LLM4Decompile: 大規模言語モデルによるバイナリコードのデコンパイル

概要

デコンパイルはバイナリコードを高レベルのソースコードに変換することを目的としていますが、GhidraのようなTraditionalなツールは、読みにくく実行困難な結果を生み出すことが多いです。大規模言語モデル（LLM）の進歩に触発され、私たちはLLM4Decompileを提案します。これはバイナリコードをデコンパイルするために訓練された最初かつ最大のオープンソースLLMシリーズ（1.3Bから33B）です。我々はLLMトレーニングプロセスを最適化し、バイナリを直接デコンパイルするLLM4Decompile-Endモデルを導入しました。その結果、HumanEvalおよびExeBenchベンチマークにおいて、GPT-4oやGhidraと比較して再実行可能率が100%以上向上しました。さらに、標準的な改良アプローチを改善してLLM4Decompile-Refモデルを微調整し、Ghidraからデコンパイルされたコードを効果的に洗練させ、LLM4Decompile-Endよりもさらに16.2%の改善を達成しました。LLM4Decompileはバイナリコードのデコンパイルを革新するLLMの可能性を示し、読みやすさと実行可能性に関して顕著な改善をもたらし、最適な結果を得るために従来のツールを補完します。

1. はじめに

デコンパイルは機械コードやバイナリコードを高レベルプログラミング言語に変換する逆プロセスで、脆弱性の特定、マルウェア研究、レガシーソフトウェアの移行などの様々なリバースエンジニアリングタスクを容易にします。デコンパイルは、コンパイルプロセスで本質的に失われる情報、特に変数名やループと条件分岐のような基本構造の詳細情報が喪失するため、困難です。これらの課題に対処するため、GhidraやIDA Proなど多くのツールが開発されてきました。これらのツールはバイナリコードを高レベルの疑似コードに戻す能力を持ちますが、その出力はしばしば読みやすさと再実行可能性に欠け、レガシーソフトウェアの移行やセキュリティ計装タスクなどのアプリケーションにとって不可欠です。

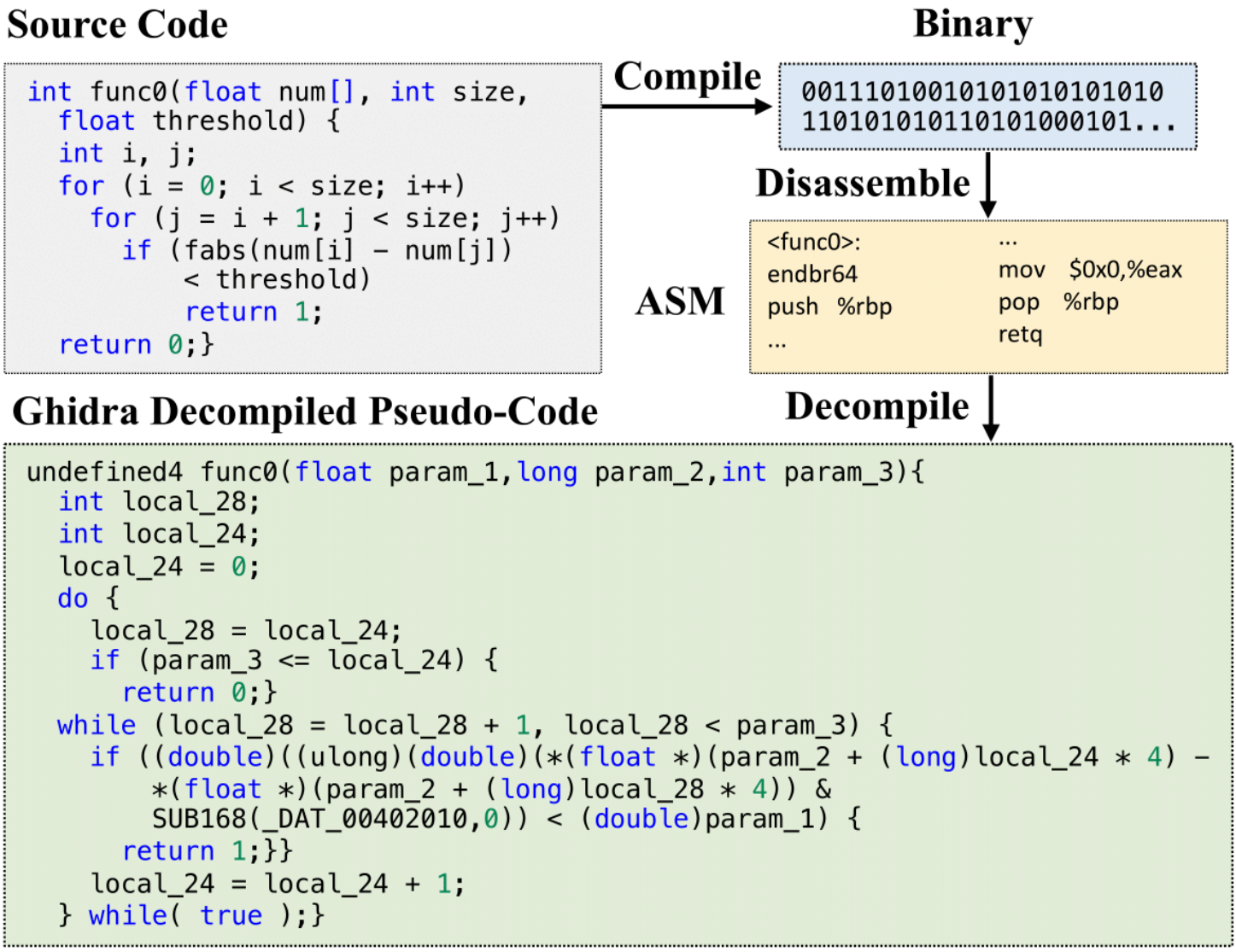


図1は、ソースCコードからバイナリファイル、アセンブリコード（ASM）、Ghidraからデコンパイルされた疑似コードへの変換を示しています。この疑似コードでは、元のネストされた「for」構造が、別の「while」ループ内の「do-while」ループの組み合わせに置き換えられています。さらに、「num[i]」のような配列インデックスは、「*(float *)(param_2 + (long)local_24 * 4)」のような複雑なポインタ算術にデコンパイルされます。デコンパイルされた出力には構文的なエラーもあり、関数の戻り値の型が「undefined4」に変換されています。全体的に、従来のデコンパイルツールは高レベル言語が提供する構文的な明確さを取り除き、構文の正確性を保証しないため、熟練した開発者でもアルゴリズムのロジックを再構築するのが非常に困難です。

最近の大規模言語モデル（LLM）の進歩により、コードのデコンパイルプロセスが大幅に改善されました。LLMベースのデコンパイルには、*Refined-Decompile*と*End2end-Decompile*という2つの主要なアプローチがあります。特に、*Refined-Decompile*はLLMを活用して従来のデコンパイルツールからの結果を洗練します。しかし、LLMは主に高レベルプログラミング言語に最適化されており、バイナリデータでは効果的でない可能性があります。*End2end-Decompile*はLLMを微調整してバイナリを直接デコンパイルします。しかし、このアプローチの以前のオープンソースアプリケーションは、約2億のパラメータしか持たない小さなモデルと限られたトレーニングコーパスに制限されていました。それに対して、より広範なデータセットで訓練されたより大きなモデルを利用することで、パフォーマンスが大幅に向上することが証明されています。

前述の研究の限界を克服するため、我々はLLM4Decompileを提案します。これはバイナリコードをデコンパイルするために特別に訓練された、1.3Bから33Bのパラメータサイズを持つ初めての最大のオープンソースLLMシリーズです。我々の知る限り、このような深さでLLMベースのデコンパイルの能力を向上させようとしたり、このような大規模なLLMを組み込んだりする前例はありません。*End2end-Decompile*アプローチに基づき、データ拡張、データクリーニング、2段階トレーニングという3つの重要なステップを導入して、LLMトレーニングプロセスを最適化し、バイナリを直接デコンパイルするLLM4Decompile-Endモデルを導入しました。具体的には、我々のLLM4Decompile-End-6.7BモデルはHumanEvalで45.4%、ExeBenchで18.0%の成功したデコンパイル率を示し、GhidraやGPT-4oを100%以上上回っています。さらに、Ghidraのデコンパイルプロセスの効率を検証し、データを拡張・フィルタリングしてLLM4Decompile-Refモデルを微調整する*Refined-Decompile*戦略を改善しました。これらのモデルはGhidraの出力を洗練させることに優れており、実験によると*Refined-Decompile*アプローチの性能上限が高くなり、LLM4Decompile-Endに対して16.2%の改善を達成しています。また、ソフトウェア保護で一般的に使用される難読化条件下でのモデルの潜在的な誤用に関連するリスクを評価しました。我々の調査結果は、我々のアプローチもGhidraも難読化されたコードを効果的にデコンパイルできないことを示しており、知的財産権の侵害のための不正使用に関する懸念を軽減しています。

