ENGINEERING Information Technology & Electronic Engineering
https://www.academax.com/EITEE; https://www.engineering.org.cn
https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=11344065
E-mail: eng.itee@zju.edu.cn

Engineering Journals
Available on IEEE Xplore®

Research Article
https://doi.org/10.1631/ENG.ITEE.2026.0017

Benchmarking large language models for binary function name prediction

Yunxiang GE[1], Bing XIA[1,3]✉, Yong DING[2], Wenbo LIU[1,2]

[1]*School of Cyberspace Security, Zhongyuan University of Technology, Zhengzhou 450007, China*
[2]*School of Computer and Information Security, Guilin University of Electronic Science and Technology, Guilin 541004, China*
[3]*Henan Key Laboratory of Computational Intelligence and Intelligent System, Zhengzhou 450007, China*

Abstract: In practical reverse engineering, stripped and optimized binaries lack high-level semantics, hindering automated function understanding. This paper adopts binary function name prediction as a benchmark to evaluate open-source large language models (LLMs) for function-level semantic inference under realistic conditions. We systematically examine key factors affecting performance, including the pretraining domain, model scale, architecture and optimization settings, prompting, and contextual information. Experiments on a large-scale multi-project binary dataset reveal clear limitations in recovering function semantics from stripped binaries. Code-oriented LLMs consistently outperform general-purpose models, while increasing the model size alone does not yield monotonic gains. We further show that realistically recoverable contextual signals, especially structurally recoverable cross-function contexts, substantially mitigate semantic sparsity and improve prediction quality. These results delineate current capability boundaries of LLM-based binary semantic understanding and suggest future directions in contextual modeling and domain-adaptive techniques for practical reverse engineering.

Key words: Binary function name prediction; Large language models (LLMs); Reverse engineering (RE); Binary code analysis

1 Introduction

Reverse engineering (RE) plays a fundamental role in software security research and serves as a critical technical foundation for key tasks such as malware analysis (Han et al., 2013), vulnerability discovery (Tian et al., 2020), and program understanding (Xia et al., 2021). In a typical RE workflow, analysts begin with binary executables and infer a program's internal logic and high-level semantics through instruction analysis, as well as control-flow and data-flow analyses. In real-world scenarios, however, binary programs are often stripped of symbol information due to intellectual property protection (Meng and Miller, 2016) or obfuscation techniques (Patrick-Evans et al., 2020). Consequently, critical semantic cues—including type information, control structures, and meaningful identifiers—are difficult to recover directly. Under such semantically sparse conditions, RE relies heavily on manual expertise, making it labor-intensive and raising the barrier to entry.

To alleviate these challenges, researchers have increasingly introduced machine learning and deep learning (DL) techniques to support RE-related tasks, such as malware classification (Song et al., 2025), vulnerability detection (Harzevili et al., 2023), and function identification (Shin et al., 2015). However, most existing approaches still rely heavily on handcrafted features or task-specific model architectures, which limits their ability to generalize across diverse programs, compiler configurations, and instruction set architectures (ISAs).

In recent years, the rapid advancement of large language models (LLMs) has opened new opportunities for RE research. Pretrained on large-scale natural language and source code corpora, LLMs have demonstrated strong contextual modeling and semantic abstraction capabilities that align with the core objective of RE—inferring high-level semantics from low-level program representations (Hu et al., 2025). Prior studies have also shown that incorporating LLMs into automated RE workflows can improve analysis efficiency and reduce manual effort (Boronat and Mustafa, 2025).

Against this backdrop, binary function name prediction

✉ Bing XIA, xiabing@zut.edu.cn
Yunxiang GE, https://orcid.org/0009-0001-0988-1233
Bing XIA, https://orcid.org/0000-0001-9467-6092
Yong DING, https://orcid.org/0000-0002-3571-7576
Wenbo LIU, https://orcid.org/0009-0001-2747-2564
CLC number: TP391

Received: Jan. 19, 2026; Revision accepted: June 30, 2026;
Crosschecked: July 14, 2026

has emerged as a representative task for evaluating automated RE techniques. In the absence of symbol information, this task examines whether a model can extract stable semantic cues from decompiled low-level representations and generate a high-level function name that reflects the underlying functionality. As such, it is widely used as a benchmark for assessing a model's capability in binary semantic recovery. Accordingly, this work investigates the following research question: under symbol-stripped (but non-obfuscated) binary settings with diverse compilation configurations, what are the capability boundaries of mainstream open-source LLMs in recovering function-level semantics?

Rather than introducing another task-specific architecture, we contribute a methodological study: a unified and realistic evaluation framework for binary function name prediction that decouples inference from evaluation and systematically stress-tests modern open-source LLMs under practical RE conditions, e.g., symbol stripping, diverse ISAs, and compiler optimizations. Building on this framework, we conduct controlled analyses of contextual variables, e.g., call-graph structure, imported application programming interfaces (APIs), and oracle symbolic cues, and distill actionable insights for designing reliable LLM-assisted RE pipelines.

Our main contributions are as follows:

1. We propose a unified evaluation framework for binary function name prediction under realistic RE settings, encompassing multiple projects, ISAs, compiler optimization levels, and symbol/debug configurations, which enables fair and systematic comparisons between general-purpose and code-oriented LLMs.

2. We conduct a comprehensive evaluation of representative open-source LLMs and analyze the impact of key factors, including the pretraining domain, model scale, input length, ISA, and compiler optimization, to characterize their performance profiles and capability boundaries.

3. We perform controlled analyses of different contextual signals, distinguishing realistically recoverable stripped-binary cues from oracle symbolic cues, and show that a structurally recoverable cross-function context is particularly effective in alleviating semantic sparsity.

2 Motivation

Although LLMs have achieved notable success in natural language and source-code understanding tasks, their effectiveness in practical binary analysis remains limited. Decompiled binaries differ substantially from the training data of LLMs in both representation and available semantic cues, while additional variability introduced by different compilers, optimization strategies, and ISAs further complicates model generalization. Consequently, the performance results obtained under a single experimental configuration provide only a partial view of practical reliability, motivating a systematic evaluation of LLM capability boundaries across realistic RE settings.

Previous work (Shang et al., 2024) has explored the application of LLMs to binary analysis tasks such as function name recovery and binary summarization, demonstrating preliminary feasibility. However, feasibility alone is insufficient for practical RE, where robustness under symbol stripping, varying compiler optimization levels, cross-architecture settings, and sensitivity to contextual information are of primary concern. Accordingly, this study focuses on delineating the capability boundaries of LLMs in realistic RE environments and adopts a unified evaluation protocol with standardized inputs and inference procedures (Fig. 1).

