LLM4CodeRE: Generative AI for Code Decompilation Analysis and Reverse Engineering

*LLM4CodeRE: GenAI for RS

Hamed Jelodar, Samita Bai, Tochukwu Emmanuel Nwankwo, Parisa Hamedi,
Mohammad Meymani, Roozbeh Razavi-Far, Ali A. Ghorbani
Faculty of Computer Science
University of New Brunswick, Canada
Emails: {h.jelodar,samita.bai,tochukwu.nwankwo,parisa.hamedi, mohammad.meymani79,roozbeh.razavi-far, ghorbani}@unb.ca

***Abstract*—Code decompilation analysis is a fundamental yet challenging task in malware reverse engineering, particularly due to the pervasive use of sophisticated obfuscation techniques. Although recent large language models (LLMs) have shown promise in translating low-level representations into high-level source code, most existing approaches rely on generic code pretraining and lack adaptation to malicious software. We propose LLM4CodeRE, a domain-adaptive LLM framework for bidirectional code reverse engineering that supports both assembly-to-source decompilation and source-to-assembly translation within a unified model. To enable effective task adaptation, we introduce two complementary fine-tuning strategies: (i) a Multi-Adapter approach for task-specific syntactic and semantic alignment, and (ii) a Seq2Seq Unified approach using task-conditioned prefixes to enforce end-to-end generation constraints. Experimental results demonstrate that LLM4CodeRE outperforms existing decompilation tools and general-purpose code models, achieving robust bidirectional generalization.**

***Index Terms*—GenAI, LLM, Source Code, Reverse Engineering**

I. Introduction

Reverse engineering of portable executable (PE) files, including both benign software and malware, remains a fundamental challenge in cybersecurity. The rapid evolution of modern malware—characterized by pervasive obfuscation, packing, and polymorphism—has made automated code decompilation a critical yet largely unresolved problem. Reverse engineers must routinely translate low-level representations, such as assembly code or raw binaries, into high-level, human-readable source code in order to understand program semantics, uncover malicious intent, and develop effective defensive signatures.

Despite the strong semantic modeling capabilities of transformer-based NLP models, reverse engineering pipelines still rely largely on traditional static and dynamic analysis techniques, which are labor-intensive, brittle under obfuscation, and difficult to scale. Even state-of-the-art decompilers struggle with stripped symbols, flattened control flow, and dynamic API resolution commonly used by malware [1, 2], motivating the need for domain-adaptive learning frameworks that directly model low-level code semantics.

Generative AI models, particularly LLMs, have enabled promising advances in program translation and code understanding by treating code as natural language [3, 4, 5]. However, most LLM-based decompilation methods rely on generic code pretraining and treat malware as out-of-distribution, limiting their ability to capture malware-specific obfuscation and anti-analysis patterns and reducing reliability in cybersecurity settings [6].

In this work, we argue that effective GenAI-based reverse engineering and decompilation require domain-specific representation learning and parameter-efficient task adaptation explicitly tailored to both benign and malicious software. In fact, we introduce LLM4CodeRE [1], a domain-adaptive LLM framework for bidirectional code reverse engineering that supports both assembly-to-source decompilation and source-to-assembly translation within a unified model.

By pretraining the backbone model on large-scale corpora of disassembled malware binaries and decompiled pseudo-code, we expose the model to obfuscation techniques, anti-debugging logic, and malware-specific API usage patterns that are characteristic of malicious programs. Inspired by recent advances in domain-adaptive pretraining for code models [7], LLM4CodeRE extends this paradigm to the malware domain and, to the best of our knowledge, is the first framework to explicitly target malicious binaries for LLM-based decompilation.

Moreover, we evaluate LLM4CodeRE on bidirectional code transformation tasks, including Assembly-to-Source (Asm→Src) and Source-to-Assembly (Src→Asm), using a unified evaluation protocol that measures three complementary dimensions: (i) semantic similarity, (ii) structural edit similarity, and (iii) re-executability of generated code in a sandboxed

Identify applicable funding agency here. If none, delete this.

[1]The model is available at: https://huggingface.co/JeloH/LLM4CodeRE-S2S-V1.

environment. Unlike prior work that focuses primarily on static similarity metrics [8, 9], our evaluation explicitly tests whether generated programs can be compiled and executed correctly, which is essential for real-world malware analysis and reverse engineering workflows. Our contributions can be summarized as follows:

- To the best of our knowledge, the first malware-aware causal language modeling (CLM) pretraining framework for LLM-based decompilation is introduced, enabling domain-specific representation learning from real-world malicious binaries.
- The first bidirectional reverse engineering framework is presented that supports both assembly-to-source decompilation and source-to-assembly translation, using a unified model capable of modeling both benign and malicious code behaviors.
- Multi-Adapters(MA) and Seq2Seq(S2S) Unified prefixing are proposed as a hybrid strategy for task-specific adaptation in bidirectional malware code reverse engineering.
- A unified evaluation framework is constructed to jointly measure semantic similarity, syntactic fidelity, and re-executability of generated code.
- A unified evaluation framework is constructed to jointly measure semantic similarity.

