (Jain et al., 2016) with binary function embeddings for efficient multi-label classification, addressing sparsity and class imbalance in function name labeling. NER (Chen et al., 2023b) starts from the perspective of binary code representation and studies the effectiveness of different representations for function name recovery using deep neural models, providing new perspectives and tools for this field. Finally, llasm (Sha et al., 2024) pioneers an encoder-decoder LLM fusion architecture for binary function name recovery, integrating a pre-trained assembly encoder with a natural language decoder. This approach leverages LLM reasoning to enhance semantic understanding, broaden function name prediction, and enable deeper binary code interpretation. These studies refine binary function name recovery through data representation, optimization effects, and NLP integration, advancing reverse engineering, malware analysis, and program comprehension.

### 2.2.2 *Binary Code Summarization*

Binary Code summarization aims to automatically describe the functionality of binary functions in natural language, assisting reverse engineers in analyzing binary files without source code. Due to the lack of high-level semantic information (such as function names and comments), decompiled pseudo-C code remains difficult to interpret. This task improves analysis efficiency through automatic summarization, facilitating malware detection and vulnerability discovery while aiding engineers in comprehending binary code behavior.

Recently, several approaches have emerged, each contributing unique solutions to the challenges of understanding and summarizing binary code. In these approaches, BinT5 (Al-Kaswan et al., 2023) is the first model focused on binary code summarization, which extends the application scope of source code pre-trained language models. This model treats the decompiled code as a special programming language, uses fine-tuned CodeT5 (Wang et al., 2021) to capture its semantics and generate a summary. The introduction of BinT5 opens up new avenues for binary code summarization research. HexT5 (Xiong et al., 2023) proposes a unified pre-training model also based on CodeT5, which allows multi-task learning, supports function name recovery, binary code summarization, and other downstream tasks, and demonstrates

promising performance. CP-BCS (Ye et al., 2023) proposes a framework based on control flow graphs and pseudo code for generating binary function summaries. This approach effectively captures the execution behavior of assembly code by combining bidirectional instruction-level control flow graphs and pseudo code, overcoming the challenges posed by the low-level representation of assembly code. Bin2Summary (Song et al., 2024) enhances the semantic understanding of binary code fragments through function-specific code embedding techniques and utilizes an attention-based seq-to-seq model to generate natural language summaries from the embedded binary code. Lastly, MALSIGHT (Lu et al., 2024) enhances binary code summarization by integrating reverse function extraction, recursive summarization, and static/dynamic annotations, capturing function call context to better handle malware’s complex interactions. It fine-tunes an LLM on malware source code and benign pseudo code and introduces BLEURT-sum to improve summary accuracy and readability.

Additionally, multi-intent code summarization has become a research focus, aiming to generate customized summaries tailored to different developer needs and intents. MiSum (Zhu et al., 2025) introduces a multi-intent code summarization framework based on a multimodal heterogeneous code graph (MM-HCG), integrating assembly code (CFG) and pseudo code (AST) for multi-level code understanding. Utilizing an intent-aware attention mechanism, MiSum generates customized summaries tailored to different code analysis needs, enhancing both the flexibility of binary code summarization and its effectiveness in reverse engineering and cross-layer code analysis.

### 2.3 Large Language Models and Our Motivation

In recent years, Large Language Models (LLMs) have garnered widespread attention from both academia and industry due to their remarkable capabilities. Typically composed of billions or even trillions of parameters, LLMs are trained on vast amounts of data to learn the relationships between programming languages and natural language. Notable examples include GPT-3 (Brown et al., 2020) and LLaMA (Touvron et al., 2023a), all of which have demonstrated outstanding performance across various Natural Language Processing (NLP) tasks. Amidst this surge in research interest, LLMs specifically designed for programming languages have rapidly emerged. These include Codex (Chen et al., 2021), GPT-NeoX (Black et al., 2022), CodeT5+ (Wang et al., 2023), PolyCoder (Xu et al., 2022), WizardCoder (Luo et al., 2023), and CodeLlama (Roziere et al., 2023), among others. These models have exhibited exceptional proficiency in code comprehension, further expanding the potential applications of LLMs in software development and analysis. Recently, A few studies (Wu et al., 2023; Zhang et al., 2023; Hou et al., 2024) find that LLMs have demonstrated excellent capabilities in dealing with natural language tasks, as well as source code understanding, indicating that they have the potential to be applied to complex analysis of source code.

Traditional binary reverse engineering and decompilation tools, such as Ghidra (NationalSecurityAgency, 2024) and IDA Pro (Hex-RaysSA, 2024), play a crucial role in converting binary code into high-level languages. However, they exhibit significant limitations in terms of readability and comprehension of complex code structures. The lack of high-level semantic information and debugging symbols makes decompilation

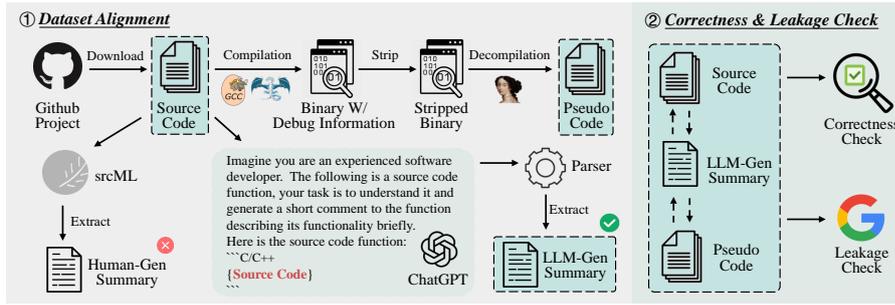


Fig. 3: An overview of the benchmark dataset construction process.

a labor-intensive process, heavily reliant on expert knowledge and domain-specific expertise. Nevertheless, binary code shares inherent similarities with source code and natural language, as they all follow specific patterns and structures that can be learned and leveraged by LLMs (Zhang, 2022).

Therefore, this study will explore the potential of LLMs in understanding binary code, aiming to evaluate whether these models can cross domain boundaries and extend their capabilities in natural language and source code to binary code analysis. This is expected not only to provide new perspectives for automated code understanding, but also to open up new application paths in areas such as reverse engineering and malware analysis.

### 3 Evaluation Design

#### 3.1 Dataset Construction

Before we can effectively evaluate the ability of LLMs to understand binary code, a comprehensive benchmark dataset is necessary to provide a consistent basis for different model evaluations and comparisons. The specific process of constructing the benchmark dataset is shown in Figure 3.

##### 3.1.1 Source Code Selection

To reflect real-world reverse engineering needs, we believe that code sources of the benchmark should meet the following requirements:

- *Authenticity*: the code should come from real projects, not toy programs or incomplete code snippets.
- *Breadth*: the selected code should cover multiple fields and not be limited to specific fields or application scenarios.
- *High quality*: the selected code should be of good quality, including clear structure, reasonable naming conventions, etc.
- *Credibility*: the selected code should ideally be sourced from projects maintained by well-known or reputable developers or organizations to accurately reflect real-world application scenarios.

Table 1: Statistics of our benchmark dataset.

Project	Domain	# Binaries	# Aligned Functions							# Select	
			x64_O0	x64_O1	x64_O2	x64_O3	x86_O0	ARM_O0	MIPS_O0		
FFmpeg (FFmpeg, 2024)	Video	14	44,157	26,215	22,722	21,295	45,443	42,979	44,518	250	
Redis (Redis, 2024)	Database	14	9,688	8,827	7,736	7,072	9,716	5,713	5,486	200	
Curl (Curl, 2024)	Network	14	1,329	995	796	706	1,364	1,329	1,329	200	
Masscan (Masscan, 2024)	Network	7	946	674	524	456	946	946	946	150	
Llama2.c (Llama2.c, 2024)	Neural Network	14	67	67	45	35	67	67	67	35	
Whisper.cpp (Whisper.cpp, 2024)	Neural Network	63	18,907	4,537	3,755	3,505	4,963	4,932	4,947	200	
OpenSSL (OpenSSL, 2024)	Crypto	14	4,859	4,193	3,509	3,366	4,764	4,776	4,778	250	
zstd (zstd, 2024)	Compress	7	2,302	956	694	557	2,284	2,279	2,279	200	
ImageMagick (ImageMagick, 2024)	Image	21	7,112	4,207	3,737	3,631	6,560	6,560	6,560	200	
Libvips (Libvips, 2024)	Image	7	3,721	3,147	2,860	2,747	4,233	3,724	3,723	208	
Libxpat (Libxpat, 2024)	Format	7	260	234	155	134	260	260	260	100	
Ultrajson (Ultrajson, 2024)	Format	14	27	11	8	7	26	26	26	7	
<b>Total (12)</b>	<b>8</b>	<b>196</b>	<b>93,348</b>	<b>54,052</b>	<b>46,533</b>	<b>43,504</b>	<b>80,600</b>	<b>73,565</b>	<b>74,893</b>	<b>2,000</b>	

Therefore, as shown in Table 1, we select 12 real-world projects implemented in C language with the highest star ratings on Github, including FFmpeg, Redis, Curl, Masscan, Llama2.c, Whisper.cpp, OpenSSL, zstd, ImageMagick, Libvips, Libexpat, and Ultrajson, which have high credibility, excellent code quality and maintenance standards, covering eight application domains, including crypto, compress, network, video, image, database, neural network, etc.

### 3.1.2 *Compile, Strip and Decompile*

We cross-compile these projects in different compilation environments to obtain binary files. Specifically, the target architectures include x64, x86, ARM, and MIPS, where the x64 architecture uses four optimization levels (O0 to O3) while the other architectures only use the O0 optimization level. As illustrated in Table 1, we generate a total of 196 binaries. Subsequently, we employ the `strip` command in Linux to remove the symbol tables from these binaries to simulate binaries without symbol information that are common in actual reverse engineering.

Previous research (Chen et al., 2023b) has found that using pseudo code as a representation of binary code is more effective for neural models than assembly instruction sequence and Intermediate Representation (IR), as it provides a higher-level code representation that facilitates model understanding. Therefore, we directly utilize IDA Pro (Hex-RaysSA, 2024) to decompile the binary files and convert the binary code into pseudo code form without considering other representation forms.

### 3.1.3 *Alignment*

We use DWARF (International, 2010) debugging information to align source code and pseudo code, which can record functions, variables in binary functions, and their locations (include source file name, line number, and column number) in source code. As shown in Table 1, we obtain a total of 466,495 functions matching source code and pseudo code. To align source code and human-written summary, we use srcML (Maletic and Collard, 2015) to analyze and parse the source files, then collect single- and multi-line summaries above the location of function declarations and definitions. Through the above steps, the alignment of source code – pseudo code – human summary is finally achieved.

### 3.1.4 *Ground-truth Identification*

For the function name recovery task, we parse the function names in the source code as labels using regular expressions. For the binary code summarization task, we first consider using human-written comments extracted from source code files as labels as in previous work (Al-Kaswan et al., 2023; Xiong et al., 2023). However, we found that only about 14.8% of functions have comments written by human developers. Worse yet, not all comments are describing the functional summary of the function, but will also contain some noisy content, and they are of varying quality and style. Therefore, using human-written comments as ground-truth is unreliable.

Presently, an increasing number of research works (Dagdelen et al., 2024; Bzdok et al., 2024; Tan et al., 2024) employ large language models such as ChatGPT (Ouyang

Table 2: Detail information of Large Language Models we apply in this work. (In the License column, "✓" indicates Open-source, "×" indicates Closed-source.)

