A Machine Learning Based Decompiler for x86 Assembly to C Code Translation

Archibald Emmanuel Carrion Claeys*, Fernando Arce Castillo†, Javier Alfredo Solano Saltachín‡
University of Costa Rica
San José, Costa Rica
Email: *archibald.carrion@ucr.ac.cr, †fernando.arce@ucr.ac.cr, ‡javier.solanosaltachin@ucr.ac.cr

***Abstract*—This paper presents a machine learning-based approach to decompile x86 assembly code (with 64-bit-adressing extensions) back to human-readable C source code. We leverage transformer-based language models, specifically DistilGPT2, datasets from large source-code datasets like The Stack and Exebench, and common metrics for Large Language Models. Our approach uses a fine-tuned model trained on the University of Costa Rica's institutional cluster with specialized GPU acceleration. The system demonstrates improved performance over baseline models, achieving a Levenshtein distance of 176.63 and BLEU score of 0.0055 for the fine-tuned version compared to the baseline's 260.89 and 0.0015 respectively. Nevertheless, other metrics like recall, precision, categorical cross-entropy loss, the f_1 score and perplexity appear to be very poor. This work contributes to automated reverse engineering and software analysis tools.**

***Index Terms*—Machine Learning, Decompilation, Assembly Code, Transformer Models, Reverse Engineering, Code Translation**

I. Introduction

A. Problem Definition

Compilation can be understood in layman terms as a text-processing task, which takes sentences of computer code in a higher-level, more abstract language, into equivalent sentences written in a lower-level, less abstract language. Decompilation can be then understood as the inverse procedure: taking sentences written in a lower-level language and writing equivalent sentences in a higher-level language. An example is illustrated on figure 1

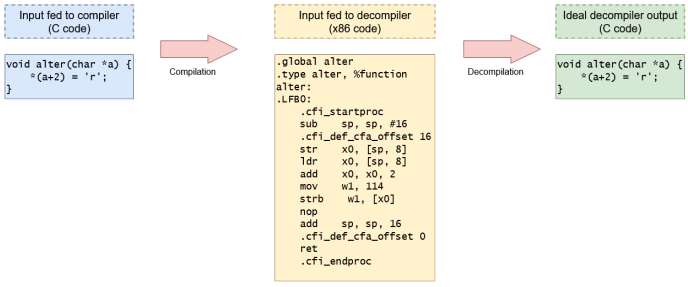

Fig. 1. Example of plausible input for decompiler, and ideal output

However, the decompilation task presumes the precedence of previous compilation: it's fundamentally more about the reversal of compilation than it is about the swapping of roles of the origin and destination languages involved in the presumed translation. This distinction comes into play when discussing the use cases and applications of decompilation in industry; like for example. the case of reverse engineering in cybersecurity applications [3] [4], which main use case is understanding the underlying workings of malware. Similarly, on software development portability efforts, it's easier to work with code written on a higher-level language due to its legibility and closeness in abstractions to the way people design software. Indeed, the goal is not merely to just end up with equivalent code in a higher-level language, but rather to also have it resemble the human-friendly structures and expressiveness that the original high-language source code — from which the low-level code is presumed to have came from — had in the first place.

Plainly put, we try to tackle decompilation from low-level, x86 ISA assembly code (with 64-bit extensions) into legible, human-readible and standard-conforming conforming C code (with no fixed standard), with a limited scope of single-source, single-destination file decompilation.

B. Related Work

1) Common challenges: Long before the introduction of Machine Learning approaches there were plenty of techniques applied to decompilation in virtue of domain-specific knowledge. Most can be summarized into four main distinct phases: syntax analysis, semantic analysis, data-flow analysis and control-flow analysis [1], each of which had their own approaches, such as pattern matching, analysis of control-flow graphs, type inference, and many more.

Nevertheless, many problems arise when attempting decompilation. Some are listed below as described by [1, p. 1-7]:

- *Recursive Undecidability*: More often than not for high-level languages, in the general case predicting which instructions may or may not execute is undecidable, which makes difficult the separation of instruction from data on von Neumann architectures.
- *Self-modifying code*: Some programs benefit from the ability to modify their own instructions, like security-critical encripted programs or viruses. Anticipating the ways on which programs modify themselves before translation is commonly also undecidable.
- *Architecture-dependent restrictions*: Some instructions carry operational semantics specific to some architectures,

which require some degree of emulation to fully anticipate the effects of.
- *Sub-routines included by the toolchain*: A great deal of instructions external to the main source code for a program are often required for it to be executable or functional at all. These routines can be embedded at compile-time or referenced and loaded at run-time. These dependencies are hard to account for, or decompile at all.

