

バイナリデータを処理する AI エージェントの試作

大坪 雄平^{1,2,a)} 大塚 玲²

概要：近年、急速に進展する大規模言語モデル（LLM）は、テキストを前提としたタスクで高い性能を示す一方、サイバーセキュリティ領域で頻繁に扱われる実行ファイル等のバイナリデータへの直接的な応用は未だ限定的である。本研究では、バイナリファイル解析を支援するAIエージェントのプロトタイプを作成し、その入力として「1 byte 区切りの 16 進数文字列」「Base64」等複数のテキスト表現を簡易的に検証した。評価対象は、ノート PC の APU で動作可能なサイズの Llama, Gemma, Mistral, Phi, GPT OSS の合計 5 モデルで、(1) バイナリデータの文字列複合タスク、(2) 位置指定した 4byte 値をリトルエンディアンとして抽出するタスクの 2 実験を行った。その結果、GPT OSS が全テキスト表現・両タスクで最も高い精度を示し、他のモデルではテキスト表現に応じて性能が大きく変動することを確認した。本研究は、テキスト表現の選択が LLM のバイナリ解析タスクに与える影響を定量的に示した初めての試みであり、今後の AI エージェント設計につながる知見を提供する。

キーワード：LLM, AI エージェント, バイナリ解析

Prototype of an AI agent that processes binary data

YUHEI OTSUBO^{1,2,a)} AKIRA OTSUKA²

Abstract: Recent advances in large language models (LLMs) have yielded remarkable performance on text-centric tasks, yet their direct application to binary data—such as executable files that are frequently encountered in cybersecurity—is still limited. In this study, we developed a prototype AI agent designed to aid binary-file analysis and conducted preliminary experiments with several textual encodings of the input data, including "hexadecimal strings separated by 1 byte" and "Base64". The evaluation employed five LLMs that can run on an average laptop APU: Llama, Gemini, Mistral, Phi, and GPT OSS. Two experimental tasks were performed: (1) decoding of binary data represented as a string, and (2) extraction of a offset-specified 4-byte value interpreted in little-endian order. Results show that GPT OSS achieved the highest accuracy across all textual representations for both tasks. The remaining models exhibited substantial performance variations depending on the chosen text encoding. This work represents the first quantitative assessment of how input representation affects LLMs' ability to perform binary-analysis tasks, and it provides insights that can inform future design of AI agents in cybersecurity contexts.

Keywords: LLM, AI Agent, Binary Analysis

1. はじめに

近年、人工知能（AI）の進展は目覚ましく、特に大規模言語モデル（LLM）が自然言語処理を中心とした多様なタス

クで高い性能を示すようになっている。また、外部ツールとの連携等を行う「AI エージェント」の登場により、LLM 単体では実現困難であった領域にも LLM の能力の適用が可能となっており、注目を集めている。

一方で、サイバーセキュリティ領域ではテキストデータだけでなく、実行ファイルなどのバイナリ形式も解析対象となる。一般に、LLM は UTF-8 文字列を前提に学習されているため、バイナリデータを直接扱うことは難しい。本

¹ 警察庁

National Police Agency

² 情報セキュリティ大学院大学

Institute of Information Security

a) dgs157101@iisec.ac.jp

研究では、バイナリデータを処理する AI エージェントで入力が想定されるバイナリデータのテキスト表現について、選択するテキスト表現による LLM の処理性能の変化を簡易的に検証した。

本研究の主な貢献は以下の通りである。

- **既存 LLM のバイナリデータ対応能力評価:** 既存の LLM 5 種類を対象にバイナリデータのテキスト表現を入力し、その処理能力を定量化した。その結果、モデルごとに最適な表現が異なることを確認し、特に GPT OSS が全体的に高い精度を示した。
- **テキスト表現選択の重要性:** 実験結果から、1 byte 区切りの 16 進数文字列がほぼすべてのモデルでバランスの良い性能を示したことから、いくつかのモデルを切り替えるなど汎用性の高い AI エージェントの設計には推奨できる形式であることを示した。

2. 関連研究

我々が調べた限りで、サイバーセキュリティ分野に特化せず一般向けに公開された素の LLM についてバイナリ解析能力の評価を行った研究は見つからなかった。ここでは機械学習でバイナリ解析している研究についていくつか取り上げる。

