Asm2SrcEval: Evaluating Large Language Models for Assembly-to-Source Code Translation

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Assembly-to-source code translation is a critical task in reverse engineering, cybersecurity, and software maintenance, yet systematic benchmarks for evaluating large language models (LLMs) on this problem remain scarce. In this work, we present the first comprehensive evaluation of five state-of-the-art LLMs on assembly-to-source translation. We assess model performance using a diverse set of metrics capturing lexical similarity (BLEU, ROUGE, METEOR), semantic alignment (BERTScore), fluency (Perplexity), and efficiency (time prediction). Our results reveal clear trade-offs: while certain models excel in text similarity metrics, others demonstrate lower perplexity or faster inference times. We further provide qualitative analyses of typical model successes and failure cases, highlighting challenges such as control flow recovery and identifier reconstruction. Taken together, our benchmark offers actionable insights into the strengths and limitations of current LLMs for program translation, establishing a foundation for future research in combining accuracy with efficiency for real-world applications.

1 INTRODUCTION

Assembly language, while powerful, presents significant challenges due to its machine-like syntax, lack of abstractions, and hardware dependence[1]. Programs written in assembly are hard to read, maintain, and debug, making development slow and error-prone[2, 3, 4]. Unlike high-level languages such as C or C++, which provide portability, modularity, and human-readable syntax[5], assembly requires deep hardware knowledge and is only practical for small-scale programs. These limitations in readability, scalability, and collaboration (Table 1) strongly motivate research into methods that can translate assembly code into higher-level, more understandable representations.

Previous studies have investigated assembly-to-source translation using rule-based or statistical approaches [6, 7, 8, 9, 10]. While these efforts demonstrated the feasibility of the task, they often relied on handcrafted rules or shallow machine learning methods [11, 12, 13], and thus struggled to capture semantic nuances across diverse assembly instructions [14, 15, 16].

Recent advances in large language models (LLMs) have shown strong capabilities in source-to-source translation, code synthesis, and natural language–to–code generation, making them natural candidates for tackling the challenges of assembly-to-source translation. Building on these strengths, several works have begun to explore LLMs for decompilation—for example, LLM4Decompile[17] demonstrated converting binaries into readable source code, [18] improved recompilability of decompiler outputs, and Decompile-Bench[19] introduced large-scale benchmarks to facilitate systematic evaluation. However, despite these advances, no work has specifically examined direct LLM-based translation from assembly to C++ code, leaving this as an open and practically significant research problem.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

Table 1: Comparison between Assembly (low-level) and High-Level (C++) programming languages.

Aspect	Assembly (Low-Level)	High-Level (C++ etc.)
Readability	Hard to read (machine-like, cryptic instructions)	Human-readable, closer to natural language
Maintainability	Very difficult to modify/debug	Easier to maintain and refactor
Portability	Hardware-specific, not portable across CPUs	Portable across architectures via compilers
Development Speed	Slow (manual registers, memory management)	Faster (loops, functions, data structures)
Error-Proneness	Easy to introduce subtle bugs	Safer abstractions, type-checking reduces errors
Abstractions	No functions, classes, or advanced data types	Supports OOP, modularity, libraries
Scalability	Only feasible for small programs	Suitable for large, complex software systems
Collaboration	Requires deep hardware knowledge (few can contribute)	Accessible to more developers, teamwork-friendly

In this work, we present a benchmark to evaluate five representative large language models (LLMs) on the task of translating assembly code into high-level source code (C++). We assess their performance using widely adopted automatic evaluation metrics, including BLEU, ROUGE, METEOR, BERTScore, Perplexity, and prediction time [20, 21].. Our findings reveal notable trade-offs between fluency and correctness, highlighting both the strengths and limitations of current LLMs. These results provide valuable insights into the challenges of assembly-to-source code translation and underscore its practical significance for software engineering, program comprehension, and security analysis.

2 RELATED WORK

Early efforts in assembly-to-source translation primarily relied on rule-based systems and statistical learning approaches. These methods mapped instruction patterns to higher-level constructs through handcrafted heuristics [6, 8, 10]. While effective on restricted subsets of assembly, they often lacked scalability and failed to capture the semantic nuances of diverse instruction sets [11, 12, 13]. More recent statistical and neural approaches have sought to improve upon these limitations by introducing learned representations of code, yet they still depend heavily on aligned training data and are brittle to out-of-distribution inputs [14, 15, 16].

