Beyond BLEU: A Semantic Evaluation Method for Code Translation

Anonymous ACL submission

Abstract

Code translation is one of the core capabilities of LLMs. However, evaluating the correctness of translations remains difficult, as commonly used metrics such as BLEU measure only syntactic similarity, disregarding program semantics. We propose a novel evaluation methodology for code translation tasks, emphasizing semantic equivalence over surface-level string similarity. Our approach applies established compiler testing methodology to a new domain, allowing the assessment of an LLM fine-tuned for binary lifting tasks (i.e. decompiling binaries to higher-level representations). We introduce a semantic correctness score, defined as the proportion of translations that produce correct execution outcomes, and demonstrate its application by evaluating LLM-based and heuristic decompilers. Our findings show that LLM-based approaches significantly outperform heuristic ones, while BLEU scores show negligible correlation with semantic correctness (r = -0.127 to 0.354), demonstrating that syntactic metrics fail to predict functional accuracy.

1 Introduction

Code translation is a key capability of large language models (LLMs), supporting a range of applications from cross-language interoperability (Dhruv and Dubey, 2025) to automated decompilation (Hu et al., 2024). Prior studies have often relied on *string similarity metrics*, e.g., the BLEU score, to assess the correctness of code translations, measuring surface-level resemblance between the translated and original programs (Lei et al., 2023; Aljagthami et al., 2025; Cummins et al., 2025). More sophisticated approaches such as RUBY (Tran et al., 2019) and CodeBLEU (Ren et al., 2020) consider code structure and incorporate AST in the calculation, but do not consider execution semantics. Such syntactic metrics are ill-suited for semantic-preserving decompilation tasks, which require evaluating semantic equivalence, i.e., that translated programs faithfully replicate the original program's semantics, rather than syntactic similarity: (a) programs that are functionally identical may differ in syntax, choice of identifiers, or structural organization, yet string similarity assigns them a low score, thereby underestimating correctness; (b) programs that share superficial textual patterns can achieve high similarity scores even when their behaviors diverge, resulting in misleading overestimation.

Another approach is to assess the correctness of code translation with tests (Chen et al., 2021). These tests run the translated program to check if it has the same functionality as the original program. This approach requires manual effort for creating each test case, and tests may not be

exhaustive, leaving the semantic equivalence of untested code in question.

In this work, we present a scalable methodology for evaluating semantic-preserving code translation, addressing the limitations of prior methods. Our approach automatically generates random programs, eliminating manual effort and enabling the creation of arbitrarily large evaluation sets. The generated programs compute check-sums after each instruction, offering greater reliability than string-based similarity metrics in checking of semantic equivalence between original and translated code.

We realize the new methodology using *Csmith*, a tool originally designed for automated compiler testing (Yang et al., 2011a), which produces compilable C programs with integrated checksum logic. This enables systematic, reproducible, and fine-grained evaluation of semantic-preserving code translation tasks. We use our evaluation methodology to assess the correctness of LLM-based decompilers, specifically, Meta's LLM Compiler (Cummins et al., 2025) [1], in the task of decompiling from x86 assembly to LLVM-bytecode. Our evaluation reveals that BLEU scores show no correlation with semantic correctness (point-biserial r = -0.13 to 0.35).

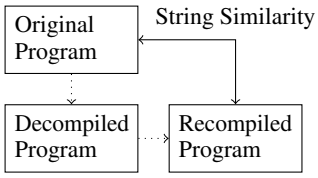
In summary, we contribute (1) an evaluation methodology for code translation based on automated program generation and semantic checksum validation, (2) a resulting *semantic correctness score* providing a direct measure of functional accuracy, and (3) empirical evidence that BLEU score is unrelated to semantic equivalence, showing the need for more sophisticated evaluation methods.

