BinMetric: A Comprehensive Binary Code Analysis Benchmark for Large Language Models

Xiuwei Shang[1], **Guoqiang Chen**[3], **Shaoyin Cheng**[1,2*], **Benlong Wu**[1], **Li Hu**[1], **Gangyang Li**[1], **Weiming Zhang**[1,2], **Nenghai Yu**[1,2]
[1]University of Science and Technology of China, Hefei, China
[2]Anhui Province Key Laboratory of Digital Security, Hefei, China
[3]QI-ANXIN Technology Research Institute, Beijing, China
{shangxw, ch3nye, dizzylong, pdxbshx, ligangyang}@mail.ustc.edu.cn
{sycheng, zhangwm, ynh}@ustc.edu.cn

Abstract

Binary analysis is crucial for software security, offering insights into compiled programs without source code. As large language models (LLMs) excel in language tasks, their potential for complex decoding binary data structures is growing. However, the lack of standardized benchmarks hinders their evaluation and progress in this domain. To bridge this gap, we introduce BinMetric, a first comprehensive benchmark designed specifically to evaluate LLMs performance on binary analysis tasks. BinMetric comprises 1,000 questions derived from 20 real-world open-source projects across 6 practical binary analysis tasks, including decompilation, code summarization, etc., which reflect actual reverse engineering scenarios. Our empirical study on this benchmark investigates various state-of-the-art LLMs, revealing their strengths and limitations. The findings indicate that while LLMs show strong potential, challenges still exist, particularly in the areas of precise binary lifting and assembly synthesis. In summary, BinMetric makes a significant step forward in measuring binary analysis capabilities of LLMs, establishing a new benchmark leaderboard, and our study offers valuable insights for advancing LLMs in software security.

1 Introduction

Binary analysis is pivotal in various fields like software reverse engineering [Sutherland *et al.*, 2006], malware prevention [Nguyen Hung *et al.*, 2023], and patch analysis [Xu *et al.*, 2017], enabling the understanding and dissecting of software functionalities without source code access. According to a research report by Statista, approximately 21.5 billion IoT devices will be connected globally by 2025 [Statista, 2024]. The diversity in instruction architectures and operating systems, coupled with the predominance of closed-source code and documentation, limits the applicability of source code analysis for securing IoT device firmware, which drives further updates of binary analysis technology.

*Corresponding Author

Unfortunately, understanding and interpreting the structure and behavior of binary files is challenging due to their complexity and lack of direct human readability [Zhang *et al.*, 2021]. Traditional tools [Hex-RaysSA, 2024] and techniques [LLVM, 2024], while effective, often require extensive manual effort and expertise, making the process time-consuming and prone to errors. The integration of automated tools, especially AI-powered ones, has the potential to revolutionize this field by increasing efficiency and reducing human oversight.

Recently, large language models (LLMs) have demonstrated increasing proficiency in a range of complex tasks, particularly showing promise in more specialized code-intensive areas like code synthesis [Jiang *et al.*, 2023] and automated programming assistance [Wei *et al.*, 2023]. This has sparked questions among many software engineering practitioners: *Can LLMs like ChatGPT and CodeLlama effectively perform binary analysis tasks?* However, the specific application of LLMs in this delicate area is still in its infancy, to some extent due to the lack of dedicated benchmarking frameworks that can adequately measure and drive progress in this area.

To address this limitation, we present BinMetric, the first comprehensive benchmark designed to evaluate LLMs capabilities on binary analysis tasks, which supports multiple tasks following realistic reverse engineering scenarios. BinMetric standardizes the evaluation process, offering a consistent and replicable framework to assess LLM performance in this critical area. Specifically, it is composed of six distinct tasks that mirror real-world binary analysis challenges, including call-site reconstruction, decompilation, signature recovery, binary code summarization, algorithm classification, and assembly instruction generation. These tasks are built on 20 open-source projects, ensuring the realisticity, diversity, quality, credibility, and maintenance of data sources. After filtering and inspection, we extract 1,000 question items from these projects. Furthermore, to evaluate these tasks from different dimensions, we built 4 evaluators and integrated them into an automated pipeline for easy one-click invocation.

Next, to quantify the binary analysis capabilities of contemporary LLMs, we conduct an empirical study on BinMetric to assess widely-used LLMs, including open-source models such as Llama2 [Touvron *et al.*, 2023], CodeLlama [Roziere *et al.*, 2023], Mistral [Jiang *et al.*, 2024], DeepSeek-

Coder [Guo *et al.*, 2024], as well as closed-source like ChatGPT [Ouyang *et al.*, 2022] and GPT-4 [Achiam *et al.*, 2023]. The aim is to answer a couple of crucial questions:

- **RQ1**: What is the overall effectiveness of LLMs in binary analysis?
- **RQ2**: Which LLM we investigated performs the best? And which type of LLMs performs better?
- **RQ3**: What factors affect the effectiveness of LLMs?

Our empirical study reveals several findings. First, LLMs show promise in binary analysis but struggle with tasks like call-site reconstruction and assembly generation. Notably, each model exhibits expertise in specific perspectives, like GPT-4 shines in binary lifting and logical analysis, while WizardCoder and CodeLlama excel in semantic comprehension and assembly synthesis. Second, GPT-4 leads overall, and open-source models like CodeLlama-34B show competitive capabilities, highlighting the potential of open-source solutions in this domain. Finally, LLMs efficiency varies with model size and tasks. Larger models tend to perform better but at the cost of efficiency. One-shot prompts enhance effectiveness, whereas longer inputs may hinder performance.