In parallel, compiler-inspired decompilers and binary analysis frameworks have been developed to reconstruct high-level semantics from low-level code [7, 9]. These tools leverage static and dynamic analysis techniques to recover control flow, data structures, and variable names, but the resulting code is often verbose, hard to read, or not recompilable, limiting their usefulness for software maintenance and reverse engineering.

The emergence of large language models (LLMs) has opened a new avenue for program translation. LLMs trained on large-scale code corpora have demonstrated strong performance in code synthesis, translation, and bug fixing [20, 21]. Several recent studies have begun to apply LLMs to decompilation tasks: LLM4Decompile [17] showed that LLMs can translate binaries into human-readable source code, Wong et al. [18] focused on improving the recompilability of generated outputs, and Decompile-Bench [19] introduced a benchmark to standardize evaluation. However, these works do not specifically target direct assembly-to-C++ translation, nor do they provide a systematic comparison across multiple models and evaluation dimensions.

Our work fills this gap by introducing the first benchmark that evaluates a diverse set of LLMs on assembly-to-C++ translation. We assess performance along lexical, semantic, fluency, and efficiency dimensions, offering new insights into the trade-offs between accuracy and practicality in this important task.

3 METHODOLOGY

In this section, we outline the methodology used to evaluate large language models (LLMs) for the task of translating assembly code into C++. First, we provide an overview of the five selected LLMs and highlight their key features, training paradigms, and relevance to code translation. Next, we describe the evaluation metrics—BLEU, ROUGE, METEOR, BERTScore, Perplexity, and prediction time—and explain the specific goal of using each in assessing model performance. We then present the dataset and reference benchmark used in our experiments, including their composition and distinguishing characteristics. Finally, we report and analyze the experimental results obtained across the different models and metrics, offering both quantitative comparisons and qualitative insights.

3.1 LLM Models

To evaluate the task of assembly-to-C++ translation, we selected five representative instruction-tuned large language models (LLMs), chosen to cover a range of scales and architectures. The models can be divided into two groups based on their parameter size. The small-scale models include DeepSeek-Coder-1.3B-Instruct[22], Phi-4-Mini-Instruct (Microsoft)[23], and Qwen2.5-Coder-1.5B-Instruct[24], which are lightweight models (1–2B parameters) designed for efficiency and suitable for deployment in resource-constrained environments. These models are particularly attractive for edge computing and IoT scenarios, where inference speed and memory footprint are critical.

The larger-scale models consist of Llama-3.1-8B-Instruct (Meta)[25] and Mistral-7B-Instruct-v0.1[26], which have significantly more parameters (7–8B) and generally provide stronger reasoning and language generation capabilities. These models, while more computationally demanding, offer higher capacity for capturing long-range dependencies and complex patterns in code.

Across both groups, all models share a common focus on instruction tuning, meaning they are optimized to follow user prompts and generate context-aware responses. However, they differ in their design philosophies: DeepSeek-Coder and Qwen2.5-Coder emphasize code-centric pretraining, Phi-4-Mini is optimized for compact general-purpose reasoning, while Llama-3.1 and Mistral focus on broad multilingual and multi-domain adaptability. This diversity allows us to assess how model size and training specialization affect performance in the assembly-to-source translation task. Table 2 summarizes the key characteristics of the five models, highlighting their size, specialization, and distinctive features.

Table 2: Comparison of the selected LLMs used for assembly-to-C++ translation.

Model	Size	Key Features
DeepSeek-Coder-1.3B-Instruct	1.3B	Code-focused, instruction-tuned
Phi-4-Mini-Instruct	1.8B	Compact, efficient, general-purpose
Qwen2.5-Coder-1.5B-Instruct	1.5B	Code-centric, multilingual
Mistral-7B-Instruct-v0.1	7B	General-purpose, strong reasoning
Llama-3.1-8B-Instruct	8B	Broad coverage, multilingual

3.2 Evaluation Metrics

To assess the performance of LLMs on assembly-to-C++ translation, we employ six evaluation metrics: BLEU, ROUGE, METEOR, BERTScore, Perplexity, and Prediction Time. Each captures a distinct dimension of translation quality or practicality, and together they provide a comprehensive evaluation framework.