We adopt a two-stage evaluation protocol that strictly decouples model inference (S1) from result evaluation (S2). In S1, the model is provided only with decompiled pseudocode obtained from symbol-stripped binaries, whereas ground-truth function names are used exclusively in S2 for metric computation. Each evaluation sample is defined at the function level and consists of a single decompiled function paired with its corresponding ground-truth name. Formally, a function instance is represented as a pair (x, y), where x denotes the decompiled pseudocode after symbol stripping and y denotes the corresponding ground-truth function name. Given a model M and a unified prompt template $P(\cdot)$, the predicted function name is defined as

$$\hat{y} = M(P(x)).$$

Automatic evaluation metrics $m(\hat{y}, y)$ are computed solely based on the predicted name $\hat{y}$ and the reference label y. Under this unified setting, zero-shot inference on stripped pseudocode serves as the default baseline. Building on this baseline, we systematically vary key factors, including compilation configurations and contextual information, to analyze the stability of model performance and the shifts in their capability boundaries under realistic RE conditions.

Within this unified evaluation framework, we do not draw conclusions solely from the overall performance under a single experimental setting. Instead, we explicitly incorporate major

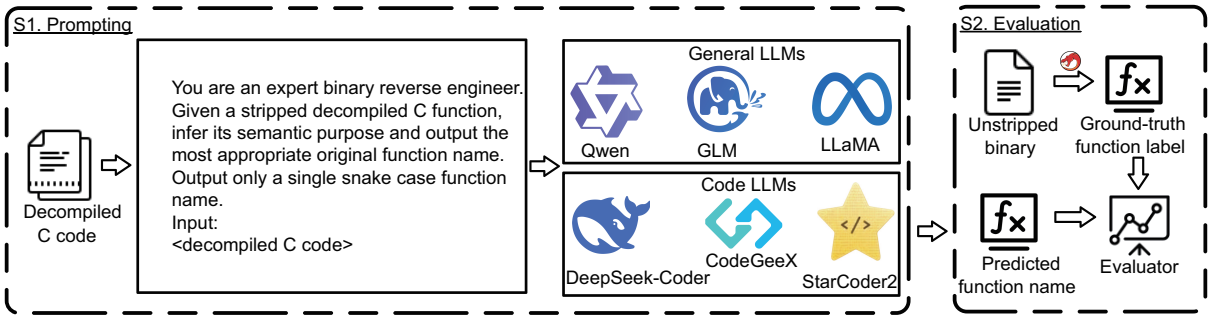

Fig. 1 Overview of the function name prediction workflow

sources of semantic distortion that commonly arise in practical RE as evaluation dimensions, including the pretraining domain, model scale, decompiled pseudocode length, ISA, and compiler optimization level. By conducting conditional analyses along these dimensions, we systematically examine the stability regimes of model behaviors under diverse environmental constraints, thereby characterizing performance profiles in realistic binary settings rather than limiting the discussion to a coarse notion of feasibility.

To address the pervasive issue of semantic sparsity in binary analysis, we further treat contextual cues as controllable analytical variables and conduct a layered investigation of their effects. Prior studies (Shang et al., 2024) have shown that richer residual symbolic information in pseudocode, such as preserved strings or identifiers, often correlates with improved performance in function name recovery. However, such correlation-based observations, which rely on information already present in the input, do not clarify which types of structured information should be prioritized when symbols are largely absent.

To this end, while keeping the model, prompt, and inference settings fixed, we incrementally introduce contextual information at different semantic levels, distinguishing realistically recoverable stripped-binary signals from oracle symbolic cues, and quantitatively compare their contributions through controlled experiments. This design helps disentangle whether performance bottlenecks stem primarily from limitations in reasoning capability or from insufficient semantic input, and further highlights the importance of a structurally recoverable cross-function context in mitigating semantic sparsity.

3 Evaluation design

3.1 Dataset

To improve domain coverage and diversify function semantics, we select eight real-world open-source projects from GitHub (GitHub, 2008), covering a wide range of application domains, including system utilities, network communication, and cryptography. To ensure data quality and reduce noise, we filter out compiler-generated functions, functions that fail to be decompiled, and duplicate samples. The resulting dataset comprises 49 935 function instances. Detailed statistics of the selected projects are provided in Section 1.1 of the supplementary materials (Table S1).

As illustrated in Fig. 2, the dataset construction pipeline consists of six main steps: (1) Source code is collected from GitHub and compiled into binary executables with debugging information under multiple ISAs (x86/x64, ARM, and MIPS) and compiler optimization levels (O0–O3). (2) Symbol tables are parsed from the unstripped binaries to extract ground-truth function names and their corresponding entry addresses. (3) The binaries are then stripped using standard architecture-specific strip tools (e.g., strip, aarch64-linux-gnu-strip, arm-linux-gnueabi-strip, and mipsel-linux-gnu-strip) to remove debugging information and most symbol information (including function and variable names), simulating realistic RE scenarios. Note that externally visible symbols required for dynamic linking (e.g., imported library functions) may remain after stripping, as is typical in real-world stripped binaries. (4) The stripped binaries are decompiled using Ghidra (version 10.1.5) in a headless batch-processing setting. Automatic analysis is enabled, and a custom Ghidra script is used to recover function boundaries, collect entry addresses, and invoke the built-in decompiler to export function-level pseudocode. No manual intervention is involved in this process. (5) Decompiled functions are aligned with their ground-truth function names based on entry addresses. (6) The matched function instances are aggregated to form the final dataset for binary function name prediction. To prevent target-name leakage in settings involving unstripped or debug-derived information, the original name of the target function is explicitly masked in the model input, including the function signature, recursive self-calls, and any direct textual references, by replacing it with an address-based placeholder.

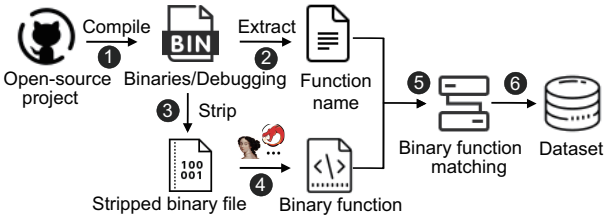

Fig. 2 Overview of the dataset construction pipeline

3.2 Setup of LLMs

We evaluate a set of representative open-source LLMs spanning general-purpose and code-specialized families. The general-purpose models include Qwen2.5 (7B/72B), LLaMA 3.1 (8B/70B), and GLM-4 (9B). The code-oriented models include Qwen2.5-Coder (0.5B/3B/7B/14B/32B), DeepSeek-Coder-V2 (16B), StarCoder2 (7B), and CodeGeeX4 (9B).

Model selection follows four criteria: (1) strong empirical performance on widely adopted benchmarks; (2) pretraining on large-scale, high-quality corpora with demonstrated code understanding capabilities; (3) coverage of a broad parameter range to examine the trade-off between model capability and computational efficiency; (4) support for local deployment, enabling fully offline and reproducible evaluation. Detailed model configurations are provided in Section 2.1 of the supplementary materials (Table S4).

Due to limitations in model context windows and the practical length of decompiled functions, we set the maximum input length to 16 384 tokens to balance input completeness with GPU memory consumption and inference efficiency; inputs exceeding this limit are truncated. In this setting, we adopt a uniform truncation strategy across all models rather than performing region-specific selection, to ensure consistency and avoid introducing additional bias. While different regions of a function (such as the function header, initial basic blocks, or later segments) encode distinct types of semantic information, truncation may inevitably discard part of the contextual information. Nevertheless, all models are evaluated under the same input constraint and on the same dataset, ensuring consistent evaluation conditions across models and settings.