In summary, our major contributions are as follows:

- **Benchmark.** We introduce BinMetric, a pioneering comprehensive benchmark for assessing LLMs performance across multiple real-world binary analysis tasks. It includes 6 distinct tasks, 1,000 questions extracted and filtered from 20 real open-source projects, and 4 evaluators integrated into the automated evaluation pipeline.
- **Empirical Study.** We conduct the first large-scale investigation of widely-used LLMs using BinMetric, studying (1) the overall effectiveness of LLMs across diverse binary analysis tasks, (2) the performance comparisons of different LLMs, (3) factors affecting their effectiveness.
- **Findings and Insights.** Our results reveal the untapped potential of LLMs in binary analysis, providing new insights and future research directions for the field.

2 Background and Related Works

2.1 Problem Definition

Given a source code S, it undergoes a compilation and stripping process to produce a binary file B, represented as $B = R(C(S))$, where C is the compiler and R denotes the stripping process of symbolic information. The binary code analyzer A, designed to support a series of binary analysis tasks $T = \{t_1, t_2, \ldots, t_n\}$, takes binary file B as input and applies these tasks to generate corresponding outputs, formalized as:

$$O = A(B) = \{o_1, o_2, \ldots, o_n\}. \quad (1)$$

where each o_i corresponds to the output of each task t_i. In this paper, we consider LLMs as binary code analyzers A, assessing their ability to perform accurate analysis, ensuring high fidelity and comprehensibility to the original source code S, and showing their effectiveness in binary analysis scenarios.

2.2 Related Works

Binary Analysis. Binary analysis involves examining binary code, the machine-level representation of software executed by the CPU, which, unlike source code, is not human-readable and requires specialized techniques and tools [David *et al.*, 2020]. Traditional tools like IDA Pro [Hex-RaysSA, 2024] have been the backbone of binary analysis for decades, but they are often labor-intensive, require expertise, and struggle to extract high-level semantic information, which is crucial for understanding the code's broader context and functionality, leading to inefficiency with large or highly optimized binaries and incomplete analysis. With the rise of deep learning, many data-driven techniques have transformed this landscape. These methods leverage large datasets and advanced algorithms to enhance and automate the process. For example, data-driven disassembly and decompilation generate more accurate, human-readable code [Tan *et al.*, 2024a]. Deep learning techniques infer variable types and function signatures [Chen *et al.*, 2022], and generative models create concise summaries of binary code [Xiong *et al.*, 2023]. These data-driven approaches promise a more automated, accurate, and efficient future for binary analysis, reducing manual effort and coping with increasingly complex software systems.

Large Language Models. In early sequential language tasks, including both natural and programming languages, task-specific fine-tuning has shown strong performance, which updates model weights to improve performance by learning input-output relationships from task datasets. The rapid advancement of LLMs, such as ChatGPT [Ouyang *et al.*, 2022], with billions to hundreds of billions of parameters, has revolutionized related fields. Trained on massive text data, LLMs possess powerful language understanding and generation capabilities, encapsulating extensive knowledge, and can perform downstream tasks via in-context learning [Dai *et al.*, 2022]. This approach allows models to handle tasks using task-related context without requiring large datasets for fine-tuning. In this paper, we leverage in-context learning to guide LLMs in understanding binary analysis tasks from multiple perspectives, enabling a thorough performance evaluation.

2.3 Challenges and Insights

Challenges. LLMs have recently been deployed to address software engineering challenges, excelling in tasks such as vulnerability detection [Gao *et al.*, 2023] and automatic program synthesis [Jiang *et al.*, 2023], boosting developer productivity and streamlining software development. Corresponding various benchmarks have emerged, like HumanEval [Chen *et al.*, 2021] for code generation and Defect4j [Just *et al.*, 2014] for automated repair, standardizing evaluation, providing clear metrics, and fostering competition to drive progress. However, in binary analysis, no official benchmarks exist yet, and the challenges of creating one are as follows:

① **Lack of Reliable Data Sources and Standardized Preprocessing.** The source and quality of binaries affect benchmark validity, yet no standardized framework exists for data collection or preprocessing, such as compilation environment settings, decompilation tool selection, and ground-truth identification. The lack of consistency, combined with the black-box nature of LLMs, complicates comparisons and

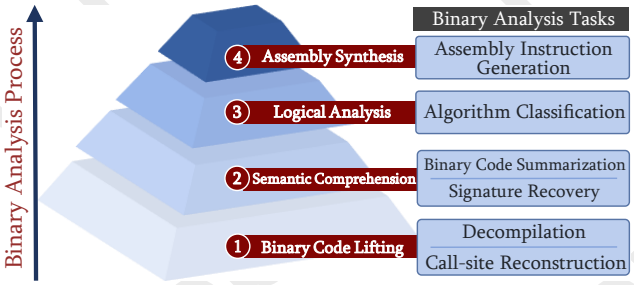


Figure 1: Binary analysis tasks in our benchmark.

risks data leakage from training sets, undermining credibility.

② **Diverse Tasks in Binary Analysis.** Existing works focus on isolated tasks like decompilation [Tan *et al.*, 2024a], or summarization [Jin *et al.*, 2023]. However, binary analysis is inherently a complex, multifaceted process requiring extraction and understanding of various information types and dependencies. Benchmarks limited to individual tasks fail to capture the comprehensive capabilities needed for effective analysis and are not representative of real-world applications.

③ **Complexity of Real-World Scenarios.** Benchmarks should avoid oversimplifying real-world binary analysis, such as relying on a narrow data scope that doesn't reflect the variety of project types and contexts encountered in practice. Additionally, benchmarks should account for the real workflows of reverse engineers to more accurately reflect the multifaceted challenges faced in actual binary analysis.

