

On Benchmarking Code LLMs for Android Malware Analysis

Yiling He
University College London
London, United Kingdom
yiling-he@ucl.ac.uk

Xinran Zheng
University College London
London, United Kingdom
xinran.zheng.23@ucl.ac.uk

Hongyu She
Zhejiang University
Hangzhou, China
hongyushe@zju.edu.cn

Zhuo Chen
Zhejiang University
Hangzhou, China
hypothesiser.hypo@zju.edu.cn

Xingzhi Qian
University College London
London, United Kingdom
xingzhi.qian.23@ucl.ac.uk

Zhan Qin
Zhejiang University
Hangzhou, China
qinzhazhan@zju.edu.cn

Lorenzo Cavallaro
University College London
London, United Kingdom
l.cavallaro@ucl.ac.uk

Abstract

Large Language Models (LLMs) have demonstrated strong capabilities in various code intelligence tasks. However, their effectiveness for Android malware analysis remains underexplored. Decompiled Android malware code presents unique challenges for analysis, due to the malicious logic being buried within a large number of functions and the frequent lack of meaningful function names.

This paper presents CAMA, a benchmarking framework designed to systematically evaluate the effectiveness of Code LLMs in Android malware analysis. CAMA specifies structured model outputs to support key malware analysis tasks, including *malicious function identification* and *malware purpose summarization*. Built on these, it integrates three domain-specific evaluation metrics—*consistency*, *fidelity*, and *semantic relevance*—enabling rigorous stability and effectiveness assessment and cross-model comparison.

We construct a benchmark dataset of 118 Android malware samples from 13 families collected in recent years, encompassing over 7.5 million distinct functions, and use CAMA to evaluate four popular open-source Code LLMs. Our experiments provide insights into how Code LLMs interpret decompiled code and quantify the sensitivity to function renaming, highlighting both their potential and current limitations in malware analysis.

CCS Concepts

• **Security and privacy** → **Software and application security**.

Keywords

Code LLM, Malware Analysis

ACM Reference Format:

Yiling He, Hongyu She, Xingzhi Qian, Xinran Zheng, Zhuo Chen, Zhan Qin, and Lorenzo Cavallaro. 2025. On Benchmarking Code LLMs for Android Malware Analysis. In *34th ACM SIGSOFT International Symposium*

on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3713081.3731745>

1 Introduction

Recent advances in Large Language Models (LLMs) have transformed natural language processing [35], and their extension to code understanding has led to the emergence of Code LLMs, i.e., specialized LLMs trained on large-scale code repositories. These models have achieved strong performance in tasks such as code generation, summarization, and repair [10, 23, 41]. Given their growing capabilities in reasoning about code, Code LLMs offer a promising direction for automating Android malware analysis, a domain where analysts must often manually inspect large volumes of low-level code to uncover malicious behavior [27].

While the potential is clear, effectively applying Code LLMs in this context remains challenging. First, decompiled Android code is often obfuscated, lacks type information, and contains incomplete control structures [30, 38], diverging sharply from the clean, structured code these models are typically trained on [50]. Second, accurate semantic interpretation is difficult due to the high-level and diverse malicious behaviors in malware [43], while the absence of reliable function-level ground truth (e.g., labeling functions as malicious or benign) further complicates evaluation [14, 16, 37]. These challenges underscore the need for a structured evaluation framework that systematically assesses and compares Code LLM performance in real-world malware analysis scenarios.

To systematically evaluate the performance of Code LLMs in Android malware analysis, we define a structured output format comprising three key elements: *function summaries*, *refined function names*, and *maliciousness scores*. While function summaries are commonly generated by Code LLMs to describe the purpose of code snippets [15], refined function names address the lack of meaningful identifiers in decompiled code and aid analysts in quickly understanding a function's intent. Additionally, maliciousness scores explicitly quantify the potential security risks associated with each function, serving as critical indicators for malicious behavior localization [9]. As these structured outputs constitute an interpretable



This work is licensed under a Creative Commons Attribution 4.0 International License.
ISSTA Companion '25, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1474-0/2025/06
<https://doi.org/10.1145/3713081.3731745>

and actionable representation, they offer potential to support both human analysts and automated systems in malware analysis [3, 12].

We consider two key malware analysis tasks to benchmark LLM performance: *malicious function identification* and *malware purpose summarization*. For each task, we propose tailored domain-specific metrics. Specifically, for malicious function identification, we define 1) *Consistency*, measuring the stability of generated function names and maliciousness scores under a self-referential process, and 2) *Fidelity*, quantifying how effectively LLM-generated maliciousness scores distinguish between benign and malicious functions. For malware purpose summarization [34], we introduce 3) *Semantic Relevance*, assessing how well aggregated function-level summaries and refined function names generated by the LLM align with ground-truth malware descriptions.

We implement our evaluation framework as CAMA and demonstrate its applicability through a detailed case study. Specifically, we construct a benchmark dataset consisting of 118 Android malware samples across 6 categories and 13 families, collectively comprising over 7.5 million distinct functions. We select 4 popular open-source Code LLMs (i.e., CodeLlama [36], StarChat [21], CodeT5 [45], and PLBART [1]) and design tailored prompting and tuning strategies to generate the desired structured outputs. This study investigates two key research questions: 1) How well do Code LLMs interpret decompiled Android code in malware analysis? and 2) How does function renaming influence the effectiveness of LLM-based analysis?