Developing more effective context-preserving strategies (such as token compression or semantic-aware input reduction) while maintaining semantic fidelity, remains a promising direction for future work.

To control generation stochasticity, the temperature parameter is uniformly set to 0.4 for all models. Temperature is a standard decoding parameter that controls the randomness of token sampling, where lower values lead to more deterministic outputs and higher values increase diversity (OpenAI, 2023). Prior work has shown the usefulness of stochastic decoding for tasks with multiple valid outputs (Chen et al., 2021). Given that function name prediction may admit multiple semantically valid names (e.g., verify vs. validate), we adopt a moderate temperature as a conservative trade-off between lexical flexibility and output stability. For the function name prediction task, the maximum number of generated tokens is fixed at 128, and all remaining inference hyperparameters follow the official default recommendations. All models are evaluated under the same decoding configuration to ensure a fair comparison.

To quantify the variance introduced by stochastic decoding, we additionally repeat the evaluation on the fixed benchmark subset with multiple decoding seeds for representative models, while keeping all other settings unchanged. The corresponding mean F1, standard deviation, and 95% confidence intervals are reported in Section 2.2 of the supplementary materials (Table S5). Model weights are obtained from the Hugging Face platform (Hugging Face, 2016), and deployed locally using vLLM (vLLM Project, 2023).

3.3 Experimental setup

All experiments are conducted on a server equipped with eight NVIDIA GeForce RTX 4090 GPUs (24 GB memory per GPU, 192 GB in total) and a 32-core AMD EPYC CPU. The software environment runs on Ubuntu 22.04 LTS, with NVIDIA driver version 560.35.03 and CUDA 12.6. All experiments are executed in a Python 3.10 environment, primarily using PyTorch 2.6.0 and the Transformers library (v4.46.2). Model deployment and inference are performed using vLLM Docker (v0.6.5). Binary decompilation and pseudocode generation are carried out using Ghidra (v10.1.5).

All LLMs are evaluated under a zero-shot setting, without any fine-tuning or additional training. This setup ensures that the experimental results faithfully reflect the models' intrinsic semantic reasoning capabilities, rather than performance gains introduced by task-specific adaptation.

3.4 Metrics

In code elements such as variables, methods, and classes, highly descriptive names are crucial for improving readability and maintainability (Høst and Østvold, 2009; Allamanis et al., 2015). Among these, method naming is particularly challenging because a name is expected to concisely (often with only a few tokens) yet accurately convey the underlying functional semantics (Lawrie et al., 2006). As summarized in Section 3.1 of the supplementary materials (Table S6), prior studies on this task predominantly adopt token-level automatic evaluation metrics, including Precision, Recall, and F1 score, while some works additionally consider ROUGE-based metrics or manual assessment.

Following prior studies (Gao et al., 2021; Shang et al., 2024), we adopt token-level Precision, Recall, and F1 score as our primary evaluation metrics. Specifically, both the predicted function name and the reference name are normalized by (1) removing non-alphabetic characters, (2) converting all tokens to lowercase, (3) treating names as unordered token sets, and (4) removing duplicate tokens. The metrics are then computed based on the resulting normalized token sets.

This evaluation strategy may not fully capture semantic equivalence expressed via different lexical choices (e.g., verify vs. validate) or common naming variants (e.g., initialize vs. init). As illustrated by the representative examples in Section 5.1 of the supplementary materials (Table S10), many predictions preserve the core functional semantics despite minor lexical differences. Therefore, the reported scores should be interpreted as conservative estimates of the models' semantic prediction capability. This also suggests that top-k and semantic-level evaluation can provide a more comprehensive assessment beyond strict token-level matching. Accordingly, we supplement the main token-level results with additional top-k and semantic-oriented evaluations in Section 3 of the supplementary materials.

4 Results

This section presents a systematic evaluation of LLMs on the binary function name prediction task, aiming to characterize their performance boundaries under realistic RE conditions. We analyze the impact of several key factors on model performance, including the pretraining domain and model scale, pseudocode length, ISA, compiler optimization level, prompt design, and availability of contextual information. These factors correspond to the following research questions:

RQ1: How do the pretraining domain and model scale influence performance on binary function name prediction?

RQ2: How does decompiled pseudocode length affect the predictive capability of LLMs?

RQ3: How does performance vary across different ISAs and compiler optimization levels?

RQ4: What is the impact of prompt design strategies on model performance?

RQ5: How does contextual information affect the ability of LLMs to recover function-level semantics?

4.1 Model domain and scale

To evaluate the impact of the model domain and parameter scale on binary function name prediction, we conduct a unified evaluation of both general-purpose LLMs and code-oriented LLMs under two input settings: unstripped and stripped. We include representative DL-based methods, including Nero (David et al., 2020) and BContext2Name (Xia et al., 2023), as reference points for comparison. For the DL-based baselines, we evaluate implementations reproduced according to the methodologies described in the original publications. No additional retraining is performed in this work. Since these models are not trained under the unified experimental protocol adopted in this study, the reported results are

presented as reference results rather than strictly comparable baselines.

The dataset is constructed following the procedure described in Section 3.1 and contains 49 935 cleaned function instances after removing compiler-generated functions, duplicate samples, and functions that fail to be decompiled. These instances are collected from eight projects under multiple compilation configurations, including different ISAs and optimization levels (Tables S1 and S2). During evaluation, we construct a fixed subset by selecting 200 functions from each project (1600 in total), where each sample is drawn from a unified pool that aggregates functions compiled under diverse architectures and optimization settings. As a result, the subset naturally includes a mixture of compilation configurations and preserves the inherent distribution of the dataset while ensuring balanced contributions across projects. Moreover, since the dataset contains semantically related function variants generated under different compilation configurations, the resulting subset captures both project-level diversity and compilation-induced variability. No model fine-tuning or task-specific training is performed. The experimental results are summarized in Table 1. To verify that the reported trends are not an artifact of a single fixed subset, we further repeat the evaluation on multiple project-balanced random subsets sampled from the full function pool. The corresponding seeds, standard deviations, and 95% confidence intervals are reported in Section 1.2 of the supplementary materials (Table S3).

The experimental results indicate that pretraining domain has a pronounced impact on binary function name prediction performance. Under both the unstripped and stripped pseudocode settings, most code-oriented LLMs outperform general-purpose LLMs of comparable scales and exhibit more consistent behaviors across different parameter sizes. Specifically, in the unstripped setting, code-oriented models achieve F1 scores primarily in the 29.2%–34.6% range, whereas general-purpose models concentrate around 27.3%–32.5%. Under the more challenging stripped setting, the performance gap further widens, with code models showing stronger robustness in both Precision and Recall. This trend is consistent with prior findings (Guo et al., 2020), suggesting that pretraining on large-scale code corpora facilitates learning structural and semantic regularities of programs, thereby improving model understanding of code.

Compared with the evaluated DL-based reference methods, LLMs achieve consistently higher scores on the sampled benchmark. The DL-based reference results also illustrate the difficulty of binary function name prediction under realistic stripped-binary conditions. This advantage may be attributed to the richer semantic knowledge acquired during large-scale pretraining, which better supports zero-shot binary function name prediction.