- BLEU (Bilingual Evaluation Understudy): Measures n-gram overlap between generated output and the reference code. It primarily evaluates syntactic correctness but may penalize valid variations in phrasing[27, 28, 29].
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a family of n-gram overlap metrics that emphasize recall by rewarding outputs which capture more of the reference content. Although originally developed for natural language evaluation, ROUGE has been widely applied across domains, including code translation. Its main variants include ROUGE-1 (unigram overlap), ROUGE-2 (bigram overlap), ROUGE-L (longest common

subsequence), and ROUGE-Lsum (a summarization-oriented extension). Each variant reflects a distinct inductive bias: ROUGE-1 measures token coverage, often yielding high scores even when tokens are scrambled; it is useful for assessing vocabulary correctness but ignores structural fidelity. ROUGE-2 captures local order by evaluating bigram overlap, rewarding adjacent-token correctness but penalizing small edits disproportionately—for example, inserting a modifier can disrupt many bigrams. ROUGE-L, based on longest common subsequence, evaluates global sequence alignment: it rewards preservation of overall token order while tolerating minor insertions or deletions, making it well-suited to distinguish meaningful control-flow consistency from benign stylistic changes. ROUGE-Lsum is a sentence-aware variant designed for summarization, but for code (where sentence boundaries are irrelevant) it closely tracks ROUGE-L without providing additional insights.

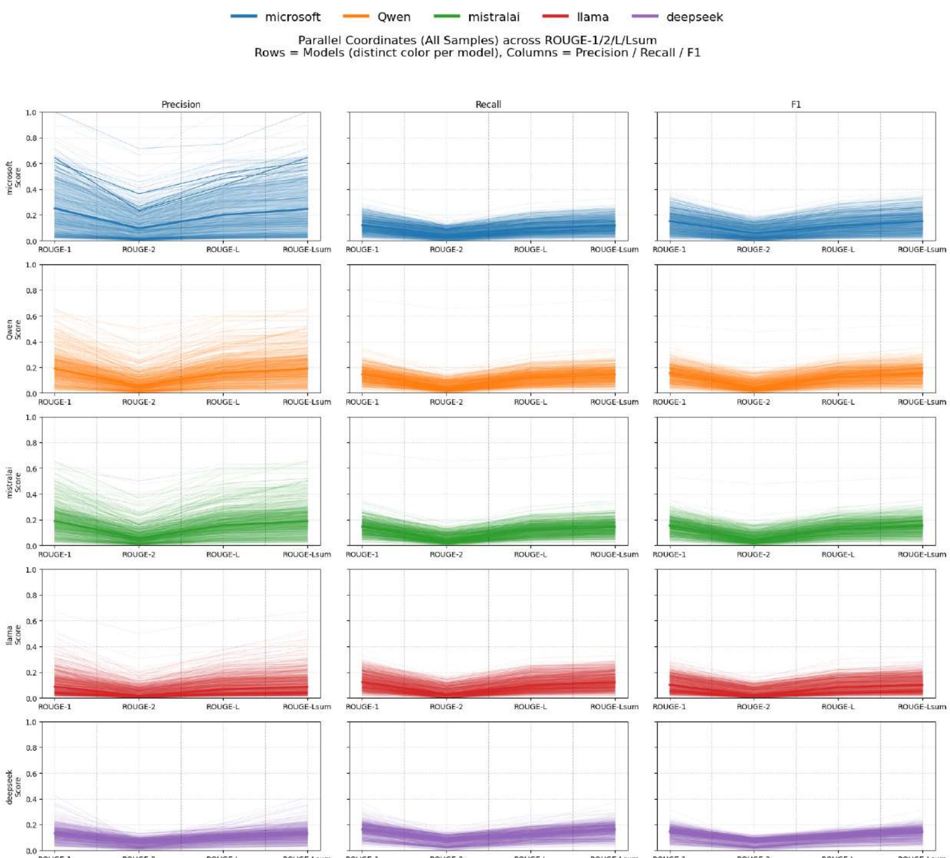

Figure 1: Parallel coordinate plots of five LLMs (Microsoft, Qwen, Mistral, Llama, DeepSeek) across ROUGE metrics (ROUGE-1, ROUGE-2, ROUGE-L, ROUGE-Lsum) for precision, recall, and F1. All models exhibit sharp performance drops on ROUGE-2 compared to ROUGE-1 and ROUGE-L. ROUGE-L and ROUGE-Lsum trends are nearly identical, indicating similar sequence-level and summarization-level matching. Precision generally exceeds recall, particularly for Microsoft and Qwen, suggesting models often produce plausible but incomplete outputs.