For the first question, we analyze the quality of LLM-generated outputs using our structured output format and domain-specific metrics. For the second, we systematically rename functions in the original decompiled code with the LLM-suggested names and measure the subsequent impact on evaluation metrics. Our findings reveal that while Code LLMs can generate informative summaries, their understanding of maliciousness remains limited. Among existing Code LLMs, instruction-tuned GPT-style models significantly outperform Seq2Seq models across different metrics; function renaming enhances fidelity and consistency but may reduce semantic clarity, indicating a trade-off that warrants careful consideration.

Contributions. This work makes the following key contributions:

- We propose a benchmarking framework for evaluating Code LLMs for Android malware analysis, incorporating structured outputs and downstream malware analysis tasks. We further define three domain-specific metrics—consistency, fidelity, and semantic relevance—to rigorously assess the stability and effectiveness of LLM-generated outputs.
- We construct a benchmark dataset of 118 representative Android malware samples and a total of 7,542,799 distinct functions to demonstrate our framework’s utility. Our empirical analysis provides critical insights into Code LLMs’ capabilities in interpreting decompiled code and quantifies the impact of function renaming on malware analysis outcomes.

2 Background and Related Work

2.1 Code Large Language Models

Code LLM refers to large language models specifically trained on programming-related data to assist with coding tasks. These models are pre-trained on extensive code repositories [5], documentation [18], and other technical resources [32], equipping them with

Model	Style	Architecture	Java [*]	Inst. [†]
CodeT5 [45]	T5	Encoder-Decoder	✓	✗
PLBART [1]	BART	Encoder-Decoder	✓	✗
CodeLlama [36]	GPT	Decoder-only	✓	✓
StarChat [21]	GPT	Decoder-only	✓	✓

^{*} Whether the model has been trained on datasets that include Java code.

[†] Whether the model uses instruction tuning to follow task-specific prompts.

Table 1: Selected Code LLMs for code summarization.

a strong understanding of syntax, semantics, and programming patterns. When fine-tuned with different datasets or optimization techniques, Code LLMs can be tailored to excel in specific tasks, such as code completion, translation, and summarization, across multiple programming languages [7, 19, 44]. These capabilities make them valuable tools for automating and streamlining various aspects of the software development process.

Code Summary Models. Code summarization aims to automatically generate concise and meaningful natural-language descriptions of code snippets. Traditional Seq2Seq models, such as CodeT5 [45] and PLBART [1], employ sequence-to-sequence architectures trained on large-scale paired datasets consisting of source code and human-written descriptions. In contrast, instruction-tuned models, such as CodeLlama [36] and StarCoder [21], incorporate additional fine-tuning with structured prompts and task-specific instructions, enabling them to generate more context-aware and adaptive code summaries. In this paper, we select the four models listed in Table 1 as they are widely used, open-source, and explicitly designed for code summarization tasks. Additionally, since all four models have exposure to Java code [15, 20, 26], they are suited for analyzing decompiled Android applications.

2.2 Learning-based Malware Analysis

Traditional machine learning (ML) based approaches primarily focus on coarse-grained malware analysis, such as family classification and benign-malicious identification [4, 28]. However, fine-grained analysis is essential for deeper malware understanding, moving beyond simple classification [9]. Recent works explore plugin-based or post-hoc methods, such as explainable AI (XAI) techniques, to extend ML models for interpretable malware analysis. These approaches have been applied to malicious snippet detection [13, 25], function identification [14], and behavioral modeling [12], providing insights into why a model detects malware.

LLM-powered Analysis. Recent efforts have explored LLM-powered malware detection, primarily operating within the established pipeline of conventional classifiers and leveraging LLMs in two ways: 1) querying LLMs with original features to generate detection outputs [22], and 2) using LLMs to encode text-based semantic representations that enrich traditional feature spaces [49]. More advanced approaches leverage GPT-4o-mini’s code summarization capabilities [42], improving malware detection via program slicing techniques and multi-tiered factual checking [34]. Despite these advancements, the effectiveness of Code LLMs in fine-grained analysis remains uncharted, largely due to the lack of ground truth. Our work goes beyond classification and complements existing studies by systematically benchmarking open-source Code LLMs on both function-level and app-level malware analysis tasks.

3 Our Evaluation Framework

We propose a benchmarking framework, named CAMA, for systematic evaluation of Code LLMs in Android malware analysis. In this section, we introduce the overview and technical details.

3.1 Overview

As illustrated in Figure 1, our benchmarking framework is structured into three main stages: dataset preprocessing, model adaptation of Code LLMs, and downstream malware analysis.