Model scaling does not yield a monotonic performance improvement in our experiments. For general-purpose LLMs, increasing Qwen2.5 from 7B to 72B brings only modest gains, and the gap between LLaMA-3.1:8B and LLaMA-3.1:70B is similarly limited. This trend is even more evident for code-oriented models: Under the stripped setting, Qwen2.5-Coder:7B outperforms the larger 14B and 32B variants, indicating a non-monotonic relationship between parameter count and accuracy. While classical scaling laws suggest that performance is often improved with model size, our results imply that, in cross-domain scenarios with substantial distribution mismatch between pretraining data and binary decompilation inputs, larger models do not necessarily confer consistent benefits (Kaplan et al., 2020).

In addition, StarCoder2:7B performs markedly poorly on this task, achieving an F1 score of about 1%. One possible explanation is that its pretraining data predominantly consist of high-level source code, providing limited exposure to lower-level program representations or decompiled pseudocode. As a result, the model may fail to learn the structural and semantic patterns required to generalize effectively to decompiled pseudocode inputs. Given the use of stochastic decoding and strict token-level evaluation, small differences in F1 scores should be interpreted with caution, particularly when comparing closely performing models or configurations.

Answer to RQ1: Code-oriented models outperform general-purpose models, highlighting the pretraining domain as the key driver. Performance is limited less by model scale than by cross-domain semantic misalignment.

Table 1 Performance comparison of evaluated methods on binary function name prediction

Domain	Model	Unstripped			Stripped		
		Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)
General-purpose LLMs	Qwen2.5:72B	28.89	32.75	30.70	14.53	16.35	15.39
	Qwen2.5:7B	26.78	27.88	27.32	13.84	14.08	13.95
	LLaMA-3.1:8B	**31.21**	30.21	30.70	15.71	14.93	15.31
	LLaMA-3.1:70B	30.83	**34.30**	**32.47**	**15.90**	**17.70**	**16.75**
	GLM4:9B	25.17	30.24	27.47	11.64	13.68	12.58
Code-oriented LLMs	Qwen2.5-Coder:0.5B	14.90	13.56	14.20	6.29	5.10	5.63
	Qwen2.5-Coder:3B	30.72	32.14	31.41	14.27	14.43	14.35
	Qwen2.5-Coder:7B	**32.34**	**37.24**	**34.62**	**16.17**	**17.99**	**17.03**
	Qwen2.5-Coder:14B	28.49	31.51	29.92	13.54	15.08	14.27
	Qwen2.5-Coder:32B	29.50	35.39	32.18	13.98	16.42	15.10
	DeepSeek-Coder-V2:16B	23.33	17.28	19.85	6.44	4.16	5.05
	StarCoder2:7B	1.03	2.05	1.38	0.56	1.39	0.80
	CodeGeeX4:9B	26.76	32.19	29.22	13.09	15.41	14.16
DL-based	Nero	–	–	–	3.84	3.52	3.67
	BContext2Name	–	–	–	**5.32**	**6.32**	**5.78**

The best results in each domain are in bold

4.2 Pseudocode length

To analyze the impact of input length on model performance, we conduct evaluations under the stripped pseudocode setting by partitioning the inputs into multiple intervals based on pseudocode length. Both general-purpose and code-oriented models are evaluated across these length ranges. Fig. 3 illustrates the variation in the F1 score as a function of input length for different models.

The results demonstrate that pseudocode length has a substantial impact on model performance. Across all models, shorter inputs (0–500 characters) generally yield higher F1 scores than longer inputs. This span often contains the function header and early body statements, where parameters, salient local variables, and partial type cues are concentrated, providing information-rich cues for inferring overall functionality. As the pseudocode length increases, the overall F1 score tends to decline and the performance becomes less stable. In practice, such outputs typically include empty predictions, degenerate repetitions, overly generic names, and over-generated outputs containing unnecessary or weakly related tokens; representative examples are provided in Section 5.3 of the supplementary materials (Table S12). This trend suggests that additional control-flow expansions and implementation-level details in long pseudocode inputs frequently introduce limited discriminative evidence for naming, while substantially increasing contextual complexity, ultimately impairing prediction quality.

In cross-model comparisons, code-specialized models exhibit greater robustness under long pseudocode inputs. For instance, Qwen2.5-Coder:3B maintains relatively stable F1 scores across multiple long-length bins, whereas general-purpose models (e.g., LLaMA-3.1:70B) show a more pronounced performance drop as input length increases. Notably, smaller models occasionally outperform larger ones in certain long-input regimes, further indicating that parameter count is not the primary determinant of long-context performance in this task.

Overall, pseudocode length is a key factor affecting binary function name prediction performance. Excessively long pseudocode often introduces substantial information that is weakly related to the function's high-level intent, such as expanded control-flow structures, repeated computations and error-handling routines, as well as intermediate variables and redundant statements introduced by decompilation. When such additions do not provide commensurate discriminative cues for naming, they substantially increase the modeling burden, leading to clear performance degradation and, in some cases, unstable outputs or degenerate predictions (i.e., output collapse).

Answer to RQ2: Excessively long pseudocode degrades performance. Code-oriented models are more robust to long inputs, indicating that pretraining-domain alignment matters more than model scale.

4.3 Architecture and optimization

Different ISAs and compiler optimization strategies can substantially alter the structural and semantic characteristics of binary code. To analyze their impact, we select a representative code-oriented model (Qwen2.5-Coder:7B) and a representative general-purpose model (LLaMA-3.1:8B), and evaluate their performance under the stripped setting across multiple ISAs (x86, x64, ARM, and MIPS) and compiler optimization levels (O0–O3). The experimental results are presented in Fig. 4.

Overall, the effects of ISA and compiler optimization level on model performance are not independent; instead, they exhibit a clear coupling effect. Under low optimization levels (O0 and O1), both models achieve their highest F1 scores on the x86 architecture. However, at the O2 optimization level, the performance on x86 degrades noticeably, followed by a partial recovery under O3.

From the perspective of compiler behavior, O2 optimization, which is widely adopted in embedded and IoT scenarios, typically aims to balance execution efficiency and code size. Techniques such as function inlining, dead-code elimination, and control-flow restructuring substantially compress function boundaries and high-level semantic structures, thereby reducing the availability of function-level discriminative cues for the models. In contrast, O3 introduces aggressive optimizations such as loop unrolling and vectorization, which partially re-expose certain computation patterns and, to some extent, mitigate the negative impact of semantic compression observed