[1] Made available under (Meta LLMCompiler License Agreement, 2025)

2 Assessing Program Equivalence with String Similarity Metrics

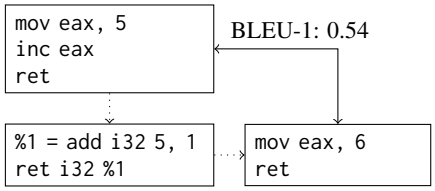
Decompilers translate the code of a low-level language to the code of a higher-level language, e.g., from x86 assembly to LLVM byte code (Avast, 2025; Yadavalli and Smith, 2019). An important property of decompilers is to preserve the semantics of the original program. More formally, two programs are observationally equivalent, if for all sequences of inputs they return the same sequence of outputs (Plotkin, 1977).

In general, deciding computationally if two programs have the same semantics is impossible due to the halting problem (Rice, 1953). As an approximate metric for semantic equivalence of programs, researchers have used string similarity metrics (Lei et al., 2023; Aljagthami et al., 2025; Evtikhiev et al., 2023; Tran et al., 2019). Specifically, the decompiled program is recompiled to the same language as the original program and then compared with a string similarity metric:

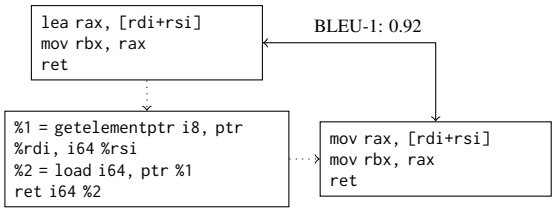


Two problems arise when using string-similarity as an approximation for program equivalence:

Problem 1: Programs are semantically equivalent, but get low string similarity score. Recompilation may optimize the program, leading to a low BLEU score between recompiled and original program:



Problem 2: Programs are not semantically equivalent, but receive high string similarity score. A slight syntactic variation may lead to different semantics, despite high BLEU score. A decompiler may wrongly interpret the `lae` instruction as a load from memory, which does not have the same semantics.



These examples show that string similarity metrics are unsuitable to assess if programs are semantically equivalent.

3 Methodology

In the field of compiler testing, semantic equivalence is a rigorous requirement used to validate the correctness of optimization steps: a program should compile to a binary with identical behavior regardless of the applied optimizations. A common way to test this property is through the automatic generation of programs whose executions can be compared to detect discrepancies. If two binaries produced from the same source yield different outcomes, the transformation has altered the program's semantics. Our evaluation methodology is inspired by this approach and implemented for the C language, though it is generalizable to any language.

We use Csmith (Yang et al., 2011b) to generate random C programs performing arbitrary computations and function calls. Binaries are produced at optimization levels `-O0` and `-O3`. Each program produces a CRC checksum over intermediate values when compiled and executed; differing checksums indicate non-equivalent semantics. This mechanism allows us to evaluate the translation accuracy of binary lifters.

Figure 1 illustrates the process. Csmith generates input programs, which are compiled to obtain reference checksums. The binaries are then processed by the decompiler under evaluation to produce source code, which is compiled and executed. The resulting checksum is compared to the reference to determine whether the lifted binary is semantically equivalent to the original.

The approach identifies and reports potential points of failure throughout the process, from decompiling to compilation and execution.

Semantic Correctness Score. We define the semantic correctness score as the proportion of successfully translated programs that produce matching checksums when executed:

$$\text{Semantic Score} = \frac{\text{\# programs with correct checksum}}{\text{\# tested programs}}$$

A score of 1.0 indicates perfect semantic preservation, while lower scores reflect the frequency of semantic errors.