II. Related Work

A. Malware Decompilation and Reverse Engineering

The analysis and decompilation of malware binaries has long been a central challenge in cybersecurity. Traditional approaches rely on static analysis techniques such as control-flow graph extraction, function call graph reconstruction, and API usage profiling [1, 2]. Dynamic analysis frameworks execute suspicious binaries in sandboxed environments to observe runtime behavior and identify malicious patterns. While effective in controlled settings, these methods are often brittle under obfuscation, packing, and anti-analysis techniques, and they require substantial manual effort from expert analysts. As a result, they struggle to scale to the volume and diversity of modern malware [6].

B. LLMs for Code Understanding and Generation

Recent advances in large language models have substantially improved automated code understanding and generation. Models such as CodeBERT [3] and Qwen2.5-Coder [4] perform well on tasks including code summarization, translation, and synthesis, with instruction tuning further enhancing code reasoning [5]. However, most LLMs are pretrained on benign open-source code and are not explicitly adapted to the statistical and semantic characteristics of malicious software [7].

C. LLMs for Decompilation and Low-Level Code Translation

Recent work applies LLMs to decompilation and low-level code translation, including LLM4Decompile [8], WADEC [9], and ASMA-Tune [5]. While these approaches demonstrate the feasibility of LLM-based decompilation, they rely on generic or task-specific fine-tuning and do not perform domain-adaptive pretraining on malware corpora.

D. Parameter-Efficient Adaptation of LLMs

To reduce the cost of fine-tuning large models, parameter-efficient adaptation methods such as adapters [10], prefix-tuning [11], and Low-Rank Adaptation (LoRA) [12] have been proposed. These techniques enable task adaptation by introducing lightweight modules or low-rank updates while keeping the backbone parameters frozen. Although widely used in natural language processing and code generation, these methods have not been systematically explored in the context of malware decompilation or evaluated for their impact on both semantic fidelity and executable correctness in low-level code translation tasks.

E. Summary and Positioning

In contrast to prior work, this study integrates malware-aware CLM pretraining, LoRA-based parameter-efficient fine-tuning, and a unified comparison of Multi-Adapter and Seq2Seq Unified strategies within a single framework. Unlike existing LLM-based decompilers, the proposed approach explicitly targets malicious software as a first-class training domain and evaluates performance using a comprehensive protocol that includes semantic similarity, structural fidelity, and re-executability. This positioning distinguishes the framework from prior efforts that focus primarily on benign code, static similarity metrics, or single adaptation paradigms.

III. Methodology

A. Problem Formulation

Let $\mathcal{D} = \{(x_i, y_i, t_i)\}_{i=1}^{N}$ denote a dataset of paired program representations, where x_i is a low-level input sequence (assembly or binary), y_i is the corresponding high-level output sequence (source code or assembly), and $t_i \in \{\text{Asm} \rightarrow \text{Src}, \text{Src} \rightarrow \text{Asm}\}$ denotes the task type. The objective is to learn a conditional generation model $P(y \mid x, t)$ that translates between representations under task conditioning.

Given a shared backbone model with parameters θ and task-specific adaptation parameters ϕ_t, training minimizes the conditional negative log-likelihood:

$$\mathcal{L}_{\text{task}} = -\sum_{i=1}^{N} \log P(y_i \mid x_i, t_i; \theta, \phi_{t_i}). \tag{1}$$

Beyond static similarity, model outputs are evaluated for *functional correctness* by recompiling generated source code and executing it in a sandboxed environment. This formulation explicitly captures both syntactic fidelity and behavioral equivalence, which are essential for real-world malware analysis.

B. Framework Overview

Figure 1 illustrates the overall architecture of the proposed **LLM4CodeRE** framework. The design philosophy decouples domain-specific representation learning from task-level specialization. Raw malware binaries are processed using