In our parallel-coordinates analysis (Fig. 1), all five models exhibit a pronounced trough at ROUGE-2 across Precision, Recall, and F1, reflecting the metric's brittleness to small local edits common in decompilation (e.g., toggling argument order, inserting casts, or relocating declarations). By contrast, ROUGE-1 often appears inflated: precision is high even when recall or token order is imperfect, since most tokens are present. ROUGE-L produces consistently higher and more stable curves than ROUGE-2 across all models, while ROUGE-Lsum overlaps ROUGE-L, confirming that sentence segmentation offers no additional value in this task. Collectively, these results suggest that sequence-aware yet order-tolerant matching provides the most reliable signal for assembly-to-source translation. Accordingly, we adopt ROUGE-L F1 as our primary ROUGE metric, since it balances precision (avoiding hallucinated tokens) and recall (capturing required tokens), while its LCS foundation preserves structural alignment—an aspect most closely aligned with compilable, semantically faithful code translations. For completeness, we also report ROUGE-1 and ROUGE-2 in the appendix.

- METEOR (Metric for Evaluation of Translation with Explicit ORdering): Incorporates stemming, synonyms, and word order, making it more sensitive to semantic equivalence compared to BLEU or ROUGE[30, 31].

- BERTScore: Uses contextual embeddings from pretrained transformers to measure semantic similarity. It can capture meaning preservation even when surface tokens differ[32, 33, 34]. Figure 2 presents the BERTScore evaluation results (Precision, Recall, and F1) for five large

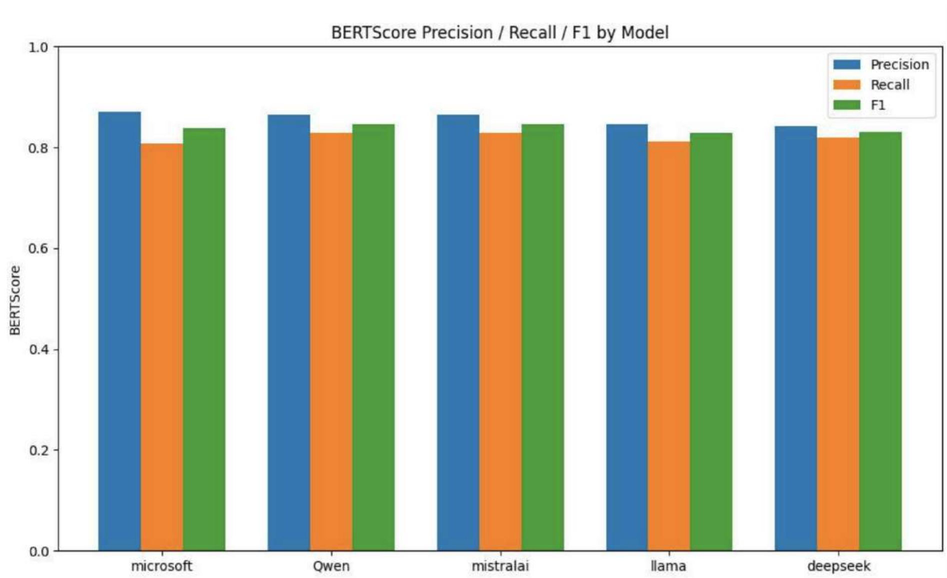

Figure 2: BERTScore evaluation (Precision, Recall, and F1) of five LLMs (Microsoft, Qwen, MistralAI, LLaMA, and DeepSeek) on the Assembly-to-Source Code Translation task.

language models—Microsoft, Qwen, MistralAI, LLaMA, and DeepSeek—on the Assembly-to-Source Code Translation task. BERTScore measures semantic similarity between the generated and reference code using contextual embeddings rather than relying solely on surface-level token overlap. As shown in the bar chart, all models achieve relatively high scores across the three metrics, with Precision typically slightly higher than Recall. For reporting purposes, the F1 score is selected as the representative metric since it balances both Precision (exactness) and Recall (coverage). This makes F1 a more robust indicator of overall performance, particularly in translation tasks where both accurate token generation and comprehensive coverage of the reference meaning are equally important.

- Perplexity: Quantifies fluency and naturalness by measuring how likely the generated sequence is under a model's probability distribution. Lower perplexity reflects greater confidence and smoother generation[35, 36].

