

LLMを用いたDecompile

発表者: 福井 遥陽

背景 | デコンパイルとは

バイナリ形式のコードから高級言語(C,Python,java)を得る手段.

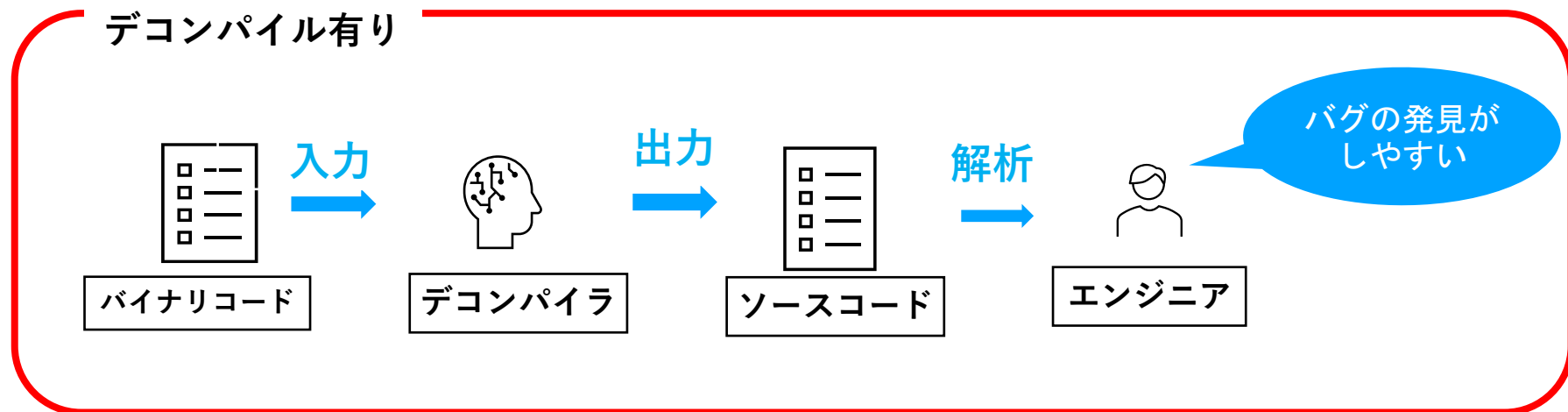
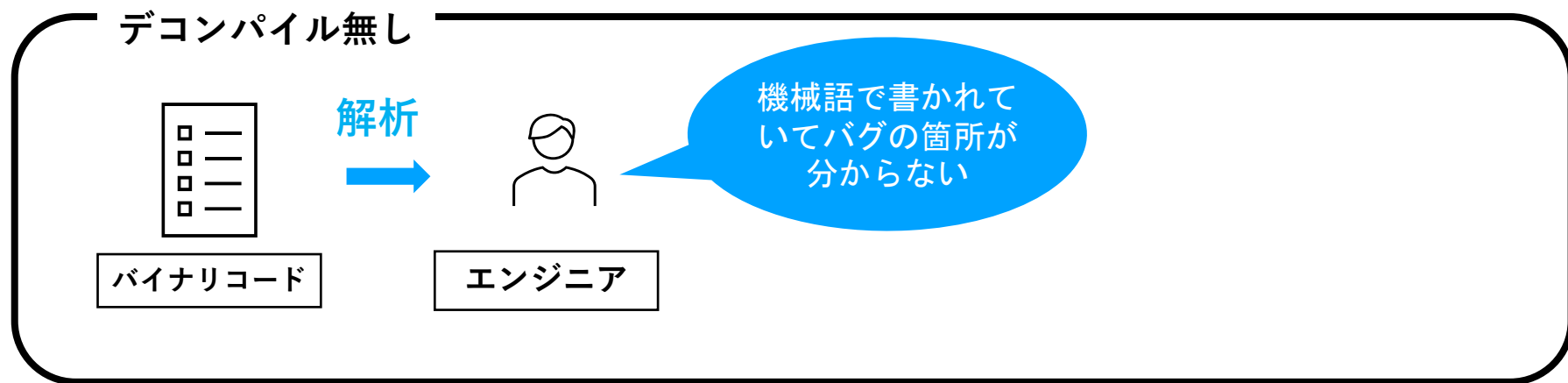


実際にソフトウェアの解析やマルウェアの解析を行う際、元のソースコードが得られない場合がある.