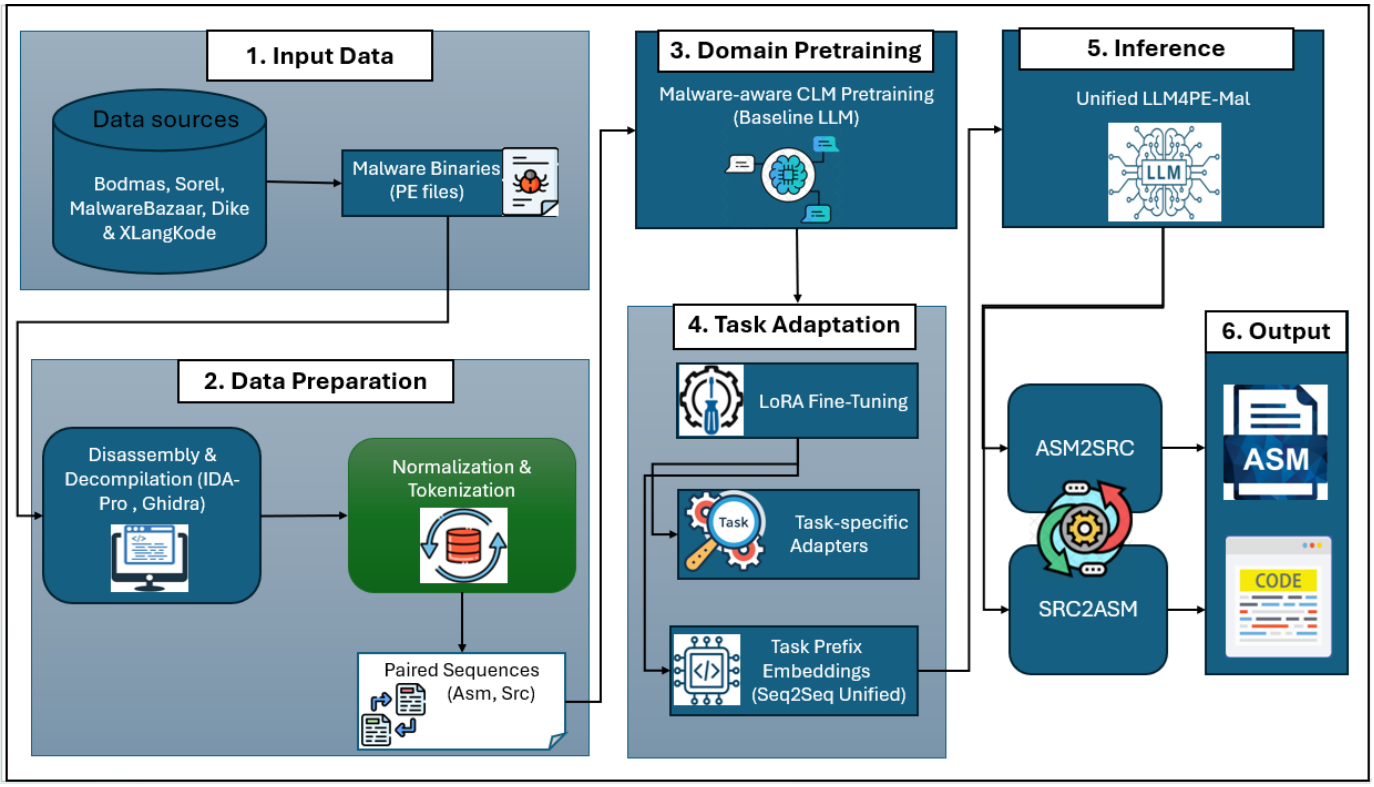

Fig. 1. System pipeline of the proposed **LLM4CodeRE** framework. Malware binaries are disassembled and normalized into paired assembly and source representations. A backbone LLM is pretrained using a causal language modeling (CLM) objective on malware corpora. Task adaptation is performed using LoRA and task-specific adapters or Seq2Seq Unified prefix tokens. The unified model supports bidirectional code transformation (Asm→Src and Src→Asm) and is evaluated using semantic similarity, edit similarity, and re-executability metrics.

disassembly and decompilation tools (e.g., IDA Pro, Ghidra) to extract assembly code and pseudo-code. These representations are normalized, tokenized, and aligned into paired sequences suitable for causal language modeling and supervised fine-tuning.

The framework consists of three hierarchical adaptation layers:

1) A **malware-aware backbone LLM** obtained via causal language model (CLM) pretraining on real-world malware corpora.
2) **Task-specific adapters** that specialize the backbone for different transformation tasks.
3) **LoRA low-rank updates** that enable parameter-efficient fine-tuning without overwriting domain knowledge.

Two complementary task adaptation strategies are supported: (i) a *Multi-Adapter* strategy with modular task heads, and (ii) a *Seq2Seq Unified* strategy using task-conditioned prefix tokens.

C. Multi-Adapter Strategy

The Multi-Adapter strategy introduces lightweight, task-specific parameter modules attached to a shared backbone model. Each adapter is responsible for learning transformations tailored to a specific task, such as Asm→Src or Src→Asm. This modular design significantly reduces fine-tuning cost while enabling flexible task specialization.

From an architectural perspective, adapters operate as residual transformation layers. Given a hidden representation $h \in \mathbb{R}^d$ from the backbone model, an adapter computes:

$$h' = h + W_2\,\sigma(W_1 h), \tag{2}$$

where $W_1 \in \mathbb{R}^{d\times r}$ and $W_2 \in \mathbb{R}^{r\times d}$ are low-rank projection matrices, $\sigma(\cdot)$ is a non-linear activation function (ReLU), and $r \ll d$ is the adapter bottleneck dimension.

By isolating task-specific knowledge within adapters, the framework avoids catastrophic forgetting when new tasks are introduced. Moreover, adapters can be dynamically activated or combined at inference time, enabling flexible multi-task execution without retraining the backbone.

D. Seq2Seq Unified Strategy

The Seq2Seq Unified strategy relies on a decoder-only causal language model augmented with task-specific prefix tokens. Instead of maintaining separate encoder–decoder pairs for each transformation, all tasks are unified under a single autoregressive decoding framework.