Insights. To address the challenges mentioned earlier, combined with our analysis of real binary analysis scenarios, we propose the following solution:

① **Establish Data Collection and Preprocessing Criteria.** Enhance the quality of binary analysis benchmarks requires robust criteria for data collection and preprocessing. Inspired by related works and following standards, we advocate that data sources should cover 5 dimensions: realisticity, diversity, quality, credibility, and maintenance. We emphasize standardized preprocessing processes, which require defining protocols for compilation, decompilation, metadata extraction, ground-truth identification, data filtering, and leak checking, minimizing biases to enhance reliability.

② **Enable Multifaceted Task Assessment.** To truly gauge LLMs' performance in binary analysis, benchmarks should assess a range of interconnected tasks across the analysis lifecycle. As shown in Figure 1, our benchmark consists of six tasks in four dimensions, such as decompilation, function signature recovery, and assembly instruction generation, to comprehensively evaluate the capabilities of LLM.

③ **Simulate Real-World Complexity.** In this paper, we aim to replicate the complex challenges of real-world binary analysis scenarios. To achieve this, we combine data from diverse project domains to ensure broad coverage and capture the variety of real-world environments. Subsequently, we dissect the complex challenges faced by reverse engineers in actual binary analysis efforts, and thus craft a series of assessment tasks to reflect the complexity of real-world analysis.

3 BinMetric Benchmark

The overview framework of BinMetric is shown in Figure 2.

3.1 Binary Analysis Tasks

BinMetric contains six representative binary analysis tasks in four dimensions that reflect the challenges faced by human reverse engineers, and LLMs also confront similar obstacles.

Call-site Reconstruction (CSR): Function call relationships reveal control flow and dependencies between program modules. CSR is crucial in reverse engineering to identify and reconstruct function calls, including function names and its calling parameters, from provided assembly code. It takes assembly code and specified call locations as input, producing high-level source code representation of the calls. Evaluation focuses on textual consistency, ensuring the recovered call sites match the original code's intent and structure.

Decompilation (DEC): Decompilation is essential for an intuitive and in-depth understanding of binary programs, which aims to reconstruct a human-readable high-level programming language representation, such as C or C++, based on assembly code. The input is assembly code with function granularity, while the output is the corresponding source code. This task is evaluated using the CodeBLEU [Ren *et al.*, 2020], which assesses both syntax and semantic accuracy by comparing the decompiled code with the original source.

Signature Recovery (SR): Function signatures represent the interface of a function, including its name, parameter types & names, and return type, which are essential for understanding program behavior [Jin *et al.*, 2022]. In this task, the input is decompiled pseudo code from stripped binaries, and the output is the complete function signature. Text consistency metrics are used to evaluate the match between the recovered signature and the original source code signature.

Binary Code Summarization (BCS): Code Summarization aims to generate concise natural language summaries of binary code, highlighting its core functions and operations without needing to examine complex details [Xiong *et al.*, 2023]. The input is decompiled pseudo code, and the output is a corresponding natural language summary. Evaluation uses text consistency metrics to ensure the prediction aligns with the ground truth in terms of relevance and clarity.

Algorithm Classification (AC): This task involves identifying and classifying algorithmic patterns within binary code, revealing key operations like sorting, encryption, etc., essential for understanding the function and purpose of the code segment during security analysis. The input is decompiled pseudo code, and the output is its corresponding algorithm category label. Evaluation is done using accuracy.

Assembly Instruction Generation (AIG): Analyzing malware and fixing vulnerabilities without source code often requires modifying assembly code, *i.e.*, assembly synthesis. The AIG task takes a natural language description of specific functionality as input, and generates corresponding assembly instructions. To evaluate whether the generated can implement intended function, we assess syntax and execution correctness, and also use text consistency to measure reliability.

3.2 Data Collection and Preprocessing

Data Collection. A reliable benchmark dataset is essential for evaluating the binary analysis capabilities of LLMs. To ensure robustness and real-world relevance, we set criteria

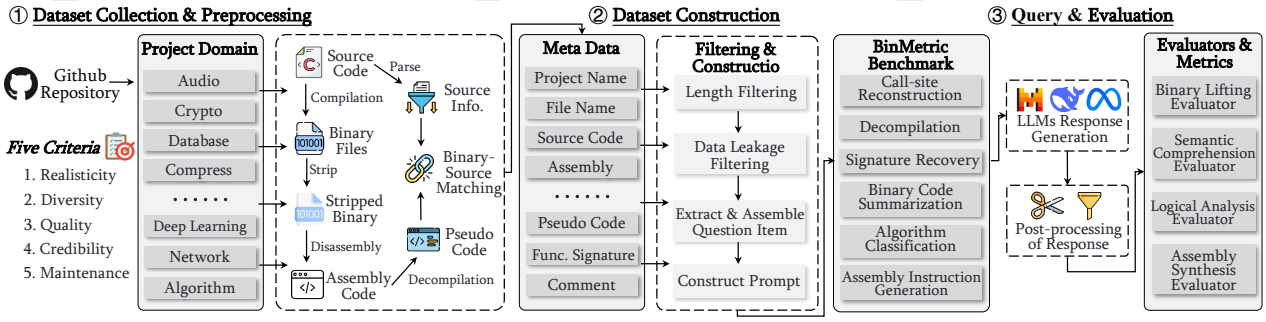


Figure 2: Overview framework of BinMetric benchmark.