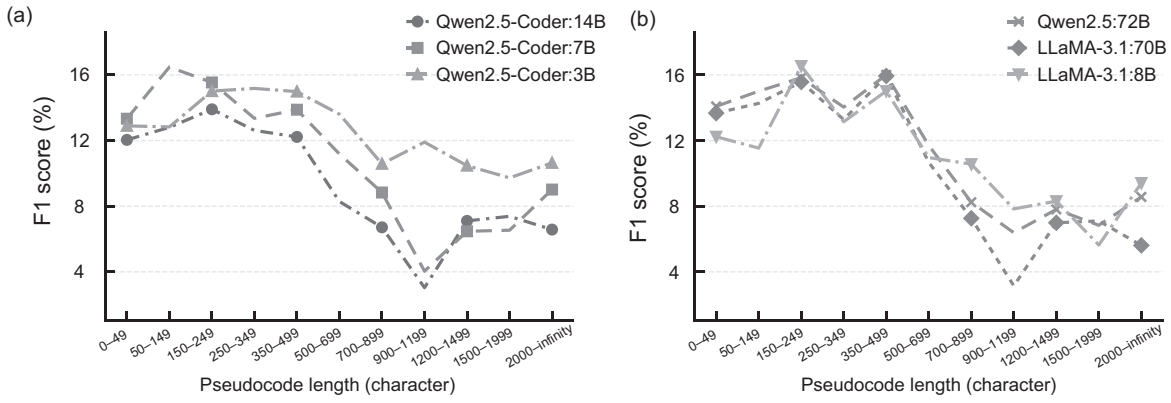

Fig. 3 Impact of pseudocode length on function name prediction performance: (a) code-oriented LLMs; (b) general-purpose LLMs

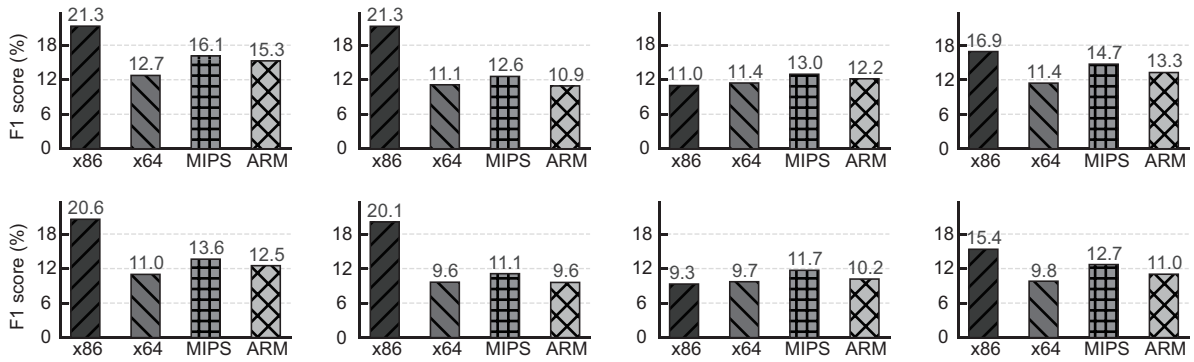

Fig. 4 F1 score comparison of different models across instruction set architectures and compiler optimization levels. The first row corresponds to Qwen2.5-Coder:7B and the second row corresponds to LLaMA-3.1:8B. From left to right, the four columns correspond to O0, O1, O2, and O3, respectively

at the O2 stage. Representative examples of prediction differences under varying compiler optimization levels are provided in Section 5.4 of the supplementary materials (Table S13).

By comparison, the MIPS architecture exhibits more regular instruction formats and control-flow structures. As a result, function-level structures are less disrupted across different optimization levels, leading to more stable and consistently observable semantic features in the decompiled pseudocode and, consequently, smaller performance fluctuations. Representative examples illustrating prediction variations across different ISAs are provided in Section 5.5 of the supplementary materials (Table S14).

Cross-model comparisons further indicate that code-oriented models exhibit higher robustness under cross-architecture and high-optimization settings. Specifically, Qwen2.5-Coder shows substantially smaller F1 fluctuations across all architectures and optimization levels than LLaMA-3.1, with particularly pronounced advantages on x64 and ARM under O2 and O3. These results suggest that structured code patterns learned during code-centric pretraining help mitigate semantic distortion introduced by architectural differences and aggressive compiler optimizations, thereby improving robustness in complex compilation scenarios.

Answer to RQ3: Higher optimization generally degrades semantic recovery. Code-oriented models are more stable across architectures and optimizations, suggesting that the pretraining domain contributes more to robustness than model scale.

4.4 Prompt type

Prompt engineering guides models' understanding of task objectives and reasoning processes, and the degree of prompt structure can directly influence model behaviors. To systematically analyze the impact of prompt design on binary function name prediction, we consider three representative prompt paradigms: zero-shot prompting, few-shot prompting, and chain-of-thought (CoT) prompting. In the zero-shot setting, the model is provided only with the decompiled pseudocode of a stripped binary and is instructed to directly output a single snake_case function name without any example or additional guidance. In the few-shot setting, several example pairs of decompiled pseudocode and their corresponding function names are included in the prompt before the target function to provide task-specific guidance through demonstration. In the CoT setting, the model is explicitly instructed to perform structured intermediate reasoning (e.g., analyzing input parameters, control-flow structure, and return semantics) before outputting a single final function name. Representative prompt examples are provided in Section 4 of the supplementary materials (Fig. S1).

Based on the above prompt designs, we conduct a unified evaluation on the general-purpose model LLaMA-3.1:8B and the code-oriented model Qwen2.5-Coder:7B. The experimental results are presented in Fig. 5.

The results indicate that the effectiveness of prompt paradigms is strongly dependent on the model's pretraining domain. For the code-oriented model Qwen2.5-Coder:7B, incorporating CoT prompts yields a consistent yet modest performance improvement, with the F1 score increasing from 13.3% under the zero-shot setting to 14.1%. This suggests that explicit step-by-step reasoning can help code-specialized models better organize control-flow and operational semantics in decompiled pseudocode, thereby supporting function-level semantic inference.

In contrast, the general-purpose model LLaMA-3.1:8B does not benefit from either few-shot or CoT prompting; its best performance is achieved under the zero-shot setting. This suggests that, for models primarily pretrained on natural language corpora, additional reasoning-oriented prompts may be poorly aligned with the semantics of decompiled pseudocode and can even interfere with modeling function behaviors. Although prior work has shown that few-shot prompting can be effective for certain natural language tasks (Bhattacharya et al., 2023), this paradigm does not translate into measurable gains in the highly domain-specific setting of binary pseudocode analysis.

Overall, while prompt design can influence model behavior to some extent, the resulting performance gains in this task remain limited, with F1 scores below 15% across all prompt configurations. These results indicate that prompt engineering alone is insufficient to effectively bridge the gap between decompiled pseudocode and high-level function semantics.