Let p_t denote the learned prefix embedding for task t. For an input sequence x, the model predicts output tokens autoregressively as:

$$P(y \mid x, t) = \prod_{k=1}^{|y|} P(y_k \mid y_{<k}, x, p_t). \tag{3}$$

The prefix conditions the model to perform the desired transformation (e.g., decompilation or recompilation) without modifying the core architecture. Figure 3 shows the relationship between the Multi-Adapter (MA) and Seq2Seq (S2S) strategies, both of which are considered for fine-tuning.

E. Malware-Aware CLM Pretraining

To equip the backbone model with domain-specific knowledge of malicious software, a large-scale causal language model (CLM) pretraining is performed on a curated corpus of real-world malware samples. The corpus consists of disassembled assembly code, decompiled pseudo-code, and recovered source-level artifacts extracted from PE binaries collected from public malware repositories and internal threat intelligence feeds.

Each malware sample is normalized via instruction canonicalization, register renaming, and address randomization to reduce syntactic noise. The resulting sequences are tokenized using a hybrid byte-level and instruction-aware tokenizer to preserve both opcode-level semantics and higher-level structural patterns.

Pretraining follows a standard autoregressive objective:

$$\mathcal{L}_{\text{CLM}} = -\sum_{t=1}^{T} \log P(x_t \mid x_{<t}; \theta), \tag{4}$$

where x_t denotes the next token in the malware sequence, and θ represents the backbone model parameters.

F. LoRA-Based Parameter-Efficient Fine-Tuning

To efficiently adapt the malware-pretrained backbone model to downstream code transformation tasks, Low-Rank Adaptation (LoRA) [12] is employed. Instead of updating the full parameter matrix $W \in \mathbb{R}^{d\times d}$, LoRA introduces two low-rank matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times d}$ such that:

$$W' = W + \Delta W, \quad \Delta W = BA, \tag{5}$$

Where $r \ll d$ is the adaptation rank.

LoRA modules are inserted into the attention and feed-forward projection layers of the decoder backbone. During training, the backbone weights remain frozen, and only the LoRA parameters are optimized.

Compared to full fine-tuning, LoRA significantly reduces memory footprint and training cost while mitigating catastrophic forgetting of malware-domain knowledge acquired during CLM pretraining. In the proposed framework, LoRA complements the Multi-Adapter strategy by capturing fine-grained task variations within each adapter module, yielding a hierarchical adaptation structure consisting of: (i) a malware-aware backbone, (ii) task-specific adapters, and (iii) LoRA low-rank deltas.

G. Training Protocol

The training configuration prioritizes efficiency by using a per-device batch size of 64 with 8-step gradient accumulation, yielding a large effective batch size while remaining within GPU memory limits. BF16 precision is employed together with the `adamw_torch_fused` optimizer to reduce computational overhead and improve training stability, enabling efficient optimization with a high LoRA learning rate (2×10^{-4}).

H. Evaluation Metrics

The framework evaluates decompilation quality using three complementary metrics that capture semantic, syntactic, and functional correctness.

a) Semantic Similarity: Semantic fidelity is measured using BERTScore, which computes token-level contextual similarity between generated and reference code sequences.

b) Edit Similarity: Structural fidelity is measured using normalized Levenshtein distance:

$$\text{Sim}_{\text{edit}} = 1 - \frac{\text{EditDist}(y, \hat{y})}{\max(|y|, |\hat{y}|)}. \tag{6}$$

c) Re-executability.: Functional correctness is evaluated by recompiling generated source code using GCC tools (v11.3) with optimization level `-O2`. Successful compilation is followed by sandboxed execution with time and memory limits. Samples that fail to compile or exceed resource limits are assigned a re-executability score of zero.

IV. Experiments and Results

A. Experimental Setup

a) Datasets: We use two complementary datasets for evaluation. For candidate LLM selection and compilation-based comparison, we employ the PE-Machine Learning dataset [13], consisting of Windows PE malware binaries, which is used solely to compare the compilation count of generated source code across models (Figure 3).

For all downstream experiments—including perplexity, edit similarity, and semantic similarity—we use the SBAN dataset [14]. SBAN provides aligned malware binaries and decompiler-recovered source representations for Assembly→Source and Source→Assembly tasks, aggregating five public malware corpora into 676,151 aligned samples (Table I).

TABLE I
Statistics of the SBAN Dataset Across Constituent Corpora. [14]

Dataset	Source	NLD	Assembly	Binary
1. BODMAS	93711	93711	92317	88605
2. MalwareBazaar	14746	14746	14051	13973
3. Sorel20m	81584	81584	81177	79166
4. Dike	17431	17431	12138	11726
5. XLangKode	468679	468679	5974	13299
Total	676151	676151	205657	206769