- Prediction Time: Measures the computational efficiency of producing translations, expressed as the time per output. This metric is particularly important for deployment in real-time or resource-constrained environments, such as edge devices or IoT systems.

No single metric fully reflects translation quality and usability. Overlap-based metrics (BLEU, ROUGE, METEOR) capture surface-level correctness, embedding-based BERTScore measures semantic fidelity, perplexity reflects fluency, and prediction time assesses practical feasibility. Together, these metrics balance syntactic correctness, semantic accuracy, fluency, and computational efficiency—dimensions that are all essential for reliable and deployable assembly-to-C++ translation.

3.3 Dataset

For our experiments, we employed a subset of the SBAN dataset, a benchmark designed for analyzing assembly code. From this collection, we selected about 700 samples, which represents approximately 0.1% of the full dataset. This subset contains a mix of malware and benign code, reflecting the diversity of real-world assembly programs. Before use, the samples were preprocessed to ensure consistency: formatting was normalized, extraneous metadata was removed, and assembly instructions were paired with their corresponding source-level representations.

To provide a ground truth for evaluation, we used a reference dataset consisting of the C++ source code aligned with the same SBAN assembly samples. This C++ reference code was manually provided and verified by human experts, ensuring semantic correctness and eliminating potential ambiguities. By aligning assembly instructions with trusted source-level counterparts, this dataset enables a reliable assessment of translation quality.

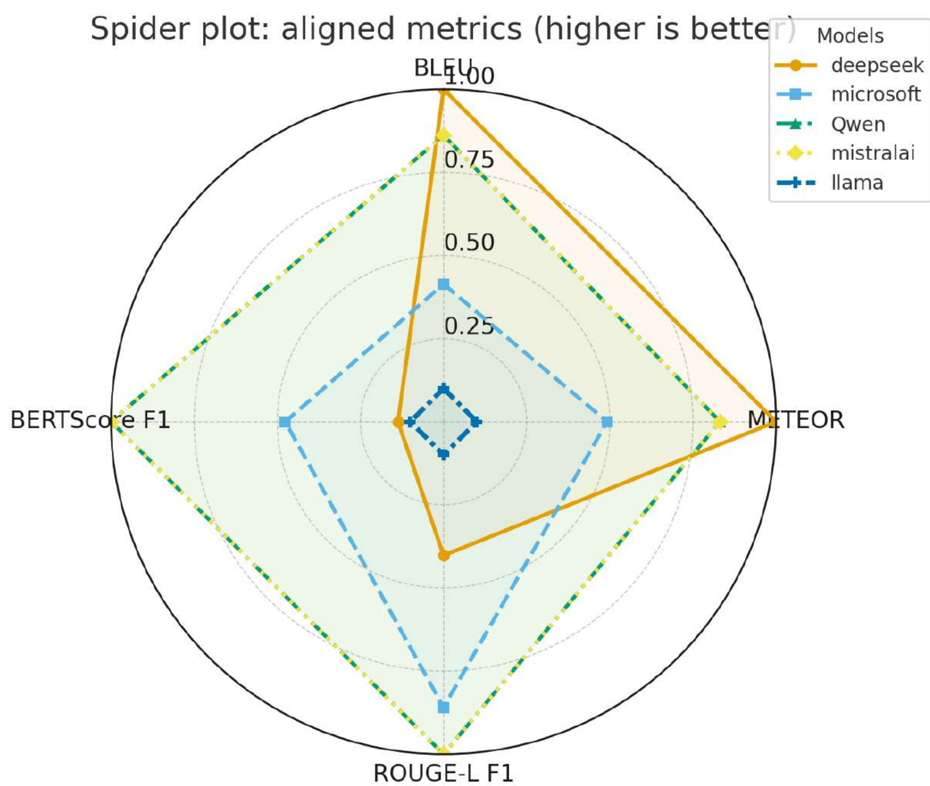

Figure 3: Spider plot comparing models across metrics that are aligned with performance (higher values indicate better results): BLEU, METEOR, ROUGE-L F1, and BERTScore F1. Each axis is min–max normalized across models to enable direct comparison. The plot highlights relative strengths and weaknesses of the models, with overlapping traces (e.g., Qwen and mistralai) indicating identical performance on these metrics.