.exeファイルはバイナリ形式

背景 | デコンパイルの目的

主にマルウェア解析の手段として使われている。



デコンパイルの存在はマルウェアの解析に**必要不可欠**

既存手法

現状のデコンパイル手法では以下の手法が使われている。

例) Ghidra

アメリカ国家安全保障国によって開発されたオープンソースのデコンパイル専用のツール

入力としてバイナリコードを受け取り，ソースコードを出力。

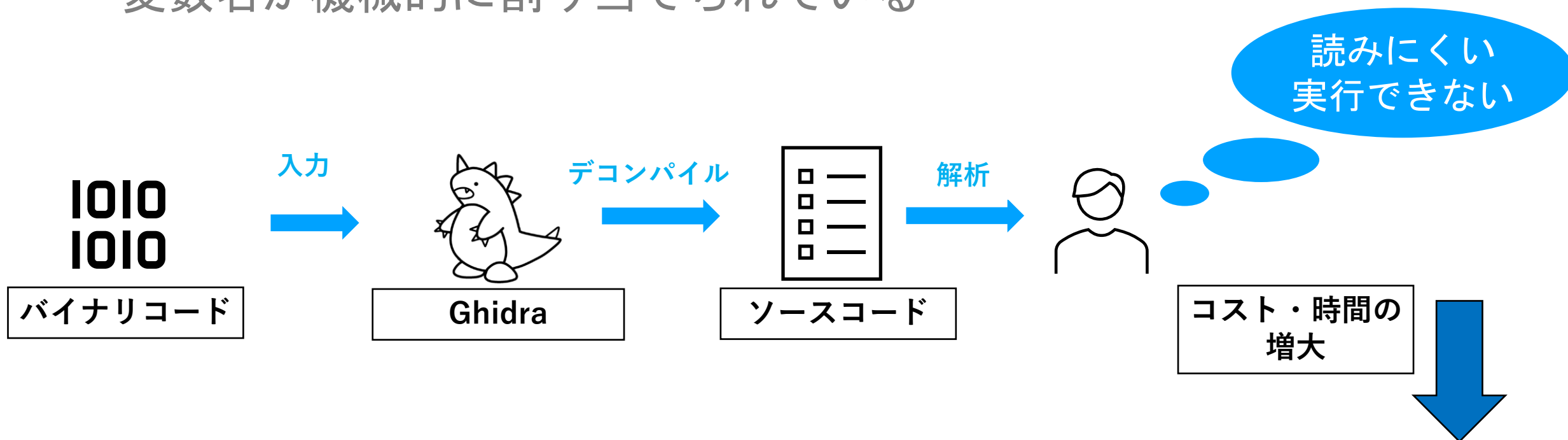
- Ghidraのフレームワーク

1. バイナリファイルをGhidraに読み込ませる
2. バイナリに記述される命令を独自の命令セットで中間表現(P-code)に変換
3. P-codeから制御フローグラフやデータフロー解析を行い，高水準コードに変換。
4. 冗長な命令の削除，ループの構造化
5. 構造化された命令列をCソースコードに変換

課題

現状のデコンパイルで取得できるソースコードは実行不可能で可読性も低い

- アセンブリコードの表現であるGOTO文を含む
- 変数名が機械的に割り当てられている



先行研究

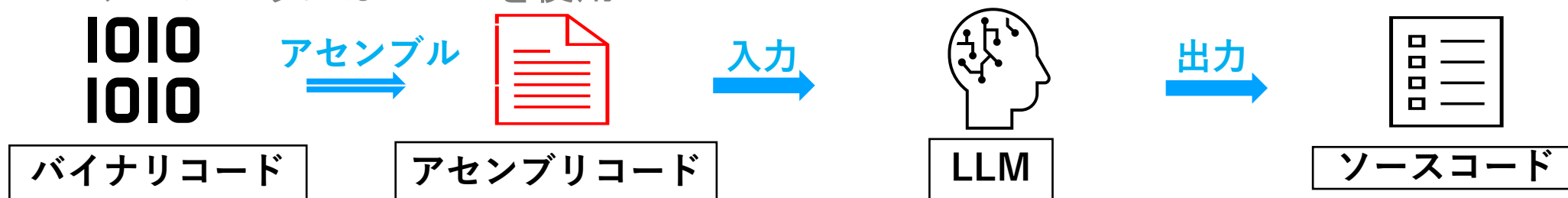
先行研究ではデコンパイルの課題点に対して以下の二つの手法を提案している.