Domain	Model	Release Time	Size	Base Model	Training Corpus			Publisher	License
					Raw Size	#Tokens	#Instances		
Code LLMs	CodeGen25-7b-instruct (Nijkamp et al., 2023)	Jul-2023	7B	CodeGen2	-	1.4T	-	Salesforce	✓
	WizardCoder-15b-V1.0 (Luo et al., 2023)	Jun-2023	15B	StarCoder	-	-	78.0K	WizardLM	✓
	WizardCoder-33b-V1.1 (Luo et al., 2023)	Jan-2024	33B	Deepseek-Coder	-	-	-	WizardLM	✓
	Code Llama-7b-instruct-hf (Roziere et al., 2023)	Jun-2023	7B	Llama-2-7b	4.4TB	525.0B	-	Meta AI	✓
	Code Llama-13b-instruct-hf (Roziere et al., 2023)	Jun-2023	13B	Llama-2-13b	4.4TB	525.0B	-	Meta AI	✓
	Code Llama-34b-instruct-hf (Roziere et al., 2023)	Jun-2023	34B	Llama-2-34b	4.4TB	525.0B	-	Meta AI	✓
	Code Llama-70b-instruct-hf (Roziere et al., 2023)	Jan-2024	70B	Llama-2-70b	-	1.0T	-	Meta AI	✓
	DeepSeek-Coder-33b-instruct (Guo et al., 2024)	Nov-2023	33B	-	-	2.0T	-	DeepSeek-AI	✓
	ChatGLM2-6B (Zeng et al., 2022)	Jun-2023	6B	-	-	1.4T	-	THUDM	✓
	Vicuna-7b-v1.5 (Zheng et al., 2023)	Aug-2023	7B	Llama-2-7b	-	-	125.0K	L.Zheng et al.	✓
General LLMs	Vicuna-13b-v1.5 (Zheng et al., 2023)	Aug-2023	13B	Llama-2-13b	-	-	125.0K	L.Zheng et al.	✓
	Llama-2-13b-chat-hf (Touvron et al., 2023b)	Jul-2023	13B	-	-	2.0T	-	Meta AI	✓
	Llama-2-70b-chat-hf (Touvron et al., 2023b)	Jul-2023	70B	-	-	2.0T	-	Meta AI	✓
	Mistral-7B-Instruct-v0.2 (Jiang et al., 2024)	Dec-2023	7B	Mistral-7B	-	-	-	Mistral AI	✓
	Mixtral-8x7B-Instruct-v0.1 (Jiang et al., 2024)	Dec-2023	47B	Mistral-7B	-	-	-	Mistral AI	✓
	ChatGPT (Ouyang et al., 2022)	Nov-2022	-	-	-	-	-	OpenAI	×

et al., 2022) for tasks like data annotation, and has demonstrated a certain degree of reliability. Inspired by these pioneering works, we utilize ChatGPT to generate summaries as ground-truth. Specifically, we use the source code of the function to construct the prompt shown in Figure 3, prompting ChatGPT to generate a short summary describing the function’s purpose and functionality.

### 3.1.5 Correctness & Leakage Check

It is crucial to ensure the correctness of the ground-truth, so we perform a correctness check on the descriptive summaries generated by ChatGPT. Specifically, we invited three experienced domain experts to review the match between the source code and the summary. Experts were asked to give each abstract a "pass" or "fail" score. If two or more experts give a "fail" rating, the data will be removed directly from the dataset; if one expert gives a "fail" rating, we will conduct a collective discussion and give a final in conclusion. Finally, as shown in Table 1, we select 2,000 functions, each of which contains seven compilation settings (i.e., x64\_00, x64\_01, x64\_02, x64\_03, x86\_00, ARM\_00, MIPS\_00), totaling 14,000 pieces of data as the benchmark dataset.

It is also imperative that benchmark datasets are not included in the training set of LLMs to mitigate the risk of data leakage. All our evaluation data are decompiled pseudo code, and the symbolic information is stripped away so that it is significantly different from the corresponding source code form, which greatly avoids data leakage. To further ensure the validity and reliability of our benchmark evaluation, we use the Google search engine to check whether the code appears on the Internet in clear text. The results show that none of the pseudo codes are retrieved by whole-word matching.

## 3.2 Large Language Models Setup

### 3.2.1 Large Language Models As Is

We select eight code domain LLMs, i.e., CodeGen25 (Nijkamp et al., 2023), DeepSeek-Coder (Guo et al., 2024), two versions of WizardCoder (Luo et al., 2023), four versions

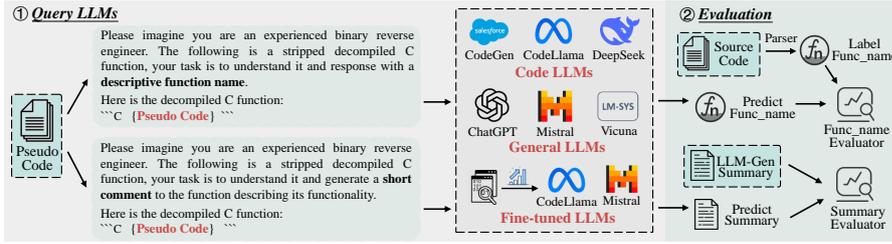


Fig. 4: An overview of the evaluation process.

of CodeLlama (Roziere et al., 2023), and select eight general domain LLMs, i.e., ChatGLM (Zeng et al., 2022), two versions of Vicuna (Zheng et al., 2023), two versions of Llama (Touvron et al., 2023b), two versions of Mistral (Jiang et al., 2024), and ChatGPT (Ouyang et al., 2022). The principles for our selection are: (1) state-of-the-art, (2) pre-trained on enough source code to be able to understand code to a certain extent, and (3) have text generation and code generation capabilities. In addition, in order to ensure that the model can follow the instructions, we all choose the instruct-tuned version. Table 2 provides detailed information, including parameter size, base model, training corpus, publisher, etc.

Limited by the context window length, we set the maximum input tokens to 4,096, and code snippets exceeding the length will be truncated. For the function name recovery task, we set the maximum new tokens to 48, and for the code summarization task, we set it to 256. We set the sampling temperature to 1,  $top_p$  to 0.95,  $top_k$  to 10, and  $num\_beams$  to 1. For all open-source models, we downloaded them from HuggingFace (HuggingFace, 2024) and deployed on our local machine with FP16 mixed precision enabled during inference. For ChatGPT, we called its latest gpt-3.5-turbo-16k version through the OpenAI’s API.

### 3.2.2 Prompt Formats

Figure 4 illustrates the prompt format we used for LLMs. We use role-play (Chen et al., 2023a; Kong et al., 2023) prompts to give LLMs the role of experienced binary reverse engineers, enabling them to better handle binary code understanding tasks. We enclose the code in the prompt with triple backticks (```) to clearly describe the code format. Considering the limitation of the length of the model context window, and in order to reduce the inference time overhead and memory usage, we adopt the zero-shot prompts. We also analyze the impact of few-shot prompts on the performance of LLMs in Section 4.3.

### 3.2.3 Fine-tuned Large Language Models

We also investigate the ability of fine-tuned LLMs to understand binary code, since fine-tuning is a common technique to adapt a pre-trained LLM to downstream tasks (Wu et al., 2023; Feng et al., 2020; Fried et al., 2022), such as function name recovery and code summary generation. Furthermore, pre-training corpora of existing LLMs

contain very few binary code, either in the form of disassembled instruction sequences or decompiled pseudo code. Therefore, we hope to explore whether injecting binary domain information into LLMs can improve its performance.

The GNU repository<sup>2</sup> is extensively used as a training or test set for many existing deep learning-based works (David et al., 2020a; Jin et al., 2022; Chen et al., 2023b; Xiong et al., 2023; Wang et al., 2022; Li et al., 2021). To build our fine-tuning dataset, we select 51 projects from the GNU repository, including binutils, coreutils, findutils, libmicrohttpd, nettle, etc. We use BinKit (Kim et al., 2022) to create a compilation environment, and then compile the selected projects using the gcc-11.2.0 compiler with x86\_64 target architecture and O0 optimization level. We obtain a total of 270 binary files. After strip, decompile and alignment, we obtain 124,819 functions matching source code and decompiled pseudo code, and randomly select 30,000 of them as the fine-tuning dataset.

Additionally, we perform a search and confirm that none of the functions in our proposed benchmark is present in the fine-tuning dataset.

### 3.3 Evaluation Setup

The evaluation environment is a machine equipped with 8 \* NVIDIA RTX A6000 GPU with 48GB of VRAM, 2 \* 28-core Intel Xeon 6330 CPU, 512GB RAM and 64TB storage, running on Ubuntu 22.04 OS. The GPU is running Nvidia driver version 525.116.03 with CUDA version 12.0.

We implement all the experiments using Python 3.8 with PyTorch (PyTorch, 2024) 2.0.1, DeepSpeed (DeepSpeed, 2024) 0.13.0 and Transformers (Transformers, 2024) 4.37.2 packages. As for model fine-tuning, we implement it based on the LLaMA-Factory (Zheng et al., 2024) framework. We use LoRA (Hu et al., 2022) fine-tuning method and specify all available modules. We adopt Adam optimizer in fp16 precision, 40 global batch size and 1 training epoch. The learning rate is set to 1e-5 and followed by a cosine decay.

## 4 Evaluation Results and Findings

In this section, we conduct extensive experiments to answer the following research questions:

- **RQ1:** How do LLMs perform in the task of function name recovery?
- **RQ2:** How do LLMs perform in the task of binary code summarization?
- **RQ3:** What factors significantly influence the performance of LLMs to understand binary code?
- **RQ4:** Can fine-tuning enhance the capability of LLMs to understand binary code?
- **RQ5:** Do LLMs have the practical ability in real-world scenarios?

---

<sup>2</sup> <http://ftp.gnu.org/gnu>

#### 4.1 RQ1: Performance of Function Name Recovery

**Metrics.** For the function name recovery task, following (Gao et al., 2021; Chen et al., 2023b), we calculate token-level Precision, Recall, F1-score to evaluate the performance of LLMs. The metrics ignore non-alphabetical characters and are case-, order-, and duplication-insensitive at the token-level. They can be expressed as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (3)$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

Specifically, function names consist of one or more discrete tokens, each encapsulating a portion of the function’s semantic information. During the evaluation, we employ a combination of empirical rules and a unigram language model (ULM)(Kudo, 2018) to segment function names into token sequences. For function names that adhere to standard naming conventions, such as camel case (`getUserId`) and snake case (`get_user_id`), we design specific rules for tokenization. However, for function names that do not follow explicit naming conventions, such as `getuserid`, we utilize a pretrained SentencePiece model in conjunction with ULM to segment names based on token frequency statistics. For instance, `setcmdfmt` is tokenized into `set`, `cmd`, and `fmt`.

For an intuitive understanding of the metrics, a ground truth of `getUserMessage` with a prediction of `get_message` is given full precision and 67% recall. And a prediction of `get_user_message_set_user_message` has 83% precision and full recall.

**Results.** Table 3 and Table 4 respectively show the performance of LLMs and DL-based methods in function name recovery tasks under different target architectures and compiler optimization options.