2) Principled approaches: Previous works on native x86 decompilation pre-existing the aid of Machine Learning techniques have existed for a while. The Phoenix decompiler developed by [5] makes use of a novel control-flow structuring algorithm that aims to produce functionally equivalent high-level C-code from a binary source, on virtue of analyzing control-flow graphs. Other intermediate representations distinct from control-flow graphs can also often used: as demonstrated by [2] by repurposing the Static-Single Assignment (SSA) form — commonly used in compiler optimization — for decompilation analysis; and as demonstrated by [4] by translation to intermediate LLVM IR code guided by encoding and semantics provided by an ISAC model, an intermediate bridge for the operational semantics in assembly back to a higher-level language.

Other industry-ready compilers based upon expert domain-knowledge of specific programming languages exist already. IDA [24], Ghidra [25] and RetDec [23] comprise some of the many examples freely-available. Nevertheless, they also suffer from poor readibility.

As showcased above, common theme in these approaches is the use of intermediate representations for the low-level code as a means of easier or richer analysis. This pattern is not lost on further Machine Learning assisted techniques, which attempt to exploit pre-existing relationships and structures — especially those related to recursive relationships present in language — to more accurately predict higher-level code with human-friendly readibility.

3) Machine Learning approaches: Machine Learning based approaches for decompilation often frame the problem as a text-processing problem; more specifically, as an instance of Neural Machine Translation (NMT). Thefore, the techniques employed are often borrowed from the field of Natural Language Processing (NLP), and more often than not make use of Sequence-to-Sequence (Seq2Seq) models [7]

[9] exemplified the use of Recurrent Neural Networks (RNNs) for binary-to-C decompilation, employing an encoder-decoder architecture. They first generate a corpus of C source-code data, which is then pre-processed into snippets of Abstract Syntax Trees (ASTs) which are then compiled into binary-C pairs. The pairs are tokenized and then fed into the model — C source code as inputs, binary as output — for training. Their tokenization scheme makes use of domain-knowledge analysis, and the training examples are bucketed according to length classes. Their work demonstrated the potential efficacy of tackling decompilation as a NMT problem with the aid of RNNs.

Multiple other studies share a similar approach, like [19] which used a transformer-based, encoder-decoder architecture instead for a Seq2Seq model.

[10] builds upon the work of [9] and attempts to "compensate for the differences between natural languages and programming languages" by augmenting the decompilation process with programming-languages domain-knowledge. Their approach makes heavy use of canonicalization of assembly and C-source code, alongside the preservation of structure via template-filling. For templated-code generation, they made use of a modified version of a standard encoder-decoder, Attention-based, model architecture "DyNmt" [8], and later compared the dependency graphs between the original source code and the predicted decompiled output. [16] similarly attempts to leverage the use of intermediate representations of high-level and low-level code. They translated high-level code into an intermediate, serialized representation with embedded semantic information but removed from the syntax structure of the high-level language. This intermediate representation is not only directly convertible back to the high-level code it was built from, but aids in compensating the information asymmetry between the high-level and the low-level code while resembling the syntactic structure found in low-level code. Likewise, they use self-attention mechanisms on the underlying model architecture. Both of these works expanded upon the NMT approach to decompilation by exploiting structure of programming languages, and making use of intermediate representations to aid in training Large Language Models (LLM)s.

[12] takes intermediate source code representation even further: by shifting the input and output spaces from texts to graphs they better account for the inherent structures in programming languages. Low-level code is represented as control/data flow graphs, while high-level source code is represented as ASTs. The source-code ASTs are predicted via inductive Graph Neural Networks (GNNs). Additionally, they split decompilation into sub-tasks: data-type resolution and source-code generation. The layers of their architecture are implemented by back-bone structural transformers and self-attention modules. Their work exemplifies the conceptual jump from framing decompilation purely as a text-processing task to reframing it as a graph-to-graph translation task, which in turn can be subdivided into mpre specialized subtasks on a pipeline.

Type inference as its own separate task is explored by [15], who try to predict the return type from functions constructed by decompilers with the aid of many machine-learning models beyond LLMs. Their problem is framed as multi-class classification, where the possible return types are the categories and detected patterns in the code — detected with the aid of LLMs — are extracted features.

[18] also tackles decompilation with the aid of intermediate graph representation of the low-level binary code taken as input, and a custom intermediate representation of the high-level C code as an output. Just like [12], they make use of GNNs to better capture dependencies in data and control flow

of low-level code, and then make use of Long-Short-Term-Memory (LSTMs) layers with a global attention mechanism to output the intermediate high-level code representation. Their work additionally leverages domain-specific knowledge to recover operands.

Due to time and logistic constraints, we opted to fine-tune causal, text-prediction LLM pre-trained on conversational tasks. Therefore, we frame the decompilation task as a question answering task. Further details are given on section II.

4) Efficacy measurements: There have been multiple proposed metrics for evaluating compilation and decompilation results, as well as code legibility. [6] proposed a methodology for validating compiler outputs. Its underlying principle lies in "dynamically executing a program on some test inputs", constructing a collection of variants which can then be compared for two distinct compilers and a given source code. Comparing these collections for equivalency can help find key differences in compiled results across compilers.