4 Evaluating the correctness of Decompilers

Study Setup. We apply our methodology to evaluate the accuracy of Meta's LLMCompiler model and two widely used state of the art heuristic lifters, RetDec (Avast, 2025) and McToll (Yadavalli and Smith, 2019). In total, we generated 1024 test files, constrained to fit the model's 8192 token context window and excluding trivial programs. Csmith generation and program execution are deterministic, CPU-based

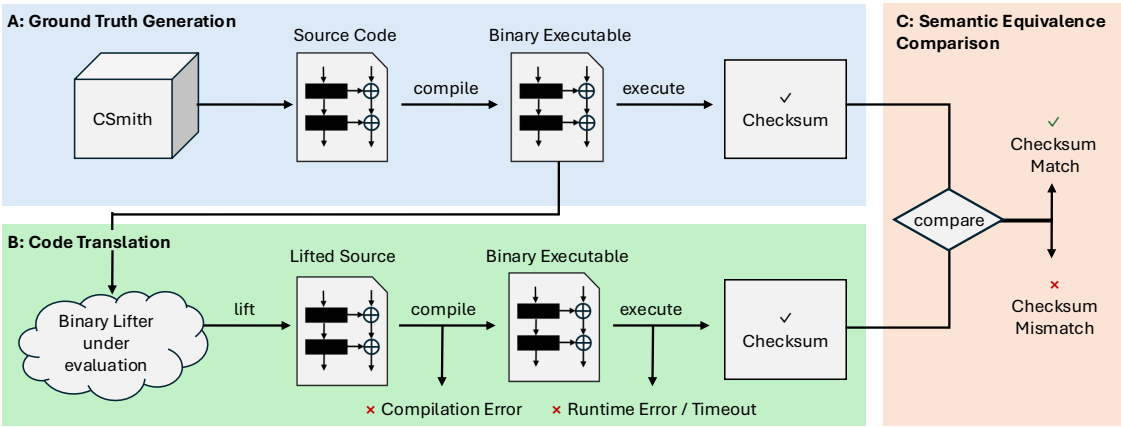


Figure 1: An overview of the evaluation framework.

processes requiring only seconds per program, producing only minimal overhead. We evaluate the 13b parameter LLMCompiler model, fine-tuned on the binary lifting task, at a temperature of 1.0. We perform a single experiment run, on two H100 GPUs, using a total of 88 GPU hours.

Comparative Performance. LLMCompiler substantially outperforms heuristic-based decompilers, achieving semantic correctness scores of 0.33 and 0.63 on O0 and O3 binaries, respectively. Neither Mctoll nor Retdec were able to produce a binary with successful execution, with Mctoll failing to lift any binaries, and RetDec producing only binaries that either failed to compile or crashed at runtime.

BLEU Scores Do Not Predict Semantic Correctness. We compute round-trip BLEU scores for all programs for which LLMCompiler produced compilable outputs. Table 1 lists point-biserial correlation coefficients between these similarity scores and semantic correctness, where values near 1 or -1 indicate strong predictive power and 0 indicating no correlation. The results reveal that BLEU scores fail to predict semantic correctness, with the strongest predictor (CodeBLEU at O3) showing only weak correlation. Mean similarity scores for succeeding and failing testcases are very close, indicating a near-overlap in distributions.

Table 1: LLMCompiler Round-Trip Similarity vs. Execution Result

Opt. Level Pass/Fail	Metric	Pass Mean	Fail Mean	Corr. (r)
O0 338/228	BLEU-1	0.81	0.84	-0.13^{**}
	BLEU-4	0.77	0.77	-0.03
	CodeBLEU	0.84	0.85	-0.05
O3 643/276	BLEU-1	0.71	0.71	0.02
	BLEU-4	0.60	0.57	0.18^{***}
	CodeBLEU	0.80	0.75	0.35^{***}

Significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

5 Risks and Limitations

Given that our framework builds on Csmith, it is currently limited to assessing semantic equivalence between only programs generated from C code and those derived from it. Including support for other languages would entail a rewrite of a comparable tool, accounting for any language-

specific constructs and semantics. Moreover, the code generated by our framework may lie outside of the distribution of training data used to train LLMCompiler. Nonetheless, the framework still provides a valuable measure of performance. A high semantic score would indicate strong applicability and a high level of semantic equivalence, and importantly, a good model should not be limited to its training data.