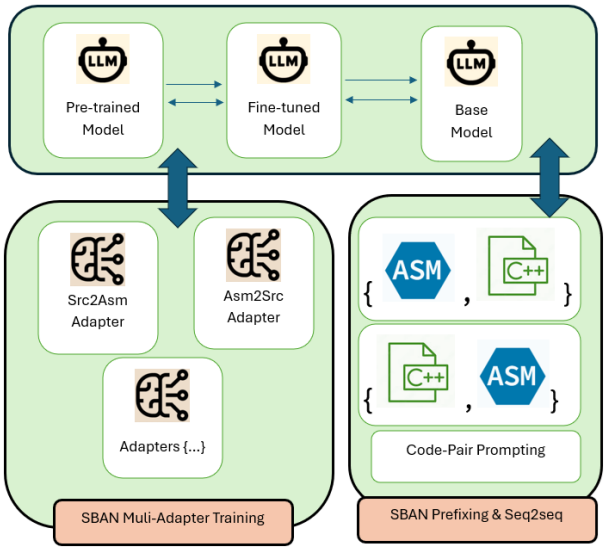

Fig. 2. Relationship between fine-tuning strategies and trained models for decompiling tasks.

b) Baseline Fairness and Comparison Protocol.: All baselines are evaluated under identical conditions, including tokenization, context length, datasets, and metrics. Backbone selection is performed on a disjoint screening dataset, and no method benefits from additional domain-specific supervision .

c) Decompilation Pipeline: All samples are drawn from datasets that were automatically decompiled using Ghidra 11.3.

d) Tokenization and Input Length: All models use a fixed maximum context length of 1024 tokens. We do not apply any truncation strategy; samples exceeding the token limit are excluded from training and evaluation to preserve full semantic content and prevent partial-function artifacts.

e) Candidate LLM Models: We evaluate the following candidate backbone models, as shown in Figure 3:

- DeepSeek-1.3B and DeepSeek-6.7B
- Qwen-1.5B and Qwen-7B
- Llama-3.2-1B and Llama-2-7B
- Mistral-7B
- Phi-3-small and Phi-4-mini

For candidate LLM selection, Figure 3 shows a Pareto chart of compilation counts on the PE-Machine Learning-200 dataset. The results exhibit a strong concentration effect: the top three models—Qwen-1.5B, DeepSeek-6.7B, and LLaMA-3.2-1B—account for over 80% of successful compilations, while the top five exceed 96

We further evaluate candidate LLMs using perplexity. Figure 4 shows that domain adaptation consistently reduces perplexity across all datasets and the four backbone language models. Overall, these results confirm that lightweight domain fine-tuning yields systematic, model-agnostic improvements, motivating its use for subsequent translation and similarity evaluation tasks.

B. Bidirectional Translation Performance

We assess bidirectional translation performance between assembly and source code using edit similarity and semantic similarity metrics. Figure 5 compares our **LLM4CodeRE** models (Multi-Adapter (MA) and Seq2Seq (S2S)) with **DeepSeek** and **LLM4Decompile** [15] across both Asm→Src and Src→Asm tasks, demonstrating consistent gains from unified bidirectional modeling and domain adaptation.

1) Assembly-to-Source (Asm→Src): Figure 5(a) shows that both LLM4CodeRE variants—the Multi-Adapter (MA) and Seq2Seq (S2S) strategies—outperform LLM4Decompile and DeepSeek on the Asm→Src task in terms of semantic and edit similarity. LLM4CodeRE (MA) achieves the highest semantic similarity (0.85) and edit similarity (0.63), followed by the S2S variant (0.81 / 0.61), while DeepSeek and LLM4Decompile obtain lower scores. These results indicate more faithful and semantically consistent source reconstruction with the proposed framework.

2) Source-to-Assembly (Src→Asm): Figure 5(b) shows that LLM4CodeRE also outperforms DeepSeek and LLM4Decompile on the more challenging Src→Asm task. The Multi-Adapter (MA) variant achieves the best performance (0.64 semantic, 0.27 edit similarity), followed by the Seq2Seq (S2S) variant, while both baselines obtain substantially lower scores. These results demonstrate the effectiveness of the proposed unified bidirectional training strategy and its robust generalization across translation directions.

3) Re-executability Analysis: Figure 6 shows re-executability results for the Asm→Src task on XLangKode. LLM4CodeRE (S2S) achieves the highest re-executability rate (86%), substantially outperforming LLM4CodeRE (MA) (53%), LLM4Decompile (48%), and DeepSeek (15%). These results highlight the importance of task-aware, unified