2.1 バイナリデータを前処理するもの

生のバイナリデータを入力とせず、前処理したものに入力とする既存研究について以下に示す。

DeepSemantic[1] は、関数の類似度推定、コンパイラの種類・最適化オプション推定などに利用されるバイナリコード埋め込み手法として提案された。この手法は、BERT を使用し、逆アセンブルした関数全体を 1 つの文 (sequence) として入力する。大量の関数に対してマスクされた命令を復元するタスクで事前学習をし、その後、関数の類似度推定などの特定領域に特化した再学習を行っている。

kTrans[2] は、関数の類似度推定、データフロー解析などに利用されるバイナリコード埋め込み手法として提案された。この手法は、BERT に類似した Transformer モデルを使用し、逆アセンブルした命令の構造情報（オペコード、オペランド、読み書き状態等）等を明示的に学習している。

NeuralCI[3] は、コンパイラの種類・最適化オプション推定を目的として提案された手法である。この手法は、CNN, Attention と CNN の組み合わせ、または GRU を使用している。入力は、逆アセンブルした命令列をそのまま使用せずに、オペランド中のベースアドレスを置き換えるなど抽象化し、Word2Vec を使用し、命令ベクトル列に埋め込みを行っている。

LLM4Decompile[4] は、バイナリコードから C 言語へ再変換する逆コンパイルを目的に提案された手法である。DeepSeek-Coder を再学習することで実現しており、

objdump で逆アセンブルした結果を入力している。

Mostafa らは、LLM をバイナリ解析に適用する際に入力前処理とトークン化手法が性能にどのように影響するか評価を行った [5]。この研究では、提案トークナイザで生成した入力シーケンスを使って既存 LLM に対し数エポック程度の再学習を行っている。

2.2 生データを入力するもの

生のバイナリデータを入力とする手法について以下に示す。

o-glassesX[6] は、実行ファイルの断片のバイト列を入力として、実行ファイルの生成に使用されたコンパイラの種類や最適化オプションを推定するモデルとして提案された。この手法は CNN と Attention を組み合わせた手法である。

BinProv[7] は、実行ファイルのコード部分のバイト列を入力として、実行ファイルの生成に使用されたコンパイラの種類や最適化オプションを推定するモデルとして提案された。この手法は BERT をベースとした Transformer モデルを使用している。

REFuSe は、関数 1 つ分の生のバイト列を入力とし、関数の特徴ベクトルとして 128 次元のベクトルを出力するモデルとして提案された [8]。この手法は CNN をベースとした手法である。

これらの手法は、いずれも特定の領域に特化して 1 からモデルを作成しており、LLM は使用していない。

3. 事前知識

3.1 LLM/SLM

LLM とは、端的に説明すると、大規模なデータを学習させた大規模なパラメータを持つ言語モデルのことを指す。つまり、LLM は大規模なテキストデータを処理し、その中に含まれる情報や関係性を学習することによって、自然言語処理能力を獲得している。

一方で、パラメータ数が比較的少ない言語モデルは SLM と呼ばれることがあり、コンパクトで効率的な言語モデルとして近年注目を集めている。一般的に、LLM が数百億から数千億のパラメータで構成されるのに対し、SLM は数億から数十億程度のパラメータで構成される。SLM は、LLM ほど高性能ではないものの、特定の用途に特化させた場合は LLM に匹敵する性能を発揮することもある。ただし、1~2 世代前の LLM よりも最新の SLM の方が性能が上という状況はよく発生する。

SLM は、LLM に比べて要求される計算リソースが少ないと考えられる。オンプレミス環境での運用に適していると考えられる。情報セキュリティ上の観点から、クラウドサービスを利用することを敬遠しがちな高機密な情報についても、オンプレミス環境に構築した SLM であれば、比較的安心して使用することが可能となる。

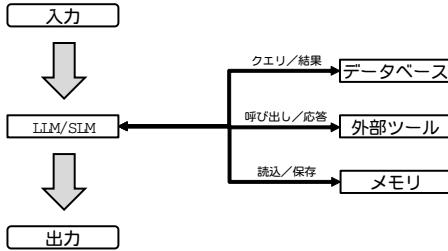


図 1 拡張言語モデル（ツールアクセス）

Fig. 1 Augmented LanguageSingle Models (Tool Access).