References

Aamer Aljagthami, Mohammed Banabila, Musab Alshehri, Mohammed Kabini, and Mohammad D. Alahmadi. 2025. Evaluating large language models for code translation: Effects of prompt language and prompt design. *Preprint*, arXiv:2509.12973.

Avast. 2025. Retdec: An open-source machine-code decompiler. Accessed: 2025-02-04.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction, CC 2025, Las Vegas, NV, USA, March 1-2, 2025*, pages 141–153. ACM.

Akash Dhruv and Anshu Dubey. 2025. Leveraging large language models for code translation and software development in scientific computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '25, page 1–9, New York, NY, USA. Association for Computing Machinery.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: How should we assess quality of the code generation models? *J. Syst. Softw.*, 203(C).

Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, volume 267622140.

Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. 2023. Creating a dataset for high-performance computing code translation using llms: A bridge between openmp fortran and c++. In *HPEC*, pages 1–7.

Meta LLMCompiler License Agreement. 2025. License for facebook/llm-compiler-13b. `https://huggingface.co/facebook/llm-compiler-13b/blob/main/LICENSE.pdf`. Accessed: 2025-10-07.

Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

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

S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, page 213–218, New York, NY, USA. Association for Computing Machinery.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011a. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011b. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294.

A Appendix

This appendix provides detailed evaluation results supporting our findings that (1) LLM-based code translation substantially outperforms heuristic approaches when measured by semantic correctness, and (2) syntactic similarity metrics fail to predict functional accuracy.

A.1 Complete Evaluation Results

Table 2 presents a comprehensive breakdown of all 1024 test programs across three lifters (LLMCompiler, McToll, RetDec) at two optimization levels (O0, O3). The table categorizes outcomes into distinct failure modes:

- **Lifting error**: The lifter failed to produce any output code
- **Compilation error**: Output code contained syntax errors preventing compilation
- **Runtime error**: Compiled code crashed (segmentation fault) or timed out (infinite loop)
- **Checksum error**: Code executed successfully but produced incorrect output
- **Checksum correct**: Code executed and produced semantically equivalent output

Table 2: Summary of evaluation, comparing the effectivity of LLMCompiler and the heuristic lifters.

Lifter	**LLMCompiler**		**mctoll**		**retdec**	
Opt. Level	**O0**	**O3**	**O0**	**O3**	**O0**	**O3**
Tested programs	1024	1024	1024	1024	1024	1024
Lifting error	0	0	1024	1024	0	0
Compilation error	458	105	0	0	181	38
Compilation success	566 (55.27%)	919 (89.75%)	0 (0.00%)	0 (0.00%)	843 (82.32%)	986 (96.29%)
Runtime error	26	12	0	0	843	982
Checksum error	202	264	0	0	0	4
Checksum correct	**338 (33.01%)**	**643 (62.79%)**	**0 (0.00%)**	**0 (0.00%)**	**0 (0.00%)**	**0 (0.00%)**

A.2 Visual Analysis of BLEU Score Distributions

Figure 2 visualizes the relationship between round-trip BLEU scores and semantic correctness for LLMCompiler outputs. Each box plot shows the distribution of similarity scores for programs with correct checksums (green) versus incorrect checksums (red), across three metrics (BLEU-1, BLEU-4, CodeBLEU) and two optimization levels (O0, O3). The substantial overlap between green and red distributions demonstrates that BLEU scores cannot reliably distinguish semantically correct from incorrect translations. Programs with high BLEU scores (>0.9) frequently fail semantic validation, while programs with lower scores (<0.7) often succeed. This visual evidence corroborates the correlation analysis in Table 1, confirming that syntactic similarity is an unreliable proxy for functional correctness in code translation tasks.