- LLM4Decompile-Endモデル : バイナリコードを入力として直接LLMでデコンパイルする手法
 - LLM4Decompile-Refモデル : バイナリコードをツールを使ってデコンパイルし, 得たコードをLLMで改良・整形する手法
-
- これらのモデルは初めて最大規模 (1.3B~33B) でLLMを使用したオープンソースLLMである.
 - 両方の手法に使用されるLLMはそれぞれ別々のファインチューニングが行われている.

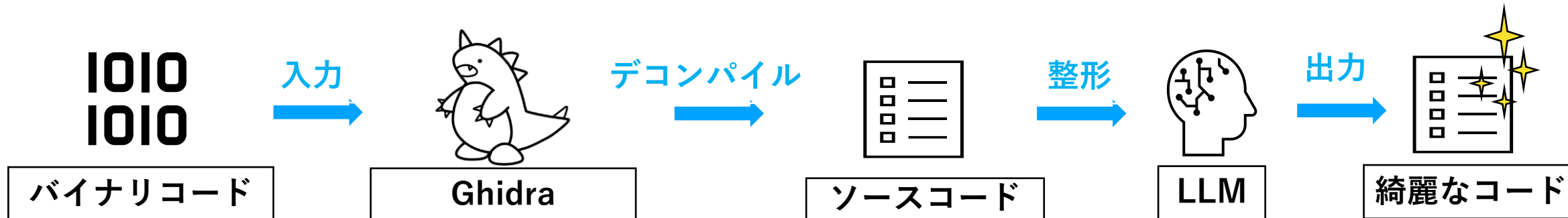
先行研究

LLM4Decompile-End

* アセンブラにはGhidraを使用



LLM4Decompile-Ref



先行研究の実験結果

実験結果

- ・ 1.3Bより6.7Bモデルの方が精度が高い。(33Bモデルは訓練データ不足により精度が落ちた)

→モデルのサイズが大きいほど精度が高まる。

- ・ また、既存手法(Ghidra, GPT-4o)よりも精度が高かった。

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
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
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54	18.86	14.65	13.96	14.11	15.40

研究目的

先行研究に対して再実行可能性を向上させる.

- ・再実行可能性はデコンパイルによって取得したコードがどれだけ元のソースコードと遜色ないものかを表す唯一の指標

先行研究で再実行可能性が50%近くに留まった理由が不透明

→原因を特定・改善することで実行可能性を向上させることが出来るのでは？

アプローチ

先行研究で判明していない原因を特定

➤ データの収集

1-LLM4Decompile-EndモデルとRefモデルの両方を使用し、それぞれのデコンパイルで得た疑似ソースコードを収集.

* 環境はGoogle Colabを使用(先行研究ではLinuxを使用)

2-収集したコードの中で、実行時エラーを起こしたコード（暫定）を選定.

3-選定したコードからASM-Sourceペアを収集.

- ・ データセットには、Human-EvalとMBPPを使用.

→前者は単体関数で構成される小規模なソースコード. 後者は、実際のソフトウェアなどにもある複数の関数で構成された中規模の関数（GitHubから収集）

- ・ 先行研究で使用されている大規模なC関数のデータは仕様が困難（一部公開）.

アプローチ

| 先行研究で判明していない原因を特定

➤ 結果分析・分類

- 1-収集したコード一つ一つに含まれるエラー原因を手動で判別.
- 2-エラー原因毎にラベル付けを行い, 含まれるエラー原因の数が多い順に整理

- ・ サンプル数は暫定100個用意. (有意水準を満たすかどうかで数は前後)
- ・ モデル毎・データセット毎・最適化レベル毎・言語毎で分割するため, サンプルの収集には少なくない時間がかかることが予想される.
(大体50個のファイル(単体関数)を逆コンパイルするのにかかる時間が20~30分)

アプローチ

分類する上でのルール

- 直接的なコンパイルエラーを起こさないもの(潜在的なエラー)は省く(使用メモリの領域の安全性、未使用の変数、アセンブリ表現など)
- コンパイルエラー情報を目視で確認+chatgpt-4oでコードとエラー情報を提供して、エラー情報を分類
- 一つのコードに複数の未定義関数の使用などが見られた場合、一つ一つカウントする。(尚、未定義関数func1が一つのコード内に複数回出現した場合は、一回のエラーとしてカウントする)
- ロジックの崩壊としてカウントするのは、無意味な変数の宣言の繰り返し・if文の繰り返しなど、明らかに冗長な処理文が見られた場合のみ。(人による主観)