Criteria	Description	Quantification
Realisticity	Code should be derived from complete, developer written real-world projects, not toy programs, incomplete or synthesized snippets	Whether from Real-world Projects
Diversity	Code should cover various fields and application domains to avoid bias and assess performance across different contexts	Quantity of Domains Encompassed
Quality	High-quality code with clear structure, logical organization, and good naming conventions	Average Github Stars / Forks
Credibility	Code should come from projects maintained by reputable developers or organizations for better reliability	Maintained by Reputable Organization
Maintenance	Code should from regularly updated, actively maintained, and well-documented projects, reflecting current practices and standards	Average Releases / Commits

Table 1: Data Sources Criteria for BinMetric benchmark.

for code sources, as shown in Table 1. These criteria are informed by industry standards like NIST's guidelines for trustworthy AI [of Standards and (NIST), 2023], as well as software lifecycle and quality assurance international standards such as ISO/IEC 25010:2011 [for Standardization, 2011] and IEEE Std-730-2014 [Heimann, 2014]. Many LLM benchmarks [Sawada *et al.*, 2023; Zeng *et al.*, 2024] also emphasize attributes like realisticity, diversity, quality, and credibility.

As shown in Table 2, we curate 20 high-star C language projects from GitHub, ensuring excellent code quality, credibility, and maintenance. These projects average 18.48K Stars, 3.9K Forks, 75.8 Releases, and 15.5K Commits, covering eleven domains such as audio, image, web, crypto, and network, ensuring both real coding practice and diversity.

Data Preprocessing. Our preprocessing pipeline, shown in Figure 2, involves the following steps:

① **Compile, Strip and Decompile.** We compile selected projects on Ubuntu 22.04 OS for the x86-64 architecture, including DWARF [International, 2010] debugging information to ensure detailed metadata is available for subsequent alignment. Each project uses its default compiler settings, reflecting typical environments. We remove all symbolic information using the `strip` command to simulate real-world conditions where symbolic is often unavailable. We then disassemble and decompile the stripped binaries with IDA Pro, obtaining assembly instructions and decompiled pseudo code.

② **Source Code Information Extraction.** We use srcML [Maletic and Collard, 2015] to parse source files and extract key information including function signatures, implementations, human-written summaries, etc. srcML converts the source files into XML format, enabling accurate extraction

Project	Domain	Project	Domain	Project	Domain
audio	Audio	Llama2.c	Deep Learning	Libexpat	Format
miniaudio	Audio	Whisper.cpp	Deep Learning	Ultrajson	Format
OpenSSL	Crypto	Mongoose	Web	Curl	Network
libsodium	Crypto	libhv	Web	Masscan	Network
Redis	Database	Libvips	Image	7z	Compress
SQLite	Database	ImageMagick	Image	zstd	Compress
FFmpeg	Video	C-Algorithms	Algorithm		

Table 2: Data sources of our benchmark dataset.

and processing through XML parsing techniques. This extracted information is then stored for subsequent alignment.

③ **Binary-Source Alignment.** We align the source and binary code using DWARF debugging information, which links binary functions and variables to their locations in the source code (file name, line and column number). This enables precise matching of assembly instructions and decompiled code to their corresponding source code locations.

3.3 Data Construction

Data Filtering. To avoid incomplete analysis from excessively long code snippets and lack of context from very short ones, we apply double-threshold filtering to remove snippets that are too short or exceed the LLMs' context window, ensuring evaluation feasibility. We also ensure our benchmark dataset is not part of LLMs' training set. All evaluation data are disassembled or decompiled code from self-compiled binaries stripped of symbol information, greatly reducing the risk of inclusion in training sets. To further verify this, we use Google search engine to check if any code appears as plain text online. Any exact whole-word matches found are removed, enhancing dataset credibility.

Extract and Assemble Question Item. After preprocessing and filtering, we obtain high-quality alignment metadata from binary and source code to construct question items.

For DEC and SR tasks, we randomly sample 250 pairs of pseudo code and output ground-truth from the metadata. For CSR, 70 assembly snippets are sampled and manually annotated with call site locations to be recovered and their ground truth. For the BCS task, we initially considered using human-written comments from source files as labels, but less than 20% of functions have comments, many of which lacked functional summaries and contained noisy content. Inspired by recent works [Tan *et al.*, 2024b] that utilize LLMs like ChatGPT to perform data annotation tasks with reasonable reliability, we turn to ChatGPT to generate summaries of

Tasks	Prompt Template
Call-site Reconstruction	Please imagine you are an experienced binary reverse engineer. The following is a disassembled assembly code, your task is to understand its semantics and behavior, and output call point in the form of C source code at 'call sub_5F57E', including only a descriptive function name and function parameters, wrapped in three backticks (```), do not explain. {Example}. Input assembly function: ```{Assmebly Code} ```
Decompilation	Please imagine you are an experienced binary reverse engineer. The following is a disassembled assembly code, your task is to understand it and output its corresponding C source code, wrapped in three backticks (```), do not explain. {Example}. Input assembly function: ```{Pseudo Code} ```
Signature Recovery	Please imagine you are an experienced binary reverse engineer. The following is a stripped decompiled C function, your task is to understand it and output the descriptive function signature in its corresponding source code. This includes the function name, parameter list and its type, and return value type. Wrap the output with three backticks (```), do not explain. {Example}. Input decompiled C function: ```{Pseudo Code} ```
Binary Code Summarization	Please imagine you are an experienced binary reverse engineer. The following is a stripped decompiled C function, your task is to understand it and generate a short comment to the function describing its functionality. {Example}. Input decompiled C function: ```{Pseudo Code} ```
Algorithm Classification	Please imagine you are an experienced binary reverse engineer. The following is a stripped decompiled C function, your task is to understand it and output its algorithm class from the following list: [Sorting, Searching, Numerical Methods, Hash, Conversions, Math, Dynamic Programming, Cipher, ...]. Wrap the output with three backticks (```), do not explain. {Example}. Input decompiled C function: ```{Pseudo Code} ```
Assembly Generation	{Example}. Design an x64 architecture assembly code for {a bubble sort algorithm, which requires an array to be input from the terminal, and then the terminal outputs the sorted result }. The generated assembly code is required to be wrapped in three backticks (```), can be compiled into an executable program by gcc, and does not contain comments.