- **Dataset Preprocessing.** We first build a representative benchmark dataset by collecting Android malware samples across different malware categories and families. The reverse engineering tool Androguard [8] is used to generate decompiled Java functions for each APK. To ensure function diversity and representativeness, we apply a **category-wise de-duplication** based on APK size and the number of extracted methods. Since APKs within the same malware category often exhibit only minor variations, this step helps eliminate near-duplicate samples.
- **Model Adaptation of Code LLMs.** Next, we carefully design prompting strategies and tuning procedures for the evaluated code LLMs. Our primary goal is to guide these models to produce **structured outputs**, specifically consisting of 1) suggestions of refined **method names**, 2) concise and meaningful **function summaries**, and 3) **maliciousness scores** indicating potential harmfulness. Such structured outputs facilitate targeted and interpretable malware analysis.
- **Downstream Malware Analysis Tasks.** We leverage the structured outputs from the LLMs to define two downstream analysis tasks essential for malware characterization: 1) **Malicious Function Identification**, where we utilize the maliciousness scores as filters to pinpoint malicious functions within Android apps, enabling analysts to efficiently locate suspicious code segments; and 2) **Malware Purpose Summarization**, where we aggregate function-level summaries into comprehensive prompts, supporting the automatic generation of concise malware descriptions detailing their overall malicious objectives and behavior.

Within this structure, we design three domain-specific metrics to evaluate model effectiveness in generating structured outputs for downstream tasks. We introduce details of the model adaptation in Section 3.2 and the three metrics in Section 3.3.

3.2 Prompting and Tuning

To effectively leverage code LLMs for Android malware analysis, we design prompting strategies and tuning mechanisms that guide models to generate the structured outputs. We adopt two complementary approaches: **prompt engineering** for instruction-tuned models and **instruction tuning** for text-to-text models.

3.2.1 Prompt Engineering. Prompt engineering involves designing effective input templates to elicit structured responses from models. Instruction-tuned models such as StarChat and CodeLlama¹ are particularly suitable for this approach, as they are optimized for instruction-following and structured generation tasks [46]. To elicit

consistent outputs that include function descriptions, name suggestions, and maliciousness scores, we design prompts that adhere to several key principles:

- **Instruction Blocks:** We wrap the main task instruction using special tokens [INST] and [/INST], following each model's best practices for instruction prompting.
- **Code Delimiters:** Decompiled function code is enclosed between [FUNC] and [/FUNC] tokens to distinguish it from the rest of the prompt and emphasize it as the primary input.
- **Role Context:** The task is contextualized from the perspective of a *cybersecurity expert* analyzing decompiled Android functions, encouraging the model to reason with a security mindset.
- **Structured Requirements:** The instruction clearly specifies that the model should return three structured outputs. The expected response is explicitly described in the prompt (see Output Requirements I–III below).

Prompt I. Structured Function Summarization

```
[INST] You are a cybersecurity expert specializing in reverse engineering and malware analysis. Your task is to analyze a decompiled Android function and generate a structured function summary based on the following aspects :
  1. Function Summary : {summary_requirement}
  2. Suggested Function Name : {name_requirement}
  3. Malicious Score(0-10) : {score_requirement} [/INST]
[FUNC] {decompiled_code} [/FUNC]
```

Output Requirement I. Summary

<Provide a brief, high-level description of what this function does. Summarize its purpose, key operations, and intent.>

Output Requirement II. Name

<Suggest a clearer, more descriptive function name that accurately represents its behavior.>

Output Requirement III. Score

<Rate the function's maliciousness on a scale from 0 to 10, where:
 0 - Benign : No suspicious activity.
 1-3 - Potentially Safe but Risky : Performs sensitive actions but could be legitimate.
 4-6 - Suspicious : Uses permissions or techniques common in malware.
 7-10 - Highly Malicious : Strong indicators of malware behavior.>

3.2.2 Instruction Tuning. Since models like CodeT5 and PLBART are pretrained for general-purpose code summarization, they inherently lack the ability to generate function names or maliciousness scores. For instance, as shown in Figure 2, when tasked with generating a function summary alongside a refined function name, CodeT5 would produce off-topic and unstructured responses, failing to generate the expected fields. To bridge this gap, we apply instruction tuning using task-specific data.

- **Function Name Prediction:** We modify the training data by replacing function names with a placeholder (unk_function). The model is then fine-tuned to predict the actual function name based on the surrounding code. This adaptation allows CodeT5

¹We use CodeLlama-Instruct and StarChat-Beta, the instruction-tuned variants of CodeLlama and StarCoder, respectively. These models are optimized for instruction-following code summarization, making them better suited for our task.

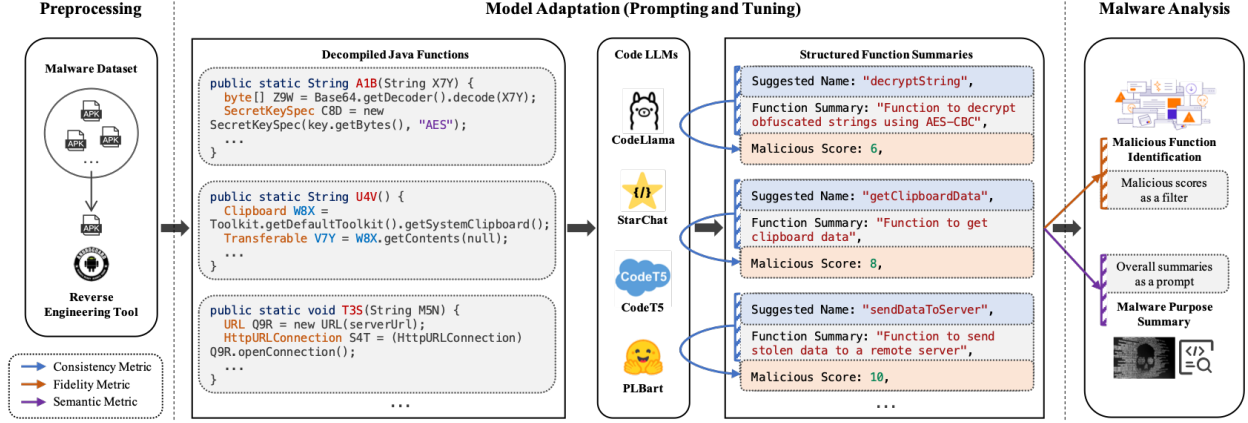
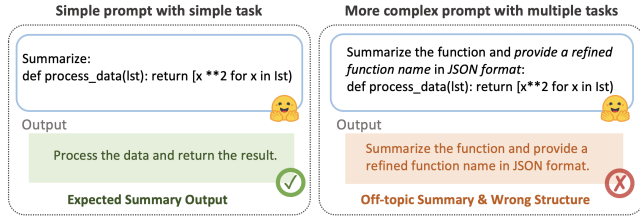