3.4 Training Settings and Hyperparameters

All experiments were conducted on an NVIDIA H100 GPU, which provides high memory bandwidth and powerful parallel processing for efficient model training. We trained the models for 10 epochs with a batch size of 32 sequences per GPU, using a maximum sequence length of 512 tokens. The learning rate was set to 3e-5 with a linear warm-up over the first 1,000 steps, and optimization was performed using AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay 0.01) with gradient clipping at 1.0. These hyperparameters were carefully selected to ensure stable training, fast convergence, and optimal performance on our dataset.

3.5 Results

We evaluate the five LLMs on six metrics spanning correctness, semantics, fluency, and efficiency. Table 3 summarizes the average results across all models, reporting values for lexical metrics (BLEU, METEOR, ROUGE-L F1), semantic similarity (BERTScore F1), fluency (Perplexity), and efficiency (Prediction Time). Model size is also listed, expressed in billions of parameters, while prediction time is measured in seconds. This table provides a unified view of performance, complementing the visual analyses presented in the spider and bubble charts.

Aligned metrics. The spider plot in Figure 3 aggregates the aligned metrics—BLEU, METEOR, ROUGE-L F1, and BERTScore F1—where higher values indicate better performance. Large-scale models (Mistral-7B, Llama-8B) generally achieve higher values, especially on ROUGE-L and BERTScore, reflecting their greater ability to capture structural and semantic fidelity. Among the smaller models, Qwen-1.5B consistently performs close to the larger models, suggesting that efficiency-focused architectures can still yield competitive accuracy. Overlapping traces in the spider plot (e.g., Qwen and Mistral) highlight instances where small and large models converge on similar aligned performance. Non-aligned metrics. In contrast, the bubble chart in Figure 4 focuses on

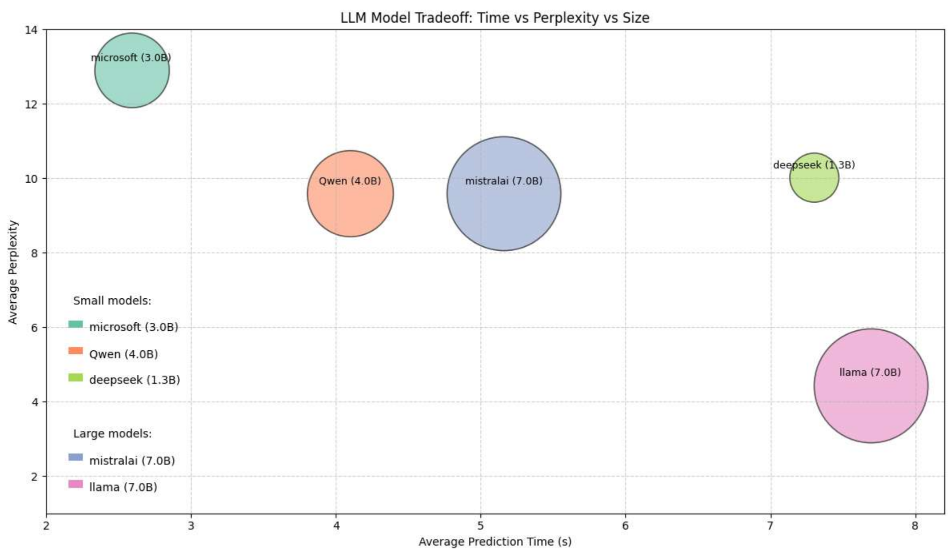

Figure 4: Bubble chart showing the efficiency trade-off across models. X-axis: average prediction time (s). Y-axis: average perplexity (lower is better for both). Bubble area $\propto$ parameter count (billions), color encodes the model. Models closer to the lower-left are more efficient; the legend groups small ($\leq$4B) vs large ($>$4B) models.

non-aligned metrics, where lower values are better. Here, the Y-axis denotes perplexity (fluency), the X-axis shows prediction time (efficiency in seconds), and bubble area encodes model size (in billions of parameters). Smaller models, such as Microsoft-Phi-3B, achieve substantially lower prediction times (as fast as 2.59 s), while DeepSeek-1.3B and Qwen-1.5B balance efficiency with competitive perplexity. Large models, particularly Llama-8B, deliver the lowest perplexity (4.44) but incur the highest inference latency (7.69 s), illustrating the cost of scaling. Overall comparison. Table 3, together with Figures 3 and 4, highlights the trade-offs between aligned and non-aligned metrics. Larger models dominate in lexical and semantic similarity but sacrifice inference speed, while smaller models are more efficient yet less consistent on accuracy-based metrics. These findings reveal that model choice depends on the intended application: accuracy-critical tasks benefit from large models, whereas efficiency-critical deployments favor smaller ones. This balance between billions of parameters and seconds of inference time underscores the dual challenge of achieving both correctness and practicality in assembly-to-source translation.