The previous methodology could be repurposed in order to compare compiled outputs for the same compiler but between a reference original source code, and a decompiled output. Further reading on C code decompilation correctness is available on [13].

Semantic correctness is not the only metric one could take to measure the efficacy of a decompiler. Code legibility is one of the key desired aspects that Machine Learning approaches to decompilation attempt to tackle. [21] proposes their own quantifiable metric for code readibility, specifically regarding the decompilation results of popular decompilers.

Automated approaches to measuring the efficacy of LLMs decompiling C code have also gone underway. [22] showcases a simple pipeline for generating pairwise examples of C and assembly code, generating decompiled outputs with both LLMs and common domain-knowledge, classic decompilers, and lastly evaluating their results. Their main metric for evaluating readibility and correctness is CodeBLEU, a composite metric [14] which combines the scores of the traditional BLEU metric, AST and data-flow matching scores, to better capture the structural and legibility similarities between the reference and generated high-level code.

Due to time and logistic constraints, we developed recall, precision, f_1, perplexity and categorical cross-entropy loss measurements. Further details are given on section II.

II. Methodology

III. Materials and Methods

This study relies on two large-scale datasets of C functions paired with x86_64 assembly code: **ExeBench** and **The Stack**. These datasets serve as the foundation for training and evaluating data-driven decompilation models. This section details the data acquisition process, filtering strategies, structure, and known limitations.

A. Dataset Overview

We employ two complementary datasets:

- **ExeBench**: A curated benchmark of executable C functions compiled at multiple optimization levels, designed for machine learning applications in compilation and decompilation [17].
- **The Stack**: A large-scale, permissively licensed source code repository that includes real-world C programs. While not originally intended for decompilation tasks, we compile and extract assembly representations from this dataset following a controlled pipeline [20].

B. Data Collection and Preprocessing

We collected examples by filterig x86 and C pairs from Exebench, and generating x86 inputs from The Stack. An overview of our process is illustrated by figure 2

1) ExeBench: We process the `.jsonl.zst` files for each split (`train`, `validation`, and `test`) using `zstandard` decompression. From each JSONL entry, we extract the C source function from the `func_def` field and the corresponding assembly from the `asm` field.

Assembly outputs include multiple compiler targets. We retain only those compiled with GCC for the x86 architecture (e.g., entries labeled `gcc_O0`, `gcc_Os`, `gcc_O3`), discarding all others. This decision ensures uniformity in instruction set architecture (ISA) and aligns with common practices in decompilation research [18].

Examples are included only if they contain valid assembly code for at least one of the selected optimization levels. Most commonly, `-O0` is present, with `-Os` and `-O3` available in a substantial subset of the corpus.

2) The Stack: To complement ExeBench with more realistic code, we extract a subset of C source files from The Stack. These were compiled using `GCC 10.2.1` on `Debian 10 (buster)`, targeting the `x86` ISA. Compilation was performed for optimization levels `-O0`, `-Os`, and `-O3`.

Files were discarded if they failed to compile, produced empty output, or required external dependencies. Successfully compiled examples were parsed to extract assembly code and paired with the original C function. Only files with at least one valid optimization level were retained.

C. Dataset Structure

Each valid example is stored as a JSON object with the following structure:

- `c`: the original C function code.
- `asm`: a dictionary mapping optimization levels (e.g., `O0`, `Os`, `O3`) to the corresponding x86 assembly code.

Examples are organized into directories according to their respective splits: `train/`, `valid/`, and `test/`. Assembly outputs are normalized to remove formatting artifacts, improving downstream processing without altering semantics.

D. Dataset Composition

The final dataset composition is summarized below, distinguishing between the ExeBench and The Stack corpora. Each example is considered valid if it contains a successfully compiled C function paired with its corresponding x86 assembly

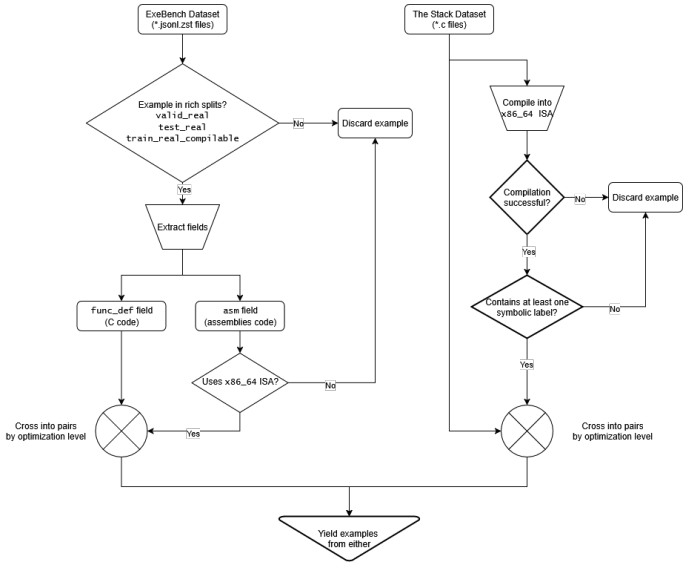

Fig. 2. Data generation and preprocessing pipeline overview

representation for at least one of the selected optimization levels (`-O0`, `-Os`, `-O3`).