要約すると、我々の貢献は以下の通りです：

- デコンパイル用に微調整された、15億トークンで訓練された最初かつ最大のオープンソースLLM（1.3Bから33Bのパラメータ範囲）であるLLM4Decompileシリーズを導入します。
- LLMトレーニングプロセスを最適化し、直接バイナリデコンパイルの新しいパフォーマンス基準を設定するLLM4Decompile-Endモデルを導入します。これらはHumanEvalとExeBenchベンチマークにおいて、再実行可能性の面でGPT-4oとGhidraを100%以上上回っています。
- *Refined-Decompile*アプローチを改善してLLM4Decompile-Refモデルを微調整し、Ghidraからデコンパイルされた結果を効果的に洗練させ、LLM4Decompile-Endよりもさらに16.2%の再実行可能性向上を達成します。

2. 関連研究

実行可能バイナリをソースコード形式に戻す「デコンパイル」という実践は、数十年に渡って研究されてきました。従来のデコンパイルはプログラムの制御フローとデータフローの分析に依存し、Hex-Rays Ida proやGhidraなどのツールで見られるようにパターンマッチングを採用しています。これらのシステムは、プログラムの制御フローグラフ（CFG）内で条件文やループなどの標準的なプログラミング構造に対応するパターンを識別しようとします。しかし、このようなデコンパイルプロセスからの出力は、変数のレジスタへの直接変換、gotoの使用、その他の低レベル演算など、アセンブリコードのソースコードに似た表現になる傾向があります。この出力は、元のコードと機能的に似ていることが多いですが、理解が難しく、再実行できない可能性があります。ニューラル機械翻訳からインスピレーションを得て、研究者はデコンパイルを機械レベルの命令を読みやすいソースコードに変換する翻訳演習として再構成しました。この分野での初期の試みでは、デコンパイルにリカレントニューラルネットワーク（RNN）を利用し、結果を強化するためのエラー修正技術を補完しました。

大規模言語モデルの成功に触発され、研究者はデコンパイルにLLMを採用し、主に「*Refined-Decompile*」と「*End2end-Decompile*」という2つのアプローチを通じて行いました。特に、*Refined-Decompile*はGhidraやIDA Proなどの従来のデコンパイルツールからの結果を洗練するようLLMにプロンプトします。例えば、DeGPTはGhidraの読みやすさを向上させ、認知負荷を24.4%削減し、DecGPTはエラーメッセージを洗練プロセスに組み込むことで、IDA Proの再実行可能率を75%以上に向上させます。しかし、これらのアプローチは、LLMが主に高レベルプログラミング言語向けに設計されており、バイナリファイルでの有効性が十分に確立されていないという事実を大きく無視しています。一方、*End2end-Decompile*はLLMを微調整してバイナリを直接デコンパイルします。BTCやSladeなどの初期のオープンソースモデルは、約2億のパラメータを持つ言語モデルを採用してデコンパイル用に微調整しています。Novaはオープンソース化されていませんが、10億のパラメータを持つバイナリLLMを開発し、デコンパイル用に微調整しています。その結果、この分野の最大のオープンソースモデルは2億に限定されています。より広範なデータセットで訓練されたより大きなモデルを利用することで、パフォーマンスが大幅に向上することが証明されています。

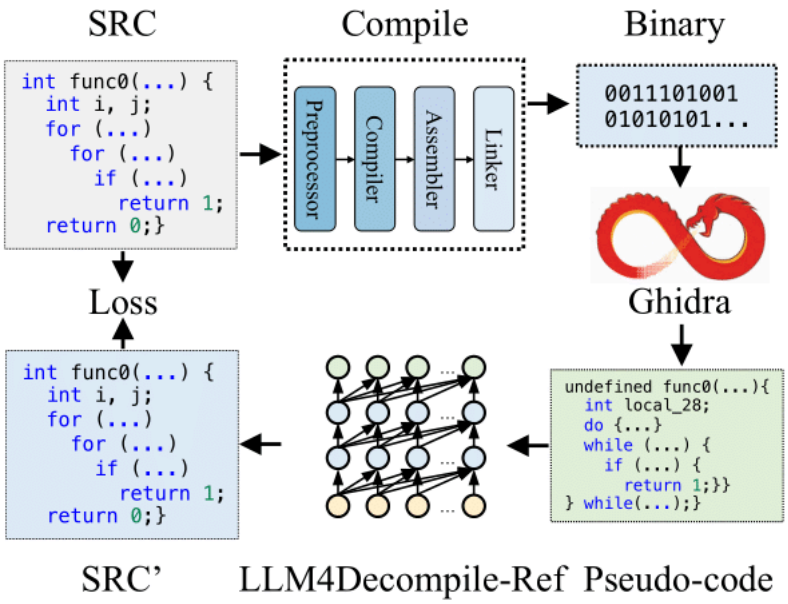
したがって、我々の目標は、LLMのデコンパイル能力を包括的に進めることを目指す、最初で最も広範なオープンソースLLM4Decompileシリーズを提示することです。まず、*End2end-Decompile*アプローチを最適化してLLM4Decompile-Endを訓練し、バイナリファイルを直接デコンパイルする効果を実証します。次に、LLMとGhidraを統合し、最適な効果を得るために従来のツールを強化するよう*Refined-Decompile*フレームワークを強化します。

3. LLM4Decompile

まず、バイナリを直接デコンパイルするためのLLMトレーニングを最適化する戦略を紹介します。この結果のモデルはLLM4Decompile-Endと名付けられています。次に、*Refined-Decompile*アプローチを強化する取り組みについて詳述します。対応する微調整されたモデルはLLM4Decompile-Refと呼ばれ、Ghidraからデコンパイルされた結果を効果的に洗練することができます。

3.1 LLM4Decompile-End

このセクションでは、一般的な*End2end-Decompile*フレームワークを説明し、LLM4Decompile-Endモデルのトレーニングを最適化する戦略について詳細を提示します。



3.1.1 End2End-Decompileフレームワーク

図2は、コンパイルからデコンパイルプロセスまでの*End2end-Decompile*フレームワークを示しています。コンパイル中、プリプロセッサはソースコード（SRC）を処理してコメントを削除し、マクロやインクルードを展開します。クリーニングされたコードはコンパイラに転送され、アセンブリコード（ASM）に変換されます。このASMはアセンブラによってバイナリコード（0と1）に変換されます。リンカーは関数呼び出しをリンクして実行可能ファイルを作成し、プロセスを完了します。一方、デコンパイルはバイナリコードをソースファイルに戻す作業を含みます。テキストでトレーニングされたLLMは、バイナリデータを直接処理する能力がありません。したがって、バイナリはまず `Objdump` によってアセンブリ言語（ASM）に逆アセンブルされる必要があります。バイナリと逆アセンブルされたASMは同等であり、互換性があるため、我々はこれらを相互に参照します。最後に、デコンパイルされたコードとソースコードの間で損失が計算され、トレーニングを導きます。

3.1.2 LLM4Decompile-Endの最適化

LLM4Decompile-Endモデルのトレーニングを3つの重要なステップで最適化します：1）トレーニングコーパスの拡張、2）データ品質の向上、3）2段階トレーニングの導入。

トレーニングコーパス

スケーリング法則が示すように、LLMの効果はトレーニングコーパスのサイズに大きく依存します。したがって、トレーニング最適化の最初のステップは、大規模なトレーニングコーパスを組み込むことです。我々は、5百万のC関数を含む最大の公開コレクションであるExeBenchに基づいてasm-sourceペアを構築します。トレーニングデータをさらに拡張するために、開発者によって頻繁に使用されるコンパイル最適化状態を考慮します。コンパイル最適化には、冗長な命令の削除、より良いレジスタ割り当て、ループ変換などの技術が含まれ、デコンパイルのデータ拡張として完璧に機能します。主要な最適化レベルはO0（デフォルト、最適化なし）からO3（積極的な最適化）です。我々はソースコードをこれら4つの段階、つまりO0、O1、O2、O3すべてにコンパイルし、それぞれをソースコードとペアにします。

データ品質

データ品質は効果的なモデルをトレーニングする上で重要です。したがって、2番目のステップはトレーニングセットをクリーニングすることです。我々はStarCoderのガイドラインに従い、コードのMinHashを計算し、局所的に敏感なハッシュ（LSH）を使用して重複を削除します。また、10トークン未満のサンプルも除外します。