From Table 3, we can observe that CodeLlama and Llama consistently perform excellently across different parameter sizes under various architectures. Among them, CodeLlama-34b outperforms all other LLMs in precision, recall, and F1-score metrics, achieving scores of 28.73, 27.33, and 26.75 in the x64 architecture, 28.06, 26.82, and 26.09 in the x86 architecture, 29.14, 27.83, and 27.16 in the ARM architecture, and 32.87, 31.22, and 30.66 in the MIPS architecture. Following closely behind, WizardCoder-15b and DeepSeek-Coder-33b also show strong performance, with F1-score of 21.76, 22.45, 22.64, 28.09 and 23.02, 21.83, 22.84, 27.05 across the four architectures, respectively. CodeGen25-7b and ChatGLM2-6B show the poorest performance, with F1-score ranging from only 48.9% to 66.7% of the other LLMs. This lagging performance may be attributed to the capability flaws of their basic models or the lack of targeted training data. In the general LLMs category, Llama-2-13b consistently outperforms other LLMs in both precision and F1-score metrics, achieving scores of 23.88 and 22.68 in the x64 architecture, 23.90 and 22.54 in the x86 architecture, 23.98 and 22.51 in the ARM architecture, and 28.17 and 26.88 in the MIPS architecture, respectively.

Table 3: Comparison of LLMs and DL-based methods on function name recovery under different target architectures. We mark the best performing methods in each domain.

Domain	Model	x64			x86			ARM			MIPS						
		Precision	Recall	Time(s)													
Code	CodeGen25-7b-instruct	9.98	12.62	10.06	1.63	9.68	12.46	9.89	1.47	10.07	12.77	10.13	1.47	14.01	16.25	13.89	1.48
	WizardCoder-15b-V1.0	23.29	22.11	21.76	1.57	23.66	23.03	22.45	1.55	23.93	23.32	22.64	1.53	29.58	28.80	28.09	1.51
	WizardCoder-33b-V1.1	20.57	21.13	19.49	5.01	20.79	20.85	19.70	4.79	20.54	20.09	19.18	4.85	25.04	24.71	23.47	4.81
LLMs	Code Llama-7b-instruct-hf	27.52	25.06	25.23	1.19	26.73	24.76	24.81	1.40	26.14	25.18	24.58	1.38	31.27	29.77	29.36	1.35
	Code Llama-13b-instruct-hf	26.64	24.83	24.68	1.72	25.44	23.84	23.63	1.68	26.41	24.84	24.67	1.69	31.35	30.04	29.57	1.68
	Code Llama-34b-instruct-hf	28.73	27.33	26.75	3.42	28.06	26.82	26.09	3.43	29.14	27.83	27.16	3.33	32.87	31.22	30.66	3.40
DeepSeek-Coder-33b-instruct	Code Llama-70b-instruct-hf	26.57	25.52	24.90	10.05	26.88	25.75	25.00	10.93	26.79	25.78	25.16	9.83	30.03	29.61	28.55	9.94
	DeepSeek-Coder-33b-instruct	23.56	24.58	23.02	3.99	22.59	22.99	21.83	4.01	23.74	24.09	22.84	3.83	27.80	28.68	27.05	3.91
	ChatGLM2-6B	13.88	12.57	12.66	1.08	15.26	13.48	13.74	1.10	13.96	12.80	12.81	1.02	17.63	16.18	16.15	1.01
General LLMs	Vicuna-7b-v1.5	18.28	18.17	17.32	1.13	17.81	16.97	16.68	1.06	17.71	17.57	16.86	1.06	23.03	23.09	22.07	1.01
	Vicuna-13b-v1.5	21.24	21.54	20.46	1.03	20.56	21.13	19.96	1.03	20.94	21.81	20.37	1.04	24.22	25.19	23.57	1.05
	Llama-2-13b-chat-hf	23.88	23.28	22.68	1.18	23.90	23.03	22.54	1.39	23.98	23.03	22.51	1.25	28.17	27.77	26.88	1.17
Mixtral-8x7B-Instruct-v0.1	Llama-2-70b-chat-hf	23.51	21.56	21.72	4.46	22.88	21.23	21.12	4.26	22.68	21.29	21.16	4.27	26.14	24.61	24.43	4.35
	Mixtral-7B-Instruct-v0.2	21.92	23.82	21.81	1.20	20.87	22.61	20.72	1.26	21.62	23.22	21.34	1.20	25.57	27.43	25.22	1.32
	Mixtral-8x7B-Instruct-v0.1	19.34	25.65	21.00	2.65	18.97	25.55	20.72	2.65	19.58	26.12	21.42	2.73	21.50	29.10	23.58	2.69
ChatGPT(gpt-3.5-turbo-16k)	19.40	18.41	18.08	-	18.23	20.81	18.66	-	18.66	21.10	19.03	-	20.90	23.71	21.39	-	
DL-based NER	SymLM (Jin et al., 2022)	9.63	5.54	6.66	0.28	9.55	5.39	6.59	0.29	9.61	5.50	6.68	0.28	10.14	5.76	6.78	0.27
	HexT5 (Xiong et al., 2023)	16.22	9.83	11.17	0.03	16.31	10.03	11.40	0.03	15.99	9.49	10.87	0.03	16.98	10.36	11.73	0.03
	HexT5 (Xiong et al., 2023)	3.39	2.97	3.00	0.17	3.44	3.10	3.10	0.16	3.33	3.01	2.95	0.16	3.59	3.24	3.31	0.18

A deeper analysis of the performance across different target architectures reveals that the average F1-score for all LLMs on the MIPS architecture is 24.62, while on x64, x86, ARM architectures, the average F1-score are 20.73, 20.47, and 20.74, respectively. Compared to other architectures, the F1-score for MIPS shows improvements of 18.77%, 20.27%, and 18.71%. The superior performance of MIPS in function name recovery tasks can likely be attributed to its simplified instruction set and unified function calling conventions. This streamlined instruction set reduces reliance on complex operations and diverse instruction variants, enabling the LLM to more effectively capture function call patterns and contextual information.

Furthermore, as a task that combines natural language and code language understanding, code domain LLMs generally perform slightly better than general domain LLMs. This may be because their pre-training datasets have a higher proportion of source code. Training on an extensive range of source code datasets allows them to gain a deeper grasp of programming syntax, structure and semantics.

**Findings 1:** CodeLlama-34b performs the best in the function name recovery task across different target architectures, achieving an F1-score of 26.75, 26.35, 26.75, and 26.59 in the x64, x86, ARM, and MIPS architectures, respectively. Additionally, LLMs perform the best in function name recovery on the MIPS architecture, with the average F1-score across all LLMs being 24.62. Code domain LLMs generally perform slightly better than general domain LLMs, likely owing to their greater familiarity with programming paradigms.

From Table 4, it is evident that CodeLlama and Llama also continue to perform excellently under different compiler optimization options. Among them, CodeLlama-34b outperforms all other LLMs in precision, recall, and F1-score metrics, achieving scores of 28.73, 27.33, and 26.75 under the -O0 optimization, 27.98, 27.50, and 26.35 under the -O1 optimization, 28.89, 27.09, and 26.75 under the -O2 optimization, and 28.40, 27.31, and 26.59 under the -O3 optimization.

In the general LLM category, Llama-2-13b outperforms other general LLMs in both precision and F1-score, achieving the following scores across the four optimization options: 23.88 and 22.68 for -O0, 24.06 and 22.71 for -O1, 24.44 and 23.00 for -O2, and 24.42 and 23.04 for -O3, respectively.

For the binaries with -O0, -O1, -O2, and -O3 optimization levels, the average F1-scores for all LLMs are 20.73, 20.62, 20.53, and 20.81, respectively. We observe that the performance differences in function name recovery tasks across different optimization levels are minimal, with the performance gap between O0 and O3 being only 0.39%.

**Findings 2:** CodeLlama-34b continues to perform the best in the function name recovery task under different compiler optimization options, achieving an F1-score of 26.75, 26.35, 26.75, and 26.59 under the -O0, -O1, -O2, and -O3 optimization levels, respectively. Additionally, all LLMs exhibit a minimal performance gap across different optimization levels, with the average F1-score difference between O0 and O3 being only 0.39%.

Table 4: Comparison of LLMs and DL-based methods on function name recovery under different compiler optimization options. We mark the best performing methods in each domain.

Domain	Model	O0			O1			O2			O3						
		Precision	Recall	Time(s)													
Code LLMs	CodeGen25-7b-instruct	9.98	12.62	10.06	1.63	9.83	12.25	9.85	1.44	10.02	12.78	10.36	1.54	10.23	12.89	10.77	1.56
	WizardCoder-15b-V1.0	23.29	22.11	21.76	1.57	24.24	23.45	22.86	1.67	24.44	23.82	23.13	1.58	24.58	23.82	23.16	1.62
	WizardCoder-33b-V1.1	20.57	21.13	19.49	5.01	20.20	20.90	19.23	5.09	20.60	21.52	19.63	5.22	20.76	21.18	19.74	5.30
Code LLMs	Code Llama-7b-instruct-hf	27.52	25.06	25.23	1.19	26.28	24.75	24.55	1.28	25.43	24.06	23.81	1.25	26.64	24.82	24.74	1.57
	Code Llama-13b-instruct-hf	26.64	24.83	24.68	1.72	26.59	24.47	24.45	1.92	26.56	24.50	24.53	1.81	27.33	24.99	25.10	1.88
	Code Llama-34b-instruct-hf	28.73	27.33	26.75	3.42	27.98	27.50	26.35	3.59	28.89	27.09	26.75	3.57	28.40	27.31	26.59	3.77
DeepSeek-Coder-33b-instruct	Code Llama-70b-instruct-hf	26.57	25.52	24.90	10.05	26.45	25.93	25.10	9.97	26.59	25.57	24.90	10.10	26.41	25.47	24.80	10.35
	DeepSeek-Coder-33b-instruct	23.56	24.58	23.02	3.99	23.69	24.92	23.21	4.11	23.42	24.39	22.87	4.21	23.76	24.95	23.25	4.22
	ChatGLM2-6B	13.88	12.57	12.66	1.08	13.75	12.20	12.39	1.04	13.34	12.00	12.11	1.02	14.10	12.42	12.67	1.04
General LLMs	Vicuna-7b-v1.5	18.28	18.17	17.32	1.13	18.36	17.37	17.03	1.10	17.50	17.27	16.63	1.15	18.72	17.80	17.44	1.13
	Vicuna-13b-v1.5	21.24	21.54	20.46	1.03	20.29	20.71	19.62	1.10	20.50	20.80	19.70	1.09	20.11	20.55	19.33	1.18
	Llama-2-13b-chat-hf	23.88	23.28	22.68	1.18	24.06	23.20	22.71	1.18	24.44	23.48	23.00	1.20	24.42	23.62	23.04	1.30
DL-based NER (Chen et al., 2023b)	Llama-2-70b-instruct-v0.2	23.51	21.56	21.72	4.46	22.68	20.95	20.99	4.45	22.15	20.50	20.53	4.52	22.93	20.87	21.02	4.81
	Mistral-7B-Instruct-v0.1	21.92	23.82	21.81	1.20	21.53	23.97	21.62	1.26	21.38	23.47	21.37	1.33	21.44	23.37	21.36	1.36
	Mixtral-8x7B-Instruct-v0.1	19.34	25.65	21.00	2.65	19.09	25.76	20.82	2.69	18.73	25.33	20.57	2.75	19.52	25.86	21.15	2.79
HexT5 (Xiong et al., 2023)	ChatGPT(gpt-3.5-turbo-16k)	19.40	18.41	18.08	-	18.63	21.58	19.21	-	18.12	21.00	18.65	-	18.36	21.12	18.85	-
	SymLM (Jin et al., 2022)	9.63	5.54	6.66	0.28	9.58	5.39	6.48	0.26	9.51	5.42	6.51	0.26	9.48	5.30	6.44	0.25
	HexT5 (Xiong et al., 2023)	3.39	2.97	3.00	0.17	3.30	2.88	2.85	0.18	3.24	2.82	2.91	0.17	3.38	3.01	3.09	0.16

Among the deep learning-based expert models, NER (Chen et al., 2023b) performs the best, with an average F1-score of 11.29 and 11.13 in different architectures and optimization options, still slightly outperforming the LLM with the worst performance, CodeGen25. However, SymLM (Jin et al., 2022) and HexT5 (Xiong et al., 2023) perform poorly, with average F1-score of 3.68 and 6.52, and 3.09 and 2.96, respectively, which differ from the performance reported in their original papers. This difference may come from the partitioning of their datasets. SymLM and HexT5 widely use projects from GNU as part of their training and testing sets. SymLM divides the training and testing sets at the binary file-level, which may result in some code appearing in both the training and testing sets. For example, in the `Binutils` project, the `ar` and `nm` files share the same binary file descriptor (BFD) processing code. This reuse of libraries and underlying code may lead to exaggeration of evaluation metrics. Although HexT5 adopts a stricter project-level dataset partitioning approach, different projects under GNU may still share programming styles or naming conventions, leading to potential data leaks. Overall, deep learning-based expert models perform worse compared to LLMs, primarily due to their limited generalization ability, which makes them prone to overfitting specific features of the training data when handling out-of-distribution data. In contrast, LLMs exhibit superior adaptability in zero-shot learning tasks, allowing them to effectively handle unseen data.