4 DISCUSSION

Our evaluation shows that model performance depends strongly on the trade-off between **accuracy** and **efficiency**.

Large-scale models. *Mistral-7B* and *Llama-8B* consistently achieve the best results on aligned metrics (BLEU, ROUGE-L, METEOR, BERTScore), reflecting their superior capacity for semantic and structural fidelity. *Llama-8B* also attains the lowest perplexity (4.44), but its prediction time is the slowest (7.69 s). *Mistral-7B* offers a better balance, combining strong accuracy with more moderate runtime.

Small-scale models. *Microsoft-Phi-3B* is the fastest (2.59 s) but lags on similarity metrics. *DeepSeek-1.3B* is similarly efficient but weaker overall. *Qwen-1.5B* stands out among small models, achieving competitive BLEU and BERTScore while maintaining reasonable prediction time (4.10 s).

Best models by use case. For accuracy-critical tasks (e.g., reverse engineering), *Mistral-7B* is the most practical, with *Llama-8B* providing the strongest fluency at higher cost. For efficiency-focused applications (e.g., real-time security analysis), *Microsoft-Phi-3B* is preferable. *Qwen-1.5B* represents the best compromise between the two extremes.

Takeaway. There is no single best model: large models maximize fidelity, while small models enable faster and more resource-conscious deployment. Future work should explore hybrid strategies (e.g., distillation, ensembles) to reduce this trade-off.

Despite providing new insights, this study has several limitations. First, our evaluation is conducted on a relatively small subset of the SBAN dataset (700 samples), which represents only a fraction of the diversity found in real-world assembly code. This restricted scale may limit the generalizability of our findings to broader domains such as obfuscated binaries or domain-specific instruction sets. Second, while we evaluate multiple dimensions of performance—including lexical similarity, semantics, fluency, and efficiency—our benchmark does not incorporate functional correctness checks such as recompilability or execution-based validation. Finally, we only benchmarked five instruction-tuned LLMs; extending the evaluation to larger and more diverse model families would provide a more comprehensive understanding of the trade-offs between accuracy and efficiency in assembly-to-source translation.

5 CONCLUSION

This work introduces the first comprehensive benchmark of large language models for assembly-to-C++ translation, evaluating five diverse models across lexical, semantic, fluency, and efficiency metrics. The study reveals trade-offs between semantic accuracy and inference speed, with larger models like Mistral-7B and Llama-8B excelling in fidelity, while smaller ones like Phi-3B and Qwen-1.5B offer faster, more deployment-friendly performance. Based on several samples from the SBAN dataset, the results mark a meaningful starting point but underscore the need for broader evaluations involving more diverse data and additional models. Future directions include exploring

Table 3: Comparison of models across evaluation metrics. Metrics with ↑ mean higher is better, and ↓ mean lower is better. Highest values are in green, lowest in red.

Metric	deepseek	microsoft	Qwen	mistralai	llama
Size (↓)	**1.3 B**	3 B	4 B	7 B	7 B
Perplexity (↓)	10.0233	12.9007	9.5993	9.5993	**4.4414**
Prediction Time (↓)	7.3 s	**2.59 s**	4.10 s	5.16 s	**7.69 s**
BLEU (↑)	**0.0426**	**0.0299**	0.0396	0.0396	0.0231
METEOR (↑)	**0.1733**	0.1383	0.1617	0.1617	**0.1113**
ROUGE-L F1 (↑)	0.1044	0.1245	**0.1307**	**0.1307**	**0.0911**
BERTScore F1 (↑)	0.8290	0.8345	**0.8430**	**0.8430**	**0.8284**

hybrid approaches (e.g., distillation, ensembles), incorporating functional correctness via human and compiler feedback, and expanding benchmarks to cover multilingual or domain-specific binaries. Overall, the study lays essential groundwork for advancing LLM-based program translation toward more robust, efficient, and real-world-applicable solutions.

References

[1] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[2] Tilman Mehler. Challenges and applications of assembly level software model checking. 2006.

[3] Wenbing Wu. Analysis of several difficult problems in assembly language programming. *Creative Education*, 10(7):1745–1752, 2019.