また、近年、学習可能なデータの枯渇が懸念されており、単純にスケーリング則 [9] (AI モデルの規模 (パラメータ数、データ量、計算量) を大きくすることで、性能が向上する関係性を表す経験則) に頼った LLM 性能の向上が困難になりつつある。したがって、学習データの規模が制限された環境下で効率的に学習する手法を生み出す必要があり、モデルサイズの増大を抑えつつ LLM の性能が向上させる研究がされている [10]。このことが SLM の性能の向上にも繋がっており、SLM の性能が実運用に耐えられる水準に近づいてきたという点が、近年、SLM に注目が集まっている要因のひとつと考えられる。

LLM と SLM の違いはモデルの大きさだけなので、以降は、基本的に SLM の概念も含めて単に LLM と記述する。ただし、サイズの違いを区別したい場合は、SLM と記述する。

3.2 拡張言語モデル

LLM 単体で利用した場合、一部の分野は苦手とされている。例えば、最新かつ専門的な情報を利用したい場合、ハッシュ値の計算など複雑な計算をする場合、条件に合致するものを漏れなく抜き出す場合、リアルタイムデータ(時刻、ファイル一覧など)を考慮した回答を生成したい場合は、LLM 単体での性能に限界があるとされる。

この問題を解決するものとして外部ツールと連携する LLM が注目を集めている。外部ツールや知識ソースを活用し、モデルの能力を強化する手法は、拡張言語モデル [11] (ALMs: Augmented Language Models) とも呼ばれる。拡張言語モデルとは、言語モデルが一般的に持っている、限られたコンテキストとパラメータによって制約される統計的言語モデルの枠組みを超えて、新しい能力や機能を追加した言語モデルを指す。

拡張言語モデルは、推論能力の強化を主眼に置いたもの (Reasoning) と外部システムとの連携に主眼をおいたもの (ツールアクセス、図 1) の大きく 2 つに分類される。本研究では外部ツールと連携するツールアクセスを扱う。外部ツールとは、本研究では外部関数、Linux コマンドや API の呼び出しを想定する。拡張言語モデルは、モデル内の知識に加えて、外部リソースを柔軟に活用することができ、

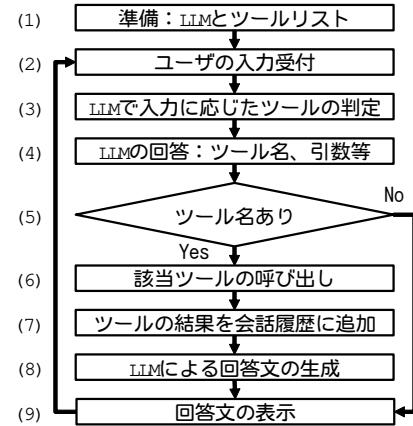


図 2 AI エージェントの実装概要

Fig. 2 High-Level Workflow of the AI Agent.

API 連携などを通じて、より正確で信頼性の高い応答を提供することが可能と考えられる。

具体的な実装方法については後述するが、外部ツールを利用することで LLM に必要な情報を渡すことができ、これにより、LLM はその内容を踏まえた回答を作成することができる。例えば、ファイルを分析したい場合には strings コマンドなどの外部ツールを呼び出し、その結果を LLM に渡すことで、ファイル中の可読文字列を踏まえた回答を生成させるという、いわゆる「目 grep」も簡易的なものであれば簡単に実現できる。

3.3 AI エージェント

AI エージェントとは、現時点において定義が必ずしも定まっていないが、本研究では、AI 機能を有するソフトウェアで、ユーザーを補助し、コンピュータ関連のタスクをユーザーに代わって行うソフトウェアを AI エージェントとする。

本研究で扱う AI エージェントは、ユーザーの入力を受け、入力に応じた外部ツールを呼び出し、その結果を踏まえた回答を LLM が生成する。一般的な AI エージェントの動作の概要を図 2 に示す。

- (1) 初期設定。使用する LLM・接続先の設定と外部ツール (ツールリスト) の定義を行う。
- (2) ユーザの入力を受け付ける。
- (3) ユーザの入力に (1) で作成したツールリストを加えて LLM を呼び出す。
- (4) ユーザの入力文から、LLM が外部ツール呼び出しが必要と判断した場合、LLM の回答に該当する外部ツールの情報が付記される。LLM が外部ツール呼び出しが不要と判断した場合、元の質問に対する回答を返す。
- (5) LLM の回答にツール名があるか否かで外部ツールの呼び出しの必要性を判断する。外部ツールを呼び出さない場合は (9) へ分岐する。
- (6) 該当する外部ツールを呼び出す。外部ツールの呼び出

表 1 バイナリデータのテキスト表現の例

Table 1 Example of a text representation of binary data.