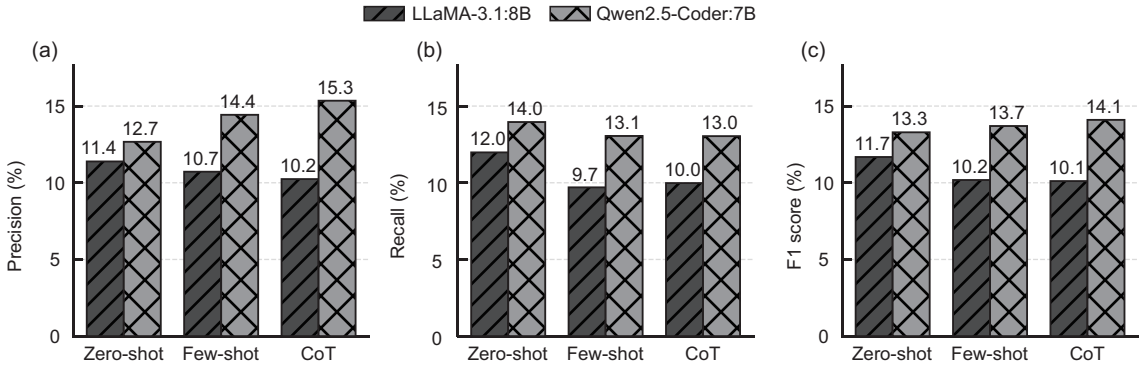

Fig. 5 Impact of different prompt designs on function name prediction performance: (a) Precision; (b) Recall; (c) F1 score

Answer to RQ4: Code-oriented LLMs obtain modest gains from CoT prompting, whereas general-purpose LLMs may degrade under more elaborate prompts; therefore, prompt design is unlikely to be the primary performance bottleneck for this task.

4.5 Context separation

To avoid conflating practically available stripped-binary signals with unrealistically preserved symbolic information, we explicitly distinguish realistic stripped-binary context from oracle context. The realistic stripped-binary context refers to signals that may remain available after stripping, whereas the oracle context refers to symbolic information that is accessible only in unstripped binaries or via debugging information.

Based on this distinction, we construct three supplementary input settings on top of the stripped pseudocode baseline:

$$C_{\mathrm{p}} = \{P\},$$
$$C_{\mathrm{r}} = \{P, \mathrm{RealisticContext}\},$$
$$C_{\mathrm{o}} = \{P, \mathrm{OracleContext}\},$$

where P denotes the decompiled function-level pseudocode obtained from stripped binaries after target-name masking. RealisticContext includes practically available stripped-binary signals, such as imported API names, string literals, constants, call-graph topology, and coarse decompiler-recovered type-like cues, whereas OracleContext includes original variable names and internal function names obtained from unstripped binaries or debugging information. To further clarify the contribution of major realistic signals, we perform a lightweight ablation over three representative components: imported API names, string literals, and call-graph topology (Table 2). In the realistic-context setting, imported external function names are retained as they are, while internal callees are anonymized with address-based placeholders. In all settings, the target function name is excluded or masked wherever it may directly appear, including the function signature, recursive self-calls, and contextual fields, to prevent direct target-name leakage.

As shown in Table 2, all contextual settings improve the performance over the pseudocode-only baseline, indicating that additional semantic signals are beneficial under stripped-binary conditions. Among the representative realistic signals, call-graph topology provides the strongest and most consistent gains for both models, while imported API names also yield stable improvements and string literals offer smaller but still useful complementary cues. When combined as realistic stripped-binary context, these signals further improve performance beyond any single component, showing that practically available cues provide complementary semantics. The oracle context yields the largest gains overall, indicating that the original variable names and internal symbolic names still provide substantially richer semantic guidance than realistically recoverable stripped-binary cues. Moreover, the code-oriented model benefits more strongly from both realistic and oracle contexts, suggesting stronger robustness in integrating structured program-level semantic information.

Table 2 Function name prediction performance under different context settings on the sampled subset

Model	F1 score (%)					
	P	P+API	P+Str	P+Topo	P+R	P+O
LLaMA-3.1:8B	15.20	16.30	15.90	17.00	17.40	22.52
Qwen2.5-Coder:7B	17.10	18.40	17.90	19.70	20.80	29.10

P denotes pseudocode only. P+API, P+Str, and P+Topo denote pseudocode augmented with imported API names, string literals, and call-graph topology, respectively. P+R denotes pseudocode plus realistic stripped-binary context, and P+O denotes pseudocode plus oracle context

Answer to RQ5: The realistic stripped-binary context improves performance, while the oracle context yields further gains and should be viewed only as an upper bound.

5 Discussion

Based on the experimental findings, we discuss the broader implications, limitations, and future directions of this work.

1. Limitations of prompt-based approaches: The prompt designs adopted in this study are intended to compare different prompting paradigms rather than to inject structural priors tightly coupled with binary semantics. Under this setting, models rely primarily on decompiled pseudocode, which limits the effectiveness of prompt engineering by the semantic sparsity of the input itself. Our supplementary context-separation analysis further shows that realistically available stripped-

binary signals can already improve performance, while oracle symbolic signals provide additional gains but should be interpreted only as an upper bound. These findings suggest that the main bottleneck lies in insufficient semantic information rather than in the prompting strategy alone. Overall, prompt engineering can help guide inference, but it cannot substitute for explicit, structured semantic signals in binary semantic understanding.

2. Challenges of fine-tuning models for decompiled binary semantics: A substantial body of work on fine-tuning large code models has primarily been conducted around high-level source code, where training and adaptation rely on well-formed syntax, explicit identifiers, and clear semantic boundaries. While such approaches have achieved strong results in source-code understanding, the learned semantic distribution fundamentally differs from that of decompiled binaries. In contrast, prior work (Tan et al., 2024) incorporated decompiled pseudo-C as model input and performed task-specific fine-tuning, demonstrating the feasibility of adapting models within a representation space that more closely matches practical RE settings. This evidence suggests that, for binary semantic understanding, the critical factor in fine-tuning is not the model scale or the fine-tuning paradigm per se, but the choice of an intermediate representation that faithfully captures the semantic characteristics—and noise patterns—of the decompiled code.

3. Implications for practical RE: The results of this study indicate that, in realistic RE scenarios, directly applying general-purpose LLMs to stripped binaries—without incorporating explicit, structured semantic information—remains subject to substantial limitations. In contrast, a more practical and effective approach is to augment decompiled outputs with realistically recoverable semantic cues, such as imported API names, string literals, and call-graph structure obtained via static analysis, and to combine these signals with code-oriented language models for inference or further adaptation. These cues provide complementary semantic evidence under stripped-binary conditions while remaining closer to realistic deployment settings than oracle symbolic information.

4. Scope and obfuscation: This study focuses on symbol-stripped but non-obfuscated binaries, with the aim of isolating the effects of symbol removal and compiler optimizations. In practical RE, binaries may also undergo advanced obfuscation transformations that further distort structural and contextual signals in decompiled code. Unlike symbol stripping, obfuscation techniques may deliberately disrupt control-flow regularity and introduce semantically irrelevant transformations. Systematic evaluation of LLM robustness under obfuscation scenarios would require dedicated experimental design and controlled obfuscation pipelines. We consider this an important direction for future work to further enhance the applicability of the proposed evaluation framework.

5. Potential training data overlap: Potential overlap with public code repositories remains an important consideration when evaluating web-scale pretrained LLMs. Since our dataset is derived from widely used open-source projects on GitHub, parts of the original source code may have appeared in pretraining corpora. However, our evaluation inputs are decompiled pseudocode generated from symbol-stripped binaries rather than the original source code. The compilation–stripping–decompilation pipeline removes the original identifiers and substantially changes the program representation, thereby reducing direct overlap with source-level training data. Moreover, the task requires function-level semantic prediction rather than token reproduction. Nevertheless, potential overlap cannot be fully excluded, and evaluation of curated unseen corpora remains an important direction for future work.