現状

実際に収集して発見できたエラー原因

```
float mean(float *data, int n)
{
    int i;
    float sum = 0;
    for(i = 0; i < n; ++i) sum += data[i];
    return sum / (float)n;
}
```

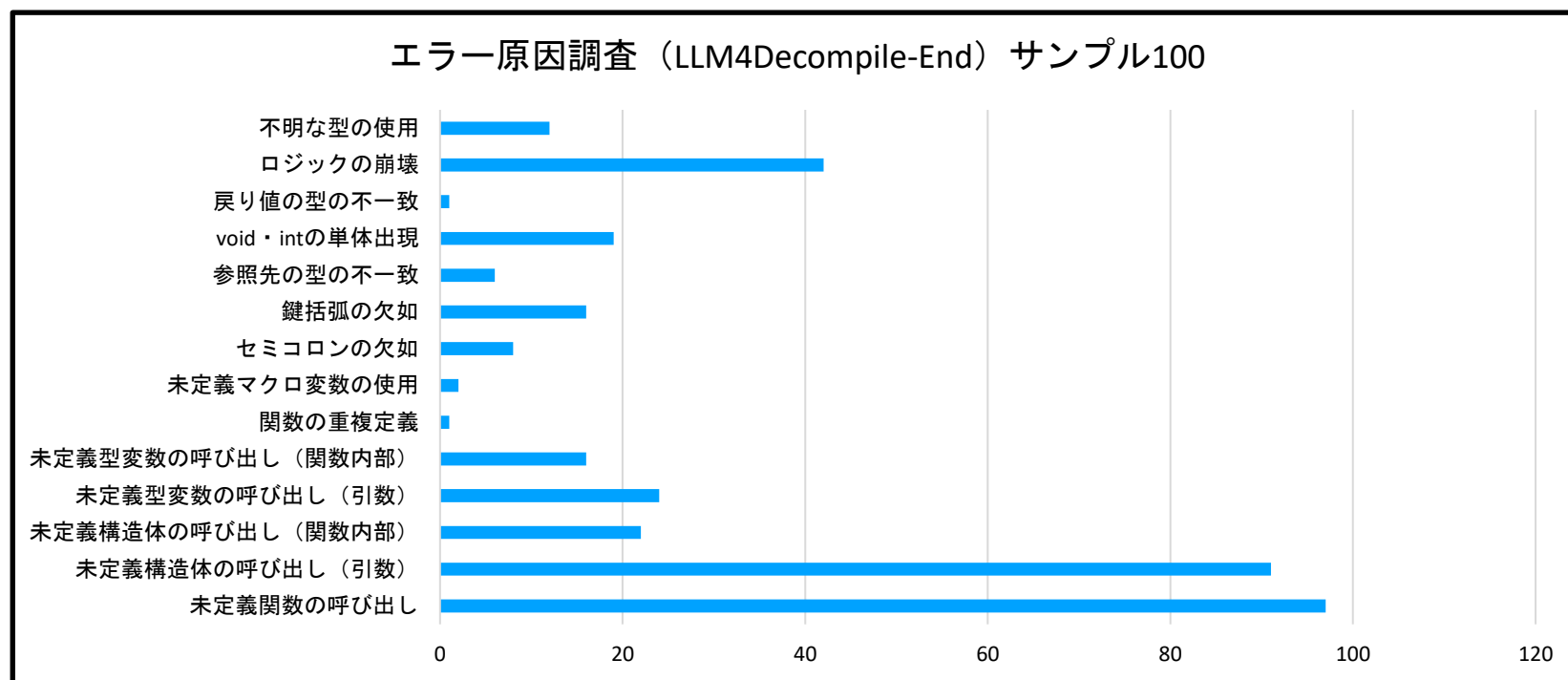
```
float
__attribute__((no))
f32_avg(f32_accumulator_t *acc)
{
    acc->sum = acc->sum + acc->last;
    acc->count++;
    if (acc->count >= acc->max_count) {
        return acc->sum / acc->max_count;
    }
    return acc->sum / acc->count;
}
```

引数の型宣言の欠
如

二つ目の関数の出
現

実験結果

RQ1 : LLM4Decompileモデルは、どのエラー原因が多く引き起こすか？



現状

今後の課題

- ・ 今回実施していない複数関数の実施も必要
- ・ 今回使用したデータに一つのバイナリコードに対して, ghidra・IDA proでそれぞれ逆アセンブリして用意したアセンブリコードがあった. それぞれのアセンブリコードでデコンパイルの精度が大きく変わる.