データ表現	実装方法	値
生データ	body	01 02 03 04 05 06 07 08
一部エスケープ処理	body.decode("utf-8", "backslashreplace")	01 02 03 04 05 06 07 08
16進数文字列(区切りなし)	body.hex()	"0102030405060708"
16進数文字列(1 byte 区切り)	body.hex(sep=' ', bytes_per_sep=1)	"01 02 03 04 05 06 07 08"
16進数文字列(2 byte 区切り)	body.hex(sep=' ', bytes_per_sep=2)	"0102 0304 0506 0708"
Base64 文字列	base64.b64encode(body).decode()	"AQIDBAUGBwg=="

しに必要な引数があった場合は、(4)において LLM が会話履歴に基づき生成している。

- (7) 外部ツールの実行結果を会話履歴に追加する。これにより、LLM が外部ツールの実行結果を参照することが可能となる。
- (8) LLM にユーザの入力文に対応した回答を再度生成させる。外部ツールの呼び出し結果も踏まえた回答が生成される。
- (9) LLM の回答文を表示し、(2)へジャンプし、AI エージェントとの会話のループを継続する。

なお、2025 年 8 月に公開された GPT OSS も外部ツール呼び出しに対応しているが、推論中に外部ツールを呼び出しするように LLM の挙動が変更されたため、外部ツール呼び出し後に推論フェーズに戻る必要があり、当該モデルを使用する場合には図 2 のままでは正しく動作することはできない。

3.4 バイナリデータのテキスト表現

一般向けに公開されている LLM/SLM は入力データが UTF-8 でエンコードされたテキストデータであることを前提に設計されており、生のバイナリデータを入力することはできない。そのため、バイナリデータを何らかの方式でエンコードしてテキストデータに変換する必要がある。バイナリデータをテキストで表現する手法は様々なものがあるが、本研究では、既存 LLM がクローリングしたデータを学習する際に入力される可能性が高いものとして以下のものを選定した。

- **一部エスケープ処理:** バイナリデータをそのまま UTF-8 で文字列に変換し、変換エラーが発生するものだけエスケープ処理を行う。クローリングしたデータを学習データに変換する際に文字化け等の情報損失を抑えた前処理の 1 つと想定。
- **16進数文字列:** バイナリデータを 16 進数文字列に変換したもので、ソースコード、CTF の Writeup、セキュリティレポート等で頻出。ただし、表現方式に以下の 3 種類がある。
 - **区切りなし:** 16 進数文字列を区切りなく並べたもの。
 - **1 byte 区切り:** 1 byte ごとに空白を挟んだもので、

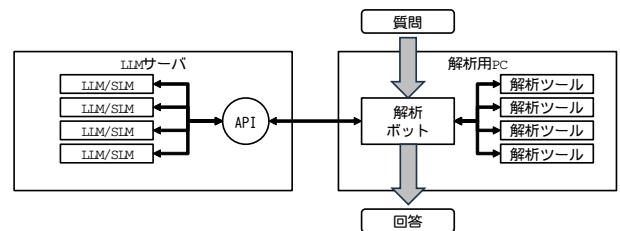


図 3 AI エージェントの構成
Fig. 3 Our AI Agent Configuration.

一般的なバイナリエディタで採用されているもの。

- **2 byte 区切り:** 2 byte ごとに空白を挟んだもので、hexdump コマンドや xxd コマンドのデフォルト。
- **Base64:** インターネット上でバイナリデータをテキスト表現で送受信する際に頻繁に使用される方式で、ソースコード、CTF の Writeup、セキュリティレポート等で頻出。

各テキスト表現について、Python による実装例とエンコード後の値の例を表 1 に示す。表 1 において、body には生データが格納されていることを想定している。「一部エスケープ処理」については、変換エラーが発生するもののみエスケープ処理するため、ほぼ生データと一致する。

4. バイナリデータを処理する AI エージェントの試作

本研究では、LLM のバイナリデータの処理能力の評価に先立ち、バイナリデータを処理する AI エージェントを試作した。

作成した AI エージェントのシステム構成を図 3 に示す。この AI エージェントは、LLM サーバと解析用 PC という 2 つの主要な機器で構成されている。

- **LLM サーバ:** LLM サーバは API 経由で解析用 PC からのリクエストを受け付け、LLM の生成文章を返す。本研究では、LM Studio 0.3.22^{*1}のローカルサーバ機能で実装した。
- **解析用 PC:** 解析用 PC については、どの解析ツールを使用するかで必要なスペックは大きく変わる。本研究の AI エージェントであれば、Python 3 が動作し、

^{*1} <https://lmstudio.ai>

表 2 AI エージェントを構築する PC 構成

Table 2 PC configuration for building AI agents.