Figure 1: Evaluation pipeline of CAMA.

Figure 2: Demonstration of the limited capability of CodeT5² in generating a meaningful output when additional requirements are specified.

and PLBART to suggest meaningful function names instead of generic or incomplete descriptions.

- **Maliciousness Score Prediction:** Since large-scale ground truth labels for maliciousness scores are unavailable, we introduce a two-step approach. First, we use a tuned model to generate structured summaries and function names. Then, we leverage larger models (e.g., GPT-4 and DeepSeek [11]) to infer maliciousness scores based on the generated summaries and function names.

This hierarchical approach allows us to enhance the structured output capabilities of CodeT5 and PLBART while leveraging more powerful models for tasks that require higher-level reasoning, such as estimating maliciousness scores.

3.3 Domain-Specific Metrics

To rigorously assess the performance of code LLMs in Android malware analysis, we define three domain-specific evaluation metrics: **consistency**, **fidelity**, and **semantic relevance**. These metrics quantitatively measure the effectiveness of structured outputs at different levels—individual function analysis, malware classification, and overall application characterization.

Notations. Given an Android application \mathcal{A} composed of a set of decompiled functions $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$, our goal is to evaluate the structured outputs generated by a target code LLM (denoted as G), including function summary $S(f)$, refined function name $N(f)$, and

maliciousness score $M(f)$. We formally define the *structured output* for function f as $O(f) = S(f) \oplus N(f) \oplus M(f)$, which encapsulates all three elements generated by the LLM³. For specific evaluation tasks, we define the *function descriptor* as $D(f) = S(f) \oplus N(f)$, which serves as the interpretable textual function representation (without including its numerical measure) and is particularly relevant in tasks that focus on function-level understanding and classification.

3.3.1 Consistency-based Metric. The consistency metric measures the internal stability of the LLM's structured outputs by checking whether the model's predictions contradict each other when examined under a self-referential process. We define two forms of consistency, i.e., *maliciousness consistency* and *name consistency*.

Maliciousness Consistency. This metric evaluates whether the maliciousness scores generated by the LLM from raw decompiled code align with those produced when the model is queried with structured descriptors (*function summaries and suggested names*). Formally, for each function f , we obtain:

- $M_{\text{raw}}(f)$, the original maliciousness score, obtained by directly querying the target model using the decompiled function.
- $M_{\text{des}}(f)$, the descriptor-based maliciousness score, obtained by prompting the same model with $D(f)$.

Prompt II. Descriptor-based Maliciousness Score

Task: Given a function descriptor, $\{\text{score_requirement}\}$
Input: A function descriptor: $\{D_{\text{raw}}(f) = S_{\text{raw}}(f) \oplus N_{\text{raw}}(f)\}$
Output: A numerical maliciousness score between 0 and 10, where 10 represents highly malicious behavior.

To measure consistency, we first normalize the score vectors over all functions in an application ($f \in \mathcal{A}$) into valid probability distributions (non-negative and summing to 1), obtaining M'_{raw} and M'_{des} . Then we compute the distributional divergence using Jensen-Shannon Divergence (JSD):

$$\text{JSD}(M'_{\text{raw}}, M'_{\text{des}}) = \frac{1}{2} D_{\text{KL}}(M'_{\text{raw}} \| M_{\text{avg}}) + \frac{1}{2} D_{\text{KL}}(M'_{\text{des}} \| M_{\text{avg}}), \quad (1)$$

²<https://huggingface.co/Salesforce/codet5-base-multi-sum>

³The operator \oplus denotes concatenation and is used consistently throughout the paper.

where M_{avg} is the average distribution and $D_{\text{KL}}(P||Q)$ is the Kullback-Leibler divergence:

$$M_{\text{avg}} = \frac{1}{2}(M'_{\text{raw}} + M'_{\text{des}}), D_{\text{KL}}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}. \quad (2)$$

Finally, we normalize the JSD to (0, 1) and define:

$$\text{MCS} = 1 - \frac{\text{JSD}(M'_{\text{raw}}, M'_{\text{des}})}{\log 2}. \quad (3)$$

A higher maliciousness consistency score (MCS) indicates higher consistency, meaning that the structured outputs retain the function's security-relevant information.

