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

Supplementary materials for

Yunxiang GE, Bing XIA, Yong DING, Wenbo LIU, 2026. Benchmarking large language models for binary function name prediction. *ENGINEERING Information Technology & Electronic Engineering*, 27(8):260017.
https://doi.org/10.1631/ENG.ITEE.2026.0017

1 Dataset statistics and sampling robustness

1.1 Statistics

Table S1 summarizes the composition of the dataset used in this study, including the project domains, the number of compiled binaries, the number of extracted functions, and the sampled instances for each project. Table S2 further presents the distribution of functions across different instruction set architectures and compiler optimization levels.

These projects span multiple real-world application domains, such as systems, networking, databases, compilers, and image processing, and are widely adopted in prior program analysis and security research. The detailed statistics are provided to facilitate reproducibility and future comparative studies.

Table S1 Statistics of the binary dataset

Project	Domain	Number of binaries	Number of functions	Number of samples
Coreutils	System	14	4500	200
Curl	Network	3	6502	200
OpenSSL	Crypto	2	5631	200
SQLite3	Database	3	4986	200
PuTTY	Network	8	7492	200
Binutils	Compiler	19	5217	200
Findutils	System	6	7281	200
libpng	Image	2	8326	200
Total			49 935	1600

Projects are compiled for ×86/×64, ARM, and MIPS at optimization levels O0–O3

Table S2 Function distribution across architectures and optimization levels

Architecture	Number of functions			
	O0	O1	O2	O3
×86	3148	510	2484	1391
×64	4259	3580	3314	2774
ARM	3795	3699	3493	3146
MIPS	4666	3237	3410	3039

1.2 Sampling protocol

To assess subset sensitivity, we repeatedly construct project-balanced random subsets from the full function pool. In each run, 200 functions are sampled from each project, resulting in a total of 1600 functions, which matches the benchmark size used in the main experiments. We use the sampling seeds

Table S3 F1 performance under different sampling seeds

Domain	Model	Sampling seeds	Mean F1 (%)	ΔF1 to benchmark (%)	Std(F1) (%)	95% CI(F1) (%)
General	LLaMA-3.1:8B	13, 21, 42, 87, 123	15.20	−0.11	0.48	[14.60, 15.80]
	LLaMA-3.1:70B	13, 21, 42, 87, 123	16.80	+0.05	0.53	[16.14, 17.46]
Code	Qwen2.5-Coder:7B	13, 21, 42, 87, 123	17.10	+0.07	0.57	[16.39, 17.81]
	Qwen2.5-Coder:32B	13, 21, 42, 87, 123	15.10	0.00	0.51	[14.47, 15.73]

ΔF1 to main benchmark is computed as the difference between the mean F1 under repeated random subset sampling and the corresponding stripped-setting F1 reported in the main benchmark. The mean F1 values remain close to the main benchmark results, while the overall ranking trend is preserved across repeated random subsets

13, 21, 42, 87, 123 and report the resulting mean F1, standard deviation, and 95% confidence intervals for representative models in Table S3.

2 Evaluated models and decoding robustness

2.1 Model configurations

Table S4 reports the model parameter sizes, maximum context lengths, training token scales, architectures, release dates, and publishers. The evaluated models include both general-purpose LLMs and code-oriented LLMs, covering a wide range of parameter scales and design choices.

Table S4 Overview of the LLMs evaluated in this study

Domain	Model	Parameter number	Maximum context length	Training token scale	Architecture	Release date	Publisher
General	Qwen2.5	7B/72B	128k	18T	Trans.	Sep-2024	Alibaba
	LLaMA-3.1	8B/70B	128k	15T	Trans.	Jul-2024	Meta AI
	GLM-4	9B	128k	10T	Trans.	Jan-2024	Zhipu AI
Code	Qwen2.5-Coder	0.5/3/7/14/32B	128k	5.5T	Trans.	Nov-2024	Alibaba
	DeepSeek-Coder-V2	16B	128k	6T	MoE	Jun-2024	DeepSeek
	StarCoder2	7B	16 384	3.5T	Trans.	Feb-2024	BigCode
	CodeGeeX4	9B	128k	1.58T	Trans.	Jul-2024	Zhipu AI