**Findings 3:** Among the existing deep learning-based expert models, SymLM performs the best. However, these models exhibit poor generalization ability beyond the training data distribution, with performance significantly lower than that of LLMs.

In terms of inference time cost, locally deployed LLMs with 6B-7B parameter quantities typically require 1 to 1.6 seconds to infer a single piece of data, LLMs with 13-15B scales require 1.2 to 2 seconds, LLMs with 33-34B scales require 3.5 to 5.1 seconds, and CodeLlama-70b requires a maximum of 10.35 seconds per piece. Considering that ChatGPT requires API access, which is affected by network latency and rate limits, we do not measure its time overhead. The DL-based model, due to its lightweight advantage, greatly reduces inference time and achieves the fastest NER of 0.03 seconds per piece.

**Findings 4:** The DL-based model has a significant advantage in inference speed benefiting from their model size. Meanwhile, the inference speed of LLMs is still within an acceptable range.

#### 4.2 RQ2: Performance of Binary Code Summarization

**Metrics.** For the binary code summarization task, same as BinT5 (Al-Kaswan et al., 2023), HexT5 (Xiong et al., 2023), we use smoothed BLEU-4 (Papineni et al., 2002), METEOR (Lavie and Denkowski, 2009), Rouge-L (Lin, 2004) as the evaluation metric.

Table 5: Comparison of LLMs and DL-based methods on binary code summarization under different target architectures. We mark the best performing methods in each domain.

Domain	Model	x64				x86				ARM				MIPS			
		BLEU-4	METEOR	Rouge-L	Time(s)												
	CodeGen25-7b-instruct	3.56	20.76	14.06	7.14	3.55	20.99	14.11	7.19	3.53	20.97	14.38	7.17	3.79	21.71	14.61	7.15
	WizardCoder-15b-V1.0	4.72	24.10	18.16	8.00	4.63	24.04	18.12	7.95	4.67	24.23	18.02	8.16	5.08	25.16	19.07	7.85
	WizardCoder-33b-V1.1	4.12	23.72	16.03	20.77	4.07	23.70	15.84	21.25	4.05	23.68	16.01	20.92	4.38	24.72	16.63	20.88
Code	Code Llama-7b-instruct-hf	4.16	21.10	16.65	5.55	4.06	21.18	16.62	5.42	4.15	20.92	16.74	5.16	4.47	21.80	17.49	5.42
LLMs	Code Llama-13b-instruct-hf	4.22	19.52	16.46	7.62	4.11	19.65	16.24	7.15	4.14	19.74	16.54	7.02	4.35	20.22	16.91	6.98
	Code Llama-34b-instruct-hf	4.41	20.97	17.60	15.70	4.45	20.96	17.61	16.37	4.34	20.98	17.64	16.29	4.77	21.44	18.41	15.44
	Code Llama-70b-instruct-hf	4.26	22.37	16.60	46.33	4.07	22.26	16.37	46.73	4.19	22.36	16.63	46.38	4.45	23.13	17.30	47.21
	DeepSeek-Coder-33b-instruct	4.66	24.08	17.38	17.32	4.53	23.97	17.13	17.98	4.64	24.22	17.37	17.36	5.05	25.11	18.18	17.15
	ChatGLM2-6B	3.54	21.42	14.49	5.21	3.55	21.62	14.69	5.14	3.54	21.78	14.72	5.05	3.80	22.40	15.19	5.00
	Vicuna-7b-v1.5	4.47	23.02	17.64	4.40	4.29	22.94	17.34	4.49	4.39	23.26	17.79	4.39	4.79	24.03	18.50	4.06
	Vicuna-13b-v1.5	5.59	22.23	19.75	4.86	5.47	22.34	19.78	5.20	5.55	22.38	19.87	5.20	6.06	23.36	20.86	5.10
General	Llama-2-13b-chat-hf	4.72	22.56	18.06	7.07	4.79	22.67	18.35	7.21	4.86	22.92	18.61	6.85	5.28	24.02	19.58	6.72
LLMs	Llama-2-70b-chat-hf	4.36	23.17	16.14	45.73	4.35	23.50	16.26	44.49	4.50	23.93	16.66	42.85	4.82	24.76	17.39	42.60
	Mistral-7B-Instruct-v0.2	6.05	25.65	20.86	3.51	5.74	25.53	20.62	3.53	6.11	25.33	21.06	3.55	6.62	26.40	22.08	3.43
	Mixtral-8x7B-Instruct-v0.1	6.41	26.02	21.04	12.41	6.16	25.34	20.69	11.23	6.47	25.86	21.19	13.63	6.77	26.88	21.97	10.71
	ChatGPT (gpt-3.5-turbo-16k)	7.69	29.50	22.09	-	7.38	28.65	21.52	-	7.80	29.48	22.48	-	8.30	30.32	23.13	-
DL-based	BinT5 (Al-Kaswan et al., 2023)	0.00	2.08	4.69	0.56	0.00	2.00	4.61	0.57	0.00	2.03	4.67	0.58	0.00	2.11	4.82	0.57
	HexT5 (Xiong et al., 2023)	0.10	6.21	8.44	0.43	0.09	6.13	8.37	0.44	0.09	6.09	8.33	0.44	0.11	6.29	8.53	0.42

BLEU (Bilingual Evaluation Understudy) is the most widely used metric in code summarization tasks. The Unigrams and bigrams measure the adequacy of the candidate, while longer trigrams and 4-grams assess its fluency. Based on standard works like CodeT5 Wang et al. (2021) and CodeSearchNet Husain et al. (2019), we choose BLEU-4 as the evaluation metric. BLEU-4 calculates the cumulative precision for 4-grams, which is the ratio of matching 4-grams in the candidate sentence to the total number of 4-grams. The score is computed as follows:

$$BLEU-4 = BP \times \exp\left(\sum_{n=1}^4 w_n \log P_n\right), \quad (5)$$

where  $BP$  represents the brevity penalty for short generated sequences,  $w_1$  to  $w_n$  are positive weights summing to 1, and  $P_n$  is the ratio of subsequences of length  $n$  in the generated summary that also appear in the reference.

METEOR (Metric for Evaluation for Translation with Explicit Ordering) calculates the harmonic mean of the unigram precision and recall, which is calculated as:

$$METEOR = (1 - \gamma \cdot frag^\beta) \cdot \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R}, \quad (6)$$

where  $frag$  is the fragmentation fraction, and  $P$  and  $R$  represent unigram precision and recall, respectively. The parameters  $\alpha$ ,  $\beta$ , and  $\gamma$  are penalties.

Rouge-L is a variant of Rouge (Recall-oriented Understudy for Gisting Evaluation), which is calculated based on the longest common subsequence (LCS) between the reference and the candidate. The LCS-based F-measure ( $F_{lcs}$ ) is called Rouge-L, which is calculated as:

$$P_{lcs} = \frac{LCS(X, Y)}{n}, \quad R_{lcs} = \frac{LCS(X, Y)}{m}, \quad F_{lcs} = \frac{(1 + \beta^2)P_{lcs}R_{lcs}}{R_{lcs} + \beta^2P_{lcs}}, \quad (7)$$

where  $\beta = P_{lcs}/R_{lcs}$ , and  $n$  and  $m$  denote the lengths of  $X$  and  $Y$ , respectively.

**Results.** The performance of LLMs and deep learning-based methods in the binary code summarization task under different architectures and compiler optimization options is listed in Table 5 and Table 6, respectively.

From Table 5, we can clearly observe that ChatGPT outperforms all other LLMs in BLEU-4, METEOR, and Rouge-L metrics across different architectures, achieving scores of 7.69, 29.50, and 22.09 in the x64 architecture, 7.38, 28.65, and 21.52 in the x86 architectures, 7.80, 29.48, and 22.48 in the ARM architectures, and 8.30, 30.32, and 23.13 in the MIPS architectures, respectively. At the same time, WizardCoder-15b also shows very competitive results, with scores of 4.72, 24.10, and 18.16 in the x64 architecture, 4.63, 24.04, and 18.12 in the x86 architecture, 4.67, 24.23, and 18.02 in the ARM architecture, and 5.08, 25.16, and 19.07 in the MIPS architecture. Similar to the function name recovery task, CodeGen25-7b and ChatGLM2-6B perform the worst in their respective domains, but narrow the performance gap with other LLMs. Similar to the function name recovery task, all LLMs perform best on the MIPS architecture, with an average Rouge-L score of 18.58, compared to 17.69, 17.58, and 17.86 on the x64, x86, ARM architectures, respectively. The Rouge-L for the MIPS architecture is improved by 5.03%, 5.69%, and 4.03% over the other architectures.

Table 6: Comparison of LLMs and DL-based methods on binary code summarization under different compiler optimization options. We mark the best performing methods in each domain.