→新たな研究テーマ (サブ)

来週までの方針

LLM4Decompile-Refモデルでの実施

扱うソースコードはEndモデルと同様だが、エラーコードの数により余分に収集する必要あり

データセットmbppでも実施する.

再来週の進捗になる可能性あり.

目標

最低限の目標

LLM4Decompile-Endモデル・LLM4Decompile-Refモデルの両方でエラー原因の特定の実施

理想の目標

上記の目標に加えて、ベースモデルであるCode-Llama（暫定）でも実施

既存研究に対する立ち位置

- ・ 既存のRefモデルを使用した手法に対して、大規模データセットの使用・データのクレンジングや二段階訓練を行っている.
- ・ 既存のEndモデルでは小規模のLLMやニューラルネットワークしか検証されていない. それに対して、大規模なLLMを使用した初の試み.

上記の試みによって、既存の手法に対してコードの再実行可能性と可読性の観点から優位に立っている.

先行研究の実験設定

比較対象

- GPT-4o : 現時点で最も高性能なLLMの一つ
- DeepSeek-Coder : オープンソースの最新コードLLM

実験結果

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
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
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54	18.86	14.65	13.96	14.11	15.40

研究目的の合理性

再実行可能性や可読性が重要視される理由

再実行可能性はデコンパイルして取得したコードが再実行できるだけでなく、その出力結果が正しいかも判定している。

→デコンパイルの研究において唯一デコンパイルコードの精度を担保する指標。

再実行可能性の低いモデルを使用する際に起こる問題点

実務において実行不可能でも元のソフトウェアの機能から大きく離れないコードを出力した場合、時間がかかっても修正可能。

しかし、元の機能と全く別の挙動をするコードを出力されるとマルウェアの解析が十分に行えない可能性が高い。

先行研究に対するアプローチ

EndモデルがRefモデルに対してより高い再実行可能性・可読性を示したい。

訓練段階で、改良の余地がある。

先行研究のモデルはオープンソースなので改良が容易

先行研究では、NLLを使用してパラメータの更新を行うことにより、トークン単位での正確性を重視している。

→トークン単位で正解かどうかを判断している。

例) `if(x>0) | if(x>0.0)` → 損失として判断（実行時の結果は一緒）

アプローチとしてセマンティック損失の導入や、全く違う方法である強化学習によって再実行成功率を報酬とする手法

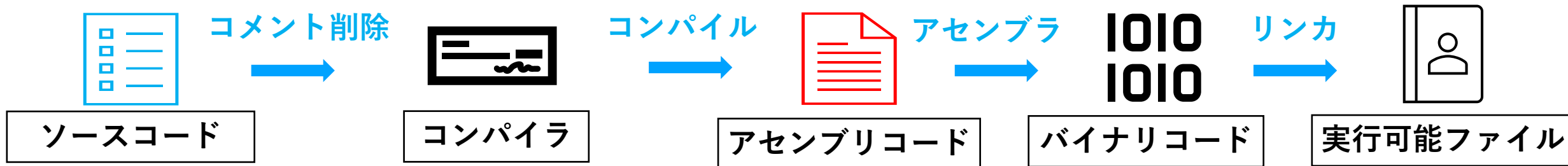
先行研究に対するアプローチ(別Ver)

- Endモデルの実行可能性が低いのはLLMがメソッド間, または関数間の関連性を把握できていないからであるかも.
- LLMに対して出力されたコード自体と“実行できない原因(エラー情報)”を投げることで実行可能性を向上させることが出来るのではないか.