2.2 Variance under repeated stochastic decoding

To quantify the variance introduced by stochastic decoding, we repeat the evaluation on the fixed benchmark subset while varying only the decoding seed. All other settings, including the model, prompt, test set, temperature, and maximum generation length, remain unchanged. We use the decoding seeds 11, 22, 33, 44, 55 and report the corresponding mean F1, standard deviation, and 95% confidence intervals in Table S5. This design isolates decoding-induced variability from dataset-induced variability.

3 Evaluation metrics and semantic assessment

3.1 Metrics used in prior studies

To provide additional background on evaluation practice in function name prediction, Table S6 summarizes the automatic evaluation metrics adopted in representative prior studies across different programming languages. This summary is intended to contextualize the metric choices adopted in this work and to highlight the diversity of evaluation criteria reported in the literature.

Table S5 F1 performance under different decoding seeds

Domain	Model	Decoding seeds	Mean F1 (%)	ΔF1 to benchmark (%)	Std(F1) (%)	95% CI(F1) (%)
General	LLaMA-3.1:8B	11, 22, 33, 44, 55	14.90	–0.41	0.58	[14.18, 15.62]
	LLaMA-3.1:70B	11, 22, 33, 44, 55	16.55	–0.20	0.61	[15.79, 17.31]
Code	Qwen2.5-Coder:7B	11, 22, 33, 44, 55	16.91	–0.12	0.64	[16.12, 17.70]
	Qwen2.5-Coder:32B	11, 22, 33, 44, 55	14.80	–0.30	0.60	[14.05, 15.55]

ΔF1 to main benchmark is computed as the difference between the mean F1 in repeated stochastic decoding and the corresponding F1 reported in the main benchmark. The results indicate that the repeated-seed means remain close to the main benchmark values, while the overall ranking trend is preserved

Table S6 Summary of evaluation metrics used in prior studies

Task	Reference	Language	Automatic evaluation metrics
Method naming	Fernandes et al., 2019	Java, C#	ROUGE-L, ROUGE-2, F1
	Alon et al., 2019	Java	Precision, Recall, F1
	Boone et al., 2019	Python	Top-K Precision, Top-K Recall, Top-K F1
	Patrick-Evans et al., 2020	C	Precision, Recall, F1
	Gao et al., 2021	C	Precision, Recall, F1
	Jin et al., 2022	C	Precision, Recall, F1
	Shang et al., 2024	C	Manual scoring
	Jiang et al., 2025	C	Precision, Recall, F1

3.2 Extended evaluation beyond token-level F1

To complement the strict top-1 token-level evaluation used in the main benchmark, we further report a supplementary top-k token-level analysis of representative models. While this setting differs from the main benchmark in that multiple candidate names are generated for each sample, we retain the same normalization and token-level scoring procedure. For each sample, the best token-level F1 within the top-k generated candidates is selected. As shown in Table S7, increasing k yields consistent but limited improvements, indicating that semantically closer alternatives may appear among the top-ranked candidates, while the overall task remains highly challenging.

3.3 Manual semantic evaluation

To complement the strict token-level evaluation used in the main benchmark, we conduct a manual semantic assessment on 100 sampled predictions for each representative model. All predictions were independently annotated by two evaluators with experience in reverse engineering and software analysis according to a predefined annotation rubric. Inter-rater agreement was measured using Cohen's κ and reached 0.72, indicating substantial agreement between annotators. Disagreements were resolved through discussion before producing the final labels. The complete annotation guideline and representative labeled examples are provided in Section 3.4 of the supplementary materials. Table S8 summarizes the resulting label distribution based on the final annotations. The results indicate that token-level F1 underestimates part of the models' ability to recover function semantics, as a portion of predictions remain semantically meaningful despite lexical differences from the reference names.