	LLM サーバ	解析 PC
OS	macOS Sequoia 15.5	Ubuntu 24.04 (WSL)
CPU	M4 Pro	—
メインメモリ	64 GB	—
GPU メモリ	最大 48 GB	—

Linux の基本的なコマンドが実行できれば動作する。本研究では、表 2 の構成で AI エージェントを実装し、AI エージェントには主に以下の機能を実装した。

- ! [Linux コマンド]: 指定された Linux コマンドを実行し、その結果を会話履歴に追加
 - !! [質問]: 質問の回答に必要な Linux コマンドを生成し、実行。その結果に基づいて最終的な回答を生成
 - ? [質問]: 外部ツールを使用せずに LLM に直接質問
 - 作業ディレクトリの変更
 - 画像ファイル内の QR コードのデコード
 - ROT13 で暗号化された文字列のデコード
 - 破損 zip ファイルの強制復元
 - 画像をテキストで説明

AI エージェントの実行例を以下に示す。

Assistant: Hello! I'm an assistant helping with binary file analysis. What would you like me to do?

(Type 'quit' to exit)

```
You: !head -c 128 test.png
Assistant: \x89PNG (略)\xe1\x84N8\xe1\x84\xfe\xba
```

You: ?入力したデータを解析して

Assistant: (略)

4. 要約

- ・ 取得した 128 バイトは PNG ファイルのヘッダーと IHDR チャンク の情報を含んでいます。
 - ・ 画像サイズは 730×730 ピクセル、1ビット/ピクセル(白黒)です。
 - ・ IDAT チャンクが続き、 zlib 圧縮データが格納されています。
 - ・ 完全な画像解析には IDAT 全体を取得し、圧縮を解凍して表示・解析する必要があります。

この例では `head` コマンドにより、PNG ファイルの先頭 128 byte のほぼ生データを一般向けに公開された素の LLM に入力し解析を試みている。

また、`usr/bin/ls`に対し `strings` を実行した結果の文字列を LLM にカテゴライズ分けさせた場合の結果を図 4 に示す。素の LLM に特にテキスト表現を最適化しなくても、ある程度解析に有用な出力が得られる可能性を示している。

本 AI エージェントの会話履歴には様々な CLI ベースの

図 4 AI エージェントの実行例

Fig. 4 AI agent execution example.

解析ツールの出力結果等が会話履歴に入力されることになる。ツール毎に出力されるログは様々であり、バイナリデータのテキスト表現方法も様々である。各テキスト表現について LLM の処理能力に差異がある場合、入力するログのテキスト表現を変換することで、LLM の能力を最大限発揮した AI エージェントを実現できる可能性がある。

5. 簡易実験の方法

本研究では、表 3 の SLM を対象にバイナリデータの処理能力について簡易的な調査を行った。SLM は、近年発表された SLM から、ノート PC 向け APU で動作可能なサイズのものを選定した。さらに、AI エージェントの作成に必要なツール呼び出しに対応している SLM を優先的に選定した。また、バイナリファイル解析には、様々なツールの解析結果等を組み合わせることから、LLM/SLM が入出力可能なトークン数の合計である Context 長の長さが重要となる。そのため、長い Context 長を処理できる SLM を優先的に選定した。なお、Phi については、上記 2 つの条件を満たさないものの、事前調査である程度高性能な結果が見込まれたため、比較対象として選定した。

バイナリデータの処理として求められる能力は様々であるが、本研究では基礎的な能力の簡易評価として、後述する2つの実験を行った。

システムプロンプトは、LLMの素の能力を評価するためなるべくシンプルなものとし、具体的には以下を使用した。

You are a helpful assistant Speak in Japanese

5.1 実験 1：バイナリデータのテキスト表現のデコード

実験1では、バイナリデータが示す文字列を解釈できるかを簡易的に評価した。具体的には、バイナリデータをテ

表 3 評価対象の SLM
Table 3 Target SLM for evaluation.

略称	Model	Params	Size	量子化	Context 長	Tool	公開
Llama	lmstudio-community/meta-llama-3.1-8b-instruct	8B	4.92GB	Q4_K_M	128K	○	2024/07
	microsoft/phi-4	15B	9.05GB	Q4	16K	-	2024/12
Gemma	gemma-3-12b-it-qat	12B	7.74GB	Q4_0	128K	○	2025/04
	mistral-small-3.2-24b-instruct-2506	24B	15.21GB	Q4_K_M	128K	○	2025/06
GPT OSS	openai/gpt-oss-20b	20B	12.11GB	MXFP4	128K	○	2025/08

表 4 デコード能力評価用データ
Table 4 Decoding ability evaluation data.