Name Consistency. This metric assesses whether the suggested function name remains stable when the LLM is prompted with its own function summary. For each function f , the LLM generates:

- $N_{\text{raw}}(f)$, the initial function name suggested as part of the structured output $O(f)$.
- $N_{\text{reg}}(f)$, a new function name generated when the LLM is re-prompted with its own function summary $S(f)$.

Prompt III. Re-generated Function Name

Task: Given a function summary, $\{\text{name_requirement}\}$
Input: A function summary: $\{S_{\text{raw}}(f)\}$
Output: A concise, descriptive function name.

To quantify name consistency, we compute the normalized edit distance between the original and revised function names:

$$\text{NCS} = 1 - \frac{\text{EditDistance}(N_{\text{raw}}, N_{\text{reg}})}{\max(|N_{\text{raw}}|, |N_{\text{reg}}|)}, \quad (4)$$

where $|n|$ represents the length of n , and EditDistance is the Levenshtein distance, which counts the minimum number of character-level insertions, deletions, or substitutions required to transform N_{raw} into N_{reg} . The result is normalized to (0, 1) by the length of the longer function name to ensure comparability across different naming conventions.

A higher name consistency score (NCS) indicates greater stability in function name generation, suggesting that the model consistently associates summaries with the same function identity.

3.3.2 Fidelity-based Metric. The fidelity metric assesses the degree to which function-level structured outputs contribute to *malicious function identification*. Inspired by explainable AI (XAI) evaluation techniques [47], we define fidelity in terms of the impact of function removals on malware classification performance.

Given a malware classifier C , which takes the function descriptor as input and predicts a malware category \hat{y} , we measure classification confidence before and after removing the *top-k most malicious* function summaries. Formally, let:

$$p_{\text{full}} = C(D(f_1) \oplus D(f_2) \oplus \dots \oplus D(f_n)) \quad (5)$$

be the malware classification probability for an application before removal. After removing the top- k most malicious function features ranked by maliciousness, the new classification probability is:

$$p_{\text{red}(k)} = C\left(\bigoplus_{f \notin \mathcal{F}_k} D(f)\right), \quad (6)$$

where $\mathcal{F}_k = \{f_i \in \mathcal{F} \mid M(f_i) \text{ is among the top } k\}$ is the set of k most malicious functions. The maliciousness-based fidelity score (MFS) is then computed as the *relative drop in confidence*:

$$\text{MFS}_{(k)} = \frac{p_{\text{full}}[\hat{y}] - p_{\text{red}(k)}[\hat{y}]}{p_{\text{full}}[\hat{y}]}. \quad (7)$$

A higher fidelity score indicates that structured outputs effectively encode function-level characteristics for malware classification, as the maliciousness-based descriptor removal leads to a significant drop in the classifier's confidence for the predicted class.

3.3.3 Semantic-based Metric. The semantic metric evaluates how function outputs contribute to accurate application-level *malware purpose descriptions*. Adopting approaches from automatic machine translation evaluation [29], we measure the similarity between LLM-generated malware descriptions and reference descriptions.

Given a set of top- v malicious function outputs, where v varies based on the target LLM's context window, we firstly generate

$$A_{\text{LLM}} = G_{\text{app}}(O(f_1) \oplus O(f_2) \oplus \dots \oplus O(f_v)), \quad (8)$$

where G_{app} is the target code LLM prompted to generate a high-level malware description. This approach mimics *context slicing*, but leverages LLM outputs (i.e., maliciousness scores) instead of heuristic-based methods such as sensitive API filtering, which can often be incomplete or overlook critical behaviors. Specifically, the prompt for G_{app} is defined as follows.

Prompt IV. Application Purpose Description

Task: Given the structured function-level analyses, generate a concise and comprehensive description of the overall application's purpose.
Input: A set of top- v malicious functions: $\{\text{Function Summary } S(f), \text{ Refined Function Name } N(f), \text{ Maliciousness Score } M(f)\}$
Output: An application purpose description summarizing the app's behavior and potential security risks.

We compare A_{LLM} against the reference malware description A_{GT} using three widely used text similarity metrics. BLEU [31] measures n-gram precision between A_{LLM} and A_{GT} :

$$\text{BLEU}(A_{\text{LLM}}, A_{\text{GT}}) = \exp\left(\sum_{n=1}^N w_n \log p_n\right) \quad (9)$$

where p_n is n-gram precision and w_n are weighting factors. Additionally, we report METEOR [6], which incorporates synonym matching and recall, and ROUGE-L [24], which evaluates the longest common subsequence overlap.

A higher BLEU, METEOR, or ROUGE-L score indicates stronger alignment between the LLM-generated description and the reference description, validating the semantic relevance of function outputs in capturing malware behavior.

4 Benchmarking Results

We conduct experiments guided by two key research questions:

- RQ1:** How well do Code LLMs understand decompiled code for malware analysis tasks?
- RQ2:** How does function renaming affect their performance (i.e., can models self-repair based on their own suggested names)?