Table 3: Base prompt templates of LLMs. {Example} represents the golden demonstration example in the One-shot prompts, and {··· ···} represents the specific input for each piece of data.

the source code, outlining code functionality. We select 250 pseudo code-summary pairs, which are manually reviewed for correctness to ensure high-quality ground truth for BCS. For the AC task, we sample 80 pseudo-code snippets from the C-Algorithms [C-Algorithms, 2024] project and manually annotate the respective algorithm categories. For the AIG task, we design 100 clear instructions to guide LLMs in generating assembly code snippets following Intel syntax for given functionalities, such as implementing bubble sort. We also provide test cases with inputs and outputs to verify the functional correctness of the generated assembly code.

Overall, the above tasks, which required manual annotation, review, and verification, cost about 60 man-hours.

3.4 Base Prompt Templates

To leverage the in-context learning capabilities of LLMs, we adopt the one-shot prompt strategy, also and explore zero-shot performance in §4.4. While zero-shot performs well on simple tasks, it often lacks sufficient context for complex binary analysis, resulting in lower accuracy and consistency. The chain-of-thought method aids step-by-step reasoning but faces challenges like context length limits, reasoning efficiency, and uncertain performance improvements.

Table 3 presents detailed one-shot prompts for different tasks. Carefully selected golden examples are included in the prompts to help the model grasp the task context and expected output format. Moreover, role-play prompts [Kong *et al.*, 2023] position the model as an "experienced binary reverse engineer" to clarify task requirements and reduce ambiguity. Code within prompts is enclosed in triple backticks (```) to specify formatting, and the model's output is similarly wrapped to facilitate easier parsing in post-processing and minimize noisy text interference.

3.5 Evaluators and Metrics

We build 4 evaluators, each targeting a specific dimension, and integrate into an automated pipeline for easy invocation.

① **Binary Lifting Evaluator.** This evaluator assesses LLMs' ability to convert binary code in assembly form into a higher-level representation, crucial for reconstructing binary program structure. It applies to CSR and DEC tasks, with evaluation metrics Rouge-L [Lin, 2004] and CodeBLEU [Ren *et al.*, 2020]. Rouge-L measures textual consistency between generated and reference call-site information, while CodeBLEU evaluates the syntactic and semantic similarity of the decompiled code to the source code.

② **Semantic Comprehension Evaluator.** This evaluator measures LLMs' understanding and interpretation of binary code, focusing on capturing the intent behind code snippets. It applies to SR and BCS tasks, using BLEU-1 [Papineni *et al.*, 2002], METHOR [Lavie and Denkowski, 2009], and Rouge-L to assess precise word matching, semantic flexibility and richness, and structural coherence.

③ **Logical Analysis Evaluator.** This evaluator tests LLMs' ability to comprehend algorithmic logic in binary code, classifying algorithms or patterns. It applies to AC tasks, with Accuracy as the metric.

④ **Assembly Synthesis Evaluator.** This evaluator assesses LLMs' ability to generate accurate, executable assembly code from natural language descriptions for the AIG task. The evaluation includes three metrics: Syntactic correctness, Execution correctness, and Rouge-L. Syntactic correctness ensures the generated code compiles without errors, adhering to proper syntax rules. Execution correctness is verified using pre-designed test cases to confirm the intended functionality. To mitigate security risks, code execution is conducted in an isolated Docker environment. Rouge-L measures the textual consistency between the generated code and expected output.

4 Evaluation

4.1 Experiment Setup

Large Language Models Setup. We evaluate 12 LLMs from 5 model families, including both locally deployable open-source and API-callable closed-source models, based on the following criteria: (1) State-of-the-art LLMs ranked on leaderboards like EvalPlus [EvalPlus, 2024]. (2) Extensive code pre-training to enhance programming understanding. (3) Strong capabilities in both natural language and code generation tasks. (4) Instruction-tuned to effectively follow detailed instructions.

Implementation Details. The experiments are conducted on an Ubuntu 22.04 server with 8 NVIDIA RTX A6000 GPUs. For closed-source LLMs, *i.e.* ChatGPT and GPT4, we call OpenAI's API to access `gpt-3.5-turbo-16k-0613` and `gpt-4-0613` backend models. Open-source LLMs are downloaded from Huggingface, with half-precision in FP16 enabled for inference. Given the context window limitations,