Domain	Model	O0			O1			O2			O3						
		BLEU-4	METEOR	Time(s)													
Code	CodeGen25-7b-instruct	3.56	20.76	14.06	7.14	3.55	20.83	13.88	7.20	20.72	13.90	7.27	3.47	20.19	13.42	7.38	
	WizardCoder-15b-V1.0	4.72	24.10	18.16	8.00	4.63	23.90	18.00	8.08	24.12	18.07	8.18	4.56	23.60	17.61	8.50	
LLMs	WizardCoder-33b-V1.1	4.12	23.72	16.03	20.77	4.16	23.79	16.09	20.68	4.18	23.81	16.20	20.73	4.08	23.26	15.99	20.70
	Code Llama-7b-instruct-hf	4.16	21.10	16.65	5.55	4.13	20.70	16.48	5.42	4.05	20.67	16.55	5.31	4.06	20.99	16.59	5.50
Code Llama-13b-instruct-hf	Code Llama-13b-instruct-hf	4.22	19.52	16.46	7.62	4.20	19.33	16.22	7.88	4.25	19.28	16.42	7.78	4.19	19.62	16.45	8.86
	Code Llama-34b-instruct-hf	4.41	20.97	17.60	15.70	4.32	20.81	17.41	16.00	4.29	20.58	17.45	16.59	4.37	20.44	17.40	16.87
Code Llama-70b-instruct-hf	Code Llama-70b-instruct-hf	4.26	22.37	16.60	46.33	4.09	22.18	16.51	46.31	4.09	22.39	16.43	47.83	4.13	22.14	16.25	49.08
	DeepSeek-Coder-33b-instruct	4.66	24.08	17.38	17.32	4.67	24.03	17.38	17.15	4.62	24.01	17.42	16.87	4.61	23.71	17.28	17.42
ChatGLM2-6B	ChatGLM2-6B	3.54	21.42	14.49	5.21	3.49	21.45	14.46	5.29	3.50	21.43	14.45	5.36	3.40	21.17	14.26	5.40
	Vicuna-7b-v1.5	4.47	23.02	17.64	4.40	4.39	22.95	17.58	4.41	4.41	22.99	17.68	4.49	4.36	22.87	17.44	4.56
General LLMs	Vicuna-13b-v1.5	5.59	22.23	19.75	4.86	5.46	21.91	19.57	5.05	5.48	22.19	19.81	5.00	5.35	21.77	19.43	5.20
	Llama-2-13b-chat-hf	4.72	22.56	18.06	7.07	4.57	22.45	17.86	7.43	4.61	22.57	17.91	7.58	4.41	22.34	17.61	8.03
LLMs	Llama-2-70b-chat-hf	4.36	23.17	16.14	45.73	4.24	23.15	15.93	48.11	4.23	23.10	16.01	46.28	4.13	22.91	15.69	50.13
	Mistral-7B-Instruct-v0.2	6.05	25.65	20.86	3.51	6.00	25.21	20.60	3.51	5.93	25.69	20.67	3.59	5.85	25.38	20.69	3.59
Mixtral-8x7B-Instruct-v0.1	Mixtral-8x7B-Instruct-v0.1	6.41	26.02	21.04	12.41	6.27	25.85	20.93	11.38	6.36	26.02	21.08	11.35	6.27	25.86	20.90	11.37
	ChatGPT (gpt-3.5-turbo-16k)	7.69	29.50	22.09	-	7.57	29.13	21.94	-	7.46	29.12	21.92	-	7.40	29.07	21.76	-
DL-based	BinT5 (Al-Kaswan et al., 2023)	0.00	2.08	4.69	0.56	0.00	2.06	4.60	0.56	0.00	2.06	4.65	0.55	0.00	2.09	4.72	0.55
	HexT5 (Xiong et al., 2023)	0.10	6.21	8.44	0.43	0.10	6.18	8.47	0.43	0.09	6.20	8.38	0.42	0.12	6.30	8.49	0.42

Unlike the function name recovery task, the performance of general domain LLMs is generally significantly better than that of code domain LLMs in binary code summary tasks. This may be attributed to the different properties of the two tasks. In the function name recovery task, the output of LLMs is usually shorter and only needs to generate a function name, which is relatively simple. In contrast, the binary code summarization task requires generating longer natural language descriptions to accurately summarize the functionality and structure of binary code, which requires the model to understand more contextual information and convert it into natural language text, which is a more complex task. General domain LLMs are better at generating longer natural language descriptions due to their inherent characteristics, while code-domain LLMs have limited capabilities in this regard.

**Findings 5:** Among all LLMs, ChatGPT performs the best in the binary code summarization task across different architectures, achieving BLEU-4 scores of 7.69, 7.38, 7.80, and 8.30 on x64, x86, ARM, and MIPS, respectively. All LLMs perform best on the MIPS architecture, achieving an average Rouge-L score of 18.58, which represents improvements of 5.03%, 5.69%, and 4.03% compared to the other architectures. General domain LLMs perform significantly better than code domain LLMs, which is attributed to its stronger long-context understanding and summarizing capabilities.

As observed in Table 6, similar to the results across different architectures, ChatGPT outperforms all other LLMs in the BLEU-4, METEOR, and Rouge-L metrics under various compiler optimization options. It achieves scores of 7.69, 29.50, and 22.09 under the -00 optimization, 7.57, 29.13, and 21.94 under the -01 optimization, 7.46, 29.12, and 21.92 under the -02 optimization, and 7.40, 29.07, and 21.76 under the -03 optimization options, respectively. Additionally, among code-domain LLMs, WizardCoder-15b leads in both BLEU-4 and Rouge metrics, outperforming other models in the same domain with scores of 4.72 and 18.16, 4.63 and 18.00, 4.64 and 18.07, and 4.56 and 17.61 across the different optimization options. As in the case of different architectures, CodeGen25-7b and ChatGLM2-6B still perform the worst in the binary code summarization task within their respective domains.

Similar to the function name recovery task, the performance differences across all LLMs in binary function summarization at the -00, -01, -02, and -03 optimization levels are also minimal, with an average Rouge-L score of 17.68, 17.55, 17.62, and 17.42, respectively. The performance gap between -00 and -03 is only 1.49%.

**Findings 6:** ChatGPT exhibits the best performance in the binary code summarization task across different compiler optimization settings, achieving BLEU-4 scores of 7.69, 7.57, 7.46, and 7.40 for the -00, -01, -02, and -03 optimization options, respectively. All LLMs exhibit minimal performance differences across different optimization levels, with the average ROUGE-L score difference between -00 and -03 being 1.49%.

For the deep learning-based expert models, BinT5 (Al-Kaswan et al., 2023) achieves average BLEU-4, METEOR, and Rouge-L scores of 0.00, 2.06, and 4.68, re-

spectively, across different architectures and optimization options. HexT5 (Xiong et al., 2023) shows slight improvement, with scores of 0.10, 6.20, and 8.43, respectively. Similar to the function name recovery task, their performance still falls significantly short of LLMs.

**Findings 7:** Similar to the previous task, existing DL-based expert models perform worse than LLMs on the binary code summarization task.

Regarding inference time, locally deployed LLMs are generally 5-6 times longer than the function name recovery task. LLMs with 6B-7B parameters usually take 3.4 to 7.4 seconds to infer a single piece of data, 13-15B scale LLMs take 4.8 to 8.9 seconds, and 33-34B scale LLMs takes 15.4 to 21 seconds. CodeLlama-70b takes the most of 49.08 seconds among LLMs. The DL-based model also shows the advantage of inference speed, with HexT5 taking only the shortest 0.42 seconds.

**Findings 8:** For the binary code summarization task, inference time for locally deployed LLMs is about five times that of function name recovery.

### 4.3 RQ3: Factors that Significantly Affect Performance

We further explore the key factors affecting the performance of LLMs in this section. As this experiment focuses on analyzing factors including the few-shot form prompts, the length of pseudo code, and the length of symbol information, the experimental environment is fixed with the x64 architecture and the O0 optimization option.

#### 4.3.1 Few-shot prompts

The pre-training datasets for LLMs contain little or no binary code, which makes directly applying LLMs to binary code understanding tasks likely not to yield optimal results. In this case, few-shot prompts become a potential solution, by providing well-designed examples to LLMs, so that LLMs can learn the unique structure and syntax of binary code and quickly adapt to new tasks. Specifically, we construct two carefully designed pairs of pseudo code and ground-truth examples, and add them to the original prompts. We conduct experiments on few-shot prompts for all LLMs in the previous experiments. The results are shown in Table 7.

For the function name recovery task, both code-domain LLMs and general-domain LLMs show significant improvements compared to zero-shot prompts. Among them, the code domain WizardCoder-33b and the general domain ChatGPT exhibit the most notable improvements, with Precision, Recall, and F1-score increasing by 11.19, 8.15, and 9.90 points for WizardCoder-33b, and 8.16, 10.70, and 9.33 points for ChatGPT, respectively. The code domain LLMs show an average improvement of 5.43 in Precision, 4.05 in Recall, and 4.84 in F1-score. In comparison, the performance improvement of general domain LLMs is slightly lower, with an average improvement of 5.24, 3.84, and 4.49 on Precision, Recall, and F1-score, respectively.

For the binary code summarization task, both code domain and general domain LLMs show a slight improvement compared to the function name recovery task.

Table 7: Performance of prompts in the form of Few-shot. The Impr. column represents the performance improvement of Few-shot compared to Zero-shot. We mark the increase and decrease of the metrics. (x64\_00)

Model	Function Name Recovery												Binary Code Summarization											
	Precision			Recall			F1-score			Time(s)			BLEU-4			METEOR			Rouge-L			Time(s)		
	Few	Impr.		Few	Impr.		Few	Impr.		Few	Impr.		Few	Impr.		Few	Impr.		Few	Impr.		Few	Impr.	
CodeGen25-7b-instruct	17.32	+7.34pt	16.33	+3.71pt	16.04	+5.98pt	1.88	+0.25s	3.83	+0.27pt	19.88	-0.88pt	13.56	-0.50pt	11.31	+4.17s								
WizardCoder-15b-V1.0	29.25	+5.96pt	26.34	+4.23pt	26.64	+4.88pt	7.85	+6.28s	4.83	+0.11pt	25.07	+0.97pt	18.89	+0.73pt	10.27	+2.27s								
WizardCoder-33b-V1.1	31.76	+11.19pt	29.28	+8.15pt	29.39	+9.90pt	4.64	-0.37s	5.16	+1.04pt	26.03	+2.31pt	18.53	+2.50pt	24.06	+3.29s								
Code Llama-7b-instruct-hf	28.07	+0.55pt	26.07	+1.01pt	26.19	+0.96pt	1.15	-0.04s	4.55	+0.39pt	23.83	+2.73pt	17.81	+1.16pt	6.39	+0.84s								
Code Llama-13b-instruct-hf	32.08	+5.44pt	30.03	+5.20pt	29.75	+5.07pt	2.10	+0.38s	4.55	+0.33pt	23.41	+3.89pt	17.28	+0.82pt	11.37	+3.75s								
Code Llama-34b-instruct-hf	31.91	+3.18pt	30.18	+2.85pt	30.08	+3.33pt	3.84	+0.42s	4.92	+0.51pt	24.45	+3.48pt	17.89	+0.29pt	24.56	+8.86s								
Code Llama-70b-instruct-hf	30.92	+4.35pt	29.55	+4.03pt	29.31	+4.41pt	12.07	+2.02s	4.74	+0.48pt	25.15	+2.78pt	17.31	+0.71pt	59.01	+12.68s								
DeepSeek-Coder-33b-instruct	28.97	+5.41pt	27.78	+3.20pt	27.24	+4.22pt	7.11	+3.12s	4.82	+0.16pt	25.92	+1.84pt	16.68	-0.70pt	31.30	+13.98s								
ChatGLM2-6B	16.50	+2.62pt	14.11	+1.54pt	14.68	+2.02pt	1.30	+0.22s	3.93	+0.39pt	22.34	+0.92pt	16.27	+1.78pt	5.64	+0.43s								
Vicuna-7b-v1.5	24.05	+5.77pt	20.93	+2.76pt	21.63	+4.31pt	1.11	-0.02s	5.07	+0.60pt	22.18	-0.84pt	18.31	+0.93pt	4.54	+0.14s								
Vicuna-13b-v1.5	27.07	+5.83pt	25.09	+3.55pt	25.09	+4.63pt	1.52	+0.49s	6.16	+0.57pt	24.58	+2.35pt	20.23	+0.48pt	8.98	+4.12s								
Llama-2-13b-chat-hf	24.94	+1.06pt	23.90	+0.62pt	23.06	+0.38pt	2.08	+0.90s	5.28	+0.56pt	23.34	+0.78pt	18.24	+0.18pt	10.83	+3.76s								
Llama-2-70b-chat-hf	27.26	+3.75pt	24.70	+3.14pt	25.01	+3.29pt	10.59	+6.13s	5.23	+0.87pt	24.57	+1.40pt	17.35	+1.21pt	59.72	+13.99s								
Mistral-7B-Instruct-v0.2	27.59	+5.67pt	27.51	+3.69pt	26.50	+4.69pt	1.28	+0.08s	6.67	+0.62pt	27.03	+1.38pt	21.06	+0.20pt	5.59	+2.08s								
Mixtral-8x7B-Instruct-v0.1	28.39	+9.05pt	30.35	+4.70pt	28.29	+7.29pt	2.67	+0.02s	6.84	+0.43pt	27.65	+1.63pt	21.92	+0.88pt	15.44	+3.03s								
ChatGPT(gpt-3.5-turbo-16k)	27.56	+8.16pt	29.11	+10.70pt	27.41	+9.33pt	-	-	7.93	+0.24pt	30.61	+1.11pt	23.16	+1.07pt	-	-								