	Consistency		Fidelity			Semantic Relevance		
	MCS	NCS	MFS ₍₂₎	MFS ₍₅₎	MFS ₍₈₎	BLEU	METEOR	ROUGE-L
CodeT5	N/A	0.233 ± 0.04	0.332 ± 0.30	0.125 ± 0.22	0.396 ± 0.32	0.059 ± 0.04	0.083 ± 0.04	0.186 ± 0.04
PLBART	N/A	0.499 ± 0.05	0.033 ± 0.14	0.031 ± 0.11	0.065 ± 0.14	0.137 ± 0.03	0.185 ± 0.05	0.228 ± 0.04
CodeLlama	0.381 ± 0.03	0.628 ± 0.04	0.158 ± 0.27	0.159 ± 0.27	0.113 ± 0.25	0.175 ± 0.05	0.247 ± 0.08	0.271 ± 0.06
StarChat	0.813 ± 0.02	0.575 ± 0.02	0.111 ± 0.20	0.254 ± 0.30	0.275 ± 0.33	0.176 ± 0.05	0.273 ± 0.09	0.272 ± 0.06
CodeLlama+	0.357 ↓6.30%	0.677 ↑7.80%	0.485 ↑207.0%	0.451 ↑183.7%	0.440 ↑289.4%	0.171 ↓2.29%	0.219 ↓11.34%	0.270 ↓0.37%
StarChat+	0.828 ↑1.85%	0.582 ↑1.22%	0.298 ↑168.5%	0.351 ↑38.19%	0.726 ↑164.0%	0.172 ↓2.27%	0.246 ↓9.89%	0.274 ↑0.74%

* Rows 1–4 correspond to RQ1, evaluating the performance of all four models on decompiled code. Results are reported as mean ± standard deviation.

* Rows 5–6 correspond to RQ2, assessing the impact of function renaming by replacing original names with LLM-suggested ones. CodeT5 and PLBART are excluded due to limited name generation capability—they often replicate names from the input code. Results are reported as mean values with relative improvement ratios.

Table 2: Benchmarking results.

4.1 Experimental Setup

Dataset Selection. We use the LAMD [34] dataset⁴, which provides Android malware samples with high-quality GPT-4-generated ground-truth application purpose summaries, making it well-suited for evaluating semantic relevance. To reduce redundancy, we perform de-duplication by filtering out near-identical APKs within each malware category, resulting in 118 APKs across 6 categories (Adware, Backdoor, PUA, Riskware, Scareware, Trojan) and 13 families. All APKs are decompiled using Androguard, resulting in a total of 7,542,799 decompiled functions across the dataset.

Implementation Details. For all selected models in Table 1, we use their official implementations from the Hugging Face Hub⁵. For models not originally instruction-tuned (i.e., CodeT5 and PLBART), we perform additional tuning on Java functions from their pre-training datasets. Each model is fine-tuned for 3 epochs, which we find sufficient to produce the structured outputs. The maliciousness score prediction of these two models is assisted by a locally deployed DeepSeek-R1-Distill-Llama-70B. In the fidelity evaluation, we use LightGBM [17] as the malware category classifier C , ensuring high reliability with an accuracy above 0.95. To assess the effect of removing suspicious code, we experiment with top- k values of 2, 5, and 8. In the semantic relevance evaluation, to ensure stylistic consistency, we prompt all models to begin their outputs with the phrase: “*This application appears to...*”, matching the format used in the ground truth. We set top- v based on model context limits: 4K tokens for CodeLlama, 8K for StarChat, and 1K for CodeT5 and PLBART. For BLEU-based evaluation, we use 2-gram precision, which is more appropriate for evaluating short summaries.

The generated outputs are made open-source⁶. Our overall experimental results are summarized in Table 2. In the following sections, we provide a detailed analysis of each research question: RQ1 in Section 4.2 and RQ2 in Section 4.3.

4.2 RQ1 - Decompiled Code

This experiment investigates how well Code LLMs interpret decompiled Android code for malware analysis. We evaluate their ability to generate the structured outputs and analyze their effectiveness using the three domain-specific metrics.

For maliciousness consistency, only CodeLlama and StarChat are evaluated, as CodeT5 and PLBART rely on external models for score generation. Among the two, StarChat achieves a notably higher score (over twice that of CodeLlama), suggesting a better understanding of high-level malware semantics. For name consistency, CodeLlama performs best, with StarChat following closely. Both models outperform the Seq2Seq baselines, reinforcing the observation that instruction-tuned, GPT-style models exhibit greater stability in structured output generation. Among the Seq2Seq models, PLBART outperforms CodeT5 by approximately 114%, likely due to its improved alignment between code and natural language.

In downstream evaluations, we observe clear performance differences across models in both fidelity and semantic relevance. StarChat consistently outperforms the others, demonstrating a stronger ability to assign meaningful maliciousness scores and produce high-level malware descriptions—results that align with its superior consistency metrics. This superior performance is likely driven by its larger model size and the StarCoder architecture, which emphasizes multilingual understanding and instruction-following. CodeLlama performs competitively, especially excelling in top-2 function removal and producing stylistically aligned summaries, suggesting it effectively captures the most critical functions but is less robust than StarChat when evaluating broader function sets.