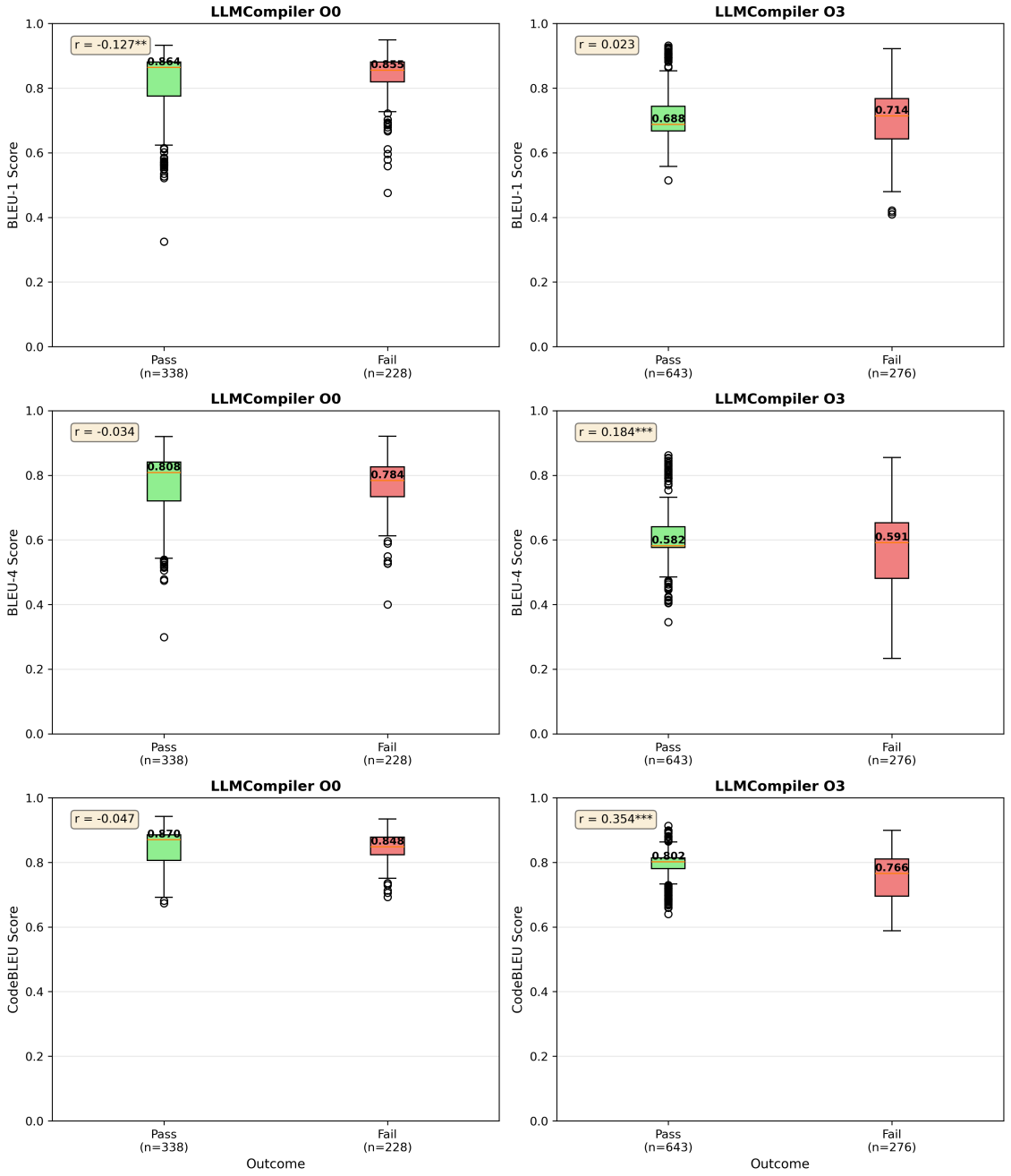


Figure 2: BLEU similarity scores comparing original assembly to round-trip assembly (compiled from lifted source), with green and red plots representing a checksum match or mismatch respectively. The plots show a distribution overlap between checksum match and checksum fail string simlilarity scores.