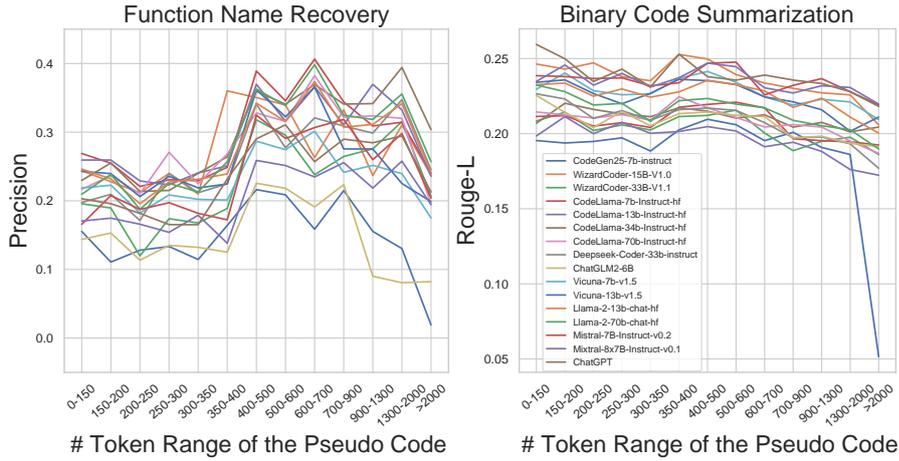


Fig. 5: Impact of pseudo code length on performance.

Among them, WizardCoder-33b exhibits the most notable improvement, with an increase of 1.04, 2.31, and 2.50 points in the BLEU-4, METEOR, and Rouge-L metrics, respectively. However, there are still some LLMs that show a decrease in METEOR and Rouge-L metrics for this task. For example, the code domain CodeGen25-7b has decreased by 0.88 and 0.50 points in METEOR and Rouge-L, respectively, while the general domain Vicuna-7b shows a decrease of 0.84 points in METEOR. Observing their outputs, we find that the introduction of the few-shot examples increased the length of prompts, causing more test data to exceed the maximum length of window context of the model (4096 tokens) and be truncated, resulting in a decrease in performance. Overall, the code domain LLMs show an average improvement of 0.41, 2.14, and 0.63 points in the three metrics for the summarization task, respectively. In comparison, the general domain LLMs show average improvements of 0.54, 1.09, and 0.84 points.

In addition, few-shot prompts will improve inference time in most cases. However, observing the outputs of WizardCoder-15b, we find that the few-shot prompts improve the model’s ability to follow instructions, reduce the output of useless information, and thus reduce the inference time. In this case, few-shot prompts not only enhance model performance but also improve inference efficiency.

**Findings 9:** When computing resources and inference time permit, few-shot prompts can be selected to improve the performance of LLMs on function name recovery and binary code summarization tasks.

#### 4.3.2 Pseudo Code Length

To study the impact of pseudo code length on the performance of LLMs, we divide the length of pseudo code token according to intervals, and controll the number of data in each interval between 100 and 200 to avoid long-tail distribution of data.

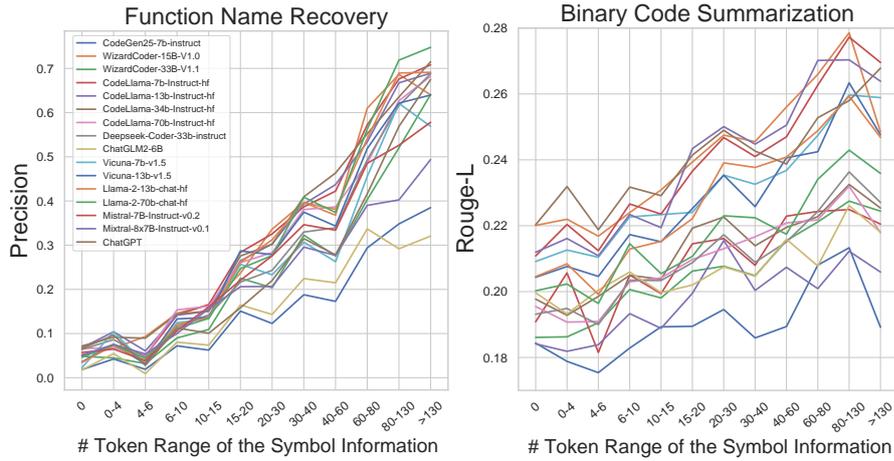


Fig. 6: Impact of symbol information length on performance.

As shown in Figure 5, when the length of pseudo code is between 0-400 tokens, the metric of function name recovery remains at a relatively low level, as shorter pseudo code may not provide enough keywords to infer the purpose and naming intention of the function. Longer pseudo code can provide more contextual information, helping the LLMs capture semantic clues related to function names. Therefore, the metrics are relatively high between 400-2000 tokens; After exceeding 2000 tokens, the structure and logic of the code are too complex, making it difficult for the LLMs to process and integrate a large amount of information, resulting in a decrease in metrics.

For the binary code summarization task, the metrics show a slowly decreasing trend as the pseudo code length increases. As code complexity increases, LLMs find it difficult to maintain both conciseness and accuracy of summaries resulting in the generation of lengthy and unfocused summaries, thereby reducing the overall quality of the summaries.

**Findings 10:** LLMs achieve the best performance for function name recovery at moderate pseudo code length, while the performance of binary code summarization slowly decreases as pseudo code length increases.

#### 4.3.3 Symbol Information Length

We define symbol information as the strings and identifiers in the pseudo code that are not stripped during the strip process, which can provide human-understandable semantic information. We also divide the length of the symbolic information token into intervals.

As shown in Figure 6, as the length of the symbol information token increases, the performance of both function name recovery and binary code summarization tasks increases significantly. This is due to the fact that longer symbol information provides richer semantic content and more context clues, helping LLMs understand the intent

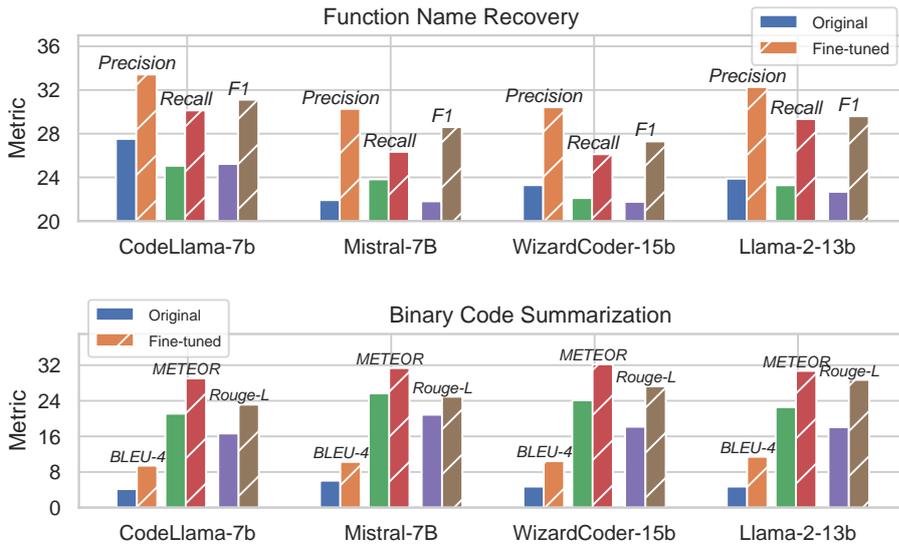


Fig. 7: Comparison of original and fine-tuned LLMs on performance.

and functionality of the code. However, we found that when the symbol information token exceeds 130, the performance of most LLMs in binary code summarization tasks slightly decreases. This is because more symbol information tokens are accompanied by longer pseudo code lengths, resulting in more code being truncated due to window context limitations, affecting the completeness of LLMs summary.

**Findings 11:** The symbol information (e.g., strings and identifiers) has rich semantics and contributes significantly to LLM’s understanding of binary code.

#### 4.4 RQ4: Fine-Tuning to Enhance the Performance

As mentioned in Section 3.2.3, we build the fine-tuning dataset from the GNU repository, with each piece of data in the form of decompiled pseudo code and ground-truth pairs. Considering the computational resource limitations, we fine-tune only the 7b-15b LLMs, selecting CodeLlama-7b-instruct-hf and WizardCoder-15b-V1.0 from the code domain, as well as Mistral-7B-Instruct-v0.2 and Llama-2-13b-chat-hf from the general domain, which have performed well in previous experiments.

Figure 7 shows the performance comparison of original and fine-tuned LLMs. For the function name recovery task, the general domain Llama-2-13b-chat-hf shows the greatest improvement, with Precision, Recall, and F1-score increasing by 8.39, 6.06, and 6.92 points, respectively. On average, all LLMs shows improvements of 7.44, 4.42, and 6.28 points on three metrics. For the binary code summarization task, Llama-2-13b-chat-hf also shows the most significant improvement, with BLEU-4, METEOR, and Rouge-L metrics increasing by 6.70, 8.15, and 10.63 points, respectively. On average, all LLMs shows improvements of 5.48, 7.46, and 7.56 points.

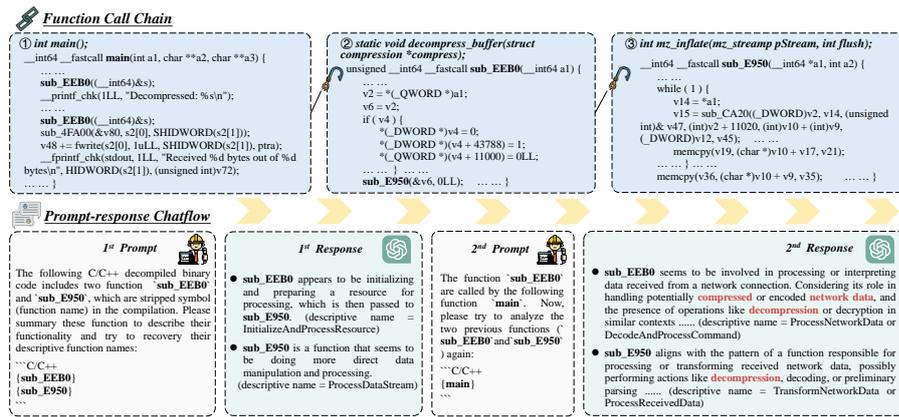


Fig. 8: An example of binary code understanding in a real-world virus with ChatGPT.

Overall, fine-tuning LLMs on downstream tasks related to binary code understanding can bring considerable performance improvements.