3.4 Annotation guideline

To improve the reproducibility of manual semantic assessment, we define four annotation categories for evaluating predicted function names against the corresponding ground-truth names.

Exact match: The predicted function name is identical to the reference function name after normalization. Minor formatting differences, such as letter case or delimiter variations, are ignored.

Semantically correct: The predicted function name differs lexically from the reference name but preserves

Table S7 Extended evaluation beyond token-level F1

Domain	Model	Top-1 token-F1 (%)	Top-3 token-F1 (%)	Top-5 token-F1 (%)
General	LLaMA-3.1:8B	8.44	11.67	13.03
	LLaMA-3.1:70B	9.72	12.71	13.96
Code	Qwen2.5-Coder:7B	10.58	13.49	14.55
	Qwen2.5-Coder:32B	8.21	11.08	12.47

Top-K token-F1: F1 score computed based on the top-K predicted tokens

Table S8 Manual semantic evaluation on a sampled subset

Domain	Model	Exact match (%)	Semantically correct (%)	Partially related (%)	Incorrect (%)
General	LLaMA-3.1:8B	0	21	32	47
	LLaMA-3.1:70B	0	27	32	41
Code	Qwen2.5-Coder:7B	2	24	34	40
	Qwen2.5-Coder:32B	1	20	43	36

the same primary functionality. Typical cases include synonym substitution, abbreviation expansion, or equivalent semantic expressions (e.g., verify versus validate, init versus initialize).

Partially related: The predicted function name captures only part of the intended functionality or describes the behavior at a different level of abstraction. This category is used when the prediction reflects a related semantic concept but omits important task-specific information or provides a more general description compared with the reference name.

Incorrect: The predicted function name is semantically inconsistent with the reference functionality and does not provide a meaningful description of the intended behavior.

Predictions expressed at different abstraction levels are categorized as "partially related" rather than "semantically correct." For example, a prediction such as create_connection for the reference name create_ssl_connection captures only a higher-level description of the functionality and therefore receives the "partially related" label. Table S9 presents representative examples for each annotation category.

4 Prompt design examples

This section presents representative prompt designs used in this study, including zero-shot, few-shot, and chain-of-thought prompting strategies. Fig. S1 illustrates the general structure of the prompts without revealing any additional experimental results beyond those reported in the main text.

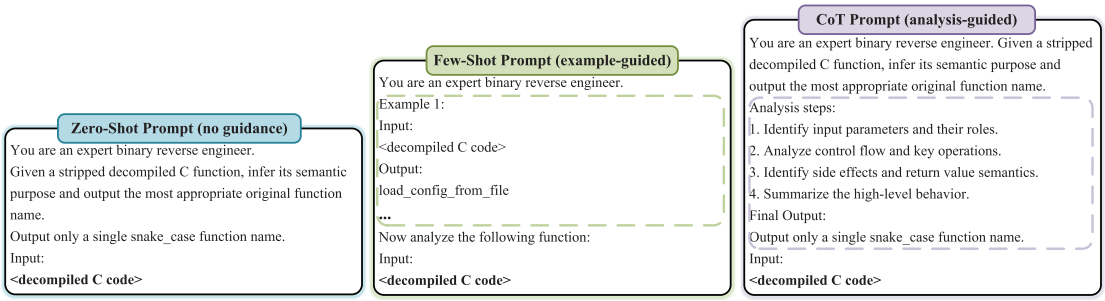

Fig. S1 Comparison of zero-shot, few-shot, and chain-of-thought prompts used in this study

Table S9 Representative examples from manual semantic evaluation

Ground truth	Prediction	Label
png_zalloc	png_zalloc	EM
set_filetime	setfiletime	EM
quote_memory	quotearg_mem	SC
safe_malloc	xmalloc	SC
ocsp_query_responder	get_ocsp_resp_from_responder	PR
check_curl_protocol_support	check_protocol	PR
compare_strings	check_end	INC
check_buffer_length	join	INC