Type	Model	Size	BinMetric Benchmark									
			CSR	DEC	SR	BCS			AC	AIG		
			Rouge-L	*CodeBLEU*	*Rouge-L*	*BLEU*	*METEOR*	*Rouge-L*	*Acc.*	*Syntax Corr.*	*Execution Corr.*	*Rouge-L*
Baseline Methods	LLM4-Decompile	1.3B	-	21.53 (↑6.7%)	-	-	-	-	-	-	-	-
		6.7B	-	22.97 (↑6.6%)	-	-	-	-	-	-	-	-
		33B	-	24.48	-	-	-	-	-	-	-	-
	IDA Pro	-	8.52	18.56	10.27	-	-	-	-	-	-	-
	BinT5	220M	-	-	-	31.58	1.82	4.28	-	-	-	-
	HexT5	223M	-	-	-	36.92	4.99	7.65	-	-	-	-
Open Source	Llama2	7B	3.83 (↑28.2%)	24.71 (↓4.4%)	6.51 (↑122.6%)	35.83 (↓54.5%)	21.64 (↑22.3%)	19.00 (↓31.3%)	15.00 (↑141.7%)	1.00 (↑6700%)	0.00 (↑0.0%)	**31.58** (↑22.6%)
	CodeLlama	7B	4.91 (↑25.7%)	23.62 (↓12.8%)	14.49 (↑83.5%)	16.32 (↑77.9%)	16.81 (↑71.9%)	13.05 (↑75.4%)	36.25 (↑79.3%)	**68.00** (↓7.4%)	0.00 (↑inf%)	24.44 (↓1.9%)
		34B	6.17	20.60	26.59	29.04	28.89	22.89	65.00	63.00	2.00	23.98
	DeepSeek	7B	4.92	16.63 (↑28.4%)	8.95 (↑30.3%)	40.44	26.97 (↑12.1%)	23.92 (↓2.0%)	20.00 (↑137.5%)	17.00 (↑141.2%)	0.00 (↑inf%)	14.50 (↑69.2%)
	DeepSeek-Coder	7B	2.99 (↑112.4%)	21.36 (↑6.6%)	11.66 (↑71.3%)	32.45 (↓34.3%)	30.23 (↑0.6%)	23.44 (↑17.7%)	47.50 (↑50.0%)	41.00 (↓61.0%)	**4.00** (↓75.0%)	24.54 (↓48.5%)
		33B	6.35	22.77	19.97	43.58	30.41	27.60	71.25	16.00	1.00	12.64
	WizardCoder	15B	0.05 (↑inf%)	22.62 (↑4.4%)	27.04 (↑4.0%)	25.76 (↑78.3%)	27.44 (↑2.4%)	20.16 (↑38.5%)	51.25 (↑46.3%)	0.00 (↑inf%)	0.00 (↑0.0%)	4.02 (↑147.5%)
		33B	8.34	23.62	**28.12**	**45.93**	28.09	**27.93**	75.00	1.00	0.00	9.95
	Mixtral	7B	4.63 (↑57.0%)	19.51 (↑13.5%)	19.49 (↑12.4%)	41.39 (↑4.3%)	**31.00** (↓3.6%)	26.63 (↑0.4%)	48.75 (↑33.3%)	2.00 (↑1500%)	0.00 (↑inf%)	7.51 (↑271.4%)
		8x7B	7.27	22.15	21.91	43.19	29.88	26.73	65.00	32.00	1.00	27.89
Close Source	GPT-3.5-Turbo	/	4.09	21.04	19.62	36.16	30.69	25.09	60.00	15.00	0.00	23.97
	GPT-4	/	**9.61**	**25.99**	20.69	44.48	24.81	23.25	**83.75**	11.00	1.00	18.62
Average			5.26	22.05	18.75	36.21	27.24	23.31	53.23	22.25	0.75	18.64

Table 4: Overall effectiveness of LLMs and baselines on BinMetric. ↑and ↓represent a relative increase and decrease between two rows.

we set `max_length` to 8192 and `max_new_tokens` to 2048. Since accuracy is prioritized over diversity in most code-related tasks, the sampling `temperature` is set to 0.1, with `top_k` and `top_p` both set to 1 for deterministic responses.

Baseline Methods. To evaluate LLMs' effectiveness, we endeavor to include as many baseline methods as possible. For static analysis tool IDA Pro [Hex-RaysSA, 2024], we use its Hex-rays decompiler for CSR, DEC, and SR tasks, though it doesn't support others. For DEC task, we evaluate three versions (1.3B, 6.7B, and 33B) of LLM4Decompile [Tan *et al.*, 2024a], a LLM fine-tuned on DeepSeek-Coder for decompilation task. For BCS task, we reproduce the methods BinT5 [Al-Kaswan *et al.*, 2023] and HexT5 [Xiong *et al.*, 2023]. For AC and AIG tasks, no existing baselines are available, and our BinMetric provides a potential baseline for future research.

4.2 Overall Effectiveness

In Table 4, we present the overall effectiveness of various LLMs across different binary analysis tasks.

Results on Binary Lifting. We evaluate binary lifting using Rouge-L and CodeBLEU to assess LLM performance in CSR and DEC, respectively. The average scores are 5.26% and 22.05%. Reconstructing call-sites without additional knowledge about callees themselves is highly challenging for general LLMs. Although GPT-4 achieves the highest CSR score (9.61%), manual inspection reveals limited useful reconstructions. Except for GPT-4, all LLMs score below the IDA Pro baseline (8.52%), with WizardCoder-15B scoring the lowest (0.05%) due to its failure to follow instructions. In contrast, DEC scores are relatively higher, suggesting that LLMs have the potential to perform the semantic mapping between disassembled code and source code. GPT-4 and DeepSeek-7B achieve the highest and lowest scores of 25.99% and 16.63%, indicating that decompilation presents consistent challenges for non-tailored models, and does not show dramatic performance variations based on model size and domain. LLM4Decompile shows slight improvements over its base model, DeepSeek-Coder, at the same scale.