- **ExeBench**:
  - Approx. 20,950 total examples in the full dataset.
  - All examples contain `O0`; around 70% also include `Os` and/or `O3`.
- **The Stack**:
  - Approx. 120,000 compiled examples considered initially.
  - A filtered subset of 348 valid examples was selected for evaluation purposes.
  - Over 85% of retained examples include multiple optimization levels.

Tables I and II provide a breakdown of optimization levels and examples across splits for both datasets.

TABLE I
DISTRIBUTION OF PROCESSED EXAMPLES FROM THE STACK BY SPLIT AND OPTIMIZATION LEVEL

Optimization Level	Train	Validation	Test
O0	93	12	11
Os	93	12	11
O3	93	11	12
Total	**279 (80.17%)**	**35 (10.06%)**	**34 (9.77%)**

TABLE II
DISTRIBUTION OF PROCESSED EXAMPLES FROM EXEBENCH BY SPLIT AND OPTIMIZATION LEVEL

Optimization Level	Train	Validation	Test
O0	5582	698	698
Os	5588	699	698
O3	5587	699	699
Total	**16757 (80.00%)**	**2096 (10.00%)**	**2095 (10.00%)**

E. Limitations

Despite the careful design and preprocessing of the datasets, several practical limitations must be acknowledged.

Storage space was a significant constraint. The total size of the dataset exceeded 60 GB, which can be challenging to manage in constrained computing environments or when reproducing results. The final dataset was deliberately capped to ensure accessibility and feasibility for training and evaluation.

Additionally, the **data preparation pipeline** required substantial **processing time**. Larger-scale processing—such as compiling all C files available in The Stack—was deemed infeasible within our compute budget, leading us to restrict the final corpus.

A further limitation involves the **compilability** of real-world source code. A significant number of C files from The Stack could not be compiled due to syntax errors, reliance on platform-specific headers, unresolved macros, or missing dependencies. These cases were automatically filtered out. Although this process ensured that all included samples were syntactically and semantically valid, it introduces a selection bias that favors simpler or more portable code.

Lastly, **example length** posed a constraint from a model architecture perspective. Transformer-based language models operate with fixed-length input windows. In our case, the DistilGPT [11] variant used during experimentation is limited to a context window of approximately 1024 tokens. Some valid examples—especially those containing large functions or multiple optimization levels—exceeded this limit and could not be processed end-to-end. Although we considered filtering out these longer samples, doing so would bias the dataset toward short and trivial functions, reducing its representativeness. As such, we retained these examples in the dataset, with the understanding that future work may require truncation, segmentation, or hierarchical modeling strategies to accommodate them.

F. Model

- **Problem formulation:** Assembly-to-C decompilation is cast as a *causal autoregressive* language-modeling task. At each timestep t the DistilGPT2 model predicts the next token w_t in the target C sequence conditioned on the full assembly prompt and all previously generated C tokens:

$$p(x) \;=\; \prod_{t=1}^{T} p_\theta(w_t \mid w_{<t})\,.$$

- **Architecture and pretrained weights:** We fine-tune *DistilGPT2*, a 6-layer, 82 M-parameter distilled variant of GPT-2, chosen for its moderate footprint and compatibility with our compute/storage constraints.
- **Input:** Prompts employ a multi-turn template (implemented in `model_loading.py`), for example:

  ```
  <|system|> You are a C decompiler model.
  <|user|> Decompile this x86_64 assembly:
  <|tool_start|> [x86 code] <|tool_end|>
  ```

  Special tokens `<|tool_start|>` / `<|tool_end|>` clearly delimit the assembly block.
- **Output:** After the `<|assistant|>` marker, the model generates the C source code:

  ```
  <|assistant|> [Predicted C code]
  ```

  Only these C tokens are unmasked and contribute to training.
- **Loss function:** We use *categorical cross-entropy* with token-level masking (implemented in `model_evaluation.py`). Prompt and assembly tokens receive a sentinel label (–100) and are excluded; only C-token positions incur loss:

$$\mathcal{L} = -\sum_{t \in \mathcal{T}_C} \log p_\theta(w_t \mid w_{<t}).$$

- **Metrics:** A custom `BatchDecompilerMetrics` class collects:
  - *Token classification:* accuracy, precision, recall, F1-score.
  - *Language modeling:* cross-entropy loss, perplexity.