EM: exact match; SC: semantically correct; PR: partially related; INC: incorrect. The examples illustrate representative cases for each annotation category

Table S10 Representative prediction variations compared with the ground truth

Category	Ground truth	Prediction
Exact match	xfts_open	**xfts_open**
Case variation	RSA_verify_loop	rsa_**verify_loop**
Semantic compression	png_image_begin_ read_from_file	**png_image_ read_from_file**
Related semantic variant	png_read_row	process_image_**row**
Argument substitution	png_set_text_ compression_window_bits	**png_set_ compression_window**_size

5 Case study

To provide deeper insights into the performance characteristics observed in our benchmark, we present several representative case studies here. These examples illustrate typical prediction behaviors of LLMs when applied to function name prediction from decompiled pseudocode, complementing the quantitative evaluation results reported in the main text.

5.1 Representative prediction variations

Table S10 presents representative examples of near-correct prediction patterns compared with the ground truth. Even when the predicted function name does not exactly match the reference, the model often preserves core semantic cues. As shown in the examples, such variations may arise from superficial differences such as letter casing or formatting, or from semantically related transformations such as abbreviation expansion, semantic compression, and related semantic variants. These cases indicate that token-level automatic metrics may underestimate the model's ability to capture underlying function semantics.

5.2 Cross-model success and failure cases

Table S11 highlights two complementary phenomena. Code-oriented LLMs often generate library-specific but semantically misplaced names, typically by overfitting to main workflows, internal routines, or specific implementation patterns. In contrast, general-purpose LLMs can still recover useful high-level function semantics when the target behavior is relatively clear, even without code-specific specialization.

5.3 Long-input failures

Table S12 summarizes several representative failure cases when the pseudocode input becomes excessively long. As the length of decompiled pseudocode increases, the model may struggle to focus on the most relevant semantic information. This may lead to unstable generation behaviors, such as empty outputs, repetitive tokens, semantic collapse, or over-generation. These cases highlight the difficulty of extracting

Table S11 Representative cross-model case analysis

Category	Model	Ground truth	Prediction
Code, failure	Qwen2.5-Coder:7B	create_pkcs12_export	pkcs12_main
		sqlite3_update_database	incrVacuumStep
		aes_cbc_decrypt_loop	AES_cbc_128_encrypt_loop
	Qwen2.5-Coder:32B	parse_pbe_options	pkcs8_main
		process_and_update_indices	sqlite3RefillIndex
		decrypt_data_loop	DES_ncbc_encrypt_loop
General, success	LLaMA-3.1:8B	parse_npn_buffer	next_protos_parse
		check_locale	hard_locale
		quote_memory	quotearg_mem
	LLaMA-3.1:70B	ssl_shutdown_loop	do_ssl_shutdown
		create_directory_path	create_dir_hierarchy
		extract_path_component	get_url_file_name

The table reports representative failure cases of code-oriented LLMs and unexpectedly strong cases of general-purpose LLMs

core functional semantics from long and noisy pseudocode, which is a common scenario in practical reverse engineering tasks.

Table S12 Examples of prediction failures for long pseudocode inputs

Failure type	Ground truth	Prediction
Empty output	last_valueStepFunc	" "
Degenerate repetition	png_ascii_from_fp	**png**_write_* repeated
	png_create_colormap_entry	**png**_write_* repeated
Semantic collapse	png_set_filter_heuristics_fixed	do_nothing
	_curl_easy_getinfo_err_long	do_nothing
Over-generation	png_data_freer	**png**_free_**data**_cleanup_**...**
	m_gt_1	generate_**gt**_metric_**...**

The table illustrates common failure patterns in model predictions for long pseudocode inputs, including empty output, degenerate repetition, semantic collapse, and over-generation

5.4 Optimization-level examples

Table S13 presents examples of predictions under different compiler optimization levels. Although the original source-level function remains unchanged, compiler optimizations (e.g., O0–O3) may significantly alter the structure and abstraction level of the generated pseudocode. As shown in the examples, predictions under higher optimization levels often shift from specific semantic descriptions toward more generic behaviors, suggesting that compiler optimizations may obscure or distort semantic cues available to the model.