Among the Seq2Seq models, PLBART shows better performance in semantic relevance, benefiting from its BART-based architecture, which favors fluent and coherent natural language generation. However, PLBART notably underperforms CodeT5 in fidelity, while both models’ maliciousness scores are generated externally by the same larger models. This difference arises because CodeT5’s summaries, though less fluent, contain more descriptions that better highlight critical code features, allowing the external scoring model to produce more discriminative maliciousness scores. Nevertheless, both Seq2Seq models exhibit limited stylistic control. For instance, even when explicitly prompted to begin with “*This application appears to...*”, they frequently prepend generic phrases like “*This function...*”, revealing limited control over stylistic constraints.

These findings highlight a fundamental **tradeoff between linguistic fluency and semantic precision** in LLM-generated outputs. Our results also reinforce the **superiority of instruction-tuned GPT-style models (especially StarChat)** for generating both accurate and interpretable outputs in fine-grained malware analysis tasks.

⁴LAMD dataset collected in the 2020s: <https://zenodo.org/records/14884736>

⁵<https://huggingface.co/meta-llama/CodeLlama-7b-Instruct-hf>, HuggingFaceH4/starchat-beta, Salesforce/codet5-base-multi-sum, uclanlp/plbart-base)

⁶Our CAMA dataset: <https://zenodo.org/records/15155917>

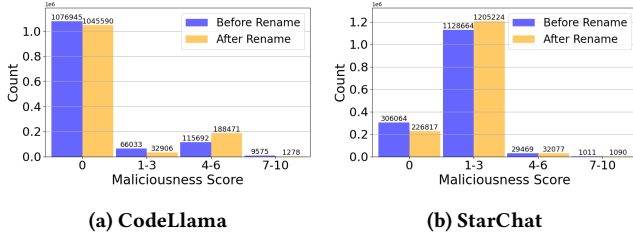


Figure 3: Maliciousness score distributions before and after function renaming. For both models, refined function names lead to more scores concentrated in the middle range.

4.3 RQ2 - Function Naming

This experiment investigates whether replacing original decompiled function names with LLM-suggested names affects model performance. The goal is to understand whether LLMs can improve their own reasoning and potentially providing more meaningful input for subsequent predictions.

We compare each model’s outputs before and after replacing function names, modifying only those names that differ from the originals, while keeping all other code aspects unchanged. CodeT5 and PLBART are excluded from RQ2 due to their limited function name refinement capability. These models often fail to produce meaningful or distinct name suggestions. For instance, 61.75% of PLBART’s generated names are exact copies of the names found in the decompiled code. Even among the remaining cases, many suggestions are only trivially modified (e.g., renaming `set_b` to `set_a`), which lacks usefulness in evaluating the impact of renaming.

Results show that replacing original function names with LLM-suggested names notably benefits fidelity, with both CodeLlama+ and StarChat+ showing substantial improvements. This suggests that improved naming significantly helps the models better prioritize and identify critical malicious functions. Consistency also improves moderately, indicating enhanced stability in model predictions when meaningful names are used. However, semantic relevance slightly decreases after renaming, likely because renaming leads to a convergence of maliciousness scores around the mid-range (Figure 3), reducing the distinctiveness of highly ranked functions when aggregated into malware descriptions.

Overall, these findings indicate that LLM-based function renaming effectively **enhances function-wise consistency and fidelity metrics**, but may **require careful handling to avoid diluting high-level semantic clarity**. To mitigate this issue, future improvements could focus on calibrating the scores to better reflect model confidence and explicitly encoding more robust knowledge about malware semantics.

5 Discussion

While CAMA enables structured evaluation of Code LLMs in malware analysis, it also exposes a *fundamental challenge*: the scarcity of reliable ground truth at the function and behavior levels. Our function-level evaluation relies on counterfactual fidelity-based methods, while APK-level summarization adopts techniques from LLM-driven malware detection [34], which leverage program slicing and prompt large models like GPT-4. Though practical, these

surrogate approaches can introduce noise and bias [40], which underscores the pressing need for high-quality, fine-grained ground truth malware datasets to advance trustworthy evaluation.

CAMA opens the door to *broader research directions*. For example, it can support studies on malware concept drift [33], enabling evaluation of whether Code LLM-based analyses generalize to evolving threats. Beyond Code LLMs and Android malware, the framework is adaptable to assess a wide range of approaches, as long as they target core sub-tasks including function summarization, naming, and maliciousness estimation. Beyond benchmarking, CAMA also supports *practical applications*: it can guide the selection, pretraining, or fine-tuning of Code LLMs specifically for malware tasks [2, 39, 48]. Its structured outputs, particularly maliciousness scores, can be used to prioritize suspicious functions, improving the precision of traditional malware classifiers [4, 13].

6 Conclusion

This paper introduces CAMA, a benchmarking framework for systematically evaluating the effectiveness of open-source Code LLMs in Android malware analysis. We define a structured output format aligned with two key analysis tasks: malicious function identification and malware purpose summarization. To address the lack of fine-grained ground truth, we propose three domain-specific evaluation metrics, enabling rigorous assessment of LLM-generated outputs. Our benchmarking results reveal both the potential and current limitations of Code LLMs. CAMA provides a foundation for future work to select and adapt Code LLMs for malware analysis, improving their effectiveness in downstream tasks such as family classification and behavioral explanation.