G. Training

- **Sampling and splitting:** We stratify by dataset origin (ExeBench vs. The Stack) and by optimization level (O0, Os, O3). Each stratum is split 80/10/10 for train/validation/test, preserving the distribution of compile-time flags across splits.
- **Hyperparameters:** Configured via a `create_training_args()` helper:
  - *num_train_epochs*: user-specified (5–10).
  - *learning_rate*: 5e-5 with 100 warmup steps.
  - *weight_decay*: 0.01.
  - *gradient_accumulation_steps*: 8 for effective large batches.
  - *fp16*: enabled when CUDA is available.
  - *eval_strategy*: "steps," evaluating every 5% of total steps.
- **Data collation:** We employ `DataCollatorForLanguageModeling` (with `mlm=False`) for causal LM batching, ensuring correct padding and attention masks.
- **Compute environment:** Training runs under SLURM (see `test_pretrained.slurm`) on GPU nodes:
  - 2 × NVIDIA A100 80 GB per node, 64-core Intel Xeon CPUs, 16 GB RAM/core.
  - PyTorch, Hugging Face Transformers, Accelerate, CuPy.
  - Memory monitoring hooks log both host and CUDA usage (`model_training.py`).
  - Checkpointing: resume from existing checkpoint if detected; save every 5% steps with a total-limit of 1.
- **Metrics collection and visualization:** The `collect_training_metrics()` function parses the trainer's history to produce:
  - *Loss vs. steps/epochs* plots (training/validation).
  - *Token metrics* over time (precision, recall, F1, accuracy).
  - *LM metrics* (perplexity and cross-entropy).
- **Memory optimizations:** The system logs virtual memory and CUDA allocations each step; uses gradient accumulation and mixed-precision to fit large effective batch sizes on limited GPU memory.

IV. Results

V. Results and Analysis

Table III presents the comparative performance of our models across different training stages, demonstrating the progressive improvement achieved through fine-tuning the DistilGPT-2 model on assembly-to-C code translation tasks.

TABLE III
Model Performance Comparison

Model	Levenshtein Distance	BLEU Score
Base Model (pre-trained)	260.89	0.0015
Fine-tuned (minimalist)	176.63	0.0055

Our results demonstrate significant improvement over the baseline model. The fine-tuned model achieved a 32.3% reduction in Levenshtein distance (from 260.89 to 176.63) and a 267% improvement in BLEU score (from 0.0015 to 0.0055).

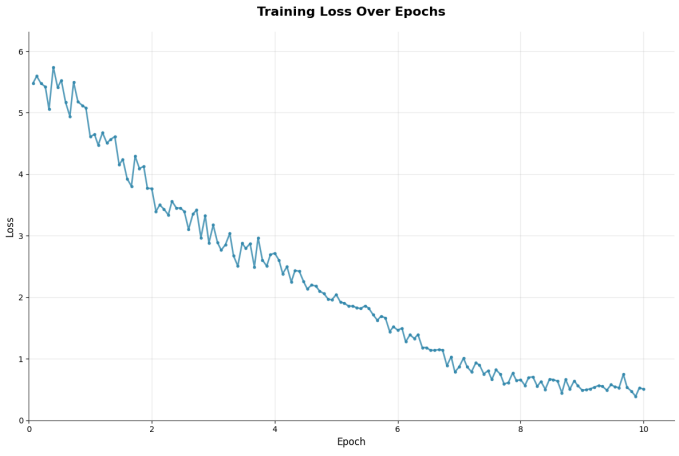

Fig. 3. Loss curve of the minimalist model showing constant convergence

A. Training Performance Metrics

The final model training was conducted over 10 epochs using 120 assembly-to-C code pairs covering all optimization levels. Table IV summarizes the key performance indicators achieved by the final model.

TABLE IV
FINAL MODEL TRAINING METRICS

Metric	Value
Accuracy	0.105
Precision	0.421
Recall	0.105
F1-Score	0.129
Cross-Entropy Loss	8.42
Perplexity	4537.0

The model achieved a final accuracy of 10.5% with a precision of 42.1%.

The F1-score of 0.129 reflects the harmonic mean between precision and recall, suggesting room for improvement in recall performance.

B. Training Dynamics Analysis

Figure 4 illustrates the training progression across 60 batches, revealing important insights into the model's learning behavior.

Fig. 4. Training metrics evolution over batches showing accuracy, loss, and perplexity trends

The training dynamics reveal several notable patterns:

- **Initial Learning Phase (Batches 0-10):** The model exhibited low accuracy (≈ 5-6%) with high perplexity values exceeding 6000, indicating initial difficulty in learning the assembly-to-C translation task.
- **Breakthrough Points:** Significant improvements occurred at batches 8, 24, 28, and 32, where accuracy jumped to 32.2%, 35.0%, 31.3%, and 43.1% respectively. These spikes correspond to perplexity drops to approximately 600, 510, 593, and 262.
- **Convergence Behavior:** After batch 32, the model showed more stable but lower performance, suggesting potential overfitting or reaching a local optimum in the loss landscape.