5.5 Cross-ISA examples

Table S14 illustrates prediction variations across different ISAs. Even for the same source function, differences in instruction sets and compilation pipelines may produce structurally different pseudocode after decompilation. Consequently, the model may rely on different semantic signals and generate inconsistent predictions across architectures. These cases demonstrate that cross-ISA variability remains an important challenge for LLM-based binary semantic understanding.

Table S13 Examples of predictions under different compiler optimization levels

Ground truth	O0	O1	O2	O3
png_zfree	**png**_free_memory	**png**_free_function	handle_**png**_callback	process_**png**_function
output_**html**_**string**	**html**_escape_**string**	**html**_escape_**string**	process_**html**_**string**	process_data_from_buffer
png_write_iCCP	**png**_handle_**iccp**_chunk	handle_**iccp**_chunk	process_data_buffer	process_data
raw_comparator	compare_values	compare_values	is_equal	is_equal
png_read_row	**png_read_row**_data	**png**_handle_**row**	process_image	process_image_data

The table shows prediction variations across different compiler optimization levels (O0–O3). Keywords matching the ground-truth function name are highlighted in bold

Table S14 Examples of predictions across different instruction set architectures

Ground truth	x64	x86	ARM	MIPS
png_process_IDAT_data	**png_process_idat_data**	**png_process_idat**_chunk	**png_process_idat_data**	handle_audio_buffer
cached_umask	get_default_**umask**	get_file_mode_mask	get_default_file_mode_mask	get_default_**umask**
sk_X509_INFO_value	openssl_**sk_value**	openssl_**sk_value**	openssl_**sk**_valueAtIndex	openssl_**sk_value**
receipt_request_print	cms_**print**_receipts	cms_**print**_receipts	cms_**print**_receipts	**print**_cms_receipts
png_crc_error	validate_checksum	validate_checksum	validate_and_process_data	validate_checksum

The table shows prediction variations across different instruction set architectures (×64, ×86, ARM, MIPS). Keywords matching the ground-truth function name are highlighted in bold

5.6 When call context helps or misleads

Table S15 illustrates the dual role of call-context information. In some cases, it helps the model recover more appropriate function semantics, while in others it introduces misleading signals from neighboring routines. This indicates that call context is useful but not uniformly beneficial under stripped-binary settings.

Table S15 Examples where call context helps or misleads function name prediction

Model	Effect	Ground truth	Without context	With context
LLaMA-3.1:8B	Helped	idxPrepareStmt	process_data_and_call_callback	prepare_statement
		newTempFile	generate_file_path	generate_temp_file_name
		sqlite3_result_int64	set_flag_and_copy_data	set_int64_value
	Misled	idxCreateFromCons	create_index_function	idxPrepareStmt
		timeOfDay	initialize_runtime_environment	initialize_vfs
Qwen2.5-Coder:7B	Helped	substExpr	process_node	substitute_expression
		bindText	handle_event	sqlite3_bind_text
		jsonRenderNode	process_data	json_render_node
	Misled	sqlite3PutVarint	decode_binary_data	decode_varint
		sqlite3BtreeOpen	handle_file_operations	sqlite3PagerCloseAndFreeCache

"Without context" denotes prediction from stripped pseudocode alone, while "With context" denotes prediction after adding call-context information

References

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

Boone C, de Bruin N, Langerak A, et al., 2019. DLTPy: deep learning type inference of Python function signatures using natural language context. https://doi.org/10.48550/arXiv.1912.00680

Fernandes P, Allamanis M, Brockschmidt M, 2019. Structured neural summarization. Proc 7$^{\text{th}}$ Int Conf on Learning Representations.

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

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

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

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