References

- [1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.
- [2] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271.
- [3] Simone Aonzo, Yufei Han, Alessandro Mantovani, and Davide Balzarotti. 2023. Humans vs. Machines in Malware Classification. In *32nd USENIX Security Symposium (USENIX Security 23)*.
- [4] Daniel Arp, Michael Spreitzerbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, Vol. 14. 23–26.
- [5] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.
- [6] Satandeep Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.
- [7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.
- [8] Anthony Desnos. [n. d.]. Androguard. <https://github.com/androguard/androguard> 2020.
- [9] Evan Downing, Yisroel Mirsky, Kyuhong Park, and Wenke Lee. 2021. DeepReflect: Discovering Malicious Functionality through Binary Reconstruction. In *30th USENIX Security Symposium (USENIX Security 21)*.
- [10] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?

- In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 829–846.
- [11] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
 - [12] Yiling He, Junchi Lei, Zhan Qin, and Kui Ren. 2024. DREAM: Combating Concept Drift with Explanatory Detection and Adaptation in Malware Classification. *arXiv preprint arXiv:2405.04095* (2024).
 - [13] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2023. MsDroid: Identifying Malicious Snippets for Android Malware Detection. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2023), 2025–2039.
 - [14] Yiling He, Jian Lou, Zhan Qin, and Kui Ren. 2023. Finer: Enhancing state-of-the-art classifiers with feature attribution to facilitate security analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 416–430.
 - [15] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
 - [16] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. Doug Tygar. 2015. Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security, AISec 2015, Denver, Colorado, USA, October 16, 2015*. doi:10.1145/2808769.2808780
 - [17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiqi Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).
 - [18] Junaed Younus Khan, Md Tawkat Islam Khondaker, Gias Uddin, and Anindya Iqbal. 2021. Automatic detection of five api documentation smells: Practitioners' perspectives. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 318–329.
 - [19] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004* (2023).
 - [20] Denis Kocetkov, Raymond Li, Li Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2023. The Stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research* (2023).
 - [21] Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlessinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research* (2023).
 - [22] Yao Li, Sen Fang, Tao Zhang, and Haipeng Cai. 2024. Enhancing Android Malware Detection: The Influence of ChatGPT on Decision-centric Task. *arXiv preprint arXiv:2410.04352* (2024).
 - [23] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
 - [24] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
 - [25] Zhijie Liu, Liang Feng Zhang, and Yutian Tang. 2023. Enhancing malware detection for android apps: Detecting fine-granularity malicious components. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1212–1224.
 - [26] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.
 - [27] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*. 2727–2745.
 - [28] E Mariconti, L Onwuzurike, P Andriotis, E De Cristofaro, G Ross, and G Stringhini. 2017. MamaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
 - [29] Nitika Mathur, Timothy Baldwin, and Trevor Cohn. 2020. Tangled up in BLEU: Reevaluating the Evaluation of Automatic Machine Translation Evaluation Metrics. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4984–4997.
 - [30] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 24–35.
 - [31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
 - [32] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.
 - [33] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordane, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX security symposium (USENIX Security 19)*. 729–746.
 - [34] Xingzhi Qian, Xinran Zheng, Yiling He, Shuo Yang, and Lorenzo Cavallaro. 2025. LAMD: Context-driven Android Malware Detection and Classification with LLMs. *arXiv preprint arXiv:2502.13055* (2025).
 - [35] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.
 - [36] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
 - [37] Aleieldin Salem. 2021. Towards Accurate Labeling of Android Apps for Reliable Malware Detection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21)*. doi:10.1145/3422337.3447849
 - [38] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2024. How Far Have We Gone in Binary Code Understanding Using Large Language Models. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12.
 - [39] Shuo Shao, Yiming Li, Hongwei Yao, Yiling He, Zhan Qin, and Kui Ren. 2025. Explanation as a Watermark: Towards Harmless and Multi-bit Model Ownership Verification via Watermarking Feature Attribution. In *Network and Distributed System Security Symposium (NDSS)*.
 - [40] Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in language models for code intelligence: A taxonomy and survey. *arXiv preprint arXiv:2310.17903* (2023).
 - [41] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, et al. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. In *Findings of the Association for Computational Linguistics ACL 2024*. 4173–4198.
 - [42] Brandon J Walton, Mst Eshita Khatun, James M Ghawaly, and Aisha Ali-Gombe. 2025. Exploring Large Language Models for Semantic Analysis and Categorization of Android Malware. *arXiv preprint arXiv:2501.04848* (2025).
 - [43] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. 2022. MalRad: Demystifying Android malware in the new era. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–27.
 - [44] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.
 - [45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.
 - [46] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*.
 - [47] Fan Yang, Mengnan Du, and Xia Hu. 2019. Evaluating explanation without ground truth in interpretable machine learning. *arXiv preprint arXiv:1907.06831* (2019).
 - [48] Yuchen Yang, Hongwei Yao, Bingrun Yang, Yiling He, Yiming Li, Tianwei Zhang, Zhan Qin, and Kui Ren. 2024. Tapi: Towards target-specific and adversarial prompt injection against code llms. *arXiv preprint arXiv:2407.09164* (2024).
 - [49] Wenxiang Zhao, Juntao Wu, and Zhaoyi Meng. 2025. Appoet: Large language model based android malware detection via multi-view prompt engineering. *Expert Systems with Applications* 262 (2025), 125546.
 - [50] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.