C. Large-Scale Model Experiment

TABLE V
LARGE-SCALE MODEL TRAINING AND EVALUATION METRICS

Metric/Parameter	Value
Evaluation Metrics	
Accuracy	0.020
Precision	0.020
Recall	0.018
F1-Score	0.019
Cross-Entropy Loss	21.526
Perplexity	2.23×10^9
Training Process Details	
Epochs	35.0
Steps	9344
Training Runtime (seconds)	20271.39
Training Samples/Second	29.414
Training Steps/Second	0.461
Training Loss (Epoch 35)	0.263
Validation Loss (Final)	0.324

During our experimental trials, we trained model with significantly larger hyperparameters and datasets which showed exceptional promise for the assembly-to-C translation task. This larger architecture, with increased dataset, generations and overall computational complexity, demonstrated superior learning dynamics compared to our standard minimalist implementation. However, due to computational resource limitations and extended training requirements, this experiment could not be completed to full convergence.

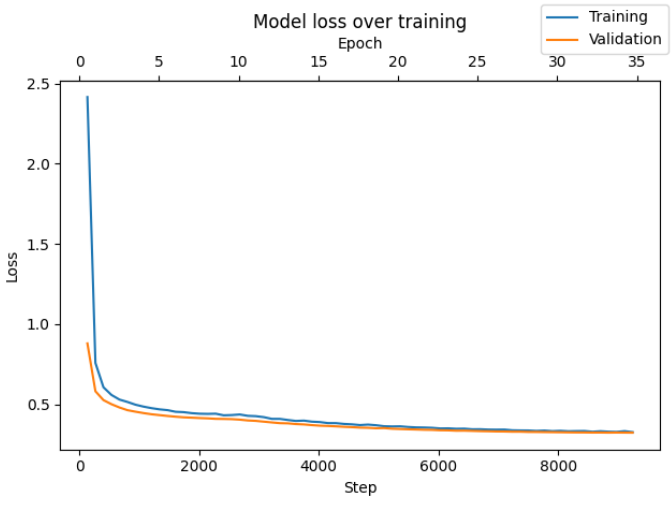

Fig. 5. Training and validation loss curves for the large-scale model showing rapid initial convergence before computational termination

Figure 5 illustrates the loss progression of this large-scale training run. The model exhibited remarkably rapid convergence, with both training and validation losses dropping from approximately 2.5 to below 0.5 within the first 2000 steps.

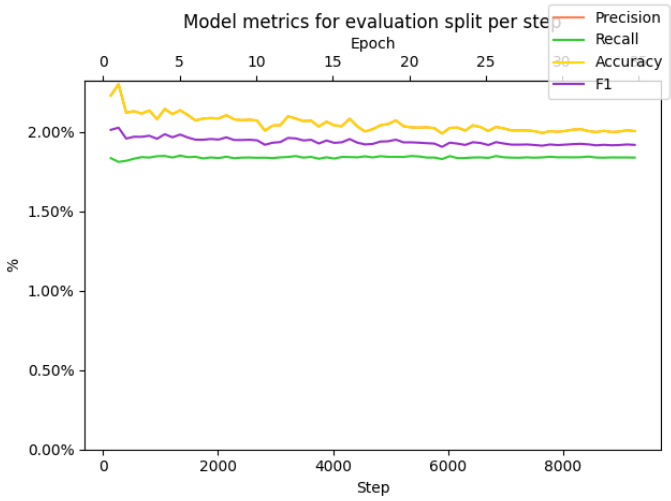

Fig. 6. Evaluation metrics progression for the large-scale model showing consistent improvement across all performance indicators

The evaluation metrics shown in Figure 6 reveal sustained improvement across all performance indicators throughout the training period. The accuracy stabilized around 2.0%, while precision remained consistently above 1.8%, and recall maintained steady performance around 1.8%. The F1-score demonstrated stable convergence at approximately 2.0%, suggesting more balanced precision-recall performance than our completed smaller models.

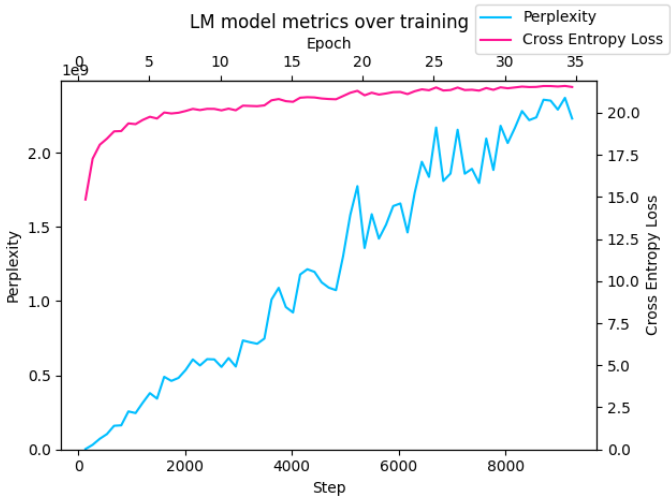

Fig. 7. Language model specific metrics showing perplexity and cross-entropy loss evolution for the large-scale model

Figure 7 presents the language modeling metrics for this large-scale training run. The cross-entropy loss exhibited steady improvement, stabilizing around 20.0 after initial fluctuations. Interestingly, the perplexity showed an upward trend reaching values above 200, which occurred alongside improving accuracy metrics. This pattern suggests the larger model was learning to generate more diverse and complex C code structures, potentially indicating superior translation capabilities that could not be fully realized due to computational constraints.