6 Related works

6.1 Source code-based function name prediction

Early studies on function name prediction primarily rely on representation learning from high-quality source code, exploiting rich structural and semantic information such as abstract syntax trees (ASTs), data flow graphs (DFGs), and code comments. Representative approaches include code2vec (Alon et al., 2019b) and code2seq (Alon et al., 2019a), which construct code representations from AST paths and generate function names via attention-based aggregation or sequence decoding, respectively. With the advent of pretraining techniques, general-purpose models such as PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021) have been trained on large-scale code corpora for code understanding and generation, substantially enhancing semantic modeling capabilities. GraphCodeBERT (Guo et al., 2020) further incorporates DFGs to capture variable-level semantic dependencies.

While these methods achieve strong performance in source-code settings, they typically depend on complete syntactic structures and high-level semantic cues. As a result, they are not directly applicable to binary programs, where symbols are stripped and compiler optimizations substantially distort or remove such information.

6.2 Binary code-based function name prediction

Binary functions are composed of low-level structures such as disassembled instructions, register operations, and control-flow transfers. They are often affected by symbol stripping, compiler optimizations, and cross-architecture variations, which make semantic recovery substantially more challenging (Jin et al., 2022).

Early approaches primarily model binary functions based on disassembled instruction sequences, e.g., NFRE (Gao et al., 2021). While such methods enable lightweight semantic learning, they often fail to capture higher-level structural dependencies and execution semantics. To address this limitation, subsequent works introduce graph-based representations, particularly control flow graphs (CFGs) (e.g., Nero (David et al., 2020)), which provide a more structured view of program execution but rely heavily on accurate disassembly and graph construction. More recent studies further exploit decompiled pseudocode to bridge the semantic gap between low-level binaries and high-level program logic, e.g., SymGen (Jiang et al., 2025). Although decompilation enhances semantic expressiveness, it inevitably introduces additional noise and errors.

Building upon these representation strategies, recent research has explored Transformer-based architectures tailored for binary-level semantic modeling. For instance, Kim et al.

(2023) proposed a Transformer-based function symbol name inference model based on assembly representations. Sha et al. (2025) introduced llasm, a hybrid framework that fuses encoder-only and decoder-only language models for binary function naming. These approaches highlight the growing trend of leveraging large-scale pretrained models for binary semantic understanding.

Overall, these studies provide useful structural and semantic priors for function naming and related tasks. However, their ability to capture semantic invariance and generalize across different compilers, optimization levels, and ISAs remains limited.

6.3 LLMs for semantic understanding and downstream applications

In recent years, LLMs have demonstrated strong semantic abstraction and language modeling capabilities in code understanding and generation, and have increasingly been explored for binary analysis and program comprehension. One line of work leverages LLMs to generate high-level code representations directly from disassembly or decompilation outputs, or to improve the readability and compilability of decompiled code, e.g., LLM4Decompile (Tan et al., 2024), DecLLM (Wong et al., 2025), and D-LiFT (Zou et al., 2025), thereby validating the potential of LLMs for semantic abstraction from the perspective of generation quality. Another line of research takes a broader security-oriented viewpoint and discusses the prospects of LLMs in automated analysis, assisted reasoning, and system integration (Yao et al., 2024; Jaffal et al., 2025; Xu et al., 2025).

However, many of these studies primarily target generation quality or functional usability, and their evaluation setups often involve task-specific designs, additional structural signals, or model fine-tuning. Under such protocols, performance gains can be jointly influenced by multiple factors, making it difficult to disentangle improvements attributable to the intrinsic semantic modeling capability of LLMs from those arising from injected priors or training strategies. This limitation is particularly salient in practical RE settings, where binaries frequently exhibit symbol stripping, optimization-induced perturbations, and cross-architecture discrepancies. These factors lead to highly sparse semantic inputs, yet systematic characterization of LLM behaviors under such realistic conditions remains limited.

Motivated by this gap, this work does not aim to maximize performance via additional structural information or task-specific training. Instead, we focus on the evaluation design itself: under a unified input representation and inference protocol, we examine to what extent contemporary open-source LLMs can recover function-level semantics in realistic, semantically sparse binary settings using only decompiled pseudocode. To this end, we adopt binary function name prediction as a representative task and conduct a systematic analysis along key factors encountered in real-world RE, including symbol stripping, compiler optimization levels, ISAs, and contextual cues, thereby delineating the applicability and capability boundaries of LLMs in this domain.

7 Conclusions

This paper investigates binary function name prediction as a representative task for function-level semantic recovery in practical reverse engineering. We develop a systematic evaluation framework that reflects real-world conditions and assess open-source LLMs across multiple dimensions, including pretraining domain, input representation, compilation settings, and contextual information. The results show that, while code-oriented LLMs generally outperform general-purpose models, the semantic patterns learned primarily from source-code corpora do not transfer reliably to binary settings, and increasing the model size does not yield consistent or monotonic performance gains.

Overall, code pretraining provides a clear benefit; however, all evaluated models exhibit substantial performance degradation under symbol stripping and aggressive compiler optimizations, indicating that cross-domain transfer from source-code semantics to binary semantics remains insufficient. Moreover, the unstable gains from scaling suggest that the dominant bottlenecks are more likely attributable to decompilation noise and missing contextual cues than to model capacity alone. These findings imply that future improvements in binary semantic recovery will depend less on naïve scaling or prompt engineering, and more on incorporating binary-specific contextual modeling and dedicated representation learning mechanisms.

Supplementary information

The online version contains supplementary materials available at https://doi.org/10.1631/ENG.ITEE.2026.0017.

Acknowledgments

This work was supported by the Key Research and Development Program of Henan Province (No. 262102210054), the Guangxi Natural Science Foundation (No. 2025GXNSFGA069004), and the Natural Science Foundation of Henan Province (No. 262300420698).

Author contributions

Bing XIA designed the research. Yunxiang GE conducted the experiments and drafted the paper. Wenbo LIU collected the data and performed the formal analysis. Yong DING validated the results and supervised the study. Yunxiang GE revised and finalized the paper.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declaration on the use of generative AI tools

During the preparation of this work, the authors used ChatGPT to improve language. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

Ahmad W, Chakraborty S, Ray B, et al., 2021. Unified pre-training for program understanding and generation. Proc Conf North American Chapter of the Association for Computational Linguistics: Human Language Technologies, p.2655-2668. https://doi.org/10.18653/v1/2021.naacl-main.211

Allamanis M, Barr ET, Bird C, et al., 2015. Suggesting accurate method and class names. Proc 10th Joint Meeting on Foundations of Software Engineering, p.38-49. https://doi.org/10.1145/2786805.2786849

Alon U, Brody S, Levy O, et al., 2019a. code2seq: generating sequences from structured representations of code. 7th Int Conf on Learning Representations.