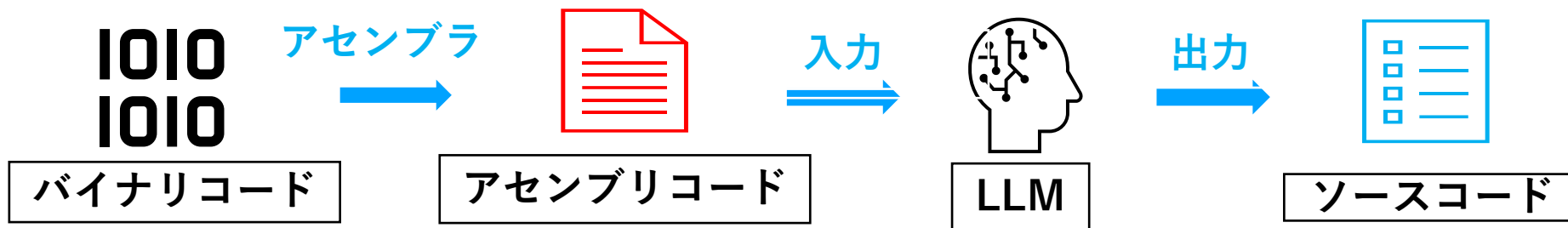
先行研究(以下補足)