D. Performance Validation

The cross-entropy loss of 8.42 and perplexity of 4537.0 indicate the model's uncertainty in predicting the next token in the sequence. This explain the pseudo-random code generation of model, which seems to successfully create output similar to compilable C code, it still struggle to generate code related to the given assembly input.

E. Comparative Analysis

When compared to the baseline results, our fine-tuned model shows substantial improvements in code generation quality:

- The 32.3% reduction in Levenshtein distance indicates that the generated C code is structurally closer to the reference implementations.
- The 267% improvement in BLEU score for the fine-tuned model demonstrates enhanced semantic similarity between generated and target code.
- The progression from base model to fine-tuned shows consistent improvement in translation quality metrics.

These results validate some degree of effectiveness in our "minimalist" fine-tuning approach for the assembly-to-C translation task, despite the inherent challenges in cross-language code generation. The large-scale setup, however, doesn't yield promising results.

VI. Conclusions

Both of our approaches show mixed results. Both figures 3 and 5 showcase improvement on the models' prediction capability on the training split over time. In particular figure 5 showcases a steep loss decrease curve, avoiding particularly overfitting the validation split.

Nevertheless, the recall, precision, and f_1 measures were very poor on both setups. Figure 6 showcases them settling around a score of 2% for the large fine-tuning setup. Even worse, figure 4 showcases unstable progress for these metrics in the "minimalist" setup, peaking at around 0.6% for precision, 0.4% for recall and 0.45% for the f_1 score. Granted, these metrics were collected by evaluating the model on one batch at a time, but one would still expect certain uniformity to appear.

The metrics collected after training are not very promising for both setups. Table IV showcases very poor performance on the "minimalist" setup model after training. Even worse, table V demonstrates significantly worse metrics on the "large-scale" setup — smaller by an order of magnitude even, except for perplexity, which is many orders of magnitude larger and exemplifies more uncertainty for predicting tokens on average.

These observations suggest that fine-tuning the model on a large-scale yielded a lesser performant LLM for decompilation. Experimental conditions and errors in implementation may be to blame. It may also be the case that a considerable

amount of examples were trimmed considerably or possibly entirely due to the maximum input length for the model.

Nevertheless, there is slight improvement for the "minimalist" setup fine-tuned model in comparison to the base model, as suggested by Levenshtein distance and the BLEU score displayed on table III. Further attempts to improving the experimental setups should consider collecting CodeBLEU [14] and R2I [21] scores for judging decompiler quality.

VII. Discussion

The results indicate that fine-tuning transformer models on domain-specific assembly-to-C translation tasks yields substantial improvements in decompilation quality. The reduction in Levenshtein distance suggests that our "minimalist" model generates C code that is structurally closer to the ground truth, while the improved BLEU scores indicate better semantic alignment.

Further attempts on similar setups should make note of our current limitations:

- The model performance is evaluated on specific compiler optimizations and may not generalize to all optimization levels; make sure to propsoe balanced splits and account for bias
- The model of choice is not the preferred choice on state-of-the-art literature; opt for encoder-decoder architectures and attention mechanisms, perhaps even GNNs to better capture dependencies in control and data flows for low-level input code and ASTs for high-level code outputs.
- The maximum length for text for single-source inputs and outputs is considerably limited, limited even further to make room for the text prompt fed to the model; opt to filter out examples which may not fit completely into the model or attempt to fine-tune models with larger input and output sizes
- There was limited evaluation on complex control flow structures and advanced C language features; opt for CodeBLEU, R2I and edit distance

We hipothesize that perhaps opting for different model architectures may improve output quality for generated high-level code, as mentioned on section I

VIII. Acknowledgments

The authors thank the University of Costa Rica for providing access to the institutional High-Perfomance Computing (HPC) cluster. The computational resources and infrastructure support were essential for conducting this research.

IX. Availability

The source code and implementation details are available at:
https://github.com/archibald-carrion/decompiler.git

References

[1] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, Jul. 1994, 56 pp. [Online]. Available: https://rgaucher.info/pub/decompilation_thesis.pdf.

[2] M. J. Van Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, University of Queensland, 2007, 334 pp. [Online]. Available: http://www.backerstreet.com/decompiler/vanEmmerik_ssa.pdf.

[3] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, ser. CERIAS '10, West Lafayette, Indiana: CERIAS - Purdue University, 2010.