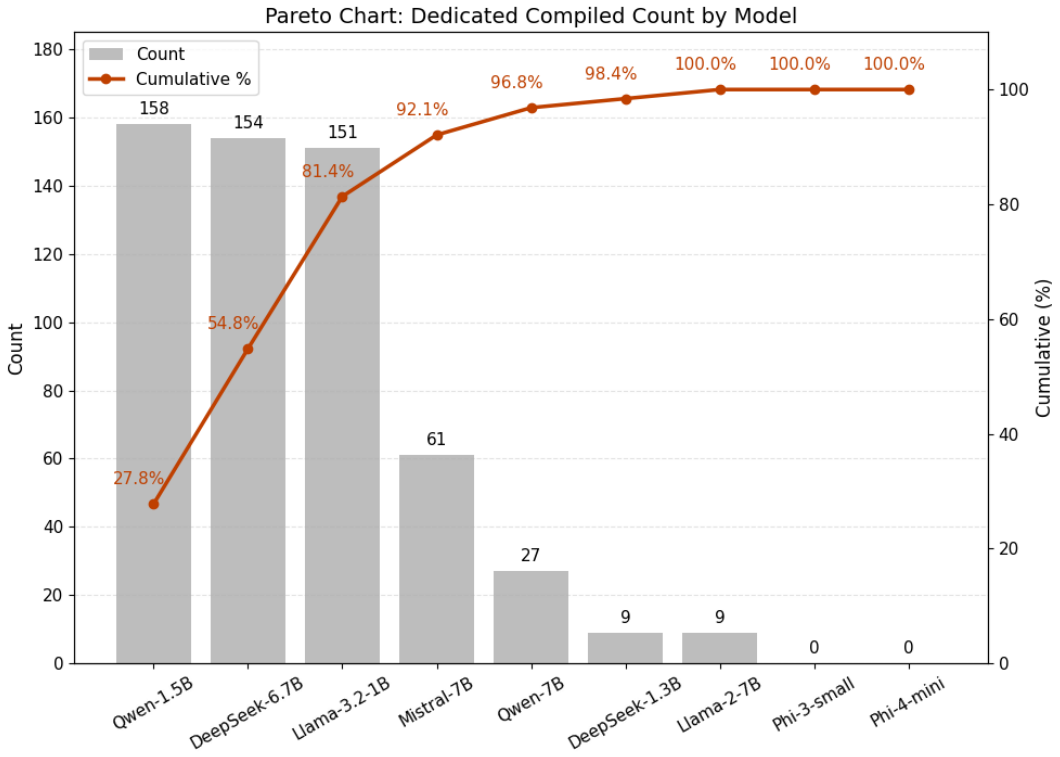

Fig. 3. Pareto chart of the dedicated compiled count on the PE-Machine Learning-200 dataset. Bars (left axis) show per-model counts (sorted in descending order), and the line (right axis) shows the cumulative percentage of the total count.

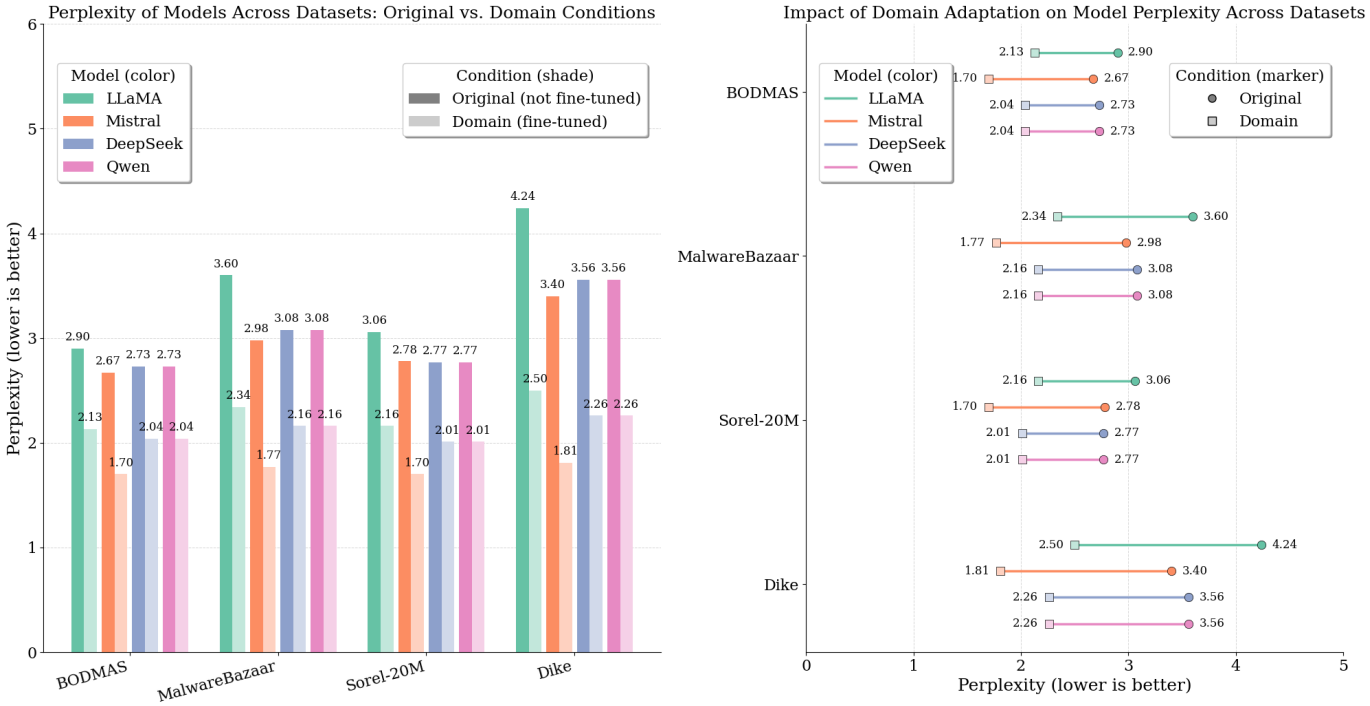

Fig. 4. Perplexity comparison across datasets for four language models under two settings: Original (non–fine-tuned) and Domain (domain–fine-tuned). Left: grouped bar chart reporting absolute perplexity on each dataset (lower is better). Right: dumbbell plot illustrating the per-model change from Original to Domain, where shorter (left-shifted) Domain markers indicate improved performance after domain adaptation.