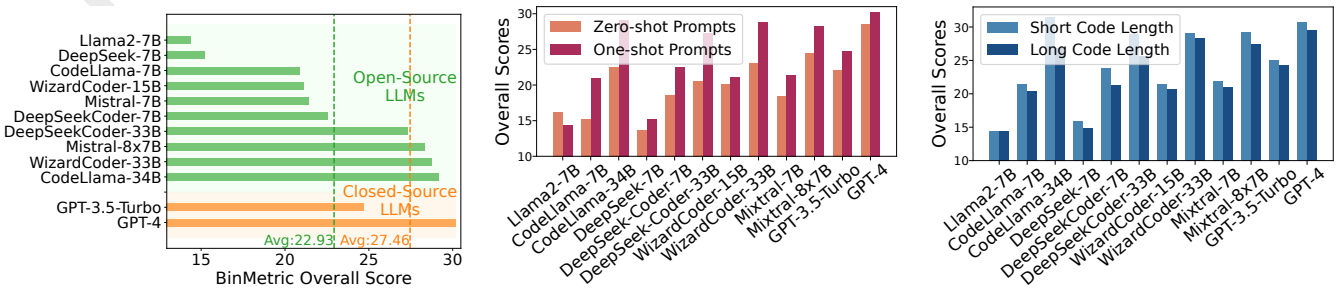
Results on Semantic Comprehension. We evaluate semantic comprehension via SR and BCS. For these two semantic understanding tasks, LLMs perform relatively well, mostly surpassing baseline methods. In SR, the average Rouge-L is 18.75%, with WizardCoder-33B scoring highest (28.12%) and Llama2-7B has struggled in it (6.51%). Furthermore, performance improved with model size (up to 42.8%) and code-specific LLMs outperform general ones by 76.45%. For BCS, we use BLEU, METEOR, and Rouge-L to provide a comprehensive evaluation. WizardCoder-33B achieves the best mean score (33.3%), while CodeLlama-7B performs the worst (22%). Interestingly, DeepSeek-Coder-7B underperforms compared to its general counterpart DeepSeek-7B, scoring 28.71% vs. 30.44%, suggesting code-specific pre-training might hinder natural language expression.

Results on Logical Analysis. The AC task evaluates LLMs' ability to identify pseudo code classes among a given candidate list. LLMs achieve an overall score of 53.23%, indicating that they can understand high-level semantics and support deeper logical reasoning to a certain extent. GPT-4 lead with 83.75%, while smaller models like Llama2-7B and DeepSeek-7B score the lowest (15.00% and 20.00%). Code-specific LLMs outperform general ones by 139.6%, and larger models show notable gains.

Results on Assembly Synthesis. AIG evaluates LLMs' ability to generate assembly code from natural language descriptions using Intel Syntax. Only compilable outputs are considered syntactically correct. CodeLlama-7B achieves the highest syntax correctness score (68%), likely due to additional pre-training on assembly code. In contrast, other models struggle, with GPT-4 scoring only 11%. Execution correctness, assessed via test cases, has an average pass rate of less than 1%. We also use Rouge-L to measure textual consistency. Llama2-7B achieves the highest score (31.58%), with manual identification revealing more similar code-blocks between its prediction and the reference. However, models like WizardCoder-15B (4.02%) and Mixtral-7B (7.51%) fail to follow instructions. These results indicate LLMs currently struggle with reconstructing low-level operations due to the complexity of assembly and lack of domain expertise.

4.3 LLMs Comparison

As shown in Figure 4, we aggregate model scores across various tasks into a radar chart to illustrate performance differences. We calculate task scores using the average of relevant metrics and determine relative scores based on the highest score in each task. Overall, GPT-4 dominates



(a) Overall scores of LLMs across 6 tasks. Dashed lines for two LLM types' average.

(b) Overall scores of LLMs with Zero-shot (Left) and One-shot (Right) prompts.

(c) Overall score of LLMs for short (Left) and long (Right) code input lengths.

Figure 3: Overall scores of LLMs, and the impact of prompts design and code length.

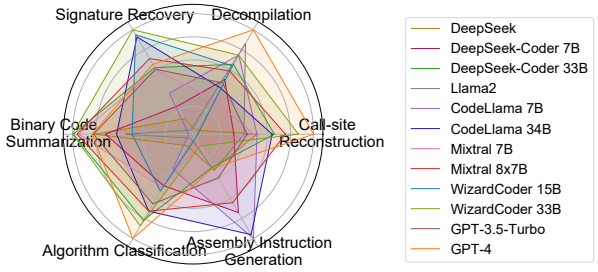


Figure 4: Relative performance against the best in each task.

the arena in DEC, CR, and AC tasks but underperforms in AIG. Trailing closely, WizardCoder-33B, CodeLlama-34B, and Mixtral-8x7B demonstrate comparable performance with distinct strengths and weaknesses.

Open-source *v.s.* Closed-source. Figure 3 (a) presents the average scores of all models across tasks. Closed-source LLMs achieve a higher mean score (27.46%) compared to open-source ones (22.93%). However, certain open-source models, like CodeLlama-34B, surpass the closed-source average. This suggests that open-source LLMs can be competitive in binary analysis, offering strong confidence in developing expert LLMs in the binary domain based on these models.

Parameter Size & Code-specific Knowledge. Our study examines two model size ranges (7-15B & 33-8x7B) and involves models with code domain knowledge from multiple LLM families. As shown in Figure 3 (a), larger models generally perform better, likely due to improved instruction-following capabilities and richer embedded knowledge. Additionally, domain-specific knowledge within the code domain enhances performance, even though code datasets rarely include decompiled binary code.

4.4 Other Factors

We examine the impact of prompts design and code length.

Prompts Design. Figure 3 (b) compares performance using zero-shot and one-shot prompts. In most cases, one-shot prompts significantly improved overall scores, with an average increase of 16.65%, emphasizing the importance of providing examples in guiding LLMs in the complex domain binary analysis. However, Llama2-7B's performance drops by 10.79% with one-shot prompts. Analysis reveals that it tends to replicate provided examples rather than generate context-based responses, likely due to its inherent capacity limitations affecting its ability to generalize from demonstrations.

Code Length. To study the impact of input code length on performance, we divide the problem items of each task into shorter and longer groups. Figure 3 (c) shows that longer code snippets result in an average score drop of 6.00% compared to shorter ones. This trend, consistent across tasks, highlights the challenges in handling extended code snippets.

