24–44

卒業論文

大規模言語モデルを用いた
逆コンパイル手法に関する検討

2025年3月

門谷 拓能

宇都宮大学工学部
基盤工学科
情報電子オプティクスコース
情報科学分野

内容梗概

近年, ソフトウェアの解析や保守の必要性が高まっており, 特にリバースエンジニアリングの技術は, セキュリティ診断や脆弱性発見, レガシーシステムの維持管理において重要な役割を果たしている. その中でも, 逆コンパイルは, バイナリコードから高水準のプログラムコードを復元する技術として広く利用されている. しかし, 逆コンパイルには多くの課題が存在する. コンパイラ最適化によりコード構造が変形されることで, 元のソースコードとは異なる形でバイナリが生成されるため, 制御フローやデータフローの復元が困難となる. また, バイナリコードには変数名や型情報, 関数名などが含まれておらず, 復元されたコードの可読性や正確性が大きく損なわれる. 従来の逆コンパイルツールでは, 静的解析や動的解析を組み合わせた手法が一般的に用いられているが, 特に最適化されたコードに対しては十分な復元精度を達成できていない.

本研究の目的は, バイナリコードから C コードを出力する大規模言語モデル（LLM）の利用を検討し, コンパイル可能かつ元のソースコードと処理内容が正確なコードを生成することである. 逆コンパイル手法が抱える課題を克服するため, 本研究では, 逆コンパイルプロセスを二段階に分割し, それぞれに特化した LLM をファインチューニングする手法を提案する. 具体的には,「バイナリコードからアセンブリコードへの変換」と「アセンブリコードから C コードへの変換」という二段階のモデルを構築し, それぞれの段階での精度向上を図る. ファインチューニングには, 数値計算アルゴリズムやプログラム構造を含む PolyBench データセットを使用し, バイナリコードとアセンブリコードの対応関係, ならびにアセンブリコードと C コードの対応関係を学習させた. これにより, 逆コンパイルツールでは復元が困難であったコードの文脈をより適切に推定し, コンパイル可能かつ処理内容が正確な C コードを出力できるようになることを目指す.

評価実験の結果, 提案手法は従来の逆コンパイルツールと比較して, 変数名や関数名の推測精度が向上し, 全体のコードの可読性が改善された. また, 文脈理解能力を活用することで, より正確な C コードを生成できることが確認された. しかし, 一部のコードにおいて冗長な処理が含まれるなど,LLM 特有の課題も見られた.

本研究の成果は, ソフトウェアのセキュリティ解析, マルウェア解析, レガシーシステムの移行など, 多岐にわたる分野への応用が可能である. 今後の研究では, より広範なデータセットを用いた学習, 異なる命令セットアーキテクチャへの適応, 生成コードの自動検証機構の導入を行うことで, より実用的な逆コンパイルシステムの構築を目指す.

A Study on Decompilation Method using Large Language Model

Hirotaka Monya

Abstract

In recent years, the need for software analysis and maintenance has increased, and reverse engineering technology in particular plays an important role in security diagnosis, vulnerability discovery, and the maintenance and management of legacy systems. Among these, decompilation is widely used as a technique for restoring high-level program code from binary code. However, decompilation has many challenges. Compiler optimization transforms the code structure, generating a binary code in a form different from the original source code, making it difficult to restore the control flow and data flow. In addition, binary code does not contain variable names, type information, function names, etc., and the readability and accuracy of the restored code is greatly impaired. Conventional decompilation tools generally use a combination of static analysis and dynamic analysis, but they have not been able to achieve sufficient restoration accuracy, especially for optimized code.

The purpose of this research is to consider the use of a large-scale language model (LLM) that outputs C code from binary code, and generate code that is compilable and has accurate processing contents with the original source code.

In order to overcome the issues that decompilation methods have, this research proposes a method to divide the decompilation process into two stages and fine-tune LLMs specialized for each stage. Specifically, we build a two-stage model of ”conversion from binary code to assembly code” and ”conversion from assembly code to C code” to improve the accuracy at each stage. For fine-tuning, we used the PolyBench dataset, which includes numerical calculation algorithms and program structures, and had the system learn the correspondence between binary code and assembly code, as well as the correspondence between assembly code and C code. This will enable us to better estimate the context of code that is difficult to restore with decompilation tools, and to output C code that is compilable and has accurate processing contents.

目 次

第1章　　はじめに

コンピュータプログラムは現代社会において, あらゆる分野で極めて重要な役割を担っている. その影響は, 私たちの身近なところに見られるインターネットやスマートフォン, 交通機関, 医療機器, 金融システムだけでなく, エンターテインメントや科学技術分野にまで広がっている. スマートフォンのアプリによって日常生活が便利になることや, 医療機器が正確な診断をすることで人命が救われることも, すべてプログラムによって構築されたソフトウェアがその機能を支えているからこそ可能となっている. このように, ソフトウェアは多くの分野で不可欠な存在となっており, その品質や性能が, 個人の生活の利便性はもちろん, 経済や社会全体の効率性にも多大な影響を与えている. そのため, ソフトウェアがどのように私たちの生活の質を高め, また将来どのように利用されるかを理解することは, 今後の社会発展において重要である.

しかしながら, ソフトウェア開発が複雑化し, 多様化が進む一方で, ソフトウェアの保守や解析に伴う課題も顕在化している. ソフトウェアの保守とは, すでに運用されているソフトウェアの問題を修正したり, 機能を更新したりする作業であり, 開発よりも多くの時間が費やされることが一般的である. 中でも, 長年にわたって運用されてきたレガシーシステム, つまり古い技術で作られたシステムや, ソースコードが失われたために十分な情報が得られない既存ソフトウェアの解析が求められる状況は増加している [1]. これらのソフトウェアは, その時代に合わせて設計された技術や要求と共に, 運用され続けているために, 保守や機能の追加が非常に困難である場合が多い. このような環境下で, プログラムの動作を理解し, 必要な改修や最適化を施すためには, リバースエンジニアリングが重要な手法となる. リバースエンジニアリングは, ソフトウェアのバイナリコードからその設計や動作を逆推論し, 理解するプロセスである.

特に, 逆コンパイルはこのプロセスにおいて重要な技術の一つであり, バイナリコードやアセンブリコードを解読し, 高水準言語で記述されたコードに復元することを目指す. 逆コンパイルの技術は, セキュリティ診断やソフトウェア保守, ライセンス検証, およびマルウェア解析など, 多くの分野で応用されている. セキュリティの観点から見ても, マルウェアの動作を理解し, 対策を講じるための初期ステップとして逆コンパイルは非常に重要である. 逆コンパイルを支えるものとして, さまざまな逆コンパイルツールが開発されており, それぞれの特性を活かした使用が求められる. 代表的なツールには,Ghidra,IDA Pro,Radare2 などがある.Ghidra は NSA によって開発されたオープンソースツール

で, 多くのプロセッサアーキテクチャに対応しており, 拡張性が特徴である [2].IDA Pro は商業向けに広く利用され, インタラクティブな解析機能とプラグインによる拡張性が特徴である [3].Radare2 は軽量でスクリプト対応の柔軟さを備え, コンソールベースでの操作が強力である [4]. しかしながら, この技術には数多くの課題が伴う. バイナリコードはコンパイラによって生成された低水準の機械語であるため, 作成時に記述されていた変数名や型情報, コメントといった情報がすべて失われる. そのため, 逆コンパイラはこれらの失われた情報を推測し, 元のコードの構造や開発者の意図を復元する必要があるのですが, これが非常に困難である [5]. 特に, 大規模で複雑なプログラムでは, 構造の復元やコードの可読性に大きな制約が生じることが多々ある. 特に安全保障分野や金融分野では, 逆コンパイル技術がセキュリティの観点からも積極的に活用されている. セキュリティの枠組みでは, マルウェア解析が危険なコードを特定し, さらにその背後に潜む脅威を理解する上で必須のプロセスである. マルウェアはその性質上, 意図的に難読化されているため, 一筋縄ではいかず, 解析には高い技術力が要求される. 逆コンパイル技術によって, この複雑な難読化コードも解明され, システムの安全性を確保するために活かされる. 逆に, 悪意ある利用を防ぐためのプロテクションとして, ソフトウェアの難読化技術が進化する背景と, 逆コンパイラとのいたちごっこのような攻防が日々続いている [6].

加えて, 逆コンパイル技術はプログラムの再利用性を高めるためのツールとしても重要である. 企業内で開発されたプログラムやアプリケーションが他のプロジェクトでも役立つことが多々あるが, オリジナルのソースコードが失われた場合や, その開発に携わった人材が退職してしまった場合, 新たにその機能を解析し再利用の可能性を探る必要がある. ここでも逆コンパイルは, オリジナルの構造を復元することで, プログラムの新しい使い道を発見する手助けをする. これは特に, エンタープライズソフトウェアやカスタムシステムを扱う企業にとって非常に価値のあるプロセスである.