元文字列	Base64 文字列
CSS2025@Okayama	Q1NTMjAyNUBPa2F5YW1h
CSS2024@Kobe	Q1NTMjAyNEBLb2J1
CSS2023@Fukuoka	Q1NTMjAyMOBGdWt1b2th
CSS2022@Kumamoto	Q1NTMjAyMkBLdW1hbW90bw==
CSS2021@Online	Q1NTMjAyMUBPbmxbmU=
CSS2020@Online(Kobe)	Q1NTMjAyMEBPbmxbmUoS29iZSk=
CSS2019@Nagasaki	Q1NTMjAxOUBOYWdhc2FraQ==
CSS2018@Nagano	Q1NTMjAxOEBOYWdhbm8=
CSS2017@Yamagata	Q1NTMjAxNOBZYW1hZ2FOYQ==
CSS2016@Akita	Q1NTMjAxNkBBA210YQ==

キスト表現として入力した場合、LLM が正確にデコードできるかをレーベンシュタイン距離 [12]（編集距離）を用いた類似度で評価した。

レーベンシュタイン距離は、二つの文字列がどの程度異なっているかを示す距離の一種であり、編集距離とも呼ばれる。具体的には、1 文字の挿入・削除・置換によって、一方の文字列をもう一方の文字列に変形するのに必要な手順の最小回数として定義される。この編集距離を以下の式で正規化することで類似度を求めることができる。

$$\text{類似度} = 1 - \frac{\text{編集距離}}{\text{長い方の文の文字数}} \quad (1)$$

この類似度は 0 から 1 までの値をとり、2 つの文が一致する場合に 1 を、全く一致しない場合に 0 をとる。

LLM のデコード能力の評価用のデータとして、表 4 に示す 10 種類のデータを準備し、デコードした結果の文字列と元文字列の類似度の平均及び標準偏差を求める事により、それぞれのテキスト表現のデコード能力を評価した。ただし、一部エスケープ処理したテキスト表現は既に decode 済みの状態であるため評価対象から除いた。

ユーザープロンプトは以下のものに、「入力」としてテキスト表現を加えたものを使用した。

#指示

入力されたバイナリデータを表す文字列をデコードして
#入力

5.2 実験 2: リトルエンディアンでの 4byte 値の抽出

実験 2 では、バイナリデータのオフセット値を指定し、

4byte を抽出し、リトルエンディアンを考慮した値に変換できるかを簡易的に評価した。

抽出元となるデータとして、0x01 から始まり 0x10 で終わる 16byte のデータを準備し、オフセット値 n を 1 から 10 まで変化させ、正しく処理できるかを評価した。本タスクは、(1) オフセット値から正しく 4byte 抽出できるか、(2) 抽出した 4byte を正しくリトルエンディアンで扱えるかの 2 つのタスクに分割できるため、LLM の最終回答だけではなく 2 つのタスクについて正しく処理できているかを人手で判断し、10 回の試行の平均値で精度を求めた。なお、タスク (2) については、(1) のタスクが正しく処理された前提で抽出した 4byte をリトルエンディアンで表現できるかを評価することとしたため、(1) のタスクが失敗しても (2) のタスクが成功することはあり得る。

実験 2 は、エンコードされていないバイナリデータを前提としたタスクであるため、テキスト表現のうち Base64 は実験対象から外した。

ユーザープロンプトは以下のものに、「入力」としてテキスト表現を加えたものを使用した。

#指示

入力されたバイナリデータの $\{n\}$ バイト目から 4 バイト分を取得しリトルエンディアンの数値として処理した結果を 16進数で表示して

#注意

最初のバイトは 1 バイト目と数える

#入力

6. 実験結果

6.1 実験 1: バイナリデータのテキスト表現のデコード

実験 1 の結果を表 5 に示す。実験の結果、GPT OSS が最も高い精度であることが確認できた。

Llama については、1 byte 区切りの 16 進数文字列を除き、類似度の値が 0.1 前後となった。この値は、元文字列と全く対応しない文字列を出力された場合にとる値であるため、そもそもデコードを前提とした学習を行っていない可能性が高い。

Gemma については、Base64 を除き高いデコード精度を示した。@の文字の処理に失敗する傾向があった。Base64 については、著しく低い精度であったため、デコードを前

表 5 バイナリデータのテキスト表現のデコード精度

Table 5 Decoding accuracy of text representation of binary data.