**Findings 12:** Introducing binary domain knowledge through fine-tuning can improve the performance of LLMs on function name recovery and summary production tasks. Among them, the general-domain Llama-2-13b-chat-hf demonstrates the most significant improvement in both tasks.

#### 4.5 RQ5: Case Study on Real-World Virus Analysis

We present a few case studies to show how much advanced general LLMs can assist participants in a real-world scenario. Specifically, we utilize ChatGPT (Ouyang et al., 2022) to facilitate virus analysis, including summarizing the functionality of decompiled binary functions in viruses and recovering their descriptive names.

An open-source Linux remote access trojan named splinter<sup>3</sup> is compiled with gcc-11.4.0 and stripped to release. In this case, Figure 8 has shown a partial analysis in a call chain, where the reduced pseudo code are shown in the upper part, as well as the function definitions in the source code. The lack of symbolic information makes it difficult for an analyst to understand the function sub\_EEB0 and sub\_E950. We first construct the first prompt with these two functions and feed it to ChatGPT, asking for the functional summaries and descriptive names. The model gives us a primary description of the operations performed in the pseudo code without any high-level insight. We then construct the second prompt with the caller function main that contains a few symbol information. The second response correctly indicates that these functions are related to data decompression. At the same time, the predicted names reflect their functionality, although the predictions do not exactly match the source code.

<sup>3</sup> <https://github.com/tuian/splinter>

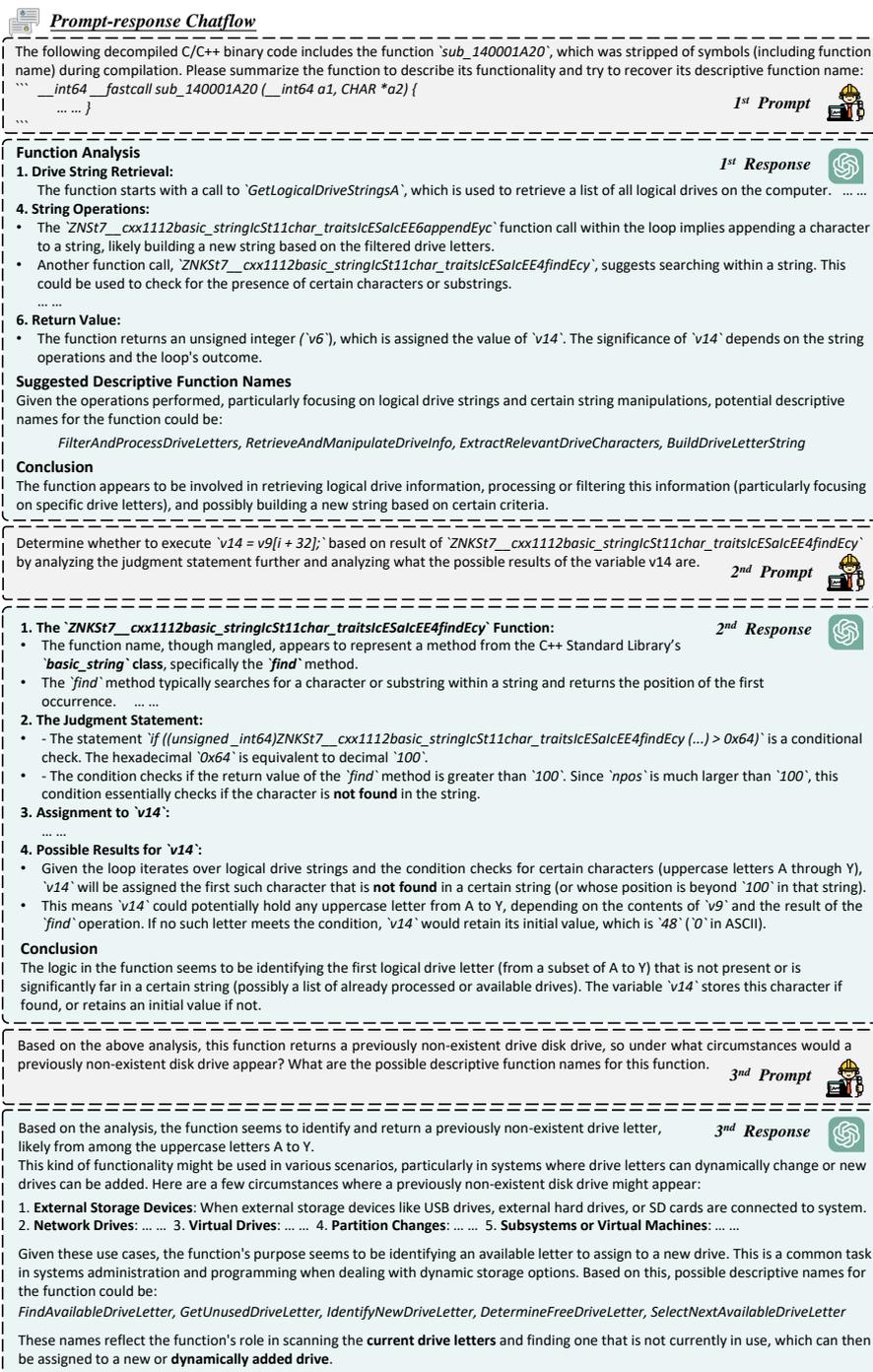


Fig. 9: The chatflow of analyzing the binary function `getRemovableDisk` from `TrojanCockroach` with ChatGPT.

In another case, we query a function named `getRemovableDisk` in stripped binary from TrojanCockroach<sup>4</sup> virus, which is used to get the recently inserted disk letter for further infection. In the Chatflow shown in Figure 9, we make several queries with ChatGPT regarding the functionality summary and function name recovery of the pseudo code `sub_140001A20` decompiled with IDA Pro. We have omitted parts of ChatGPT’s responses that deviated from our desired goals and were inconsistent with the facts. We find that ChatGPT provided an answer close to the fact in its first response, but it lacks further analysis of the key judgment statement. We lead it to successfully find out the callee function `find` mangled in the binary. With the help of calling context, we further prompt ChatGPT to analyze the functionality and provide high-level insights. Finally, it correctly summarizes the behavior of the function and tell us that the return value of the function is a dynamically added drive letter.

**Findings 13:** ChatGPT demonstrates the potential ability to analyze binaries in the real world. The information from the calling context will boost the predictions of LLMs.

## 5 Discussions

Based on the experimental findings, we summarize directions for future work and limitations of the current evaluation.

### 5.1 Future Works

Current LLMs have indeed shown potential in understanding binary codes, and we believe that future work can be conducted in-depth from the following aspects.

- *Develop domain-specific LLM:* Binary code typically lacks annotations and rich context, making it difficult for LLMs to correctly understand the semantics and functionality of the code. Future research could focus on developing domain-specific LLMs that are pre-trained specifically for the characteristics of binary code. For example, by incorporating extensive binary domain knowledge during the pre-training phase to enhance the LLMs’s grasp of code semantics and structure.
- *Extend context window:* Many existing LLMs have fixed context window sizes, which are often insufficient for handling long and complex binary code. Binary code typically involves multiple functions, call relationships, and intricate control flow structures, requiring longer context windows for effective analysis. Future research should explore architectures that support longer sequence lengths, such as enhanced attention mechanisms or extended Transformer models, to better analyze complex binary code.
- *Enhance processing of non-intuitive code:* Current LLMs rely on identifiers and descriptive strings in binary code to understand its functionality. However, binary code often lacks these elements, especially after being obfuscated or stripped

<sup>4</sup> <https://github.com/MinhasKamal/TrojanCockroach>

of symbol tables. Future research should focus on developing new algorithms or enhancing LLM capabilities to understand binary code functionality without identifiers, using techniques such as static analysis or dynamic tracking (e.g., by integrating call chains and execution traces).

- *Integrate multi-modal information*: Binary code analysis should not rely on a single source of information. By integrating multiple data sources, such as human expert annotations, assembly instructions, and dynamic execution data, LLMs’ understanding of binary code can be significantly enhanced. Future research should focus on multi-modal information integration, incorporating these heterogeneous data sources into LLM inputs to provide a more comprehensive analysis of binary code.
- *Enhancing transfer learning capabilities*: The behavior of binary code often varies due to compiler optimizations, target platforms (e.g., x86, x64, ARM architectures), and different operating system environments. Therefore, future LLM research should focus on enhancing models’ transfer learning capabilities across different platforms. This could involve constructing training datasets that span multiple architectures and environments, enabling models to better predict function names and understand binary code in unknown platforms and environments.
- *Robustness in analyzing obfuscated binary code*: Current binary code analysis methods struggle with obfuscated and encrypted code, especially in malware analysis. To improve LLM robustness, techniques like execution-aware code embeddings and dynamic execution tracing can help better interpret obfuscated code, recover its functionality, and enhance analysis accuracy.

## 5.2 Limitations

Although this paper provides a systematic evaluation of the performance of LLMs on binary code understanding tasks, we need to acknowledge existing limitations.

- *Evaluation metrics for code summarization tasks*: Current practices are mainly based on text coherence metrics such as BLEU-4 (Papineni et al., 2002) and Rouge-L (Lin, 2004), which are originally designed for text translation tasks. However, these metrics may not be fully applicable to binary code summarization tasks. Reverse engineers typically rely on specific key terms to understand the design of a function, where text fluency is not the most critical factor. It may be beneficial to develop a new metric to better capture the essence of binary summarization.
- *Binary code obfuscation*: The evaluation dataset in this paper does not consider any form of binary code obfuscation, such as encryption or compiler-based obfuscation (Junod et al., 2015). In fact, code obfuscation significantly alters the structure and control flow of the code, often leading to substantial changes in the form of the pseudo code, thereby increasing the difficulty of understanding and interpreting the code. Models may perform poorly when faced with obfuscated code, which could affect their applicability in real-world reverse engineering tasks. Future research could explore ways to improve the robustness of LLMs in dealing with obfuscated code, particularly in scenarios like malware analysis, where obfuscation is prevalent.

- *Understanding function relationships*: In binary code understanding, function name recovery and code summarization are two critical tasks, but these tasks are usually handled in isolation. However, functions in binary code are usually interdependent and call each other, making the understanding of function relationships essential for comprehending the overall code. This paper focuses more on analyzing individual functions, ignoring the mutual relationships and call dependencies between different functions and the overall program structure.

## 6 Conclusion

Large Language Models (LLMs) have demonstrated significant potential in the field of binary code understanding. In this paper, we select two representative tasks: (1) function name recovery and (2) binary code summarization, and design an automated method to construct a benchmark for a comprehensive evaluation of LLMs' ability to understand binary code. The research findings indicate that LLMs, particularly models such as CodeLlama, WizardCoder, and ChatGPT, have achieved impressive results in certain aspects of binary code understanding. However, the models' performance still requires improvement when dealing with more complex binary structures and unseen code samples. Additionally, we observe that LLMs' performance varies across different target architectures and compiler optimization options. In particular, LLMs perform better on the MIPS architecture compared to other architectures, which may be attributed to the simplified instruction set and the unified function call conventions of the MIPS architecture. Furthermore, code domain LLMs generally outperform general domain LLMs, as they are better equipped to handle the syntax and structure of binary code, leading to superior performance.

Therefore, we call for more research to focus on this important area of software engineering, exploring ways to improve and optimize LLMs so that they can play a more pivotal role in complex binary code analysis tasks. This will open up new possibilities and application paths in the field of understanding binary code, particularly in tasks such as reverse engineering, malware analysis, and vulnerability detection.