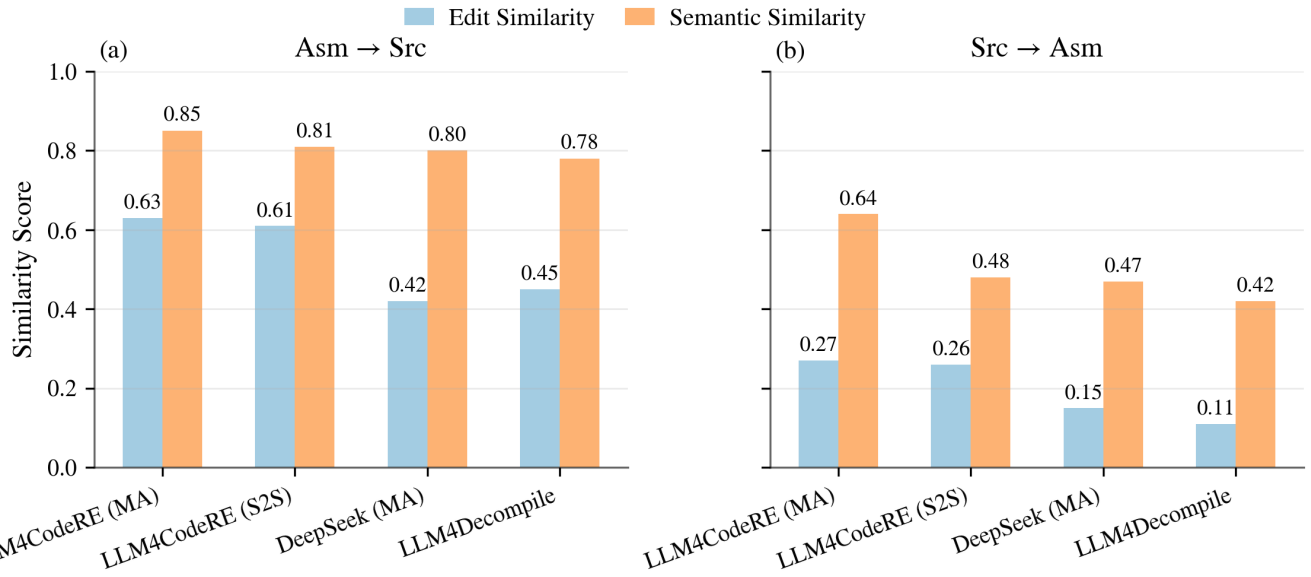

Fig. 5. Quantitative evaluation of conversion quality in both directions. (a) Asm-to-Src and (b) Src-to-Asm: Edit similarity and semantic similarity across models (higher is better)

bidirectional training for generating functionally correct and executable code

V. Limitations, Ethical Considerations and Future Work

A. Limitations and Threats to Validity

The proposed framework primarily targets Windows PE malware and may not generalize to other formats such as ELF or mobile binaries. Automated decompilation introduces label noise, particularly for heavily obfuscated samples. Re-executability is evaluated in a sandboxed environment with limited behavioral coverage, and embedding-based semantic metrics may not fully capture functional equivalence. Future work will explore cross-platform generalization and symbolic execution-based evaluation.

B. Executability Metric Formalization

Re-executability is defined as successful compilation with GCC (v11.3, `-O2`) [16, 17] followed by sandboxed execution [18] within resource limits. Programs failing compilation or execution receive a zero score, and results are reported as execution success rates.

C. Ground-Truth Approximation and Label Noise

Source-level references are derived via automated decompilation, providing an approximate [19] but standard ground truth. Uniform preprocessing and filtering are applied, and all models are evaluated under identical conditions to ensure fair comparison.

D. Length Bias and Dataset Coverage

All models use a fixed context length of 1024 tokens, with over-length samples uniformly excluded. The SBAN dataset aggregates multiple heterogeneous malware corpora, promoting robust generalization across diverse malware families.

E. Future Work

While this work focuses on Windows PE malware, future work will extend LLM4CodeRE to Android malware analysis. This includes supporting Android-specific representations such as APK packages, Dalvik bytecode (DEX), and smali code, as well as modeling Android framework APIs, and permission-based behaviors [20, 21] . We also plan to evaluate cross-platform generalization between Windows and Android malware in reverse engineering tasks.