[4] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools*. pearson Education, 2007.

[5] Bjarne Stroustrup. An overview of the c++ programming language. *Handbook of object technology*, 72, 1999.

[6] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

[7] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[8] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.

[9] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *proceedings of the 38th annual computer security applications conference*, pages 508–518, 2022.

[10] Iman Hosseini and Brendan Dolan-Gavitt. Beyond the c: Retargetable decompilation using neural machine translation. *arXiv preprint arXiv:2212.08950*, 2022.

[11] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[12] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: A transformer-based model for decompiled variable name recovery. *NLP4Prog 2021*, page 48, 2021.

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

[14] Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of c decompiler fidelity issues. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 379–396, 2024.

[15] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 491–502, 2024.

[16] Zixu Zhou. Decompiling rust: An empirical study of compiler optimizations and reverse engineering challenges. *arXiv preprint arXiv:2507.18792*, 2025.

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

[18] Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

[19] Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. Decompile-bench: Million-scale binary-source function pairs for real-world binary decompilation. *arXiv preprint arXiv:2505.12668*, 2025.

[20] Taojun Hu and Xiao-Hua Zhou. Unveiling llm evaluation focused on metrics: Challenges and solutions. *arXiv preprint arXiv:2404.09135*, 2024.

[21] Nik Bear Brown. Enhancing trust in llms: Algorithms for comparing and interpreting llms. *arXiv preprint arXiv:2406.01943*, 2024.

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

[23] Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, Congcong Chen, et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. *arXiv preprint arXiv:2503.01743*, 2025.

[24] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[25] A. Grattafiori. The llama 3 herd of models. *arXiv preprint*, 2024.

[26] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b: A 7-billion-parameter language model engineered for superior performance and efficiency. *arXiv preprint*, 2023.

[27] Matt Post. A call for clarity in reporting bleu scores. *arXiv preprint arXiv:1804.08771*, 2018.

[28] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176, 2019.

[29] Ehud Reiter. A structured review of the validity of bleu. *Computational Linguistics*, 44(3):393–401, 2018.

[30] Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380, 2014.

[31] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[32] Michael Hanna and Ondřej Bojar. A fine-grained analysis of bertscore. In *Proceedings of the Sixth Conference on Machine Translation*, pages 507–517, 2021.

[33] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

[34] Tianxiang Sun, Junliang He, Xipeng Qiu, and Xuanjing Huang. Bertscore is unfair: On social bias in language model-based metrics for text generation. *arXiv preprint arXiv:2210.07626*, 2022.

[35] Clara Meister and Ryan Cotterell. Language model evaluation beyond perplexity. *arXiv preprint arXiv:2106.00085*, 2021.

[36] Zachary Ankner, Cody Blakeney, Kartik Sreenivasan, Max Marion, Matthew L Leavitt, and Mansheej Paul. Perplexed by perplexity: Perplexity-based pruning with small reference models. In *ICLR 2024 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2024.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction explicitly state that SBAN is a large-scale, multi-dimensional dataset integrating source code, binaries, assembly, and natural language descriptions, and the paper consistently supports these claims with dataset construction details and applications.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper includes a dedicated discussion of limitations, noting challenges such as dataset balance, potential biases, and difficulties in covering all programming languages.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not include theoretical results.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Dataset construction steps, data sources, and evaluation protocols are described in detail, and the dataset is openly released with clear documentation.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The SBAN dataset is publicly available on Hugging Face at `https://huggingface.co/datasets/JeloH/SBANx20250610/tree/main`, along with scripts and documentation to reproduce results.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We used a sub-datest of SBAN datset.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: No experiments with statistical evaluation are included.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We used GPU H100 for working on LLMs.

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The dataset uses legally distributable sources, and malware binaries are disarmed for safe release. Responsible use guidelines are provided.

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Positive impacts include enabling safer reverse engineering, improving software maintenance, and supporting cybersecurity research. Potential negative impacts include misuse for malware analysis or malware code analysis.

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [Yes]

Justification: Harmful binaries are released only in disarmed form, and access requires agreement to usage guidelines.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All reused datasets and code sources are cited in References.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: SBAN is released with full documentation describing dataset structure, preprocessing steps, and limitations.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The paper does not involve human subjects or crowdsourcing.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: No human subjects research was conducted.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research?

    Answer: [NA]

    Justification: The core contribution of the paper is the dataset; LLMs are evaluated on the dataset but not developed as a core method.