	Llama	Phi	Gemma	Mistral	GPT OSS
16進数文字列(区切りなし)	0.075±0.110	0.764±0.092	0.913±0.080	0.984±0.032	0.929±0.074
16進数文字列(1 byte 区切り)	0.368±0.286	0.762±0.191	0.937±0.062	0.854±0.094	0.960±0.053
16進数文字列(2 byte 区切り)	0.105±0.130	0.688±0.219	0.790±0.092	0.728±0.179	0.942±0.076
Base64	0.096±0.073	0.146±0.097	0.106±0.141	0.490±0.256	0.877±0.062

表 6 リトルエンディアンで 4byte 値を抽出する精度

Table 6 Accuracy of extracting 4-byte values in little-endian.

	Llama	Phi	Gemma	Mistral	GPT OSS
一部エスケープ処理					
抽出	0 %	90 %	50 %	70 %	90 %
エンディアン	0 %	40 %	40 %	90 %	100 %
最終回答	0 %	40 %	20 %	60 %	90 %
16進数文字列(区切りなし)					
抽出	20 %	90 %	70 %	90 %	90 %
エンディアン	0 %	70 %	50 %	100 %	100 %
最終回答	0 %	60 %	30 %	90 %	90 %
16進数文字列(1 byte 区切り)					
抽出	50 %	100 %	90 %	80 %	100 %
エンディアン	10 %	70 %	80 %	90 %	100 %
最終回答	10 %	70 %	70 %	80 %	100 %
16進数文字列(2 byte 区切り)					
抽出	0 %	70 %	60 %	30 %	90 %
エンディアン	10 %	70 %	60 %	40 %	100 %
最終回答	0 %	60 %	40 %	20 %	90 %

前提とした学習をしていない可能性が高い。

Mistralについては、区切りなしの16進数文字列において著しく高い精度を示した。1 byte 区切り、2 byte 区切りとなるに従ってデコード精度の低下及び標準偏差の増大が見られるため、デコード処理に関しては区切りなし 16進数文字列に統一した方が安定性が増すと思われる。Base64 のデコードについては、標準偏差の値が高く不安定であるものの、ある程度学習している可能性が高い。

Phiについては、Llama よりはデコード精度が高いものの、その他のモデルほどの性能は得られなかった。Base64 については、著しく低い精度であったため、デコードを前提とした学習をしていない可能性が高い。

GPT OSSについては、評価対象のすべてのテキスト表現について高い精度を示した。完全一致に成功したものはなかったものの、Base64 のデコードについて非常に高い精度をしたことから、Base64 のデコードを前提とした学習をしている可能性が高い。

6.2 実験 2: リトルエンディアンでの 4byte 値の抽出

実験 2 の結果を表 6 に示す。実験の結果、GPT OSS が最も高い精度であることが確認できた。

Llama については、1 byte 区切りの16進数文字列から

4byte を抽出する精度は 50 % であったものの、それ以外はほとんど失敗した。位置を指定したデータの抽出であったり、リトルエンディアンでの処理について、そもそも学習していない可能性が高い。

Gemma は 1 byte 区切りの16進数文字列の処理が最も得意であり、Mistral は区切りなしの16進数文字列の処理が最も得意であった。一方、2 byte 区切りの16進数文字列の処理については、Mistralにおいて、著しい性能低下が見られた。そのため、Mistral は 2 byte 区切りの文字列について学習データが比較的少ない可能性が高い。

Phiについては、位置を指定したデータの抽出精度が安定して高い傾向があった。リトルエンディアンを考慮した処理については、一度リトルエンディアンを考慮したバイト列の並び替えをしているにも関わらず再度並び替えをしたり、わざわざ 10 進数に戻して計算して 16 進数に戻す処理をして計算ミスをしたりするなど、深い推論によるハルシネーションの結果、低い値となった。

ほぼ生データの一部エスケープ処理したテキスト表現の処理については、そもそも Llama では想定していない、または学習データに含まれていない可能性がある。その他のモデルでは学習データに入っている可能性は高いが、0x0A が改行コードであるため、Gemma と Mistral では 0x0A を含むデータの処理に失敗する傾向が見られた。

7. 考察

7.1 既存 LLM のバイナリデータ処理能力の向上

本研究では、一般向けに公開されている既存 LLM を対象にバイナリデータの処理能力の簡易評価を行った。既存 LLM は、バイナリデータの処理などサイバーセキュリティに特化した学習を謳っていないものの、2024 年に公開されたモデルと 2025 年に公開されたモデルの間には明らかな性能向上が見られた。

既存 LLM を特定領域に特化させるため、追加学習手法が多く提案されているが、追加学習には学習済みの知識を忘れてしまうという破局的忘却 [13] という課題があり、その解決策についても様々な研究がなされている [14]。サイバーセキュリティ分野における分析は、特定の処理能力だけでなく様々な領域の知識を組み合わせて分析を行うことが多い。適用範囲をサイバーセキュリティ全般からより狭い領域に限定した分析において追加学習の有効性は示さ