Alon U, Zilberstein M, Levy O, et al., 2019b. code2vec: learning distributed representations of code. *Proc ACM Programm Lang*, 3:40. https://doi.org/10.1145/3290353

Bhattacharya P, Chakraborty M, Palepu KNSN, et al., 2023. Exploring large language models for code explanation. https://doi.org/10.48550/arXiv.2310.16673

Boronat A, Mustafa J, 2025. MDRE-LLM: a tool for analyzing and applying LLMs in software reverse engineering. IEEE Int Conf on Software Analysis, Evolution and Reengineering, p.850-854. https://doi.org/10.1109/SANER64311.2025.00090

Chen M, Tworek J, Jun H, et al., 2021. Evaluating large language models trained on code. https://doi.org/10.48550/arXiv.2107.03374

David Y, Alon U, Yahav E, 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc ACM Programm Lang*, 4(OOPSLA):225. https://doi.org/10.1145/3428293

Gao H, Cheng SY, Xue YX, et al., 2021. A lightweight framework for function name reassignment based on large-scale stripped binaries. Proc 30th ACM SIGSOFT Int Symp on Software Testing and Analysis, p.607-619. https://doi.org/10.1145/3460319.3464804

GitHub, 2008. GitHub: a Platform for Hosting and Collaborating on Software Development. https://github.com/ [Accessed on Jan. 11, 2026].

Guo DY, Ren S, Lu S, et al., 2020. GraphCodeBERT: pre-training code representations with data flow. https://doi.org/10.48550/arXiv.2009.08366

Han K, Lim JH, Im EG, 2013. Malware analysis method using visualization of binary files. Proc Research in Adaptive and Convergent Systems, p.317-321. https://doi.org/10.1145/2513228.2513294

Harzevili NS, Belle AB, Wang JJ, et al., 2023. A survey on automated software vulnerability detection using machine learning and deep learning. https://arxiv.org/abs/2306.11673

Høst EW, Østvold BM, 2009. Debugging method names. 23rd European Conf on Object-Oriented Programming, p.294-317. https://doi.org/10.1007/978-3-642-03013-0_14

Hu XY, Fu ZW, Xie SC, et al., 2025. SoK: potentials and challenges of large language models for reverse engineering. https://doi.org/10.48550/arXiv.2509.21821

Hugging Face, 2016. Hugging Face. https://huggingface.co/ [Accessed on Jan. 15, 2026].

Jaffal NO, Alkhanafseh M, Mohaisen D, 2025. Large language models in cybersecurity: a survey of applications, vulnerabilities, and defense techniques. *AI*, 6(9):216. https://doi.org/10.3390/ai6090216

Jiang LX, Jin X, Lin ZQ, 2025. Beyond classification: inferring function names in stripped binaries via domain adapted LLMs. Proc ACM SIGSAC Conf on Computer and Communications Security.

Jin X, Pei KX, Won JY, et al., 2022. SymLM: predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. Proc ACM SIGSAC Conf on Computer and Communications Security, p.1631-1645. https://doi.org/10.1145/3548606.3560612

Kaplan J, McCandlish S, Henighan T, et al., 2020. Scaling laws for neural language models. https://arxiv.org/abs/2001.08361

Kim H, Bak J, Cho K, et al., 2023. A Transformer-based function symbol name inference model from an assembly language for binary reversing. Proc ACM Asia Conf on Computer and Communications Security, p.951-965. https://doi.org/10.1145/3579856.3582823

Lawrie D, Morrell C, Feild H, et al., 2006. What's in a name? A study of identifiers. 14th IEEE Int Conf on Program Comprehension, p.3-12. https://doi.org/10.1109/ICPC.2006.51

Meng XZ, Miller BP, 2016. Binary code is not easy. Proc 25th Int Symp on Software Testing and Analysis, p.24-35. https://doi.org/10.1145/2931037.2931047

OpenAI, 2023. OpenAI API Documentation. https://developer.neureality.ai/docs/user-guide/openai-api-doc.html [Accessed on Jan. 11, 2026].

Patrick-Evans J, Cavallaro L, Kinder J, 2020. Probabilistic naming of functions in stripped binaries. Proc 36th Annual Computer Security Applications Conf, p.373-385. https://doi.org/10.1145/3427228.3427265

Sha ZH, Wang H, Gao ZY, et al., 2025. llasm: naming functions in binaries by fusing encoder-only and decoder-only LLMs. *ACM Trans Softw Eng Methodol*, 34(4):93. https://doi.org/10.1145/3702988

Shang XW, Cheng SY, Chen GQ, et al., 2024. How far have we gone in binary code understanding using large language models. IEEE Int Conf on Software Maintenance and Evolution, p.1-12. https://doi.org/10.1109/ICSME58944.2024.00012

Shin ECR, Song D, Moazzezi R, 2015. Recognizing functions in binaries with neural networks. 24th USENIX Conf on Security Symp, p.611-626.

Song YF, Zhang DD, Wang J, et al., 2025. Application of deep learning in malware detection: a review. *J Big Data*, 12(1):99. https://doi.org/10.1186/s40537-025-01157-y

Tan HZ, Luo Q, Li J, et al., 2024. LLM4Decompile: decompiling binary code with large language models. Proc Conf on Empirical Methods in Natural Language Processing, p.3473-3487. https://doi.org/10.18653/v1/2024.emnlp-main.203

Tian JF, Xing WJ, Li Z, 2020. BVDetector: a program slice-based binary code vulnerability intelligent detection system. *Inform Softw Technol*, 123:106289. https://doi.org/10.1016/j.infsof.2020.106289

vLLM Project, 2023. vLLM: a high-throughput and memory-efficient inference and serving engine for LLMs. https://github.com/vllm-project/vllm [Accessed on Jan. 13, 2026].

Wang Y, Wang WS, Joty S, et al., 2021. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Proc Conf on Empirical Methods in Natural Language Processing, p.8696-8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

Wong WK, Wu D, Wang H, et al., 2025. DecLLM: LLM-augmented recompilable decompilation for enabling programmatic use of decompiled code. *Proc ACM Softw Eng*, 2:ISSTA081. https://doi.org/10.1145/3728958

Xia B, Pang JM, Wang J, et al., 2021. Study on binary code evolution with concrete semantic analysis. 7th Int Conf of Pioneering Computer Scientists, Engineers and Educators, p.30-43. https://doi.org/10.1007/978-981-16-5943-0_3

Xia B, Ge YX, Yang RN, et al., 2023. BContext2Name: naming functions in stripped binaries with multi-label learning and neural networks. IEEE 10th Int Conf on Cyber Security and Cloud Computing and IEEE 9th Int Conf on Edge Computing and Scalable Cloud, p.167-172. https://doi.org/10.1109/CSCloud-EdgeCom58631.2023.00037

Xu HX, Wang SN, Li NK, et al., 2025. Large language models for cyber security: a systematic literature review. *ACM Trans Softw Eng Methodol*, in press. https://doi.org/10.1145/3769676

Yao YF, Duan JH, Xu KD, et al., 2024. A survey on large language model (LLM) security and privacy: the good, the bad, and the ugly. *High-Confid Comput*, 4(2):100211. https://doi.org/10.1016/j.hcc.2024.100211

Zou MQ, Cai HY, Wu HW, et al., 2025. D-LiFT: improving LLM-based decompiler backend via code quality-driven fine-tuning. https://doi.org/10.48550/arXiv.2506.10125