VI. Conclusion

In this work, we introduced LLM4CodeRE, a malware-aware large language model framework for bidirectional code reverse engineering that unifies assembly-to-source decompilation and source-to-assembly translation within a single architecture. Extensive experiments on standard datasets demonstrate that LLM4CodeRE consistently improves semantic fidelity, structural alignment, and end-to-end re-executability compared to general-purpose code models and prior decompilation-focused methods. We believe that LLM4CodeRE provides a strong foundation for future research in executable code generation, automated malware analysis, and robust evaluation of LLM-driven reverse engineering systems.

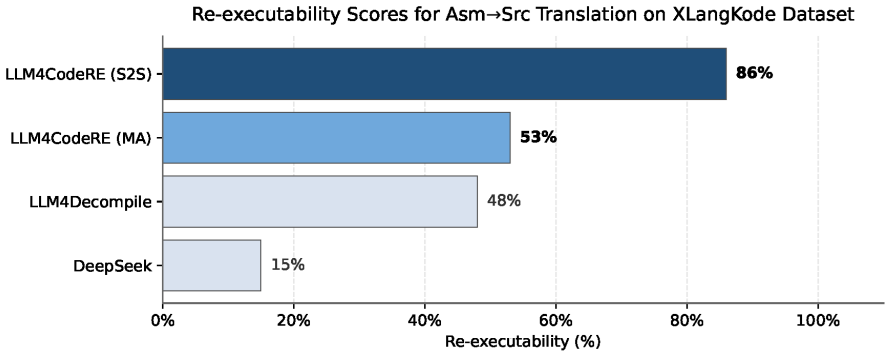

Fig. 6. Re-executability scores (percentage of translated programs that recompile and execute successfully) for assembly-to-source (Asm→Src) translation on the XLangKode dataset. Methods are ranked by performance; LLM4CodeRE (S2S) achieves the highest re-executability (86%), followed by LLM4CodeRE (MA) (53%), LLM4Decompile (48%), and DeepSeek (15%).

REFERENCES

[1] P. K. Tiwari, "Malware detection using control flow graphs," in *2024 2nd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT)*, 2024, pp. 216–220.

[2] M. Vu Minh, H.-T. Nguyen, H. V. Le, T. D. Nguyen, and X. C. Do, "A static method for detecting android malware based on directed api call," *International Journal of Web Information Systems*, vol. 21, no. 3, pp. 183–204, 2025.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[4] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," 2024. [Online]. Available: https://arxiv.org/abs/2409.12186

[5] X. Wang, J. Wang, J. Su, K. Wang, P. Chen, Y. Liu, L. Liu, X. Li, Y. Wang, Q. Chen, R. Chen, and C. Jia, "Asma-tune: Unlocking llms' assembly code comprehension via structural-semantic instruction tuning," 2025. [Online]. Available: https://arxiv.org/abs/2503.11617

[6] X. Hu, Z. Fu, S. Xie, S. H. H. Ding, and P. Charland, "Sok: Potentials and challenges of large language models for reverse engineering," 2025. [Online]. Available: https://arxiv.org/abs/2509.21821

[7] H. Jelodar, M. Meymani, and R. Razavi-Far, "Large language models (llms) for source code analysis: applications, models and datasets," 2025. [Online]. Available: https://arxiv.org/abs/2503.17502

[8] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," p. 3473–3487, 2024. [Online]. Available: http://dx.doi.org/10.18653/v1/2024.emnlp-main.203

[9] X. She, Y. Zhao, and H. Wang, "Wadec: Decompiling webassembly using large language models," *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[10] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," 2019. [Online]. Available: https://arxiv.org/abs/1902.00751

[11] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," 2021. [Online]. Available: https://arxiv.org/abs/2101.00190

[12] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[13] pracsec, "PE Malware Machine Learning Dataset," Jun. 2021. [Online]. Available: https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/

[14] H. Jelodar, M. Meymani, S. Bai, R. Razavi-Far, and A. A. Ghorbani, "Sban: A framework & multi-dimensional dataset for large language model pre-training and software code mining," 2025. [Online]. Available: https://arxiv.org/abs/2510.18936

[15] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024, p. 3473–3487. [Online]. Available: http://dx.doi.org/10.18653/v1/2024.emnlp-main.203

[16] G. Belinassi, R. Biener, J. Hubička, D. Cordeiro, and A. Goldman, "Compiling files in parallel: A study with gcc," in *2022 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 2022, pp. 1–8.

[17] H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "A neural network based gcc cost model for faster compiler tuning," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024, pp. 1–9.

[18] C. Greamo and A. Ghosh, "Sandboxing and virtualization: Modern tools for combating malware," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 79–82, 2011.

[19] F. Alotaibi, E. Goodbrand, and S. Maffeis, "Deep learning from imperfectly labeled malware data," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 3990–4004.

[20] J. Li, S. Chen, C. Wu, Y. Zhang, and L. Fan, "Foredroid: Scenario-aware analysis for android malware detection and explanation," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 1379–1393.

[21] M. Ibrahim, G. S. Tuncay, Z. B. Celik, A. Machiry, and A. Bianchi, "Lm-scout: Analyzing the security of language model integration in android apps," *arXiv preprint arXiv:2505.08204*, 2025.