れているものの、より広い範囲の領域を考慮した分析を行いたい場合に追加学習が有効かは明らかではない。そのため、追加学習を検討する前に、まずは素の状態の既存 LLM について適応したい領域の能力を評価し、追加学習の必要性について判断する必要があると思われる。

7.2 適切なテキスト表現の選択の必要性

簡易評価の結果、LLM およびタスク毎に LLM の最高性能を出すことができるテキスト表現はバラバラであったが、すべての LLM およびタスクにおいて最適な 1 つのテキスト表現というものは存在しないと思われる。しかしながら、実験の結果からは 1 byte 区切りの 16 進数文字列は最高性能でないものの最もバランスの良い性能であった。

AI 分野の進展は「毎日がゲームチャンジ」と呼ばれるほど著しいため、AI エージェントのエンジン部分については特定の LLM に依存せず新しい LLM に容易に入れ替え可能なものとした方がとしたものの方が LLM の進展に追随できる汎用性の高い手法である可能性がある。

8. おわりに

本研究では、サイバーセキュリティ分野で頻繁に扱われるバイナリデータを対象とした AI エージェントのプロトタイプを作成し、その上で既存 LLM がテキスト表現されたバイナリデータを処理する際の性能を定量的に評価した。

実験の結果、GPT OSS が最も高いバイナリデータの処理能力があることを示した。また、最新の LLM に容易に切り替え可能な汎用性の高い AI エージェントの設計のためには、テキスト表現として 1 byte 区切りの 16 進数文字列を選択することが最もバランスが良いことを示した。

本研究では、AI エージェントのプロトタイプを作成したもの、性能評価としては既存 LLM の素のバイナリデータ処理能力の評価に留まっている。今後は、本研究の知見を踏まえた改良を AI エージェントのプロトタイプに実装し、AI エージェントとしてのバイナリファイル解析の処理能力の評価を行っていきたい。

参考文献

- [1] Koo, H., Park, S., Choi, D. and Kim, T.: Semantic-aware binary code representation with bert, *arXiv preprint arXiv:2106.05478* (2021).
- [2] Zhu, W., Wang, H., Zhou, Y., Wang, J., Sha, Z., Gao, Z. and Zhang, C.: ktrans: Knowledge-aware transformer for binary code embedding, *arXiv preprint arXiv:2308.12659* (2023).
- [3] Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L. and Wu, D.: Fine-grained compiler identification with sequence-oriented neural modeling, *IEEE Access*, Vol. 9, pp. 49160–49175 (2021).
- [4] Tan, H., Luo, Q., Li, J. and Zhang, Y.: Llm4decompile: Decompiling binary code with large language models, *arXiv preprint arXiv:2403.05286* (2024).

- [5] Mostafa, A., Nahid, R. A. and Mulder, S.: How Different Tokenization Algorithms Impact LLMs and Transformer Models for Binary Code Analysis.
- [6] Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T. and Ukegawa, H.: o-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment, *Proc. Workshop Binary Anal. Res.*, pp. 1–12 (2020).
- [7] He, X., Wang, S., Xing, Y., Feng, P., Wang, H., Li, Q., Chen, S. and Sun, K.: Binprov: Binary code provenance identification without disassembly, *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 350–363 (2022).
- [8] Saul, R., Liu, C., Fleischmann, N., Zak, R., Micinski, K., Raff, E. and Holt, J.: Is function similarity over-engineered? building a benchmark, *Advances in Neural Information Processing Systems*, Vol. 37, pp. 21636–21655 (2024).
- [9] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J. and Amodei, D.: Scaling laws for neural language models, *arXiv preprint arXiv:2001.08361* (2020).
- [10] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A. et al.: Training compute-optimal large language models, *arXiv preprint arXiv:2203.15556* (2022).
- [11] Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A. et al.: Augmented language models: a survey, *arXiv preprint arXiv:2302.07842* (2023).
- [12] Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals, *Soviet physics-doklady*, Vol. 10, No. 8 (1966).
- [13] McCloskey, M. and Cohen, N. J.: Catastrophic interference in connectionist networks: The sequential learning problem, *Psychology of learning and motivation*, Vol. 24, Elsevier, pp. 109–165 (1989).
- [14] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A. et al.: Overcoming catastrophic forgetting in neural networks, *Proceedings of the national academy of sciences*, Vol. 114, No. 13, pp. 3521–3526 (2017).