5 Discussion

Potential and Limitations of LLMs. This paper presents an empirical study of various LLMs, highlighting their strengths and weaknesses in binary analysis. GPT-4 and WizardCoder-34B excel in algorithm classification and binary code summarization, showcasing their ability to extract high-level semantics and support logical reasoning. However, LLMs face challenges in call-site reconstruction and assembly generation due to the complexity of assembly instructions and limited domain knowledge. While LLMs show promise, challenges remain, and future work should address these gaps.

Future Development to Binary-Specific LLMs. The advancement of foundation LLMs and domain-specific models makes binary expert LLMs both feasible and promising. Like source code models, binary-specific LLMs should cover diverse tasks across the binary analysis lifecycle. As a multi-task benchmark leaderboard, BinMetric will be crucial for evaluating and guiding progress in this area. Binary expert LLMs should incorporate extensive binary domain knowledge during pre-training, including various assembly languages, compiler optimizations, instruction set architectures, etc., enabling a deeper grasp of code semantics and structure. Techniques like retrieval-augmented generation and specialized instruction fine-tuning can expand LLMs' binary knowledge. Our studies indicate different LLMs excel in specific areas, pointing to the potential of combining their strengths to create an expert ensemble model for binary analysis.

6 Conclusion

In this paper, we present a pioneering study on LLMs' binary analysis capabilities. We develop a standardized data pipeline, resulting in BinMetric, a comprehensive benchmark with 1,000 question items across 6 key tasks reflecting real-world challenges. Our evaluation of various LLMs highlights their strengths and limitations, showing promising potential but also significant challenges. As LLMs evolve, we believe BinMetric will serve as a crucial benchmark leaderboard for assessing and advancing progress in this field.

Acknowledgments

This work was supported in part by the Natural Science Foundation of China under Grant U20B2047, 62072421, 62002334, 62102386, and 62121002.

References

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

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

[C-Algorithms, 2024] C-Algorithms. https://github.com/TheAlgorithms/C, 2024.

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

[Chen *et al.*, 2022] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[Dai *et al.*, 2022] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers. *arXiv preprint arXiv:2212.10559*, 2022.

[David *et al.*, 2020] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[EvalPlus, 2024] EvalPlus. https://evalplus.github.io/leaderboard.html, 2024.

[for Standardization, 2011] International Organization for Standardization. *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. ISO, 2011.

[Gao *et al.*, 2023] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. How far have we gone in vulnerability detection using large language models. *arXiv preprint arXiv:2311.12420*, 2023.

[Guo *et al.*, 2024] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[Heimann, 2014] D Heimann. Ieee standard 730-2014 software quality assurance processes. *IEEE Computer Society, New York, NY, USA, IEEE Std*, 730:2014, 2014.

[Hex-RaysSA, 2024] Hex-RaysSA. "ida pro". https://www.hex-rays.com/products/ida, 2024.

[International, 2010] I. UNIX International. Dwarf debugging information format version 4. https://dwarfstd.org/doc/DWARF4.pdf, 2010.

[Jiang *et al.*, 2023] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.

[Jiang *et al.*, 2024] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[Jin *et al.*, 2022] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 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*, pages 1631–1645, 2022.

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

[Just *et al.*, 2014] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[Kong *et al.*, 2023] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, and Xin Zhou. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*, 2023.

[Lavie and Denkowski, 2009] Alon Lavie and Michael J. Denkowski. The meteor metric for automatic evaluation of machine translation. *Machine Translation*, 23(2–3):105–115, sep 2009.

[Lin, 2004] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, 2004. Association for Computational Linguistics.

[LLVM, 2024] LLVM. "clang static analyzer". https://clang-analyzer.llvm.org/, 2024.

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

[Nguyen Hung *et al.*, 2023] Thinh Nguyen Hung, Hai Nguyen Phuc, Khoa Tran Dinh, Nhan Le Tran Thanh,

Nghia To Trong, Khoa Ngo Khanh, Duy Phan The, and Hau Pham Van. Binary representation embedding and deep learning for binary code similarity detection in software security domain. In *Proceedings of the 12th International Symposium on Information and Communication Technology*, pages 785–792, 2023.

[of Standards and (NIST), 2023] National Institute of Standards and Technology (NIST). Trustworthy and responsible ai., 2023.

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

[Papineni *et al.*, 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics.

[Ren *et al.*, 2020] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

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

[Sawada *et al.*, 2023] Tomohiro Sawada, Daniel Paleka, Alexander Havrilla, Pranav Tadepalli, Paula Vidas, Alexander Kranias, John J Nay, Kshitij Gupta, and Aran Komatsuzaki. Arb: Advanced reasoning benchmark for large language models. *arXiv preprint arXiv:2307.13692*, 2023.

[Statista, 2024] Statista. https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/, 2024.

[Sutherland *et al.*, 2006] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.

[Tan *et al.*, 2024a] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[Tan *et al.*, 2024b] Zhen Tan, Alimohammad Beigi, Song Wang, Ruocheng Guo, Amrita Bhattacharjee, Bohan Jiang, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. Large language models for data annotation: A survey. *arXiv preprint arXiv:2402.13446*, 2024.

[Touvron *et al.*, 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[Wei *et al.*, 2023] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[Xiong *et al.*, 2023] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 774–786. IEEE, 2023.

[Xu *et al.*, 2017] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[Zeng *et al.*, 2024] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. Coderujb: An executable and unified java benchmark for practical programming scenarios. *arXiv preprint arXiv:2403.19287*, 2024.

[Zhang *et al.*, 2021] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.