**Acknowledgements** This work was supported in part by the Natural Science Foundation of China under Grant U20B2047, 62072421, 62002334, 62102386 and 62121002, and by Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation under Grant CSSAE-2021-007.

## References

- Al-Kaswan A, Ahmed T, Izadi M, Sawant AA, Devanbu P, van Deursen A (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, pp 260–271
- Black S, Biderman S, Hallahan E, Anthony Q, Gao L, Golding L, He H, Leahy C, McDonnell K, Phang J, et al. (2022) Gpt-neox-20b: An open-source autoregressive language model. arXiv preprint arXiv:220406745

- Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, et al. (2020) Language models are few-shot learners. *Advances in neural information processing systems* 33:1877–1901
- Bzdok D, Thieme A, Levkovskyy O, Wren P, Ray T, Reddy S (2024) Data science opportunities of large language models for neuroscience and biomedicine. *Neuron* 112(5):698–717
- Canfora G, Di Penta M, Cerulo L (2011) Achievements and challenges in software reverse engineering. *Communications of the ACM* 54(4):142–151
- Chen B, Zhang Z, Langrené N, Zhu S (2023a) Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:231014735*
- Chen G, Gao H, Zhang J, He Y, Cheng S, Zhang W (2023b) Investigating neural-based function name reassignment from the perspective of binary code representation. In: *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, IEEE, pp 1–11
- Chen M, Tworek J, Jun H, Yuan Q, Pinto HPDO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, et al. (2021) Evaluating large language models trained on code. *arXiv preprint arXiv:210703374*
- Curl (2024) URL <https://github.com/curl/curl>
- Dagdelen J, Dunn A, Lee S, Walker N, Rosen AS, Ceder G, Persson KA, Jain A (2024) Structured information extraction from scientific text with large language models. *Nature Communications* 15(1):1418
- Dai D, Sun Y, Dong L, Hao Y, Ma S, Sui Z, Wei F (2022) Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers. *arXiv preprint arXiv:221210559*
- David Y, Alon U, Yahav E (2020a) Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4(OOPSLA):1–28
- David Y, Alon U, Yahav E (2020b) Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4(OOPSLA):1–28
- DeepSpeed (2024) URL <https://www.deepspeed.ai/>
- Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al. (2020) Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:200208155*
- FFmpeg (2024) URL <https://github.com/FFmpeg/FFmpeg>
- Fried D, Aghajanyan A, Lin J, Wang S, Wallace E, Shi F, Zhong R, Yih Wt, Zettlemoyer L, Lewis M (2022) InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:220405999*
- Gao H, Cheng S, Xue Y, Zhang W (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*, pp 607–619
- Gellenbeck EM, Cook CR (1991) An investigation of procedure and variable names as beacons during program comprehension. *Tech. rep.*, USA

- Giffin JT, Jha S, Miller BP (2004) Efficient context-sensitive intrusion detection. In: Network and Distributed System Security Symposium
- Guo D, Zhu Q, Yang D, Xie Z, Dong K, Zhang W, Chen G, Bi X, Wu Y, Li Y, et al. (2024) Deepseek-coder: When the large language model meets programming—the rise of code intelligence. arXiv preprint arXiv:240114196
- He J, Ivanov P, Tsankov P, Raychev V, Vechev M (2018) Debin: Predicting debug information in stripped binaries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp 1667–1680
- Hex-RaysSA (2024) "ida pro". <https://www.hex-rays.com/products/ida>
- Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, Luo X, Lo D, Grundy J, Wang H (2024) Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33(8):1–79
- Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, Wang L, Chen W, et al. (2022) Lora: Low-rank adaptation of large language models. *ICLR* 1(2):3
- HuggingFace (2024) URL <https://huggingface.co/>
- Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:190909436
- ImageMagick (2024) URL <https://github.com/ImageMagick/ImageMagick>
- International IU (2010) Dwarf debugging information format version 4. <https://dwarfstd.org/doc/DWARF4.pdf>
- Jain H, Prabhu Y, Varma M (2016) Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pp 935–944
- Jiang AQ, Sablayrolles A, Roux A, Mensch A, Savary B, Bamford C, Chaplot DS, Casas Ddl, Hanna EB, Bressand F, et al. (2024) Mixtral of experts. arXiv preprint arXiv:240104088
- Jin X, Pei K, Won JY, Lin Z (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, pp 1631–1645
- Junod P, Rinaldini J, Wehrli J, Michielin J (2015) Obfuscator-llvm—software protection for the masses. In: 2015 IEEE/ACM 1st international workshop on software protection, IEEE, pp 3–9
- Kim D, Kim E, Cha SK, Son S, Kim Y (2022) Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* 49(4):1661–1682
- Kong A, Zhao S, Chen H, Li Q, Qin Y, Sun R, Zhou X, Wang E, Dong X (2023) Better zero-shot reasoning with role-play prompting. arXiv preprint arXiv:230807702
- Kudo T (2018) Subword regularization: Improving neural network translation models with multiple subword candidates. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 66–75
- Lavie A, Denkowski MJ (2009) The meteor metric for automatic evaluation of machine translation. *Machine translation* 23:105–115
- Li X, Qu Y, Yin H (2021) Palmtree: Learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC conference on

- computer and communications security, pp 3236–3251
- Libexpat (2024) URL <https://github.com/libexpat/libexpat>
- Libvips (2024) URL <https://github.com/libvips/libvips>
- Lin CY (2004) Rouge: A package for automatic evaluation of summaries. In: Text summarization branches out, pp 74–81
- Llama2c (2024) URL <https://github.com/karpathy/llama2.c>
- Lu H, Peng H, Nan G, Cui J, Wang C, Jin W, Wang S, Pan S, Tao X (2024) Mal-sight: Exploring malicious source code and benign pseudocode for iterative binary malware summarization. arXiv preprint arXiv:240618379
- Luo Z, Xu C, Zhao P, Sun Q, Geng X, Hu W, Tao C, Ma J, Lin Q, Jiang D (2023) Wizardcoder: Empowering code large language models with evol-instruct. arXiv preprint arXiv:230608568
- Maletic JI, Collard ML (2015) Exploration, analysis, and manipulation of source code using srcml. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol 2, pp 951–952
- Masscan (2024) URL <https://github.com/robertdavidgraham/masscan>
- Naeem MR, Amin R, Alshamrani SS, Alshehri A (2022) Digital forensics for malware classification: An approach for binary code to pixel vector transition. *Computational Intelligence and Neuroscience* 2022(1):6294058
- NationalSecurityAgency (2024) "ghidra". <https://github.com/NationalSecurityAgency/ghidra>
- Nijkamp E, Hayashi H, Xiong C, Savarese S, Zhou Y (2023) Codegen2: Lessons for training llms on programming and natural languages. arXiv preprint arXiv:230502309
- OpenSSL (2024) URL <https://github.com/openssl/openssl>
- Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, et al. (2022) Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35:27730–27744
- Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp 311–318
- Patrick-Evans J, Dannehl M, Kinder J (2023) Xfl: Naming functions in binaries with extreme multi-label learning. In: 2023 IEEE Symposium on Security and Privacy (SP), IEEE, pp 2375–2390
- Pei K, Xuan Z, Yang J, Jana S, Ray B (2020) Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:201208680
- PyTorch (2024) URL <https://pytorch.org/>
- Redis (2024) URL <https://github.com/redis/redis>
- Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Sauvestre R, Remez T, et al. (2023) Code llama: Open foundation models for code. arXiv preprint arXiv:230812950
- Sha Z, Wang H, Gao Z, Shu H, Zhang B, Wang Z, Zhang C (2024) llasm: Naming functions in binaries by fusing encoder-only and decoder-only llms. *ACM Transactions on Software Engineering and Methodology*

- Shang X, Cheng S, Chen G, Zhang Y, Hu L, Yu X, Li G, Zhang W, Yu N (2024) How far have we gone in binary code understanding using large language models. In: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
- Song Z, Chen J, Zhang K (2024) Bin2summary: Beyond function name prediction in stripped binaries with functionality-specific code embeddings. *Proceedings of the ACM on Software Engineering* 1(FSE):47–69
- Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, pp 43–52
- Tan Z, Li D, Wang S, Beigi A, Jiang B, Bhattacharjee A, Karami M, Li J, Cheng L, Liu H (2024) Large language models for data annotation and synthesis: A survey. In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp 930–957
- Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, et al. (2023a) Llama: Open and efficient foundation language models. arXiv preprint arXiv:230213971
- Touvron H, Martin L, Stone K, Albert P, Almahairi A, Babaei Y, Bashlykov N, Batra S, Bhargava P, Bhosale S, et al. (2023b) Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:230709288
- Transformers (2024) URL <https://huggingface.co/>
- Ultrajson (2024) URL <https://github.com/ultrajson/ultrajson>
- Vector35 (2024) "binary ninja". <https://binary.ninja/>
- Wang H, Qu W, Katz G, Zhu W, Gao Z, Qiu H, Zhuge J, Zhang C (2022) Jtrans: Jump-aware transformer for binary code similarity detection. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp 1–13
- Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:210900859
- Wang Y, Le H, Gotmare A, Bui N, Li J, Hoi S (2023) Codet5+: Open code large language models for code understanding and generation. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp 1069–1088
- Whispercpp (2024) URL <https://github.com/ggerganov/whisper.cpp>
- Wu Y, Jiang N, Pham HV, Lutellier T, Davis J, Tan L, Babkin P, Shah S (2023) How effective are neural networks for fixing security vulnerabilities. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp 1282–1294
- Xiong J, Chen G, Chen K, Gao H, Cheng S, Zhang W (2023) Hext5: Unified pre-training for stripped binary code information inference. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, pp 774–786
- Xu FF, Alon U, Neubig G, Hellendoorn VJ (2022) A systematic evaluation of large language models of code. In: *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pp 1–10

- Xu Z, Chen B, Chandramohan M, Liu Y, Song F (2017) Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, pp 462–472
- Ye T, Wu L, Ma T, Zhang X, Du Y, Liu P, Ji S, Wang W (2023) Cp-bcs: Binary code summarization guided by control flow graph and pseudo code. arXiv preprint arXiv:231016853
- Zaremski AM, Wing JM (1995) Signature matching: a tool for using software libraries. ACM Transactions on Software Engineering and Methodology (TOSEM) 4(2):146–170
- Zeng A, Liu X, Du Z, Wang Z, Lai H, Ding M, Yang Z, Xu Y, Zheng W, Xia X, et al. (2022) Glm-130b: An open bilingual pre-trained model. arXiv preprint arXiv:221002414
- Zhang Y (2022) Leveraging artificial intelligence on binary code comprehension. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp 1–3
- Zhang Y, Song W, Ji Z, Meng N, et al. (2023) How well does llm generate security tests? arXiv preprint arXiv:231000710
- Zhang Z, You W, Tao G, Aafer Y, Liu X, Zhang X (2021) Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, pp 659–676
- Zheng L, Chiang WL, Sheng Y, Zhuang S, Wu Z, Zhuang Y, Lin Z, Li Z, Li D, Xing E, et al. (2023) Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in Neural Information Processing Systems 36:46595–46623
- Zheng Y, Zhang R, Zhang J, Ye Y, Luo Z, Feng Z, Ma Y (2024) Llamafactory: Unified efficient fine-tuning of 100+ language models. arXiv preprint arXiv:240313372
- Zhu K, Tian Z, Wang S, Chen W, Dong Z, Leng M, Mao X (2025) Misum: Multi-modality heterogeneous code graph learning for multi-intent binary code summarization. In: Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering
- zstd (2024) URL <https://github.com/facebook/zstd>