さらに, コンパイラの最適化によってコードの構造が大幅に変形されることも, 逆コンパイルを困難にする大きな要因の一つである. 最適化とは, プログラムの実行効率を向上させるために行われる処理であるが, その過程でループの展開や関数のインライン化, 不要なコードの削除などが施される. この結果, 逆コンパイラが生成するコードは元のソースコードと大きく異なる構造を持つことが多く, 解析や再利用が難しいものとなる [7]. さらに, バイナリコードにはデータと命令が混在しており, これらを正確に区別し理解することもまた大きな課題となる.

このような課題を攻略するために, 主に用いられてきた逆コンパイル手法では, 主に静的解析と動的解析の技術が使われてきた. 静的解析は, プログラムを実行することなくコード自体を分析し, プログラムの構造や制御フローを理解しようとする方法である. 一方, 動的解析は実際にプログラムを実行し, その挙動やデータの流れを観察することでプログラムの動作を理解する手法である. しかし, これらの方法は万能ではなく, 特に最適化されたコードや難読化されたコードに対しては, 十分な結果を得ることが難しい場合が多い.

こうした中で, 近年の人工知能や機械学習の進歩は, 逆コンパイルの分野にも新しいアプローチを

提供する可能性を開いている [8]. 特に, 大規模言語モデル（LLM, Large Language Model）の発展は, 逆コンパイルにおける多くの課題解決に向けて大きな可能性を示している.[8]LLM は膨大なテキストデータを用いた事前学習によって, 自然言語の文脈や意味を理解する能力を備えている [9]. この技術を逆コンパイルに応用することによって, 従来の手法では扱いにくかったバイナリコード中の曖昧な部分を文脈から推測し, より正確で可読性の高いコードを生成することが期待されている. 本研究における大規模言語（LLM）の活用に際して, 特に注目すべきはファインチューニングの有益性である. 特定のタスクやドメインに対してファインチューニングを施すことによって, そのパフォーマンスをさらに高めることがある. ファインチューニングを行うことで,LLM は特定の逆コンパイルタスクに対する適応力を大幅に強化することができる. 一般的に自然言語理解能力を持つ LLM に対し, バイナリコードやプログラム特有の構造を学習させることで, より精度の高い逆コンパイル結果を期待できる. ファインチューニングは, モデルがバイナリコード中の曖昧な部分をより正確に推測し, 逆コンパイルツールでは得られない高い精度での復元を可能にする. これによって, より正確かつ可読性の高いコードを生成する能力が向上するだろう. 逆コンパイルの主な障壁である, 最適化されたコードや難読化されたコードの解析においても, ファインチューニングによって LLM はこれらのコードパターンをより効果的に対処することが可能になるだろう.

本研究では, この大規模言語モデルを活用した逆コンパイル手法を提案する. 具体的には, バイナリコードからアセンブリコードへ変換する学習モデルと, アセンブリコードから C コードへ変換する学習モデルを構築し, 二段階に分けて逆コンパイルを行うアプローチを取る. この分割アプローチにより, 各ステップで解析の精度を高め, 全体としての復元精度と効率の向上を目指す. さらに, ファインチューニングを通じて LLM を逆コンパイルタスクに特化させることで, 様々なプログラムパターンを学習することを試みる. この試みの一環として,PolyBench データセットを使用し, バイナリコードと対応するアセンブリコード, およびアセンブリコードと C コードのペアデータを用いてモデルを訓練する.PolyBench は, 数値計算アルゴリズムやプログラム構造に関する多様なコードを含むデータセットであり, モデルが幅広いコードパターンを学ぶのに適している.

本研究の意義は,LLM の文脈理解能力を利用することで, 逆コンパイルツールでは達成が難しかった逆コンパイルの精度向上を実現する点にある. 特に, コンパイラの最適化や難読化の影響を受けたコードに対しても, 本手法は高い復元性能を発揮することが期待される. これにより, ソフトウェアの開発者や解析者は, より迅速かつ正確にコードを理解し, 必要とされる修正や最適化を行うことが可能となる. さらには, 本研究の成果は, セキュリティ診断や脆弱性発見, レガシーコードの保守, そしてプログラムの再利用といった広範な分野において有用であると期待される. この研究が逆コンパイルにおける新たな視点を提供し, この分野の技術進展を促進することを目指している. 次章以降では, これらの目標を達成するための詳細な手法とアプローチについて検討し, 新たな知見を提供する.

本論文は, 以下の構成で進める. 第 2 章では, 大規模言語モデルと逆コンパイルについて述べる. 第

3 章では, ファインチューニングによる手法の提案と, その実施過程を詳述する. 第 4 章では, 改善手法の評価結果について議論し, 生成されたコードの性能とその議題を明らかにする. 最後に, 第 5 章では本論文をまとめ, 今後の研究の方向性について述べる.

第2章　大規模言語モデルと逆コンパイル

本章では本研究が前提とする大規模言語モデルと逆コンパイルについて述べる. まず, 逆コンパイルの概要とその技術的背景を説明し, 次に大規模言語モデルの特性とその逆コンパイルへの適用について検討する.

2.1　逆コンパイルの概要

逆コンパイルは, コンピュータプログラムの機械語やバイナリコードを高水準プログラミング言語のコードに変換するプロセスを指す [10]. リバースエンジニアリングの重要な技術として, ソフトウェアの解析, バグ修正, セキュリティ診断, 互換性検証など, 多岐にわたる用途で活用されている [11]. 通常, プログラムは高水準言語で記述され, コンパイラにより機械語に変換されるが, 逆コンパイルはこのプロセスを逆にたどり, 機械語を元に人間が理解可能なコードへ変換する試みである.

逆コンパイルが特に重要視される理由の一つは, ソフトウェアの脆弱性を発見し, セキュリティリスクを評価するための手段としての役割である. 商用ソフトウェアや第三者が提供するバイナリコードは, 通常ソースコードが公開されていないため, 逆コンパイルを通じてその内部構造を解析する必要がある. これにより, 潜在的な脆弱性を特定し, 攻撃のリスクを軽減するための対策が可能になる.

また, レガシーシステムのメンテナンスや更新作業においても, 逆コンパイルは重要な役割を果たす. 古いソフトウェアのソースコードが失われている場合でも, 逆コンパイルを通じてコードを再生成し, 新しいプラットフォームへの移行や機能の追加が可能となる. このように, 逆コンパイルは, システムの長期的な維持管理や進化においても不可欠な技術である.

逆コンパイルのプロセスは, バイナリコードを解析し, 中間表現を生成し, 最終的に高水準コードへ変換するという一連のステップから成る. バイナリコードの解析では, プログラムの制御フローやデータフローをマッピングし, その構造を理解することが求められる. 次に, これを中間表現に変換することで, バイナリコードと高水準コードの間のギャップを埋める役割を果たす. この中間表現は, 最適化や再構築が行いやすく, 最終的に可読性の高いコードを出力するための基盤となる.

逆コンパイルには多くの課題が伴う. 特に, 機械語には型情報やコメントが含まれないため, 生成されるコードの可読性や正確性を確保することが困難である. また, コンパイラの最適化により, オリジナルの構造が大きく変化していることが多く, 元のコードを完全に再現することは理論的に不可能な

場合も多い [12]. 逆コンパイルは, 理想的には健全な (バイナリと機能的に同等) かつ再コンパイル可能な C コードを生成する [13]. そのため, 逆コンパイルの目的は, バイナリ実行可能ファイルのソースコード形式を復元することである [14].

さらに, コンパイラの最適化による影響も逆コンパイルの精度に大きく影響を与える. 最適化されたコードは, 実行効率を向上させるために, 元の高水準コードの構造を大きく変化させる. これにより, 逆コンパイラが正確な復元を行うことが難しくなる. 複雑な制御フローを持つプログラムでは, 条件分岐やループ構造の解析が難しいという問題もある. 逆コンパイラは, これらの制御構造を正確に再構築し, 元のプログラムの動作を再現しなければならないが, 誤った解析結果が生じる可能性がある. 最後に, 逆コンパイルはさまざまなプラットフォームやアーキテクチャに対応する必要があるが, これも技術的なハードルを引き上げている. 異なるアーキテクチャ間での逆コンパイルは, 異なる命令セットやデータフォーマットに対応するため, 専門的な知識と技術が求められる.

このように, 逆コンパイルはプログラムの意図や機能を解析するための重要な手段であるが, その実現には高度な解析技術と専用のツールが必要となる. 次節では, 代表的な逆コンパイルツールについて述べ, それらがどのように逆コンパイルを実現しているのかを検討する.

2.1.1 逆コンパイルツール

逆コンパイルを行うために開発されたツールには, さまざまな種類が存在し, それぞれが異なる特徴と利点を持つ. 以下に代表的なツールについて詳述する.

- IDA Pro

  IDA Pro は, 商用の逆コンパイラであり, リバースエンジニアリングの分野で広く利用されている. インタラクティブなデコンパイル機能を備えており, バイナリコードを高水準コードに変換するだけでなく, 動的解析ツールとしても機能する. また, 幅広いアーキテクチャをサポートしており, 強力なプラグインシステムを備えているため, ユーザーが機能を拡張しやすい. この柔軟性が, セキュリティ研究者やリバースエンジニアにとっての人気の理由となっている.

- Ghidra

  Ghidra は, 米国国家安全保障局（NSA）が開発し, オープンソースとして公開された逆コンパイラである.IDA Pro と同等の機能を持ちながら, 無料で利用できる点が大きな魅力である.Ghidra は, 複数のアーキテクチャをサポートし, スクリプトによるカスタマイズ性が高い. また, チームでの共同作業をサポートするための機能も充実しており, 企業や政府機関における逆コンパイル作業に広く利用されている.

- Radare2

  Radare2 は, オープンソースのリバースエンジニアリングフレームワークであり, バイナリ解析, デバッグ, 逆コンパイルを行うための豊富なツールセットを提供する.CLI（コマンドラインインターフェース）を主体としているが,GUI（グラフィカルユーザーインターフェース）もサポートしており, 柔軟な操作性を持つ. また, プラットフォーム間の移植性が高く, 幅広いアーキテクチャをサポートしているため, エキスパートユーザーにとって強力な選択肢となる.

これらのツールは, 逆コンパイル技術の進化を支えているが, 特定のプラットフォームや最適化レベルに依存することが多い. したがって, 逆コンパイラが直面するさまざまな課題についても理解する必要がある. 次節では, これらのツールが抱える制限について詳述する.

2.1.2 逆コンパイルツールの制限

逆コンパイルツールには多くの利点がある一方で, いくつかの重要な制限が存在する. これらの制限は, ツールの有効性を制約し, 生成されるコードの品質に影響を与える. 以下に主な制限について詳述する.

- 最適化コードへの対応

  コンパイラによる最適化は, プログラムの実行効率を向上させる一方で, コードの構造を大きく変化させる. 例えば, 関数のインライン化, ループの展開, 不要なコードの削除などが行われると, 元の高水準コードの構造が失われる. 逆コンパイラは, このような最適化によって変形されたバイナリコードを解析し, 元の意図を再現することが難しくなる. このため, 最適化コードに対応する際の精度や可読性が低下する.

- メタ情報の欠如

  機械語やバイナリコードには, 変数名, 型情報, コメントなどのメタ情報が含まれていない. これらの情報は, 高水準言語においてプログラムの意図や構造を理解するために重要であるが, 逆コンパイラはそれらを推測しなければならない. このプロセスには限界があり, 特に複雑なプログラムや、コードの意図を正確に推測するための文脈が不足している場合には, 生成されるコードの正確性や可読性が低下する.

- 制御フローとデータフローの復元の難しさ

  逆コンパイラは, バイナリコード内の制御フローやデータフローを解析し, 元のプログラム構造を再構築する必要がある. しかし, 制御フローの複雑な分岐やループ構造, データの依存関係を

正確に解析することは非常に困難である. 特に, コンパイラが最適化の一環としてこれらの構造を変更している場合, 逆コンパイラが誤った復元を行う可能性が高まる.

- 可読性の向上の困難さ

  逆コンパイルされたコードは, 通常, オリジナルのソースコードに比べて可読性が低い. 変数名が一般的な名前（例えば、var1、temp2）に置き換えられ, コードの意図を理解するのが難しくなる. また, 最適化によって複雑化した制御構造がそのまま再現されるため, コードのメンテナンスや解析が困難になる.

- プラットフォーム依存性

  逆コンパイラは, 特定のプラットフォームやアーキテクチャに依存することが多い. 例えば, ある逆コンパイラが x86 アーキテクチャに特化している場合,ARM アーキテクチャのバイナリコードを適切に解析することは難しい. このため, 異なるプラットフォームに対応するためには, 追加のツールやモジュールが必要になることがある.

- 関数やライブラリの解釈の限界

  バイナリコードには, 外部ライブラリや関数の呼び出しが頻繁に含まれている. 逆コンパイラは, これらの関数の目的や挙動を正確に再現するために, 標準ライブラリやカスタムライブラリの知識を持つ必要がある. しかし, 未知のライブラリ関数や独自のコードに遭遇した場合, その解釈が困難であり, 誤ったコード生成の原因となる.

- ユーザー介入の必要性

  多くの逆コンパイラツールは, 生成されたコードの精度を向上させるためにユーザーの手動介入を必要とする. ユーザーは, 生成されたコードを精査し, 誤りや不適切な部分を修正する必要がある. しかし, このプロセスは時間と労力を要し, 逆コンパイラの自動化の利点を損なうことになる.

- 計算リソースの消費

  高度な解析を行うためには, 逆コンパイラは多くの計算リソースを消費する. 特に, 大規模なプログラムや複雑なバイナリコードを解析する場合, 実行時間が長くなり, 実用性が低下する. このため, 逆コンパイラの効率化やリソース使用の最適化が求められる.

これらの制限を克服するためには, より高度な解析技術や新しいアプローチの導入が必要である. 本研究では, 大規模言語モデルを活用することで, これらの制限を克服し, 逆コンパイルの精度と効率を向上させることを目指す.

2.2 大規模言語モデル

大規模言語モデル（LLM, Large Language Model）は, 膨大な数のテキストデータを用いて訓練され, 自然言語処理タスクにおいて高い性能を発揮する深層学習モデルである. この技術は, 言語の文法構造や意味の理解に優れ, 複雑な文脈を処理しながら高品質なテキストを生成する能力を有している.LLM は単なる言語モデルの枠を超え, 多様なタスクに適用可能な汎用性を備えた人工知能技術として広く注目されている.

その基盤となるのが Transformer アーキテクチャであり, 自己注意機構（Self-Attention）によって入力データ内の重要な依存関係を効率的に捉えることが可能である.Transformer は, エンコーダー・デコーダー構造を持つ場合と, エンコーダーやデコーダーの片側のみを利用する場合がある. 特に,GPT（Generative Pre-trained Transformer）シリーズはデコーダーのみを用いた構造であり, 生成タスクに特化している.

LLM の訓練には, 膨大な計算資源とデータセットが必要である. 例えば, インターネット上の大規模なテキストデータ, 書籍, コードリポジトリなど, 多様な情報源を収集し, 事前学習を行う. これにより, モデルは言語の文法構造, 文脈, 意味を理解し, 高度な推論や生成を可能にする. また, 事前学習後に特定タスクに特化したファインチューニングを施すことで, 精度をさらに向上させることができる. この訓練過程では, データの質と量がモデルの性能に大きく影響するため, 大規模データセットの活用が不可欠である.

最近の LLM では, ゼロショット学習や数ショット学習といった新たな学習パラダイムが導入されている. これにより, モデルは特定のタスクに対して直接訓練されていなくても, そのタスクを遂行する能力を持つようになった. ゼロショット学習では, タスクに関する事前情報が全く与えられない場合でも, 高い精度で問題に対応することが可能である. また, 数ショット学習では, タスク固有の少量のデータを提示するだけで, 迅速に学習し, 適応する能力を発揮する. これらの特性により,LLM は従来の自然言語処理技術を大きく凌駕する柔軟性を持っている.

特に,OpenAI による GPT シリーズは大規模言語モデルの代表例として挙げられる.GPT-3 は 1,750 億のパラメータを持ち, その卓越した文脈理解力とテキスト生成能力で広く注目を集めた. その後継モデルである GPT-4o は, さらに多くのパラメータを持ち, より高度な推論や文脈理解を可能にしている. このような進化により,LLM は自然言語処理のみならず, プログラムコード生成, 翻訳, 要約など, 幅広い応用分野で利用されるようになった.

一方で,LLM はいくつかの課題も存在する. 大規模なモデルを訓練するためには, 膨大な計算資源と時間が必要であり, コストが高いという問題がある. また, モデルのブラックボックス性も指摘されており, 生成された出力がどのような理由で導き出されたのかを説明することが難しい. この特性は, 信頼性が求められる分野での LLM の利用を妨げる要因たなっている. それにもかかわらず,LLM の

応用範囲は拡大し続けている. 例えば, プログラムコード生成の分野では, モデルが文脈を理解して適切なコードを提案し, 開発者の生産性を向上させている. また, 医療分野では, 患者データの解析や診断支援に LLM が活用されており, 医療の効率化と精度向上に寄与している. さらに, 学術分野や創造的な分野でも,LLM は文章の要約や創作活動を支援するために利用されている.

このように, 大規模言語モデルは, 技術的な進化を遂げながら多くの分野でその有用性を証明している. 次に, 本研究で用いるモデルについて述べる.

2.2.1 GPT-4o

GPT-4o は,OpenAI によって開発された第 4 世代の大規模言語モデルであり, そのアーキテクチャは Transformer に基づいている.GPT-4 は, 数百億から数兆のパラメータを持ち, 大量のテキストデータを用いた事前学習によって, さまざまな自然言語処理タスクにおいて高度な性能を発揮する. 一般的なコード理解に優れていますが, 詳細な技術およびセキュリティ分析の有効性はさまざまである [15]. 特に, 従来のモデルと比べて文脈の理解力が向上し, 複雑な命題や推論, 生成に対応可能である.GPT-4o の最大の特徴は, その膨大なパラメータ数にある. このモデルは数百億から数兆に及ぶパラメータを持ち, これにより膨大な情報を効果的に学習することが可能となっている. パラメータの増加は単なるスケールアップにとどまらず, 長文の文脈を維持しつつ, 複雑な依存関係を正確に解析する能力を強化している. その結果,GPT-4o は高度な推論や複雑な質問応答, 長文生成などのタスクにおいて, 従来モデルを大きく上回る性能を発揮している. さらに,GPT-4o はマルチモーダル学習にも対応している. 従来のモデルがテキストデータの処理に特化していたのに対し,GPT-4o は画像や音声データも扱うことが可能である. 例えば, 画像を入力として受け取り, その内容を説明するキャプションを生成することや, 音声データを文字起こしすることができる. このように, テキスト以外のデータを処理可能な能力を持つことで,GPT-4o はさらに多様な応用分野での利用が期待されている. また,GPT-4o はゼロショット学習や数ショット学習においても顕著な性能を示している. ゼロショット学習では, 特定のタスクに対する事前の訓練がなくても, そのタスクを遂行する能力を発揮する. この特徴は, モデルが膨大な事前学習データから抽象的なパターンを学習し, 新しいタスクに適応できる柔軟性を持つことを示している. 一方, 数ショット学習では, 少量のタスク関連データを与えることで, モデルが迅速にそのタスクに適応し, 高い精度で応答を生成する. この柔軟性は, 特定のタスク向けのデータセットが限られている場合にも効果的である. しかしながら,GPT-4o にはいくつかの課題も存在する. モデルの規模が大きくなるにつれて, 訓練や推論に必要な計算資源とエネルギー消費が急増しており, 環境負荷やコストの増加が問題視されている. また, 生成されるテキストの信頼性も課題の一つであり, モデルが不正確な情報や偏った出力を生成する可能性がある. このような課題を解決するため, 効率的な訓練手法の開発やモデルの解釈性向上に向けた取り組みが進められている.

GPT-4o による逆コンパイルは単純な命令であれば, 出力が可能であった. しかし, 複雑な処理や行数が多いプログラムには対応できなかった. ソースコードとは異なり, セマンティック情報がないため, バイナリコードの理解はリバースエンジニアにとって困難である [16].

GPT-4o の特性として, 以下が挙げられる.

- 高い汎用性：複数のタスクにおいて少ない例示 (Few-shot Learning）やゼロ例示（Zero-shot Learning）で優れた結果を得られる.
- マルチモーダル性：テキストだけでなく, 画像やコードなどの異なる形式のデータに対応可能である.
- スケーラビリティ：大規模な計算資源を用いた並列処理に最適化されており, 高い処理速度を実現する.

GPT-4o が逆コンパイルにおいて有望である理由は, その文脈理解力と生成能力にある. バイナリコードの機械的な命令列から抽象的な意図を推測し, それを人間が理解可能な形で表現することが可能である. また, 大規模データセットを用いて事前学習されたため, プログラミング言語やアルゴリズムに関する知識が豊富であり, 逆コンパイルに必要な論理的推論やパターン認識に長けている. 本研究では,GPT-4o を活用することで, 従来手法の課題である可読性の低さや並列化の困難さを克服し, より効率的な逆コンパイルを実現することを目指す.

2.3 逆コンパイルにおける課題

逆コンパイルは, 低水準の機械語コードから高水準のソースコードを復元する技術であり, プログラムの解析や再利用, セキュリティ診断において重要な役割を果たす. しかしながら, 逆コンパイルのプロセスは本質的に情報の欠如に起因する複雑性を伴い, 多くの課題が存在する. 本節では, 特に大規模言語モデル（LLM）を用いた逆コンパイル技術そのものが本質的に抱える課題について論じる.

大規模言語モデルは, 文脈理解と自然言語生成において顕著な性能を示しているものの, 逆コンパイルの領域に適用する場合, いくつかの問題が浮上する.

- 情報の欠如

  バイナリコードは高水準コードに含まれる変数名, 型情報, コメントといったメタ情報を欠いている. このため,LLM はバイナリコードの断片からその意図を推測し, コードを生成しなければならない. しかし, この推測には限界があり, 特に最適化が施されたバイナリでは, 元のコード構造が大きく変化しているため正確な復元が困難となる.

- 生成コードの妥当性

  LLM が生成するコードは文法的には正確である場合が多いが, 必ずしも入力バイナリと同一の機能を実現しているとは限らない. 特に, 関数の境界の識別や条件分岐を伴う制御フローの再構築において, 不正確な復元が発生する可能性がある. これにより, 逆コンパイル結果の信頼性が低下する.

- 計算リソースの消費

  LLM を用いた逆コンパイルは, モデルのパラメータ数や計算負荷の増加に比例して大量の計算リソースを必要とする. 特に大規模なバイナリコードや複雑なプログラムに対する処理では, 実行時間やコストが実用化への障壁となる.

特に困難なケースとして 2 つ挙げられる.1 つ目は, 最適化されたバイナリコードである. コンパイラによる最適化は, プログラムの実行効率を向上させる一方で, 元のコード構造を破壊する. 例えば, 関数のインライン化, 命令の再配置, 不要なコードの削除などは元のコードの意図を推測することを著しく困難にする.2 つ目は, 曖昧な命令の解釈である. 同一の機械命令が文脈に応じて異なる意味を持つ場合がある. このような曖昧性の解釈は, モデルの文脈理解能力に依存するが, 複雑な制御フローやジャンプ命令の解析は依然として大きな課題である.

以上の課題に対処するためには,LLM の文脈理解力や推論能力をさらに高めるとともに, 生成されたコードの妥当性を検証・補正する仕組みが求められる. 本研究では, 大規模言語モデルの特性を活かし, 逆コンパイルにおける課題を克服する新たな手法を提案することを目指す.

2.4 課題点に対する改善手法

逆コンパイルにおける課題を克服するために, 大規模言語モデル（LLM）の特性を活かしたファインチューニング手法を導入する. 本研究では,LLM をファインチューニングすることにより, 逆コンパイルプロセスの精度を向上させる手法を提案する. このアプローチは, バイナリコードから C コードへの変換を 2 段階に分ける構成を採用している. 具体的には, まずバイナリコードをアセンブリコードに変換するモデルを訓練し, その後, アセンブリコードを C コードに変換するモデルを訓練する. この分割されたプロセスにより, 各ステップでの精度向上を図り, 全体的な復元性能を高めることを目指す.

本研究では, 逆コンパイルタスクに特化した LLM を構築するために, 既存の大規模モデルを特定のデータセットを用いてファインチューニングする. この手法は, モデルが特定タスクに必要なパターンや構造を学習しやすくするため, 汎用的なモデルに比べて高い精度を達成できることが期待される. ファインチューニングに際しては,PolyBench データセットを使用する. このデータセットは, 数値計算やアルゴリズムに関する多様なプログラムを含んでおり, 逆コンパイルタスクの学習に適している. PolyBench を用いたファインチューニングでは, まず, バイナリコードと対応するアセンブリコードのペアを使用して, バイナリコードからアセンブリコードを生成するモデルを訓練する. このステップでは, バイナリコードの命令セットやメモリ操作を正確に解析し, それを人間が解釈可能なアセンブリコード形式で表現する能力をモデルに付与することが目的である. 次に, アセンブリコードと対応する C コードのペアを使用して, アセンブリコードから高水準プログラミング言語である C コードを生成するモデルを訓練する. このステップでは, アセンブリコードの制御フローやデータフローを解析し, それを C コードの文法や構造に適合させる能力をモデルに習得させる. 特に, 複雑なループ構造や条件分岐を正確に再現することが重視される.

本研究で採用するバイナリコードからアセンブリコード, アセンブリコードから C コードへの 2 段階モデルには, 次の利点がある. 第一に, 各段階のタスクが比較的限定的であるため, モデルが特定の解析能力を深く学習できる点である. バイナリコードから直接 C コードを生成する従来の手法では, 抽象化レベルのギャップが大きく, モデルに過剰な負担をかける可能性がある. 一方, 本研究の方法では, バイナリコードとアセンブリコード, アセンブリコードと C コードという明確な関係性を学習するため, 全体的な復元精度の向上が期待される. 第二に, この分割アプローチはエラーの追跡と修正を容易にする. 例えば, バイナリコードから生成されたアセンブリコードにエラーが含まれる場合, そのエラーを特定し修正することで, 次の段階での生成精度を確保できる. これにより, モデルのデバッグや改善が効率的に行える. ファインチューニングされたモデルは. 従来の逆コンパイルツールと比較して次の点で優位性を持つと予想される.LLM は,PolyBench データセットを通じてコードの文脈的情報を学習するため, 欠損したメタ情報をより適切に補完する能力を獲得する. また, ファインチューニングにより, 複雑な制御フローやデータフローの構造を学習し, それを C コードで正確に再現する能力が向上する. そして, 文法的に正しいだけでなく, 可読性や意図が明確なコードを生成する能力をモデルに付与することが可能となる. さらに,PolyBench データセットの多様性により, モデルは広範なアーキテクチャやコードスタイルに対応する能力を習得できる.

これらのモデルを用いることで, 逆コンパイルの課題である情報欠如や生成コードの妥当性問題に対処し, 効率的で信頼性の高い変換を実現する. これにより, さまざまなバイナリコードの解析や再利用が可能となり, 実用的なソリューションを提供できることが期待される.

第3章　　ファインチューニングによる改善

逆コンパイルに関わる課題を解決するために, ファインチューニング技術を活用した大規模言語モデルの応用に焦点を当てる. 本章では, ファインチューニングの概要, データセットの準備方法, および具体的な変換プロセスを詳しく述べる.

3.1　ファインチューニングの概要

ファインチューニングとは, 既存の大規模言語モデルに対して, 特定のタスクに最適化された追加の学習を施す手法である. これにより, モデルは特定のドメインや用途に向けたパフォーマンスを引き出すことができる. 特に, 本研究のような逆コンパイルタスクでは, 一般的な自然言語処理 (NLP) の知識だけでは十分ではなく, プログラムコードやバイナリ解析に特化した知識をモデルに付加する必要がある. そのため, ファインチューニングは不可欠な手法であるといえる.

ファインチューニングは, 次の主要なステップで構成される. まず, 適切なデータセットを選定することから始まる. 本研究では, プログラムコードに関連する高品質なデータセットとして PolyBench を採用した. このデータセットは, 数値計算やアルゴリズムを中心とした多様なコード例を含み, 逆コンパイルタスクにおける訓練に最適である. 次に, モデルに与えるタスクを明確に定義する必要がある. 本研究では, バイナリコードからアセンブリコードへの変換, アセンブリコードから C コードへの変換という 2 段階のタスクを設定している. これにより, 各ステップでモデルが特定の目標に集中しやすくなり, 全体的な精度が向上すると考えられる.

ファインチューニングの意義は, リソース集約的に訓練された汎用モデルを, ニッチで専門化されたタスクに適応させられる点にある. 特に GPT-4 のようなモデルは, 広範な基礎知識を持つため, 逆コンパイルタスクに応用することで, 最適化されたバイナリの複雑な構造を理解し, 高級言語へ再構築する能力を強化できる.

ファインチューニングの主な利点は, 汎用モデルに比べて特定タスクへの適応力が向上する点にある. 事前学習済みの LLM は, 広範なテキストデータを基に文脈や文法, 意味を学習しているが, プログラムコードの構造やバイナリ解析に特化しているわけではない. そのため, 逆コンパイルのような専門的なタスクでは, 一般的な LLM だけでは十分な性能が発揮されないことがある. ファインチューニングを施すことで, モデルは特定タスクに関連する特徴を深く学習し, バイナリコードからの情報

抽出や高水準コードへの変換といった課題において, より高い精度を実現することが可能となる. さらに, ファインチューニングは, データセットの規模や質に応じて柔軟にモデルを最適化できる点でも優れている.PolyBench のようなドメイン特化型のデータセットを活用することで, モデルは一般的なプログラムコードだけでなく, 特定のアルゴリズムやデータ構造に関する知識を効率的に学習できる. これにより, 逆コンパイルタスクに必要な高度な推論や抽象的な構造の復元が可能となる.

特に GPT-4 を逆コンパイルタスクに適用することの意義は大きい.GPT-4 は自然言語の文脈理解のみならず, プログラムコードの構造的なパターン認識にも能力を発揮することができる. ファインチューニングによって, バイナリコードから高級言語への変換精度を向上させることが可能である. 大規模言語モデルとプログラム分析を組み合わせて, ローカル変数とユーザー定義データ構造の名前と型の両方を復元する新しい手法も提案されている [8]. また, すでに事前学習済みのモデルを使用することで, ゼロからモデルを構築するのに比べて学習に要するリソースや時間を大幅に削減できる. さらに, ファインチューニングは柔軟性に富んでおり, 異なる最適化手法やプラットフォームに対しても適切にモデルを調整し, さまざまな逆コンパイルのシナリオに対応する能力をもたらす. 逆コンパイルタスクにおいて, ファインチューニングは従来の静的解析や従来型ツールでは解決が困難であった課題に対処するための鍵となる技術である. 特に, バイナリコードの曖昧な部分を解釈し, 欠落しているメタ情報を補完する能力をモデルに付与することができる. これにより, 従来手法では可読性や正確性が低かった生成コードの質が大幅に向上する. 加えて, ファインチューニングされたモデルは, バイナリコードとアセンブリコードの間, またはアセンブリコードと C コードの間の対応関係をより深く理解するため, エラーの発生頻度を低減し, 復元精度を高めることができる. この精度の向上は, セキュリティ診断や脆弱性検出といった応用分野においても重要な意味を持つ.

このように, ファインチューニングは大規模言語モデルを目的のタスクに効果的に適用するための重要なアプローチであり, 特に逆コンパイルの分野においてその価値が示されることが期待される. モデルの適応力と柔軟性を向上させ, 逆コンパイルプロセス全体の精度と効率を高める上で, ファインチューニングは不可欠な手段としての位置を占める.

3.2 データセットの概要

モデルのファインチューニングには, 適切なデータセットが不可欠である. 今回の研究では,PolyBench データセットを用いて実験を行った. 以下の表 3.1 に PolyBench の Benchmarks について示した.PolyBench は, 数値計算に特化したベンチマーク集であり, 多様なアルゴリズムとその実装を含むため, 逆コンパイルに適した多様性を備えている [17]. データ選定においては,PolyBench の持つ特性が決定的な要因となった. このデータセットは, 行列計算や配列操作, 動的プログラミングといった多岐にわたるプログラム例を含み, 制御構造やアルゴリズムパターンの多様性に優れている. また, バイ

ナリコード, アセンブリコード,C コードの対応関係が明確に定義されているため, 逆コンパイルタスクにおける入力と出力のペアデータを簡便に生成することが可能である. さらに, 多くの研究で使用されていることから, データの正確性と再現性が保証されており, 本研究においても信頼できる基盤として利用できる. データセットの準備に際しては, まずデータを「バイナリコードからアセンブリコードへの変換」と「アセンブリコードから C コードへの変換」という 2 つのサブタスクに分類した. この 2 段階のタスク設定は, モデルが各段階の特徴を明確に学習できるようにするためのものである. 次に, アセンブリコードや C コードに含まれるコメントやフォーマットのばらつきを統一する正規化処理を施した. この過程では, 不要な空白やコメントを除去することで, モデルが本質的なプログラム構造に集中して学習できるよう工夫した.

データセットの作成においては, 以下のプロセスを採用した. まず,PolyBench のソースコードを x86-64 アーキテクチャ上でコンパイルし,16 進数バイナリコードを生成した. この段階では,GCC などの一般的なコンパイラを使用し, 標準的なコンパイルオプションを適用した. 続いて, 生成された実行ファイルに対して,‘objdump‘を用いてアセンブリコードを取得した.‘objdump‘は, バイナリコードをディスアセンブルし, アセンブリコードを生成するためのツールであり, その出力により, バイナリとアセンブリの対応関係を詳細に確認できる.

このようにして,PolyBench の各ベンチマークに対する 16 進数バイナリコードとアセンブリコード、アセンブリコードと C コードのペアを作成した. 逆コンパイルのタスクにおいては, このペアデータがモデルのファインチューニングに使用される. このデータセットに含まれるコードは, 実際のアプリケーションにおける多様な計算と処理のパターンを反映しており, モデルが幅広いパターンを学習し, 一般化能力を高めるための基盤となる.

表 3.1: PolyBench のプログラム

プログラム名	内容
2mm	2 つの行列計算 式は alpha*A*B*C+beta*D
3mm	３つの行列計算 具体的には (A*B)*(C*D)
adi	Alternating Direction Implicit ソルバー
atax	行列の転置とベクトルの乗算
bicg	双共役勾配 (bicg) 法のサブカーネル
cholesky	対称正値行列のコレスキー分解
correlation	相関行列の計算
covariance	共分散行列の計算
deriche	画像処理のエッジ検出フィルタ
doitgen	多重解像度カーネルを実行
durbin	トープリッツ系ソルバー
fdtd-2d	2 次元有限差分時間領域法の計算カーネル
floyd-warshell	すべてのノード間の最短経路探索
gemm	行列積 式は C=alpha*A*B+beta*C
gemver	ベクトル乗算と行列加算を実行
gesummv	スカラー, ベクトル, 行列の乗算
gramschmidt	グラム・シュミット分解
heat-3d	3 次元データ領域に対する熱方程式
jacobi-1d	1 次元ヤコビ反復法
jacobi-2d	2 次元ヤコビ反復法
lu	LU 分解
ludcmp	LU 分解と前進代入
mvt	行列とベクトルの積と転置行列の積
nussinov	動的計画法を用いた配列整列
seidel-2d	2 次元ザイデル反復法
symm	対称行列の行列積
syr2k	対称ランク 2k 行列更新
syrk	対称ランク k 行列更新
trisolv	三角行列ソルバー
trmm	三角行列の行列積

3.3 バイナリコードからアセンブリコードへの変換

バイナリコードからアセンブリコードへの変換は, 逆コンパイルプロセスの第一段階を構成する重要なタスクである. このプロセスは, 機械語で記述された低水準の命令を, 人間が解釈可能なアセンブリ命令へと変換するものであり, 逆コンパイル全体の精度と可読性に大きく影響を及ぼす. この変換は, 大規模言語モデル（LLM）を用いることで, より高精度かつ効率的に行うことが可能だが, その実施には特有の課題がある. ここでは, ファインチューニング手法を用いたモデルの適用と, それに伴う課題を詳述する.

バイナリコードは,CPU が直接実行可能な形式で記述されており, 命令セットアーキテクチャ（ISA）に依存する. たとえば x86,ARM,RISC-V といった異なるアーキテクチャでは, それぞれ異なる命令セットが採用されているため, バイナリコードをアセンブリコードに変換する際には, 対象となるアーキテクチャの詳細を正確に理解する必要がある. このプロセスは単に命令を解釈するだけでなく, メモリアドレスやレジスタの操作, 条件分岐の構造など, 複雑な動作を含む情報を抽出する工程を伴う. そのため, 変換の精度は, アセンブリコードの可読性や正確性に直接的に影響を与える.

本研究では, バイナリコードからアセンブリコードへの変換を,LLM を活用したファインチューニングによって実現した. この変換プロセスを図 3.1 に示す. このモデルは,PolyBench データセットを基盤とし, バイナリコードと対応するアセンブリコードのペアを学習することで, バイナリ命令を正確に解析し, それに対応するアセンブリ命令を生成する能力を獲得する. モデルの学習においては, 特定の命令セットに特化したデータセットを使用することで, 対象アーキテクチャに対する理解を深め, 精度を向上させた. さらに, バイナリコードからアセンブリコードへの変換には, 条件分岐やジャンプ命令といった制御フローの解析が重要な役割を果たす. これらの命令は, プログラムの実行順序やロジックに大きく影響を及ぼすため, モデルが制御フローの構造を正確に理解する能力を備える必要がある. モデルは,PolyBench のプログラム例を通じて, 条件分岐の構造やループの開始位置と終了位置を正確に特定し, それに対応するアセンブリ命令を生成する能力を学習した.

このプロセスを通じて生成されたアセンブリコードは, 次の段階であるアセンブリコードから C コードへの変換において基盤となる. このため, アセンブリコードの精度, 逆コンパイル全体の成功にとって極めて重要である. 本研究では,PolyBench を活用したデータセット設計とファインチューニングにより, バイナリコードからアセンブリコードへの変換プロセスを大幅に改善し, 逆コンパイルタスクにおける新たな可能性を示した.

しかしながら, 逆コンパイルを行う際, 最大トークン数の制限が大きな課題となる. 長大なバイナリコードを一度に処理しようとする際, モデルは出力トークン数の制約により, アセンブリコードの一部しか生成できない可能性がある. この制限により, 期待される完全な対応アセンブリコードを獲得するためには工夫が必要となる.

これにより,16 進数バイナリコードからアセンブリコードを生成するには困難であることがわかった. 本研究では, アセンブリコードから C コードを出力することにする. 実行ファイルからアセンブリコードを取得して, それを入力として C コードを出力として得ることを目指す.

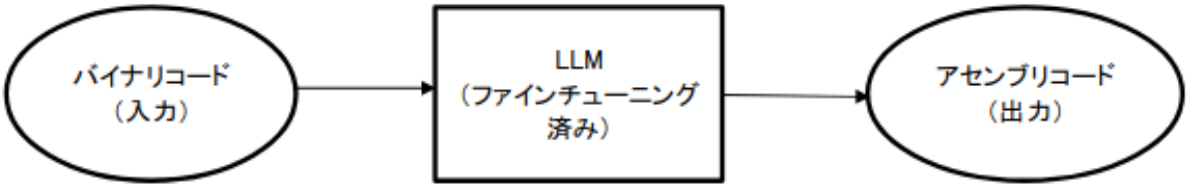

図 3.1: バイナリコードからアセンブリコードへのプロセス

3.4　アセンブリコードから C コードへの変換

アセンブリコードを C コードに変換するプロセスは, プログラムの可読性を向上させ, メンテナンスや解析を容易にするために重要な手法である. この段階では, 人間にとって直接的に解釈可能なアセンブリ命令を, 抽象度の高い高水準プログラミング言語である C コードへと変換する必要がある.C コードは, プログラムの論理構造や抽象的な設計を明示するものであり, アセンブリコードに比べて簡潔で可読性が高いが, その分, より高度な解析能力が求められる. アセンブリコードは,CPU 命令に対応する低水準の命令セットで構成されており, 直接的な動作を記述している. 一方,C コードは, 制御フローやデータ操作を抽象化した構造を持ち, プログラムの意図を表現することに重きを置いている. このため, アセンブリコードを C コードに変換する際には, 制御フロー, データフロー, 関数構造などを詳細に解析し, それを C 言語の文法やスタイルに適合させる高度な変換が必要となる. しかし, このプロセスにはいくつかの課題があり, 大規模言語モデルを用いることでこれらに対処しつつ, 正確な変換を実現することが求められる.

本研究では, アセンブリコードから C コードへの変換を担うモデルに対して,LLM を用いたファインチューニングを実施した. このモデルは,PolyBench データセットを基盤として訓練されており, 各アセンブリ命令がどのような C 言語構文に対応するかを学習する. アセンブリコードから C コードへの変換プロセスを図 3.2 に示す. 具体的には, 条件分岐, ループ構造, 関数呼び出しといったプログラムの制御構造をアセンブリコードから正確に抽出し, それを適切に C コードとして表現する能力を獲得することを目指した.

前節で触れたように, バイナリコードから直接アセンブリコードを高精度で出力することは, 最大トークン数の制限などもあり, 技術的に非常に困難であると判明した. その結果として, 本研究では

PolyBench の実行ファイルから直接アセンブリコードを取得し, そのアセンブリコードを入力として利用し, 対応する C コードを出力するプロセスを重点的に検討する.

アセンブリコードは低レベルの詳細な命令セットで構成されており, これを高レベルの C コードに変換することで, コードの可読性とメンテナンス性を向上させる. この変換では, 言語構造の抽象化と意図の再構築が求められる. 大規模言語モデルを活用したファインチューニングを通じて, この過程を支援する. アセンブリコードから C コードへの変換において, 制御フローの復元は重要な課題である. アセンブリコードでは, 条件分岐やジャンプ命令を用いて制御の流れが記述されるが,C コードではこれらを if 文や for ループ,while ループといった高水準の構文で表現する必要がある. 本研究では, モデルが制御フローグラフ（CFG）を構築し, それを解析することで, 複雑な分岐やループを正確に C コードに変換する手法を採用した. 例えば, アセンブリコードにおける条件付きジャンプ命令は, 条件式を抽出し, それに基づいて if 文を生成する. 一方, ループ構造の場合, 開始条件, 継続条件, 終了条件を特定し, それに対応する for 文や while 文を生成する. このような解析を可能にするため, モデルにはアセンブリ命令間の依存関係を学習させるだけでなく, 文脈的な判断を行う能力を付与した. データフローの解析もまた, アセンブリコードから C コードへの変換における重要な課題である. アセンブリコードでは, データ操作がレジスタやメモリアドレスを介して行われるため, それを C コードの変数や配列に変換する必要がある. 本研究では, データフローグラフ（DFG）を用いて, アセンブリ命令間のデータ依存性を解析し, それに基づいて C コードの変数やデータ構造を復元する方法を採用した. 例えば, アセンブリコード内でレジスタを介したデータ操作が記述されている場合, それが単純な変数操作なのか, 配列アクセスなのかを文脈から判断し, 適切な C コードを生成するようモデルを訓練した. また, 関数間でのデータの受け渡しや, グローバル変数の使用状況も解析し, それに基づいてコード全体の整合性を確保した. アセンブリコードに記述された命令の集合を高水準の関数として再構築するプロセスも, 本研究の重要な焦点である. アセンブリコードでは, サブルーチンとして記述される一連の命令が, そのまま関数に対応するとは限らない. このため, モデルには, アセンブリコード内の命令パターンを学習させ, それを関数として抽象化する能力を持たせた. さらに, 関数名や引数リストを文脈から推測し,C コードの可読性を向上させることを目指した.

本研究で構築したモデルは,PolyBench データセットを用いたファインチューニングにより, アセンブリコードから C コードへの変換精度を大幅に向上させた. 特に, 条件分岐やループ構造, データ依存性の解析といった複雑なタスクにおいて高い精度を示した. また, モデルは未見のアセンブリコードに対しても, 文脈を基に適切な C コードを生成する汎用性を備えていることが確認された.

- 構造識別とマッピング

  アセンブリコードで表現されるループや条件分岐, 関数呼び出しといったプログラム構造を識別し, それを C 言語の構文に精確にマッピングする. この識別は, モデルが異なるプログラムパ

ターンを理解し, 処理するために重要である.

- ファインチューニングの適用

  アセンブリコードとそれに対応する C コードのペアを用いてファインチューニングを行う. この過程では, モデルが言語間の対応を理解し, 正確な C コードを生成できる能力を強化する.

- データ構造の再構築

  アセンブリコード中のレジスタ操作やメモリ参照を解析し,C 言語の変数やデータ構造に置き換えて, 理解しやすい形に組み替える.

アセンブリコードから C コードへの変換には, 多くの課題が存在する. アセンブリコードはハードウェアに強く依存した命令を持ち, その多くが C 言語の抽象的表現には直接は対応しない. この結果, 情報の抽象化において曖昧さが生じる. また, 実行時効率を高めるためのコンパイル最適化は, アセンブリコードを元の C コードとは異なる形にしてしまう. さらに, 自動変換された C コードは多くの場合可読性に欠け, 維持管理が困難になる. アセンブリコードから C コードへの変換は, 逆コンパイル全体の成否を左右する重要なタスクである. 本研究では,LLM の文脈理解能力と生成能力を最大限に活用し, このタスクにおける課題を克服するための一歩を示した. 今後は, より大規模なデータセットや異なる命令セットアーキテクチャを用いたモデルの汎用性拡張を目指し, 逆コンパイル技術のさらなる発展に寄与することを期待している.

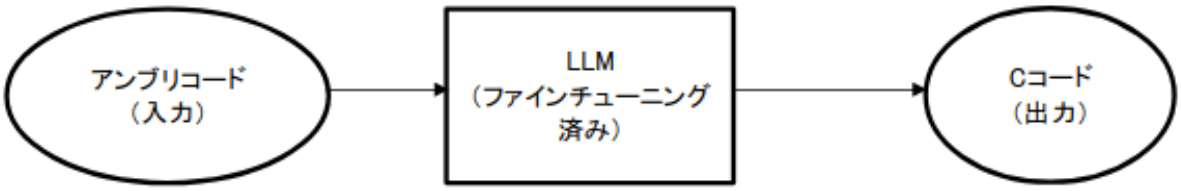

図 3.2: アセンブリコードから C コードへのプロセス

第4章　　評価

本章では, ファインチューニングされた大規模言語モデルによるアセンブリコードから C コードへの変換の性能を評価する. 特に, 生成された C コードが実際にコンパイル可能であるかどうか, そして元の PolyBench コードと同じ結果をもたらすかを検証する.

4.1　評価方法

モデルの性能を評価するため, 以下の手法を用いる.

- コンパイル可能性

  大規模言語モデルによって生成された C コードが標準 C コンパイラを使ってコンパイルできるかを確認する. これにより, コードの構文的な正しさと基本的なプログラム規則の遵守が評価される.

- 動作一致率

  コンパイルされた大規模言語モデルによって生成された C コードを実行し, 元の PolyBench プログラムと結果を比較する. 計算結果の一致度をチェックすることで, コードの機能的な正確性を評価する.

- 構文エラー修正後も再評価

  構文エラーが発生したコードについては、手動で軽微な修正（括弧の追加、必要最低限の引数補完など）を加え、再コンパイルし、動作を確認する。

4.2　評価結果

以下に各プログラムの評価結果について考察を述べる.

- 2mm

問題点は括弧や改行が欠如しているため, 構文エラーが発生した. このプログラムには基本的な構文チェックが適用されていない可能性が高い. エラーチェック機能の強化と, シンタックス修正アルゴリズムの導入が必要であるように思う.

- 3mm / nussinov / seidel-2d / floyd-warshell / gemver / gesummv

  コンパイルも可能で, 結果も一致した. アセンブリコードから C コードへの変換が正確に適用されている.

- adi

  問題点はコンパイルは可能だが, 計算結果が異なった. アルゴリズム変換において重要な計算や条件分岐が適切に処理されていない可能性がある. 変数の初期化や制御フローの忠実な再現が求められる.

- atax / lu / ludcmp / mvt / gemm

  問題点は構文エラーで括弧の欠如である. 文法構造の解析と整合性を保つ機能がモデルに欠けている可能性がある. シンタックスエラーチェックの強化が必要であるように考えれる.

- bicg

  問題点は print-array 関数の引数不足である. 関数定義と呼び出しを正確に一致させる機能が求められる. 関数シグネチャのチェック機能を追加することで改善される可能性がある.

- cholesky

  問題点はコンパイル可能だが, 計算結果が異なった. 演算における精度に問題がある可能性がある.

- correlation / covariance

  問題点は初期化段階の変数定義不足である. データ初期化の完全性がモデルによって確保されていないことである. 変数宣言の明確化と省略された部分の補完機能が求められる.

- deriche

  問題点は配列や変数の定義不足によるエラーである. 配列管理やスコープの扱いが不十分である. 大規模言語モデルにおけるデータ管理メカニズムの強化が必要である.

- doitgen

  成果はコンパイル可能で結果も一致した. 正常な変換が行われている. 他のプログラムにも見られる課題の克服例として活用できると思われる.

- durbin

  問題点はコンパイルは可能だが, 計算結果が異なった. 数値演算や浮動小数点演算の精度に問題がある可能性が高い.

- fdtd-2d

  成果はコンパイル可能で結果は一致である. 誤りの少ない変換ができた. 精度と信頼性が確保された例である.

- gramschmidt / heat-3d / jacobi-1d / jacobi-2d / symm / trisolv /trmm

  問題点はコンパイル可能だが, 計算結果が異なることである。変換アルゴリズムの微妙な誤りが結果の不一致に寄与している. 制御フローやデータフロー解析の精度を高める必要がある.

- syr2k / syrk

  問題点は関数引数の不足によるエラーである. 関数 API の変換で一貫性が欠如していると思われる. 各関数のシグネチャを正確に保つ工夫が求められる.

これらの評価と考察から, より正確で信頼性の高い逆コンパイルを実現するためには, モデルの構文エラーチェック及び補完の強化, 数値演算精度向上のための訓練, および関数間の整合性チェックの統合が必須である.

また, 以下に各プログラムの手動修正後の評価と考察を述べる.

- 2mm

  結果はコンパイル不可で結果も不一致であった.Kernel-2mm 関数に関する型の不整合によるエラーである. 同じ種類の問題が他のプログラムにも潜在している可能性があり, 型チェックの徹底が必要である.

- atax

  結果はコンパイルは可能であるが, 結果は不一致であった. 構文は正されているが, アルゴリズム変換における数値誤差またはロジックエラーが結果に影響している. さらなる検証が必要である.

- bicg

  結果はコンパイル不可であり結果も不一致であった. 型の不一致が原因であると思われる. 変数の型安全性チェックと, 詳細な型定義が重要である.

- correlation

  結果はコンパイル可能であり, 結果も一致した. 正しい修正がなされ, 期待通りの動作が確認された. 修正手法を他のエラーにも適用できる可能性がある.

- covariance

  結果はコンパイル可能であるが結果は不一致であった. コンパイルは成功しているが, 計算結果が異なった. データ操作部分に潜む論理エラーの検出が求められる.

- deriche

  結果はコンパイル不可で結果も不一致である. 配列や変数の型に関する不備によるものである. 管理すべきスコープやデータ構造の明確化が必要である.

- gemm

  結果はコンパイル不可で結果も不一致である. 変数 A の再宣言によるエラーである. 識別子の重複防止機能の導入と検証が必要である.

- lu / ludcmp

  結果はコンパイルは可能であるが結果不一致である. 数値誤差または最適化結果の不具合が原因と推測される. 最適化の適用や複雑なアルゴリズム変換に対する精度向上が望ましい.

- mvt

  結果はコンパイル不可で結果も不一致である. 同様に括弧や型の不正確さによる構文エラーである. 文字列やコードブロックの解析強化が必要である.

- syr2k / syrk

  結果は, コンパイル不可で結果も不一致である. 関数の引数不足が原因である.API の一貫性と, 自動的な引数補正機能の開発が必要である.

これらの評価結果から, 手動修正によってある程度の改善が見られたが, コンパイル成功率や結果の一致度をさらに高めるために, 型安全性とデータ構造の管理を強化する必要がある. 識別子の衝突と型ミスマッチを検出する機能の強化が求められ, これにより生成コードの信頼性と一貫性を向上させるべきである.

表 4.1: LLM による生成コードの評価

プログラム名	コンパイル	元のコードとの結果比較	備考
2mm	x	x	構文エラー
3mm	◯	◯	
adi	◯	x	構文エラー
atax	x	x	
bicg	x	x	print_array 関数の引数が足りない (int n) temp-q[0:N]=0 の書き方が C ではできない
cholesky	◯	x	
correlation	x	x	init_array 内の LARGE __ FLOAT_N が N である DATA_TYPE fn=(DATA_TYPE)LARGE_FLOAT_N; が不必要
covariance	x	x	init_array 内の LARGE __ FLOAT_N が N である DATA_TYPE fn=(DATA_TYPE)LARGE_FLOAT_N; が不必要
deriche	x	x	init_array の imgIn[i][j] での括弧がたりない imgOut がない kernel_deriche 内で定義されていない変数がいる
doitgen	◯	◯	
durbin	◯	x	
fdtd-2d	◯	◯	
floyd–warshell	◯	◯	
gemm	x	x	構文エラー
gemver	◯	◯	
gesummv	◯	◯	
gramschmidt	◯	x	
heat-3d	◯	x	
jacobi-1d	◯	x	
jacobi-2d	◯	x	
lu	x	x	構文エラー
ludcmp	x	x	構文エラー
mvt	x	x	構文エラー
nussinov	◯	◯	
seidel-2d	◯	◯	
symm	◯	x	
syr2k	x	x	print_array 関数の引数が足りない (int m)
syrk	x	x	print_array 関数の引数が足りない (int m)
trisolv	◯	x	
trmm	◯	x	

表 4.2: 構文修正後の再評価

プログラム名	コンパイル	元のコードとの結果比較	備考
2mm	x	x	kernel_2mm 関数に渡す引数の ポインタ型が一致しない
atax	◯	x	
bicg	x	x	A の型が関数内で定義された型と 一致していない
correlation	◯	x	
covariance	◯	x	
deriche	◯	◯	
gemm	x	x	A という識別子が異なる種類で再宣言されている 配列 C の定義やアクセスの方法に問題
lu	x	x	
ludcmp	x	x	
mvt	x	x	
syr2k	x	x	関数の引数の呼び出し不足
syrk	x	x	関数の引数の呼び出し不足

第5章　　おわりに

本研究では, 大規模言語モデル（LLM）を用いた逆コンパイル手法を提案し, その有効性を評価した. 逆コンパイルツールには, コンパイラ最適化によるコード構造の変形, コメントの欠如, 制御フローとデータフローの復元の困難さなど, 多くの課題が存在していた. 本研究では, これらの課題を克服するために,LLM を活用し, バイナリコードから C コードへの変換を二段階に分けるアプローチを採用した. 具体的には, バイナリコードからアセンブリコードへの変換モデル, およびアセンブリコードから C コードへの変換モデルを構築し, それぞれの段階において精度を向上させることを目指した.

また, 本研究では,LLM を逆コンパイルタスクに適応させるためにファインチューニングを実施した. その際,PolyBench データセットを活用し, バイナリコードと対応するアセンブリコード, およびアセンブリコードと C コードのペアデータを用いて学習を行った. このデータセットは, 数値計算アルゴリズムやプログラム構造に関する多様なコードを含んでおり, モデルの汎用性を高める上で有効であった.

評価実験の結果, 提案手法は逆コンパイルツールに比べて高い精度で C コードを生成できることが確認された. 特に, 変数名や関数名の推測精度が向上し, 可読性の高いコードを生成できることが分かった. ただし, 生成された C コードの一部において, 意図しないコードの冗長性が見られるケースもあり, 今後の改善が必要である.

本研究の成果は, セキュリティ診断やソフトウェア保守, プログラムの再利用, 教育など, 幅広い分野での応用が期待される. 特に, マルウェア解析や脆弱性検出においては,LLM を活用することで, より精度の高いコード解析が可能となり, セキュリティ分野における利便性向上に寄与する可能性がある. また, レガシーシステムの保守においても, 既存のソフトウェアの構造を復元し, 新しいプラットフォームへの移行を支援する手段として有効である.

一方で, 本研究にはいくつかの限界が存在する. 第一に,LLM の学習には膨大な計算資源が必要であり, 特にファインチューニングには大規模な GPU クラスタが求められる. そのため, リソースの制約がある環境では, 本手法の適用が難しい場合がある. 第二に,LLM はブラックボックス的な特性を持ち, 生成されたコードの正確性を保証する仕組みが不足している. 特に, バグの混入を防ぐための検証手法の確立が課題となる. 第三に, 本研究では PolyBench データセットを用いたが, より多様なコードを学習させることで, さらに汎用性の高いモデルを構築する余地がある.

今後の研究課題としては, 以下の点が挙げられる. まず,LLM のファインチューニング手法を改善し, より少ないデータで高精度なモデルを構築することが求められる. 特に, 逆コンパイル専用の事前学習モデルを開発し, 特定の命令セットアーキテクチャ（ISA）や最適化パターンに対応できるようにすることが重要である. また,LLM の解釈性を向上させるために, 生成されたコードの妥当性を検証する仕組みを導入し, バグの自動検出・修正が可能なシステムの開発が望まれる. さらに, 他のデータセットとの組み合わせによる学習や, 異なるアーキテクチャ（例: ARM, RISC-V）への対応を進めることで, より汎用的な逆コンパイルシステムの構築を目指す.

また,LLM と従来の静的解析・動的解析技術を統合することで, より強力な逆コンパイルシステムを実現することも考えられる. 例えば,LLM による初期コード生成後に, 静的解析ツールを用いてコードの整合性を確認し, 最終的な修正を加えるハイブリッドアプローチが有望である. 加えて, 逆コンパイルの精度向上だけでなく, 解析時間の短縮や計算コストの削減も今後の重要な課題となる.

本研究は, 大規模言語モデルを用いた逆コンパイルの可能性を示すものであり, 今後のさらなる発展に向けた基盤を提供するものである. 今後の研究を通じて, 本手法の実用性を向上させ, 逆コンパイル技術の進展に貢献することを期待しつつ, 本論文を締めくくる.

謝辞

本研究の機会を与えていただき，また，日頃から貴重な御意見，御指導いただいた，大津 金光教授，横田 隆史教授に深く感謝致します．そして，本研究において多大な御力添えを頂いた研究室の方々に感謝致します.

参考文献

[1] レガシーシステムに関わる過去の調査結果. 独立行政法人情報推進機構 デジタル基盤センター デジタルエンジニアリング部 ソフトウェアエンジニアリング G.

[2] NSA. ghidra. https://github.com/NationalSecurityAgency/ghidra. Accessed 2025-1-20.

[3] hex rays. Ida pro. https://hex-rays.com/ida-pro. Accessed 2025-1-20.

[4] Sergi Alvarez. Radare2. https://github.com/radareorg/radare2. Accessed 2025-1-20.

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

[6] 匂坂勇仁, 玉田春昭ほか. 逆難読化に向けての適用された難読化手法の特定. 第 78 回全国大会講演論文集, Vol. 2016, No. 1, pp. 389–390, 2016.

[7] 染谷実奈美, 大塚玲. 大規模言語モデルを用いたバイナリコードの機能推定手法. 人工知能学会全国大会論文集 第 38 回 (2024), pp. 4M1GS1005–4M1GS1005. 一般社団法人 人工知能学会, 2024.

[8] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 4554–4568, 2024.

[9] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, Vol. 267622140, 2024.

[10] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, Vol. 25, No. 7, pp. 811–829, 1995.

[11] Dylan Manuel, Nafis Tanveer Islam, Joseph Khoury, Ana Nunez, Elias Bou-Harb, and Peyman Najafirad. Enhancing reverse engineering: Investigating and benchmarking large language models for vulnerability analysis in decompiled binaries. *arXiv preprint arXiv:2411.04981*, 2024.

[12] 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.

[13] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings 18*, pp. 247–264. Springer, 2020.

[14] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, et al. Leveraging generative models to recover variable names from stripped binary. *arXiv preprint arXiv:2306.02546*, 2023.

[15] Saman Pordanesh and Benjamin Tan. Exploring the efficacy of large language models (gpt-4) in binary reverse engineering. *arXiv preprint arXiv:2406.06637*, 2024.

[16] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in stripped binary code understanding using large language models. *arXiv preprint arXiv:2404.09836*, 2024.

[17] Louis-Noel Pouchet. Polybenchc-4.2.1. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1. Accessed 2024-12-26.