LLM4Decompile-Endのフレームワークについて紹介

・コンパイル処理



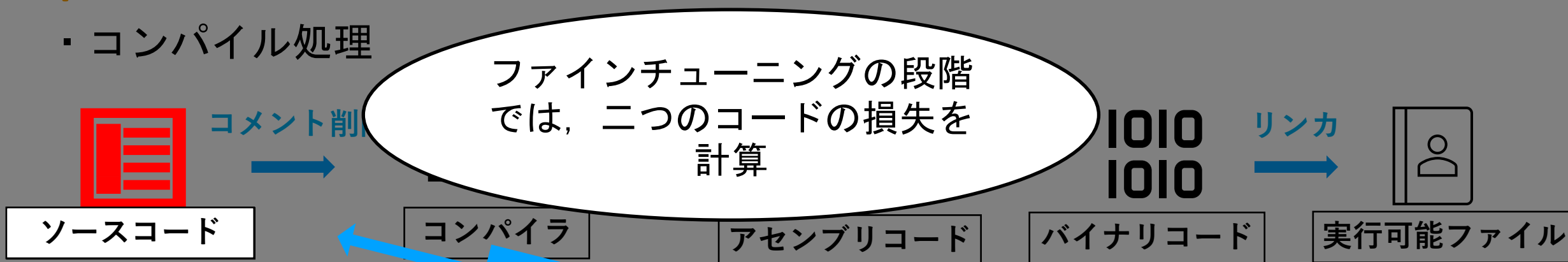
・デコンパイル処理



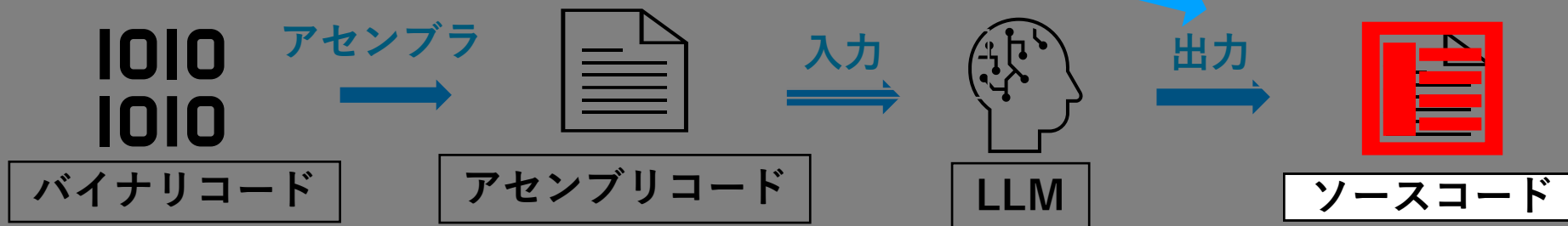
先行研究

LLM4Decompile-End フレームワーク

・コンパイル処理



・デコンパイル処理

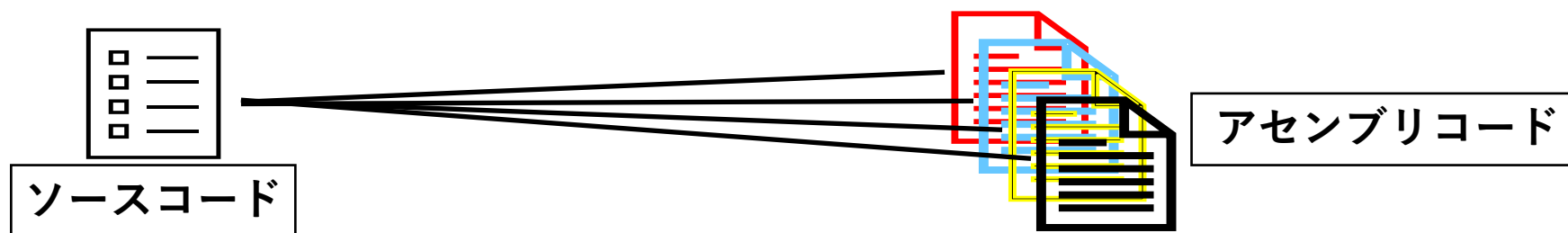


先行研究

訓練のステップ

1. 訓練データの拡張

- ・ データセットのExeBenchはC関数のソースコードを提供している.
- ・ ソースコードを最適化レベル(o0~o3)ごとにコンパイルし4つのアセンブリコードを取得
- ・ コードのASM-Sourceペアを構築



2. データ品質の向上

- ・ MinHash法とLSHを用いて類似コードを除去
- ・ 10トークン未満の短いサンプルも除外

先行研究

3.二段階訓練

非実行形式のバイナリと実行形式のバイナリで段階分け

→実行ファイルの生成には時間がかかる.

第一段階

- ・コンパイルはできるがリンク不可な非実行形式データを大量に使用.

→外部関数へのリンクが欠けているだけで, 実行形式に非常に近く, 訓練には理想的

第二段階

- ・通常の実行形式バイナリを用いてモデルを洗練
- ・応用性の向上のため, 現実的なデータでファインチューニング

先行研究の実験設定

訓練データ

ExeBenchから取得したコンパイル可能及び実行可能なC関数を基に、ASMとCコードのペアを取得

ExeBench : 実世界のCコード関数をベースとした大規模データセット
入出力例付き

- ・フィルタリング後のコンパイル可能な訓練データ : 720万関数
- ・実行可能な訓練データ : 160万関数

評価ベンチマーク

- ・HumanEval-Decompile : 標準ライブラリのみを使用した単体関数
- ・ExeBench : ユーザー定義構造体や関数が含まれたより現実的な関数

先行研究の実験設定

評価指標

再実行可能性。可読性スコアを使用

モデル構成

- ・アーキテクチャ：DeepSeek-Coderに準拠
- ・ファインチューニング方式：Seq2Seq
- ・目的は入力シーケンスに対して出力シーケンスを予測すること
- ・以下の負の対数尤度(NLL)を最小化

$$L = - \sum_i \log P_{\theta}(x_i, \dots, x_j | x_1, \dots, x_{i-1})$$

先行研究の実験設定

比較対象

- GPT-4o : 現時点で最も高性能なLLMの一つ
- DeepSeek-Coder : オープンソースの最新コードLLM

実験結果

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
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
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54	18.86	14.65	13.96	14.11	15.40

先行研究の実験結果

実験結果

- ・ 1.3Bより6.7Bモデルの方が精度が高い. (33Bモデルは訓練データ不足により精度が落ちた)

→モデルのサイズが大きいほど精度が高まる.

- ・ また, 既存手法(Ghidra, GPT-4o)よりも精度が高い.

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
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
LLM4Decompile-End-33B	51.68	25.56	24.15	24.75	31.54	18.86	14.65	13.96	14.11	15.40

先行研究の実験結果

EndモデルとRefモデルの比較

- ・ 全体的にRefモデルの方が再実行可能性は8%近く高いのに対して、テキスト類似度はほとんど値に違いがない。

→Endモデルは基礎的なロジック（制御フロー構造，メモリアクセス）を構築できていない可能性が高い。

Model/Metrics	Re-executability Rate					Edit Similarity				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37	15.57	12.92	12.93	12.69	13.53
Ghidra										
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

先行研究の実験結果

EndモデルとRefモデルの比較

- Refモデルはデコンパイルツールの出力を入力にしているため、直接アセンブリコードからコードを出力するEndモデルよりロジックを捉えられるのは明らか

→EndモデルのLLMの学習時に対して焦点を当てる。

Model/Metrics	Re-executability Rate					Edit Similarity				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37	15.57	12.92	12.93	12.69	13.53
Ghidra										
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