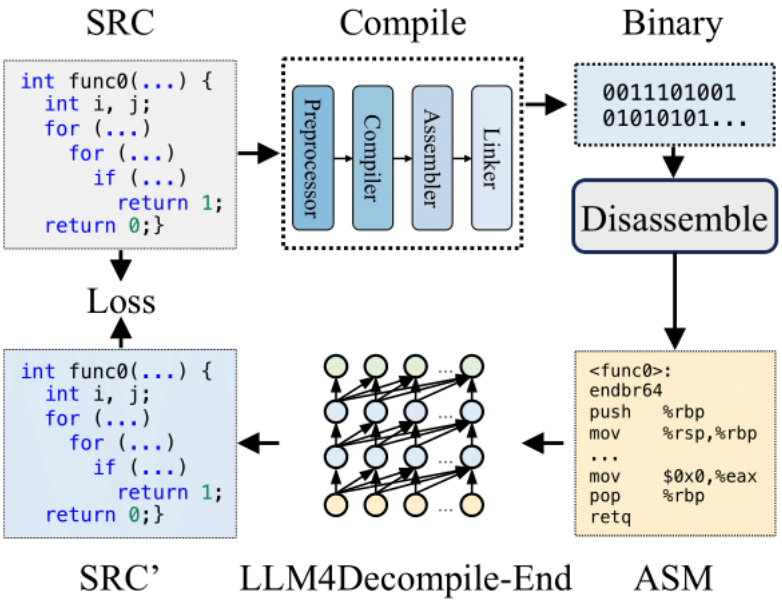
2段階トレーニング

トレーニング最適化の最後のステップは、モデルにバイナリ知識を教育することを目的とし、2段階トレーニングを含みます。第1段階では、コンパイル可能だがリンク可能（実行可能）ではない大規模なデータのコーパスでモデルをトレーニングします。コンパイル可能だがリンク可能ではないCコードを抽出するのが大幅に容易であることに注意してください。そのような非実行可能なバイナリオブジェクトコードは、外部シンボルのリンクされたアドレスが欠けている点を除いて、その実行可能バージョンと非常に似ています。したがって、第1段階では、広範なコンパイル可能コードを使用してバイナリ知識でモデルを基礎付けます。第2段階では、実用的な適用可能性を確保するために実行可能コードを使用してモデルを洗練します。我々は2段階トレーニングのアブレーション研究も実施しています。コンパイル可能データと実行可能データの比較は付録8で詳述されています。

3.2 LLM4Decompile-Ref

ここで、従来のデコンパイルツールであるGhidraがLLMと統合することでどのように大幅に改善できるかを検討します。我々のアプローチは、Ghidraからの出力全体を洗練することを目的としており、単に名前やタイプを回復するよりも広範な戦略を提供します。まず一般的な*Refined-Decompile*フレー

ムワークを詳述し、LLM4Decompile-RefによるGhidraの出力強化の戦略について議論します。



3.2.1 Refined-Decompileフレームワーク

*Refined-Decompile*アプローチは図3に示されています。このアプローチは図2のそれとは、LLMの入力という点でのみ異なります。*Refined-Decompile*の場合、入力はGhidraのデコンパイル出力から来ます。具体的には、Ghidraを使用してバイナリをデコンパイルし、その後LLMを微調整してGhidraの出力を強化します。Ghidraは読みやすさの問題と構文エラーがあるかもしれない高レベルの疑似コードを生成しますが、基礎となるロジックを効果的に保持します。この疑似コードを洗練することで、不明瞭なASMを理解する課題が大幅に軽減されます。

LLM4Decompile-RefによるGhidraの洗練

Ghidraを使用したデコンパイル

Ghidraで実行可能コードをデコンパイルすることは、多数の外部関数とIOラッパーを含むExeBenchの実行可能ファイルの複雑な性質のため、時間がかかります。Ghidra Headlessは128コアマルチプロセッシングを使用して1サンプルあたり2秒を要します。このような高い計算負荷と、非実行可能バイナリと実行可能バイナリの高い類似性を考慮して、我々はGhidraを使用して非実行可能ファイルをデコンパイルすることを選択しました。この選択により、サンプルあたりの時間が0.2秒に大幅に削減され、大量のトレーニングデータを効率的に収集できるようになりました。

最適化戦略

セクション3.1.2と同様に、最適化レベルO0、O1、O2、O3でコンパイルすることでデータセットを拡張します。さらに、LSHを使用してデータセットをフィルタリングし、重複を削除します。図1に示されているように、Ghidraはしばしば過度に長い疑似コードを生成します。その結果、我々のモデルが受け入れる最大長を超えるサンプルを除外します。

4. 実験

このセクションでは、LLM4Decompile-EndとLLM4Decompile-Refの実験設定と結果についてそれぞれ議論します。

4.1 LLM4Decompile-End

4.1.1 実験設定

トレーニングデータ

セクション3.1.2で議論したように、ExeBenchからのコンパイル可能および実行可能データセットに基づいてasm-sourceペアを構築します。ここでは、x86 Linuxプラットフォーム下でGCCでコンパイルされたC関数のデコンパイルのみを考慮します。フィルタリング後、洗練されたコンパイル可能トレーニングデータセットには720万サンプル（約70億トークン）が含まれます。実行可能トレーニングデータセットには160万サンプル（約5.72億トークン）が含まれます。モデルをトレーニングするために、次のテンプレートを使用します：

```
# This is the assembly code: [ASM code] # What is the source code? [source code]
```

ここで、`[ASM code]`はバイナリから逆アセンブルされたアセンブリコードに対応し、`[source code]`は元のC関数です。モデルをソースコードを生成するように微調整するため、テンプレートの選択はパフォーマンスに影響しないことに注意してください。

評価ベンチマークとメトリクス

モデルを評価するために、HumanEvalとExeBenchベンチマークを導入します。HumanEvalはコード生成評価の主要なベンチマークであり、164のプログラミング課題と付随するPythonソリューションとアサーションを含みます。我々はこれらのPythonソリューションとアサーションをCに変換し、GCCコンパイラで標準Cライブラリを使用してコンパイルでき、すべてのアサーションに合格することを確認し、HumanEval-Decompileと名付けました。ExeBenchは、IOの例を含むGitHubから取得した5000の実世界C関数で構成されています。HumanEval-Decompileは標準Cライブラリにのみ依存する個別の関数で構成されていることに注意してください。対照的に、ExeBenchには、ユーザー定義の構造体と関数を持つ実世界のプロジェクトから抽出された関数が含まれています。

評価メトリクスについては、先行研究に従って再実行可能率を計算します。評価中、Cソースコードはまずバイナリにコンパイルされ、アセンブリコードに逆アセンブルされ、デコンパイルシステムに入力されてCコードに再構成されます。このデコンパイルされたCコードはアサーションと組み合わせて、正常に実行しアサーションに合格できるかどうかを確認します。

モデル構成

LLM4DecompileはDeepSeek-Coderと同じアーキテクチャを使用し、対応するDeepSeek-Coderチェックポイントでモデルを初期化します。我々はSequence-to-sequence予測（S2S）を採用しています。これは、入力シーケンスを与えられた出力を予測することを目的とするほとんどのニューラル機械翻訳モデルで採用されているトレーニング目標です。式[eq2]に示されているように、ソースコードトークン $x_i, ..., x_j$ の負の対数尤度を最小化します：

$$\mathcal{L} = -\sum_{i} \log P_i(x_i, ..., x_j | x_1, ..., x_{i-1}; \theta)$$

ここで、損失は出力シーケンス $x_i ... x_j$ 、つまりソースコードに対してのみ計算されます。

ベースライン

比較のために2つの主要なベースラインを選びました。まず、GPT-4oは最も有能なLLMを代表し、LLMパフォーマンスの上限を提供します。次に、DeepSeek-Coderは現在の最先端のオープンソースコードLLMとして選ばれています。これはコーディングタスク専用に特別に調整された、公開されているモデルの最前線を代表しています。最近の研究Sladeはデコンパイル用に言語モデルを微調整していますが、コンパイラの中間出力、特に*.sファイルに依存しています。しかし、実際にはこのような中間ファイルは開発者によって公開されることはほとんどありません。したがって、我々はより現実的なアプローチに焦点を当て、バイナリからのデコンパイルのみを考慮します。詳細な議論は付録8をご参照ください。

実装

Hugging Faceから入手したDeepSeek-Coderモデルを使用します。LLaMA-Factoryライブラリを用いてモデルをトレーニングします。1.3Bと6.7Bモデルについては、バッチサイズ2048、学習率2e-5を設定し、2エポック（150億トークン）モデルをトレーニングします。実験はNVIDIA A100-80GB GPUクラスターで実行されます。LLM4Decompile-End 1.3Bと6.7Bの微調整には、8×A100でそれぞれ12日と61日かかります。リソースの制約により、33Bモデルについては2億トークンでのみトレーニングします。評価には、生成（デコンパイル）プロセスを加速するために*vllm*を使用します。ランダム性を最小限に抑えるために貪欲デコーディングを採用しています。

4.1.2 実験結果

主な結果

表[table:main_results]は、研究対象モデルの最適化状態ごとの再実行可能率を示しています。DeepSeek-Coder-33Bの基本バージョンはバイナリを正確にデコンパイルすることができません。正しいように見えるコードを生成しますが、元のプログラムセマンティクスを保持できません。GPT-4oは注目すべきデコンパイルスキルを示し、非最適化（O0）コードを30.5%の成功率でデコンパイルできますが、最適化されたコード（O1-O3）では約11%に大幅に減少します。一方、LLM4Decompile-Endモデルは優れたデコンパイル能力を示しています。1.3Bバージョンは平均で27.3%のケースでプログラムセマンティクスを保持して正常にデコンパイルし、6.7Bバージョンは45.4%の成功率を達成しています。この改善は、プログラムのセマンティクスをより効果的に捉えるために大きなモデルを使用する利点を強調しています。33Bモデルの微調整を試みる際、高い通信負荷に関連する大きな課題に直面し、トレーニングプロセスが大幅に遅くなり、2億トークンのみ使用することになりました。この制限にもかかわらず、33Bモデルは依然として1.3Bモデルを上回っており、モデルサイズのスケールアップの重要性を再確認しています。

モデル/ベンチマーク	HumanEval-Decompile					ExeBench				
最適化レベル	O0	O1	O2	O3	平均	O0	O1	O2	O3	平均
DeepSeek-Coder-6.7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GPT-4o	30.49	11.59	10.37	11.59	16.01	4.43	3.28	3.97	3.43	3.78
LLM4Decompile-End-1.3B	47.20	20.61	21.22	20.24	27.32	17.86	13.62	13.20	13.28	14.49
LLM4Decompile-End-6.7B	**68.05**	**39.51**	**36.71**	**37.20**	**45.37**	**22.89**	**16.60**	**16.18**	**16.25**	**17.98**

モデル/ベンチマーク	HumanEval-Decompile					ExeBench				
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54	18.86	14.65	13.96	14.11	15.40

表1: モデル別のHumanEval-DecompileとExeBenchのパフォーマンス比較

モデル/ベンチマーク	HumanEval-Decompile					ExeBench				
最適化レベル	O0	O1	O2	O3	平均	O0	O1	O2	O3	平均
Compilable-1.3B	42.68	16.46	16.46	17.07	23.17	5.68	4.46	4.16	4.43	4.68
Compilable-6.7B	**51.83**	**33.54**	**32.32**	**32.32**	**37.50**	7.52	6.49	6.71	6.60	6.83
Executable-1.3B	19.51	12.80	12.80	11.59	14.18	21.94	19.46	19.31	19.50	20.05
Executable-6.7B	37.20	18.29	22.56	17.07	23.78	**29.38**	**25.98**	**25.91**	**25.49**	**26.69**

表2: トレーニングデータ種類別のHumanEval-DecompileとExeBenchのパフォーマンス比較

アブレーション研究

セクション4.1.1.1で議論したように、我々のトレーニングデータは2つの異なるセットで構成されています：720万のコンパイル可能関数（非実行可能）と160万の実行可能関数です。これらのデータセットを使用してアブレーション研究を実施し、結果は表2に表示されています。ここで、「Compilable」はコンパイル可能データのみでトレーニングされたモデルを示し、「Executable」は実行可能データのみでトレーニングされたモデルを示します。コンパイル可能関数からのバイナリオブジェクトは関数呼び出しへのリンクがなく、標準Cライブラリにのみ依存する単一関数から成るHumanEval-Decompileデータのテキスト分布と類似していることに注目する価値があります。その結果、コンパイル可能データのみでトレーニングされた6.7Bモデルは、HumanEval-Decompileの関数の37.5%を正常にデコンパイルしましたが、広範なユーザー定義関数を特徴とするExeBenchでは6.8%のみでした。一方、実行可能データのみでトレーニングされた6.7BモデルはExeBenchテストセットで26.7%の再実行可能率を達成しましたが、トレーニングコーパスが小さいため、単一関数に課題があり、HumanEval-Decompileでは23.8%の成功率のみでした。スペースの制限により、さらなる分析は付録9で提示します。

Source Code

```
int func0(float num[], int size,
  float threshold) {
  int i, j;
  for (i = 0; i < size; i++)
    for (j = i + 1; j < size; j++)
      if (fabs(num[i] - num[j])
          < threshold)
          return 1;
  return 0;}
```

LLM4Decompile-End-6.7B

```
int func0(float *x, int n,
  float eps){
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (fabsf(x[i] - x[j]) < eps)
        return 1;
  return 0;}
```

Ghidra

```
undefined4 func0(float param_1,long param_2,int param_3){
  int local_28;
  int local_24;
  local_24 = 0;
  do {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;}
  while (local_28 = local_28 + 1, local_28 < param_3) {
    if ((double)((ulong)(double)(*(float *)(param_2 + (long)local_24 * 4) -
        *(float *)(param_2 + (long)local_28 * 4)) &
        SUB168(_DAT_00402010,0)) < (double)param_1) {
      return 1;}}
    local_24 = local_24 + 1;
  } while( true );}
```

Ghidra+LLM4Decompile-6.7B

```
int func0(float *x, int n,
  float eps){
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (fabsf(x[i] - x[j]) < eps)
        return 1;
  return 0;}
```

GPT-4o

```
int func0(float **arr, int length,
  float threshold) {
  int outer = 0;
  while (outer < length) {
    int inner = 0;
    while (inner < length) {
      float diff = arr[outer][inner]
        - arr[inner][inner];
      if (fabs(diff) <= threshold) {
        return 1;}
      inner++;}
    outer++;}
  return 0;}
```

Ghidra+GPT-4o

```
typedef unsigned int undefined4;
undefined4 func0(float param_1,
  long param_2, int param_3)
  {
  int local_28;
  int local_24;
  local_24 = 0;
  while (1) {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;
    }
    while (local_28 + 1 < param_3) {
      local_28++;
      if ((double)((*(float *)
        (param_2 + (long)local_24 * 4)
        - *(float *)(param_2 +
        (long)local_28 * 4)))
        < (double)param_1) {
          return 1;
      }
    }
    local_24++;}}
```

図4は異なるアプローチのデコンパイル結果を示しています。GPT-4oの出力は妥当に見えますが、配列の次元（不正確な2次元配列 `arr[outer][inner]` ）を回復できません。Ghidraの疑似コードは図1で議論されているように著しく読みにくくなっています。GPT-4oによって洗練されたGhidraの結果（Ghidra+GPT-4o）は読みやすさをわずかに向上させますが、 `for` ループと配列インデックスを正しくレンダリングできません。対照的に、LLM4Decompile-EndとLLM4Decompile-Refは正確で読みやすい出力を生成します。

4.2 LLM4Decompile-Ref

4.2.1 実験設定

実験データセット

トレーニングデータはExeBenchを使用して構築され、Ghidra Headlessを使用してバイナリオブジェクトファイルをデコンパイルします。計算リソースの制約により、O0からO3までの最適化レベルを持つ40万関数（160万サンプル、10億トークン）のみがトレーニングに使用され、評価はHumanEval-Decompileで実施されます。モデルはセクション4.1.1.1で説明したのと同じテンプレートを使用してトレーニングされます。さらに、先行研究に従って、デコンパイルされた結果の読みやすさを編集類似度スコアの観点から評価します。

実装

モデルの構成設定はセクション4.1.1.1と一致しています。1.3Bと6.7Bモデルについては、微調整プロセスは2エポックで20億トークンを含み、8×A100でそれぞれ2日と8日を要します。リソースの制限により、33Bモデルについては2億トークンのみでトレーニングします。評価のために、まずGhidraの再実行可能率にアクセスしてベースラインを確立します。その後、GPT-4oを使用してGhidraのデコンパイル結果を強化
し、
`Generate linux compilable C/C++ code of the main and other functions in the supplied snippet without using goto, fix any missing headers. Do not ex`
というプロンプトを使用します。これはDecGPTに従っています。最後に、LLM4Decompile-Refモデルを使用してGhidraの出力を洗練します。

4.2.2 実験結果

ベースラインと*Refined-Decompile*アプローチの結果は表3にまとめられています。再実行に最適化されていないGhidraによってデコンパイルされた疑似コードでは、平均で20.1%のみがテストケースに合格します。GPT-4oはこの疑似コードの洗練と品質向上を支援します。LLM4Decompile-Refモデルは、Ghidraの出力に対して大幅な改善を提供し、6.7Bモデルは再実行可能性を160%向上させます。セクション4.1.2.1での議論と同様に、33Bモデルは大幅に少ないトレーニングデータを使用したにもかかわらず、1.3Bモデルを上回っています。そして、10倍多くのトレーニングデータの恩恵を受けた6.7Bモデルよりも3.6%下回るパフォーマンスを達成しています。LLM4Decompile-End-6.7Bと比較して、LLM4Decompile-Ref-6.7Bモデルは、LLM4Decompile-Refモデルのデータの10%のみでトレーニングされたにもかかわらず、16.2%のパフォーマンス向上を示しており、*Refined-Decompile*アプローチの大きな可能性を示唆しています。さらなる分析は付録10で提示します。

モデル/メトリクス	**再実行可能率**					**編集類似度**				
最適化レベル	O0	O1	O2	O3	平均	O0	O1	O2	O3	平均
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37	15.57	12.92	12.93	12.69	13.53
Base	34.76	16.46	15.24	14.02	20.12	6.99	6.13	6.19	5.47	6.20
+GPT-4o	46.95	34.15	28.66	31.10	35.22	6.60	5.63	5.67	4.99	5.72
+LLM4Decompile-Ref-1.3B	68.90	37.20	40.85	37.20	46.04	15.17	13.25	12.92	12.67	13.50
+LLM4Decompile-Ref-6.7B	74.39	46.95	47.56	42.07	52.74	**15.59**	13.53	13.42	12.73	13.82
+LLM4Decompile-Ref-33B*	70.73	47.56	43.90	41.46	50.91	15.40	**13.79**	**13.63**	**13.07**	**13.97**
+LLM4Decompile-Ref-22B*	**80.49**	**58.54**	**59.76**	**57.93**	**64.18**	15.19	**14.04**	13.58	**13.40**	13.85

表3: LLM4Decompile-EndとLLM4Decompile-Refモデルの再実行可能率と編集類似度の比較

異なる方法間の読みやすさの分析も実施され、表3に示されており、例示的な例は図4に提示されています。テキスト類似性について、すべてのデコンパイル出力は元のソースコードから逸脱しており、編集類似度は5.7%から14.0%の範囲にあります。これは主にコンパイルプロセスが変数名を削除し、ロジック構造を最適化するためです。Ghidraは特に読みにくい疑似コードを生成し、平均で6.2%の編集類似度です。興味深いことに、GPTによる洗練（Ghidra+GPT-4o）では、編集類似度が若干減少します。GPTは「undefined4」や「ulong」などの型エラーの洗練を支援します（図4）。しかし、「for」ループと配列インデックスを正確に再構築するのに苦労しています。対照的に、LLM4Decompile-EndとLLM4Decompile-Refの両方は、ソースコードの形式に合った、より理解しやすい出力を生成します。要約すると、デコンパイル出力の再実行可能性と読みやすさを向上させるためには、ドメイン固有の微調整が重要です。

さらに、読みやすさを評価するためにGPT-4oを採用しました。具体的には、GPTに構造化されたテンプレートを使用して、構文の類似性（変数、ループ、条件）と構造的整合性（ロジックフロー、構造）を評価するよう指示しました。その後、元のコードとデコンパイルされたコードの詳細な比較に基づいて、1（不良）から5（優れている）までのスコアで読みやすさをまとめました。テンプレートは我々のGitHubリポジトリで利用可能です。表4は、様々なモデルと最適化レベルにわたるHumanEval-Decompileの読みやすさ評価をまとめています。

最適化レベル	O0	O1	O2	O3	平均
GPT-4o	2.8171	2.3537	2.2927	2.311	2.4436
Ghidra	2.9756	2.4085	2.5183	2.3841	2.5716
LLM4Decompile-End-6.7B	4.0732	3.4634	3.4024	3.2378	3.5442

表4: 様々な手法からデコンパイルされた結果の読みやすさに関するGPT-4oによる評価

表3の結果と比較すると、編集類似度（ES）がGPT評価と同様の傾向を示していることがわかります。ESは数学的に基づいていますが、その値は解釈が難しいことがあります。例えば、LLM4Decompileモデルによって得られた15のESスコアは低く見えるかもしれませんが、デコンパイルされた関数とソースコードは高度に一致しています。対照的に、読みやすさを概念的に測定するGPT評価はより直感的です。GPTスケールで4のスコアは、デコンパイルされたコードがオリジナルとほぼ同一であることを示唆しています。それにもかかわらず、これらのスコアはGPTの「主観的な」判断から導き出されたものです。ESとGPT-Evalの両方からの洞察を組み合わせることで、コードの読みやすさのより徹底的な評価につながる可能性があります。

5. 難読化に関する議論

デコンパイルプロセスは、開発者によって配布されるバイナリからソースコードを明らかにすることを目的としており、知的財産の保護に対する潜在的な脅威となります。倫理的懸念を解決するために、このセクションでは我々のデコンパイルモデルの誤用の可能性に関するリスクにアクセスします。

ソフトウェア開発では、エンジニアは通常、バイナリファイルを一般に公開する前に難読化技術を実装します。これは、ソフトウェアを不正な分析や改変から保護するために行われます。我々の研究では、Obfuscator-LLVMで提案されている2つの基本的な難読化技術に焦点を当てています：制御フロー平坦化（CFF）とボーガス制御フロー（BCF）です。これらの技術は、ソフトウェアの真のロジックを偽装するように設計されており、それによりソフトウェアの知的財産を保護するためにデコンパイルをより困難にします。これら2つの技術の詳細は付録11で紹介しています。

モデル/難読化	制御フロー平坦化					ボーガス制御フロー				
最適化レベル	O0	O1	O2	O3	平均	O0	O1	O2	O3	平均
LLM4Decompile-End-6.7B	4.27	4.88	4.88	3.05	4.27	9.76	7.32	7.93	9.76	8.69
Ghidra	12.20	6.71	6.10	6.71	7.93	6.10	4.27	3.05	4.27	4.42
+LLM4Decompile-Ref-6.7B	6.71	3.66	4.88	5.49	5.19	15.85	14.02	8.54	7.93	11.59

表5: 難読化技術に対する各モデルの性能比較

表5にまとめられた結果は、基本的な従来の難読化技術が、GhidraとLLM4Decompileの両方が難読化されたバイナリをデコードするのを防ぐのに十分であることを示しています。例えば、最も高度なモデルであるLLM4Decompile-Ref-6.7Bの成功率は、CFF下で90.2%（0.5274から0.0519へ）、BCF下で78.0%（0.5274から0.1159へ）大幅に低下します。ソフトウェアリリース前に複数の複雑な難読化方法を採用する業界標準を考慮すると、表5の実験結果は、知的財産権侵害のための不正使用に関する懸念を軽減します。

6. 結論

我々はLLM4Decompileを提案しました。これはバイナリコードをデコンパイルするためにトレーニングされた、1.3Bから33Bのサイズ範囲を持つ最初で最大のオープンソースLLMシリーズです。*End2end-Decompile*アプローチに基づき、LLMトレーニングプロセスを最適化し、バイナリを直接デコンパイルするLLM4Decompile-Endモデルを導入しました。その結果、6.7BモデルはHumanEvalで45.4%、ExeBenchで18.0%のデコンパイル精度を示し、GhidraやGPT-4oなどの既存ツールを100%以上上回っています。さらに、*Refined-Decompile*戦略を改善してLLM4Decompile-Refモデルを微調整し、これらはGhidraの出力を洗練させることに優れており、LLM4Decompile-Endに対して16.2%の改善を達成しています。最後に、難読化実験を実施し、知的財産権侵害のためのLLM4Decompileモデルの誤用に関する懸念に対処しました。

制限

この研究の範囲はx86プラットフォームを対象とするC言語のコンパイルとデコンパイルに限定されています。ここで開発された方法論は他のプログラミング言語やプラットフォームにも容易に適応できると確信していますが、これらの潜在的な拡張は将来の調査のために留保されています。さらに、我々

の研究は財政的制約によって制限されており、1年間8×A100 GPUを使用することに相当する予算で、すべての試行と反復が含まれています。その結果、6.7Bまでのモデルを完全に微調整し、小さなデータセットで33Bモデルに関する初期の探索を実施することしかできず、70Bおよびより大きなモデルの探索は将来の研究に残されています。それにもかかわらず、我々の予備テストはモデルサイズのスケールアップの潜在的な利点を確認し、より大きなモデルへの将来のデコンパイル研究のための有望な方向性を示唆しています。

倫理声明

セクション5で我々のデコンパイルモデルの誤用の可能性に関するリスクを評価しました。制御フロー平坦化やボーガス制御フローなどの基本的な難読化方法は、GhidraのようなTraditionalなツールやLLM4Decompileのような高度なモデルの両方による不正なデコンパイルから保護することが経験的にテストされ、証明されています。この組み込みの制限により、LLM4Decompileは正当な使用のための強力なツールである一方で、知的財産権の侵害を容易にしないことが保証されます。

産業界の実際の応用では、ソフトウェア開発者は通常、ソフトウェアをリリースする前に一連の複雑な難読化方法を採用します。この慣行はデコンパイルに対するセキュリティと知的財産保護の追加層を提供します。LLM4Decompileの設計と意図された使用はこれらの対策を尊重し、レガシーコードの理解やサイバーセキュリティ防御の強化など、法的かつ倫理的なシナリオでの支援として機能し、それらを損なわないことを保証します。

LLM4Decompileの開発と展開は厳格な倫理基準によって導かれています。このモデルは主に、許可が与えられている場合や、ソフトウェアが著作権によって保護されていない場合のシナリオでの使用を意図しています。これには、学術研究、デバッグ、学習、および企業が自社のソフトウェアの失われたソースコードを回復しようとする状況が含まれます。

謝辞

この研究は、中国国家自然科学基金（No. 62372220）、香港特別行政区研究助成委員会（プロジェクトNo. PolyU/25200821）、中国NSFC若手科学者基金（プロジェクトNo. 62006203）、イノベーションおよび技術基金（プロジェクトNo. PRP/047/22FX）、およびRC-DSAIからのPolyU内部基金（プロジェクトNo. 1-CE1E）によって部分的に支援されています。

参考文献

- 01-AI. 2024. Yi-coder.
- Jordi Armengol-Estape, Jackson Woodruff, Alexander Brauckmann, Jose Wesley de Souza Magalhaes, and Michael F. P. O'Boyle. 2022. Exebench: An ml-scale dataset of executable c functions. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, MAPS 2022, page 50?59, New York, NY, USA. Association for Computing Machinery.
- Jordi Armengol-Estape, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. 2023. Slade: A portable small language model decompiler for optimized assembler. CoRR, abs/2305.12520.
- Andrei Z Broder. 2000. Identifying and filtering nearduplicate documents. In Annual symposium on combinatorial pattern matching, pages 1?10. Springer.
- David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013, pages 353?368. USENIX Association.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. CoRR, abs/2107.03374.
- Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. arXiv preprint arXiv:2407.02524.
- Anderson Faustino da Silva, Bruno Conde Kind, Jose Wesley de Souza Magalhaes, Jeronimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quintao Pereira. 2021. ANGHABENCH: A suite with one million compilable C benchmarks for code-size reduction. In IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021, pages 378?390. IEEE.
- Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1497?1511.
- Inc. Free Software Foundation. 2024. The gnu c library. Ghidra. 2024a. Ghidra software reverse engineering framework.
- Ghidra. 2024b. Headless analyzer readme.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming?the rise of code intelligence. arXiv preprint arXiv:2401.14196.

- Hex-Rays. 2024. Ida pro: a cross-platform multiprocessor disassembler and debugger.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. 2024. Training compute-optimal large language models. In Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.
- Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the C: retargetable decompilation using neural machine translation. CoRR, abs/2212.08950.
- Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. Degpt: Optimizing decompiler output with llm. In Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/- CorpusID, volume 267622140.
- Huaqingweiyang. 2024. Machine language model.
- Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova+: Generative language models for binaries. CoRR, abs/2311.13721.
- Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM ? software protection for the masses. In Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, pages 3?9. IEEE.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. Preprint, arXiv:2001.08361.
- Deborah S. Katz, Jason Ruchti, and Eric M. Schulte. 2018. Using recurrent neural networks for decompilation. In 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, pages 346? 356. IEEE Computer Society.
- Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards neural decompilation. ArXiv, abs/1905.08325.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles.
- Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. Advances in Neural Information Processing Systems, 34:14967?14979.
- Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pages 628?639. IEEE.
- Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004., pages 75?86. IEEE.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020a. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871?7880, Online. Association for Computational Linguistics.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktaschel, et al. 2020b. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems, 33:9459?9474.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Joao Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Munoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! Preprint, arXiv:2305.06161.
- Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 2511?2522, Singapore. Association for Computational Linguistics.
- Zhibo Liu and Shuai Wang. 2020a. 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, ISSTA 2020, page 475?487, New York, NY, USA. Association for Computing Machinery.
- Zhibo Liu and Shuai Wang. 2020b. How far we have come: testing decompilation correctness of C decompilers. In ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, pages 475?487. ACM.
- Jerome Miecznikowski and Laurie J. Hendren. 2002. Decompiling java bytecode: Problems, traps and pitfalls. In International Conference on Compiler Construction.
- Mistral-AI. 2024. Codestral: Empowering developers and democratising coding with mistral ai.
- Steven S. Muchnick. 1997. Advanced compiler design and implementation.
- Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT : A transformer-based model for decompiled identifier renaming. In Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pages 48?57, Online. Association for

Computational Linguistics.
- Godfrey Nolan. 2012. Decompiling android. In Apress.
- OpenAI. 2023. GPT-4 technical report. CoRR, abs/2303.08774.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jeremy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. CoRR, abs/2308.12950.
- Richard M Stallman et al. 2003. Using the gnu compiler collection. Free Software Foundation, 4(02).
- Jiaan Wang, Yunlong Liang, Fandong Meng, Zengkui Sun, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023. Is ChatGPT a good NLG evaluator? a preliminary study. In Proceedings of the 4th New Frontiers in Summarization Workshop, pages 1?11, Singapore. Association for Computational Linguistics.
- Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In NDSS.
- Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A new algorithm for identifying loops in decompilation. In Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings, volume 4634 of Lecture Notes in Computer Science, pages 170?183. Springer.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. CoRR, abs/1910.03771.
- Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. Refining decompiled C code with large language models. CoRR, abs/2310.06530.
- Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2023. Lmpa: Improving decompilation by synergy of large language model and program analysis. CoRR, abs/2306.02546.
- Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2024. Leveraging generative models to recover variable names from stripped binary. Preprint, arXiv:2306.02546.
- Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. arXiv preprint arXiv:2403.13372.

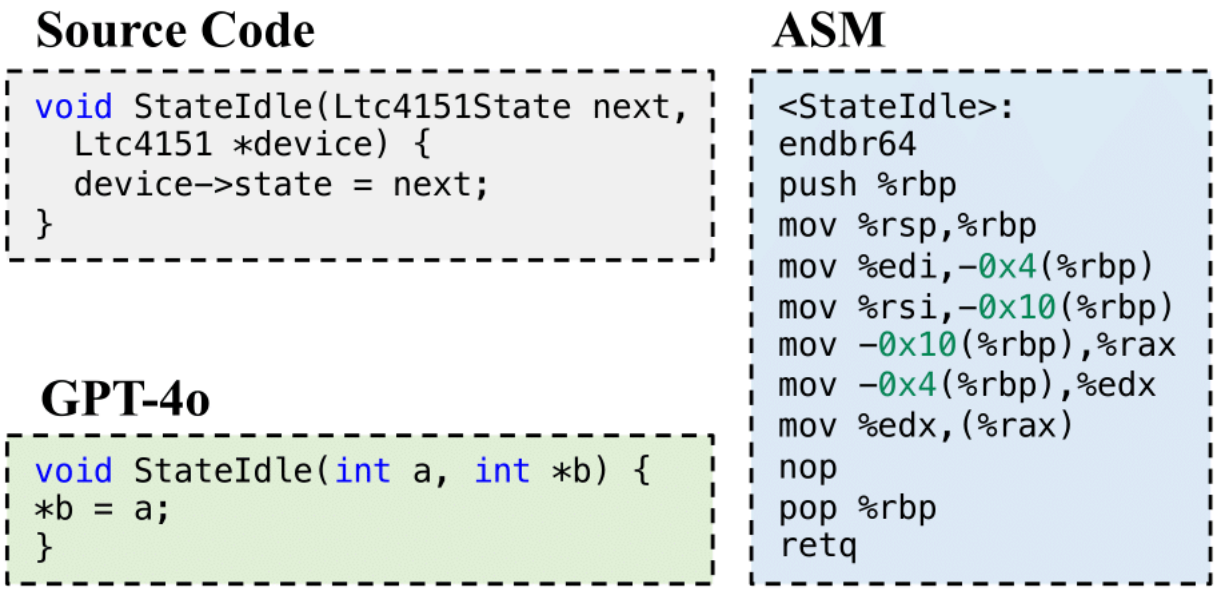
A. ExeBenchのセットアップ

ExeBenchの実行可能分割の各サンプルについて、*.sファイル（セクション3.1と図1で議論されているコンパイラの中間出力）からのアセンブリコードは、サンプルをバイナリにコンパイルするために必要です。しかし、具体的なコンパイル設定と処理の詳細は著者から提供されていません。そのため、標準的な方法でコードをコンパイルすることを選択し、サンプルの半分だけをコンパイルすることができました。これにより、実行可能トレーニングセット用に797Kのうち443Kのサンプル、実行可能テストセット用に5000のうち2621のサンプルが残りました。したがって、我々は443Kのサンプルでモデルをトレーニングし、これらの2621のサンプルで再実行可能性評価を実施しました。結果は表[table:main_results]に示されています。

モデル/メトリクス	**再実行可能性**		**編集類似度**	
最適化レベル	O0	O3	O0	O3
Slade	59.5	52.2	71.0	60.0
ChatGPT	22.2	13.6	44.0	34.0
GPT-4o(ours)	4.4	3.4	7.9	6.6

表6: ExeBenchにおける再実行可能性と編集類似度

SladeとExeBenchを開発した研究者たちは、ExeBenchに関するデコンパイルの研究結果を発表しています。彼らは中間出力、つまり*.sファイルからのアセンブリコードを、バイナリにさらにコンパイルすることなく直接デコンパイルすることを選択しました。実際には、このような中間出力はソフトウェア開発者によって公開されることはほとんどありません。表6に示されている彼らの報告結果は、我々のものと大きく異なります。彼らのバージョンのChatGPTはO0最適化下で22.2%の再実行可能率と44.0%の編集類似度を達成しました。一方、我々のGPT-4oモデルは4.4%の再実行可能率と7.9%の編集類似度に留まりました。Sladeが採用したアプローチは、実用的なデコンパイルシナリオでは一般的に利用できない設定を含んでおり、これが彼らの結果が我々のものと大きく異なる理由を説明しています。我々はより現実的な設定を守り、外部情報なしで、その内在的なデータのみに基づいてバイナリファイルをデコンパイルします。



我々の設定をさらに説明するために、図5はソース関数が「Ltc4151State」、「Ltc4151」、「device」などの特定のユーザー定義型を含む例を提供しています。しかし、これらの型はコンパイル後に完全に失われます。つまり、これらのユーザー定義に関連する情報はバイナリ（逆アセンブルされたASMコード）には見つかりません。その結果、GPT-4oはASMのみに基づいてこれらの型を再構築することができず（現実的な設定）、それらをデフォルトの型「int」または「pointer」に変換し、実行不可能なコードを生成します。この問題はExeBenchテストセット全体に広がっており、現実的な設定でExeBenchサンプルをデコンパイルするGPT-4oモデルの失敗につながりました。

B. コンパイル可能バイナリと実行可能バイナリ

トレーニングおよびテストデータセットの統計は表7にまとめられています。また、これら2つのデータセットの違いを説明する例も提示します。

データセット/コード	ASM	SRC
Train-Executable	205.08	119.35
Test-Exebench	280.27	162.68
Train-Compilable	711.89	241.16
Test-Decompile-Eval	808.07	186.84

表7: トレーニングおよびテストセットの統計情報

Object file (only compile)

```
1     endbr64
2     push   %rbp
3     mov    %rsp,%rbp
4     mov    %rdi,-0x18(%rbp)
5     mov    %esi,-0x1c(%rbp)
6     movss  %xmm0,-0x20(%rbp)
7     movl   $0x0,-0x8(%rbp)
8     jmp    88 <func0+0x88>
9     mov    -0x8(%rbp),%eax
10    add    $0x1,%eax
11    mov    %eax,-0x4(%rbp)
12    jmp    7c <func0+0x7c>
13    mov    -0x8(%rbp),%eax
14    cltq
```

Executable file (linked)

```
1     endbr64
2     push   %rbp
3     mov    %rsp,%rbp
4     mov    %rdi,-0x18(%rbp)
5     mov    %esi,-0x1c(%rbp)
6     movss  %xmm0,-0x20(%rbp)
7     movl   $0x0,-0x8(%rbp)
8     jmp    11f1 <func0+0x88>
9     mov    -0x8(%rbp),%eax
10    add    $0x1,%eax
11    mov    %eax,-0x4(%rbp)
12    jmp    11e5 <func0+0x7c>
13    mov    -0x8(%rbp),%eax
14    cltq
```

図6に示されているように、コンパイル可能バイナリと実行可能バイナリの主な違いは、関数操作アドレスの処理です。コンパイル可能ファイルでは、ジャンプ操作のアドレスはプレースホルダーであり、関数内の相対オフセットのみを表しています。対照的に、実行可能ファイルでは、このジャンプ操作アドレスはリンクプロセス中に解決され、コードが実行される特定のメモリ位置を直接指します。

C. LLM4Decompile-Endのさらなる分析

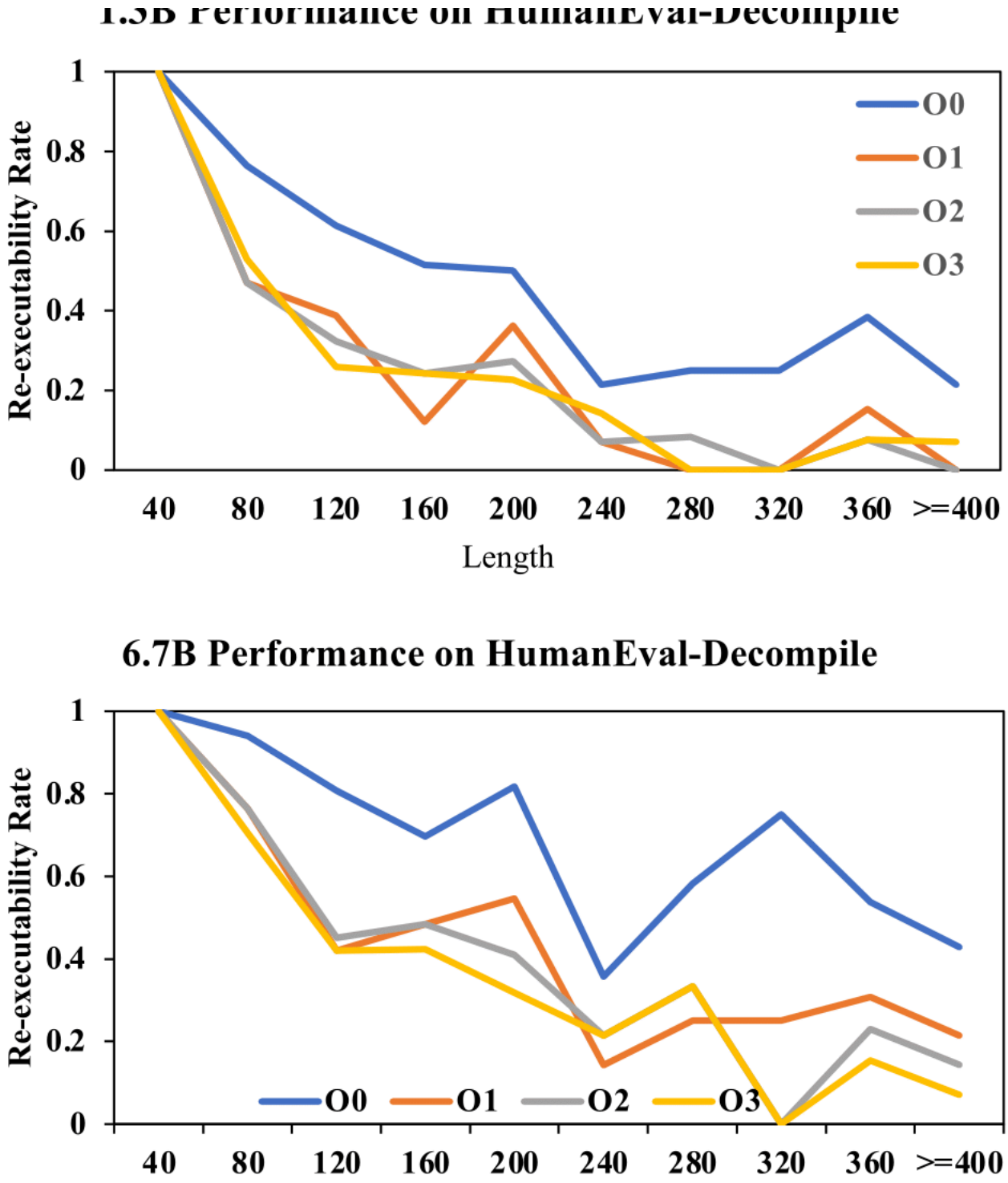


図7は、入力長が増加するにつれて再実行可能率が低下し、コードの最適化レベルが高いほどパフォーマンスが著しく低下することを示しており、長く高度に最適化されたシーケンスをデコンパイルする難しさを強調しています。重要なのは、図に示されている1.3Bモデルと6.7Bモデル間のパフォーマンスの違いが、このようなタスクにおける大きなモデルの利点を強調していることです。拡張された計算リソースとより深い学習能力を持つ大きなモデルは、複雑なデコンパイルが提示する課題を解決するのに本質的に優れています。

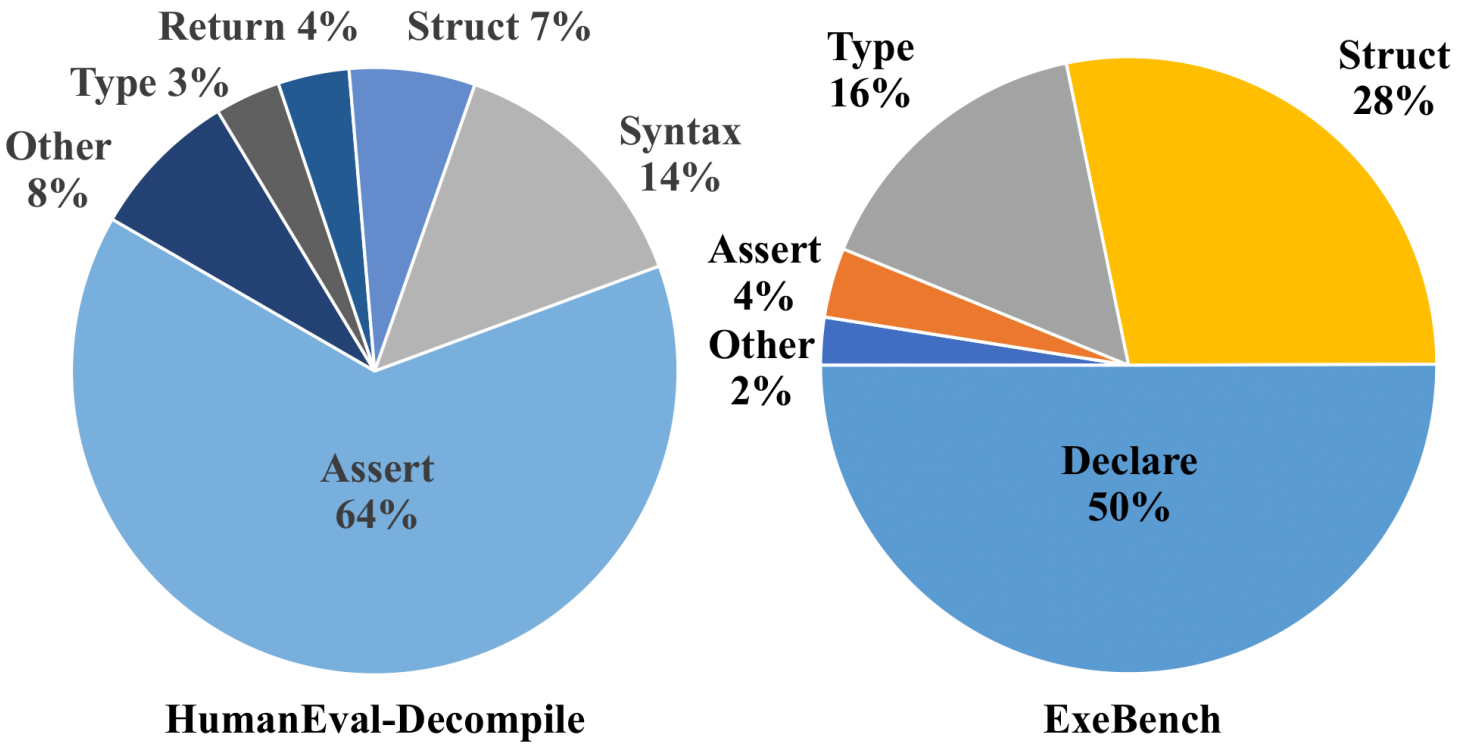


図8に示されているLLM4Decompile-End-6.7Bのエラー分析は、HumanEval-Decompileシナリオでは論理エラーが一般的であり、エラーの64%はデコンパイルされたコードが合格しないアサーションによるものであることを示しています。ユーザー定義の構造体と型を特徴とする実際の関数を含むExeBenchデータセットでは、主な課題はこれらのユーザー固有のコンポーネントの取り戻しに関連しています。エラーの50%は未宣言関数から、28%は構造体の不適切な使用から来ています。これらのユーザー定義の詳細は通常コンパイルプロセスで失われるため、それらを再構築することは特に困難です。検索拡張生成（RAG）のような技術を統合することで、必要な外部情報でデコンパイルプロセスを補完できる可能性があります。

D. データ品質、量、およびモデル

データ品質

このプロジェクトでは、短いテキストのフィルタリングや重複の削除などの古典的な技術にデータ前処理を意図的に制限しました。このアプローチは、潜在的なバイアスを最小限に抑えるデコンパイルのための公正なベースラインモデルを確立するために選択され、多様なシナリオを反映する広範で未洗練のベースラインモデルを提供することを目指しています。標準Cライブラリと互換性のないデータを除外するなどの選択的なデータ削除が、パフォーマンスを向上させることができることを認識しています。これは標準Cライブラリのみに依存するDecompile-Evalを用いた表8で証明されています。データセットの洗練がパフォーマンスの向上につながる可能性がありますが、この研究における我々の主な目標はコミュニティのための基本的なベースラインを設定することでした。このベースラインは、将来の研究が洗練し拡張することを奨励するための出発点として機能すると信じています。

モデル	再実行可能率				
最適化レベル	O0	O1	O2	O3	平均
Compilable-6.7B	51.83	33.54	32.32	32.32	37.50
+Executable (2B tokens)	68.05	39.51	36.71	37.20	45.37
+Exe w. Standard C (100M tokens)	71.80	42.68	41.31	41.46	49.31

表8: トレーニングデータの特性と再実行可能率の関係

※トレーニングデータ（+Exe w. Standard C）がテストセットのパターンに近い場合、パフォーマンスが向上します。

データ量

表9は、LLM4Decompile-Ref-1.3Bモデルのパフォーマンスとトレーニングエポック（1エポックで20億トークン）の関係をまとめています。これから、単一のエポックが強力なベースラインとして機能し、2つのエポックがパフォーマンスを最適化することが明らかです。追加のエポックは過学習と結果の低下につながる傾向があります。

エポック	O0	O1	O2	O3	平均
1	67.68	41.46	41.46	35.37	46.49
2	68.29	40.85	40.85	37.20	46.80
3	62.80	37.80	36.59	29.88	41.77
4	51.22	32.93	27.44	26.22	34.45

表9: 異なるトレーニングエポックに対するLLM4Decompile-Ref-1.3BのHumanEval-Decompileでのパフォーマンス

さらに、スケーリングの問題に対処するために6.7Bモデルの結果を含め、表10で1.3Bおよび33Bモデルとの比較を提供しました。特に、6.7Bモデルはデータの20%だけで1.3Bモデルと同等のパフォーマンスを達成し、33Bモデルはデータのわずか10%で6.7Bと同様の結果に達していますが、これらの比率はデータセットによって異なる可能性があります。

サイズ	エポック	O0	O1	O2	O3	平均
1.3B	1.0	67.68	41.46	41.46	35.37	46.49
6.7B	0.05	55.49	31.71	34.76	30.49	38.11
6.7B	0.1	57.93	36.74	32.77	32.01	39.86
6.7B	0.2	65.85	37.80	40.24	34.76	44.66
6.7B	0.5	65.55	45.73	43.29	43.75	49.58
6.7B	1.0	72.56	45.73	43.90	42.68	51.22
33B	0.1	70.73	47.56	43.90	41.46	50.91

表10: 異なるサイズとトレーニングエポックに対するLLM4Decompile-RefモデルのHumanEval-Decompileでのパフォーマンス

我々の発見は、モデルのスケーリングがトレーニングデータが十分に大きい（1億トークン）場合にパフォーマンスを大幅に向上させることができることを示唆していますが、繰り返しのトレーニングは数エポック後に過学習のリスクがあります。

モデル

デコンパイルトレーニングのために適切なベースモデルを選択することはパフォーマンスに大きな影響を与えます。我々の最初の選択であるDeepseek-Coder-6.7Bは、HumanEval-Decompileベンチマークで平均52.74%の再実行可能率という励みになる結果を提供しました。対照的に、ソースコードをLLVM IRにコンパイルするためにトレーニングされたLLM-Compiler-7B（デコンパイルの逆）は、より効果的な基盤として機能し、Deepseek-Coder-6.7Bと比較してパフォーマンスを3.5%向上させました。さらに、2024年9月に導入された現在の最先端モデルであるYi-Coder-9Bは、デコンパイルトレーニングの結果を23.1%著しく向上させました。さらに、より大きなアーキテクチャの恩恵を受けるCodeStral-22Bは、小さなモデルに比べて21.7%の改善を提供しました。

モデル/最適化レベル	O0	O1	O2	O3	平均
DeepSeek-Coder-6.7B	74.39	46.95	47.56	42.07	52.74
LLM-Compiler-7B	72.56	51.83	48.78	45.12	54.57
Yi-Coder-9B	79.27	**62.20**	**61.59**	56.71	**64.94**
CodeStral-22B	**80.49**	58.54	59.76	**57.93**	64.18

表11: LLM4Decompile-Refシリーズのベースモデルにおける再実行可能率の比較

E. 難読化技術

Obfuscator-LLVMで提案されている2つの古典的な難読化技術の詳細を提供します。

制御フロー平坦化

これはソフトウェアの単純で階層的な制御フローをより複雑で平坦化された構造に変換することで、セキュリティを強化します。ワークフローには、関数を基本ブロックに分割し、これらのブロックを同じレベルに配置し、ループ内のswitch文でカプセル化することが含まれます。

ボーガス制御フロー

これは既存のブロックの前に追加の基本ブロックを挿入することで、関数の実行シーケンスを変更します。この追加されたブロックには不透明な述語が含まれ、それに続いて元のブロックに戻る条件付きジャンプがあります。さらに、元の基本ブロックはランダムに選択された無意味な命令で汚染されています。