[4] L. Ďurfina et al., "Design of a retargetable decompiler for a static platform-independent malware analysis," in *Information Security and Assurance*, T.-h. Kim, H. Adeli, R. J. Robles, and M. Balitanas, Eds., Berlin, Heidelberg: Springer, 2011, pp. 72–86, ISBN: 978-3-642-23141-4. DOI: 10.1007/978-3-642-23141-4_8.

[5] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," presented at the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 353–368, ISBN: 978-1-931971-03-4. Accessed: Apr. 11, 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13 / technical - sessions / presentation / schwartz.

[6] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, Jun. 5, 2014, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2666356.2594334. Accessed: Apr. 11, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2666356.2594334.

[7] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. arXiv: 1409.3215. [Online]. Available: http://arxiv.org/abs/1409.3215.

[8] R. Aharoni. "Dynmt, a dynet based neural machine translation. "[Online]. Available: https://github.com/roeeaharoni/dynmt-py.

[9] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso: IEEE, Mar. 2018, pp. 346–356, ISBN: 978-1-5386-4969-5. DOI: 10.1109/SANER.2018.8330222. Accessed: Apr. 2, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/8330222/.

[10] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, *Towards neural decompilation*, 2019. DOI: 10.48550/ARXIV.1905.08325. Accessed: Apr. 2, 2025. [Online]. Available: https://arxiv.org/abs/1905.08325.

[11] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter," *CoRR*, vol. abs/1910.01108, 2019. arXiv: 1910.01108. [Online]. Available: http://arxiv.org/abs/1910.01108.

[12] C. Fu, K. Yang, X. Chen, Y. Tian, and J. Zhao, "N-bref : A high-fidelity decompiler exploiting programming structures," Oct. 2, 2020. Accessed: Apr. 2, 2025. [Online]. Available: https://openreview.net/forum?id=6GkL6qM3LV.

[13] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA: ACM, Jul. 18, 2020, pp. 475–487, ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397370. Accessed: Apr. 2, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397370.

[14] S. Ren et al., *Codebleu: A method for automatic evaluation of code synthesis*, 2020. arXiv: 2009.10297 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2009.10297.

[15] J. Escalada, T. Scully, and F. Ortin, *Improving type information inferred by decompilers with supervised machine learning*, Feb. 24, 2021. DOI: 10.48550/arXiv.2101.08116. arXiv: 2101.08116. Accessed: Apr. 2, 2025. [Online]. Available: http://arxiv.org/abs/2101.08116.

[16] R. Liang, Y. Cao, P. Hu, J. He, and K. Chen, *Semantics-recovering decompilation through neural machine translation*, 2021. DOI: 10.48550/ARXIV.2112.15491. Accessed: Apr. 2, 2025. [Online]. Available: https://arxiv.org/abs/2112.15491.

[17] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. D. S. Magalhães, and M. F. P. O'Boyle, "ExeBench: An ML-scale dataset of executable c functions," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, San Diego CA USA: ACM, Jun. 13, 2022, pp. 50–59, ISBN: 978-1-4503-9273-0. DOI: 10.1145/3520312.3534867. Accessed: Apr. 12, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3520312.3534867.

[18] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in *Proceedings of the 38th Annual Computer Security Applications Conference*, Austin TX USA: ACM, Dec. 5, 2022, pp. 508–518, ISBN: 978-1-4503-9759-9. DOI: 10.1145/3564625.3567998. Accessed: Apr. 2, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3564625.3567998.

[19] I. Hosseini and B. Dolan-Gavitt, "Beyond the c: Retargetable decompilation using neural machine translation," in *Proceedings 2022 Workshop on Binary Analysis Research*, San Diego, CA, USA: Internet Society, 2022, ISBN: 978-1-891562-76-1. DOI: 10.14722/bar.2022.23009. Accessed: Apr. 2, 2025. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/bar2022_23009_paper.pdf.

[20] D. Kocetkov et al., *The stack: 3 TB of permissively licensed source code*, 2022. DOI: 10.48550/ARXIV.2211.15533. Accessed: Apr. 3, 2025. [Online]. Available: https://arxiv.org/abs/2211.15533.

[21] H. Eom, D. Kim, S. Lim, H. Koo, and S. Hwang, "R2i: A relative readability metric for decompiled code," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 383–405, FSE Jul. 12, 2024, ISSN: 2994-970X. DOI: 10.1145/3643744. Accessed: Apr. 11, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3643744.

[22] M. H. Vaaden, "Using large language models for binary decompilation," Master thesis, NTNU, 2024. Accessed: Apr. 2, 2025. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3168005.

[23] Avast Software. "Retdec," Accessed: Jul. 2, 2025. [Online]. Available: https://github.com/avast/retdec.

[24] Hex-Rays. "Ida decompilers," Accessed: Jul. 2, 2025. [Online]. Available: https://hex-rays.com/decompiler.

[25] National Security Agency. "Ghidra software reverse engineering framework," Accessed: Jul. 2, 2025. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra.