BlockLens: Detecting Malicious Transactions in Ethereum Using LLM Techniques

Chi Feng[1][0009−0002−3908−4726] and Lei Fan[1][0000−0003−3975−091X]

Shanghai Jiao Tong University, Shanghai 200240, China
{gdpeaceminusone, fanlei}@sjtu.edu.cn

Abstract. This paper presents BlockLens, a supervised, trace-level framework for detecting malicious Ethereum transactions using large language models(LLMs). Unlike prior approaches limited to static features or storage-level abstractions, BlockLens processes complete execution traces, capturing opcode sequences, memory information, gas usage, and call structures to accurately represent the runtime behavior of each transaction. This framework harnesses the exceptional reasoning capabilities of LLMs for long input sequences and is fine-tuned on transaction data. We design a tokenization strategy aligned with Ethereum Virtual Machine (EVM) semantics, mapping execution traces into interpretable tokens. Each transaction captures its complete execution trace through simulated execution and is then sliced into overlapping chunks using a sliding window, allowing for long-range context modeling within memory constraints. During inference, the model outputs both a binary decision and a probability score indicating the likelihood of malicious behavior. We implement the framework based on LLaMA 3.2-1B backbone and fine-tune the model using Low-Rank Adaptation (LoRA). We evaluate it on a curated dataset containing both real-world attacks and normal DeFi transactions. BlockLens outperforms representative baselines, achieving higher F1 scores and recall at top-k thresholds than representative baselines. Additionally, BlockLens offers interpretable chunk-level outputs by localizing suspicious trace segments that enhance explainability, facilitating rapid forensic analysis and actionable decision-making in security-critical environments.

Keywords: Ethereum · Malicious Transaction Detection · Large Language Models.

1 Introduction

1.1 Background

Ethereum [4], as a representative blockchain platform, has rapidly emerged as the cornerstone infrastructure for decentralized applications (DApps) due to its native support for smart contracts and highly programmable nature. Its versatile transaction model encompasses a diverse range of operations, including asset transfers, smart contract invocations, non-fungible token (NFT) transactions,

and decentralized finance (DeFi) activities. To date, the Ethereum network has processed more than 2.6 billion transactions, representing a cumulative monetary value exceeding trillions of dollars.

However, alongside this growth, Ethereum's rapid ecosystem expansion has exposed security vulnerabilities. Malicious transactions exploiting smart contract vulnerabilities, such as reentrancy attacks, front-running attacks, and price manipulation attacks, have become increasingly frequent. Such incidents not only result in substantial financial losses for users but also threaten the stability and credibility of the blockchain network. According to recent statistics from Etherscan, over 405 security incidents have targeted decentralized finance (DeFi) protocols operating on the Ethereum platform, leading to financial losses of approximately $3.8 billion. One prominent example is the reentrancy attack on the DAO smart contract in April 2016, leading to the theft of $3.6 million [2].

Timely identification of malicious transactions is critical for enabling attack forensics, mitigating financial damage, and maintaining user confidence in the integrity of blockchain systems. Accurate detection allows researchers to analyze attack mechanisms and processes. For example, when smart contract vulnerabilities are reported with sufficient detail, researchers can trace suspicious transactions to reconstruct the corresponding call graph and interaction pathways, thereby understanding attackers' methodologies and identifying potential attack vectors. Also, detection mechanisms enable protocols to swiftly trigger emergency actions, such as suspending compromised smart contracts or issuing timely alerts, thus mitigating further financial damages and maintaining protocol stability and liquidity. For instance, prompt detection of front-running [6] or price manipulation attacks allows protocols to protect user funds proactively.

1.2 Related Works

Existing research on detecting malicious transactions within Ethereum can be categorized into three main methodologies: classical machine learning (ML) methods, rule systems, and large language model (LLM) approaches.

Classical ML-based methods have been widely adopted in blockchain malicious detection. For example, the Doc2Vec method introduced by Le and Mikolov [13] represents blockchain transactions as a collection of words and employs distributed word embeddings to generate vectorized transaction representations. These vectors are then analyzed using Gaussian Mixture Models (GMM) to estimate the likelihood of a transaction being malicious [19]. Although this probabilistic approach is capable of capturing malicious behaviors based on distributional deviations, its limitation lies in the inherent treatment of transactions as unordered word sets. This approach fails to capture contextual relationships, which are essential in distinguishing differences between normal and malicious transactions. As a result, Doc2Vec-based methods struggle with detecting attacks characterized by specific contextual conditions.

Rule-based methods, combined with ML techniques, have also been proposed to detect malicious behaviors by modeling transaction patterns or heuristic properties. For instance, LSTM-based methods can learn normal transaction pat-

terns and identify malicious behaviors based on deviations from the learned patterns [1]. Moreover, heuristic approaches use simplistic assumptions, such as identifying malicious behaviors by input sequence length under the assumption that malicious transactions are generally longer than normal ones [16]. Although these approaches can quickly identify well-understood malicious patterns, their primary drawback is their lack of adaptability and robustness in detecting new or evolving attacks. As malicious behaviors become increasingly sophisticated, relying solely on static, predefined rules significantly diminishes detection efficacy. Consequently, their utility diminishes significantly in real-world environments where attacks vary widely in complexity, length, and execution patterns.

Recently, LLM-based methods have gained attention due to their semantic modeling capabilities. BlockGPT [9] employs a GPT-like causal transformer model trained on quantities of normal transaction data. It aims to detect malicious behaviors by measuring the conditional likelihood of each token within transaction sequences. While BlockGPT integrates dynamic execution information, it limits its analysis to persistent storage read-write operations, explicitly avoiding opcode trace analysis due to computational complexity concerns. Furthermore, due to its causal design, BlockGPT inherently relies solely on prior tokens to predict subsequent tokens. This sequential, causal prediction approach diminishes its sensitivity in detecting malicious transactions, as blockchain transaction sequences do not follow a natural generative order akin to language.

Another representative LLM-based method, BlockFound [20] leverages a BERT-based Masked Language Modeling (MLM) approach to learn normal transaction patterns. Despite effectively capturing bidirectional contextual information, BlockFound notably neglects the dynamic execution context, particularly opcode-level transaction details. This omission restricts the model's capability to capture hidden attack behaviors embedded within execution traces. Consequently, BlockFound's performance suffers when faced with malicious transactions that differ subtly in execution rather than in textual representations alone.

Also, several recent studies have explored generative LLMs for modeling execution sequences. EarlyMalDetect [12] leverages an autoregressive transformer to analyze early-stage API call sequences during program execution, enabling the detection of malicious behavior before it fully manifests. TRACED [8] introduces execution-aware pre-training for source code by incorporating dynamic runtime traces alongside static code, significantly improving downstream tasks such as execution path prediction and vulnerability detection. These works collectively highlight that generative modeling of execution traces, whether system calls, virtual machine instructions, or runtime program states, provides a paradigm for detecting malicious behaviors in structured environments, and motivates our application of this approach to Ethereum transaction opcode traces.

1.3 Our Contributions

In this paper, we propose BlockLens, the first detection framework to fully input Ethereum execution-level opcodes into LLMs, enabling reasoning over complete EVM traces that include gas usage, call depth, and control flow.

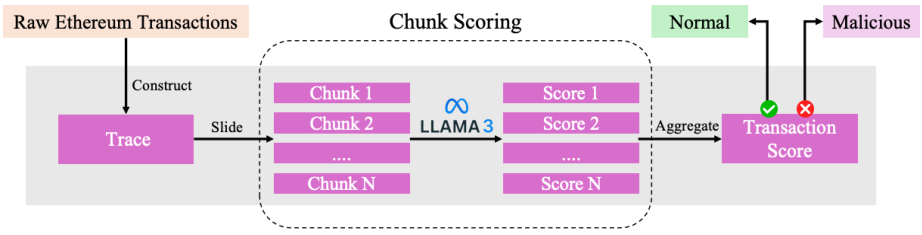

Fig. 1: Overview of BlockLens Detection Framework.

Unlike traditional rule-based or heuristic-driven approaches, which detect attacks through pattern matching or static vulnerability analysis, we construct a transaction-level detection pipeline using dynamic execution traces, which capture runtime behaviors such as opcode sequences, gas usage, and control flow, providing richer semantics than static features.

To effectively analyze the complexity of transaction data, we integrate recent advancements in Large Language Models into our detection framework. LLMs are particularly suited to detection due to their contextual modeling capabilities gained through extensive pre-training on large-scale datasets. Unlike conventional ML methods, which typically truncate lengthy context sequences abruptly, thereby losing critical contextual information, we can utilize a chunk scoring mechanism that preserves more comprehensive context information. This strategy improves the model's ability to interpret complex and lengthy transaction traces. We also propose an aggregation strategy to convert chunk-level scores into transaction-level predictions, demonstrating that LLMs can effectively distinguish malicious behaviors even in long, multi-step transactions.

Based on this method in Figure 1, we systematically leverage historical attack data to train an LLM model capable of recognizing diverse malicious transaction patterns, which can explicitly learn characteristics from various known attacks and integrate internal contract execution details.

2 Preliminaries

2.1 Large Language Models (LLMs)

Built upon the transformer architecture [18], Large Language Models (LLMs) process input sequences through stacked self-attention and feedforward layers, enabling them to capture long-range dependencies and complex token interactions. LLMs are typically pre-trained on large-scale corpora using self-supervised learning objectives. Common pretraining strategies include masked language modeling (MLM), as employed in BERT [7], and next-token prediction, as used in autoregressive models like GPT [3]. Pretraining enables LLMs to learn generalizable representations that can be transferred to various downstream applications. While initially designed for natural language, LLMs have been successfully adapted to domains with structured or symbolic data, such as programming

languages [15], scientific formulas [14]. Their ability to understand structured sequential patterns makes them promising tools for analyzing program-like execution traces in areas such as smart contract transactions or runtime behaviors.

2.2 Generative Decoder-Only Models

A major class of LLMs is the *decoder-only* or *generative* language model, which predicts sequences in a left-to-right, autoregressive manner. Compared to bidirectional models, generative LLMs naturally preserve temporal order and can model the probabilistic structure of an execution trace as it unfolds. This autoregressive nature is particularly beneficial in domains where the execution context is directional or causally dependent, such as bytecode interpretation, program simulation, or blockchain transaction modeling, where each step logically follows the previous. Such models are trained to generate the next token given prior context. Well-known examples include GPT [3], PaLM [5], and LLAMA [17].

2.3 Blockchain

Blockchain is a distributed ledger technology characterized by decentralization, immutability, and transparency. Structurally, a blockchain is composed of sequential blocks, each containing a cryptographic hash that links it to the preceding block, thereby forming an immutable chain. Each block includes a set of validated transactions, timestamps, and consensus-related metadata.

Among various blockchain platforms, Ethereum, introduced in 2015, stands out as the most prominent environment. It extends the original concept by integrating the EVM, a Turing-complete virtual machine, which enables general-purpose computation on-chain. Transactions are fundamental operations within Ethereum, representing requests from accounts to modify the blockchain state. Ethereum transactions primarily include asset transfers, smart contract executions, and interactions within decentralized applications.

3 Methodology

3.1 Problem Definition

We address the problem of identifying malicious transactions through a learned model, using both statistical transaction features (e.g., transaction hashes, values, addresses) and dynamic opcode-level execution sequences (VMtrace).

Threat Model. We assume an adversarial threat model primarily focused on malicious transactions that exploit vulnerabilities or logic flaws in smart contracts within the Ethereum blockchain ecosystem.

An attacker may build transactions specifically designed to trigger known or unknown vulnerabilities. He can interact directly with deployed smart contracts by crafting specially designed transactions containing abnormal opcode

sequences or unusual execution paths. He might intentionally obfuscate the characteristics of malicious transactions to evade traditional static detection mechanisms. Therefore, he can achieve malicious objectives such as asset theft, market manipulation, or disrupting contract execution.

Detection Model. Given a set of transactions T, each transaction $t \in T$ can be represented by its associated statistical features and dynamic execution trace. The primary objective is to determine whether a given transaction t belongs to the set of anomalous transactions $A \subset T$. To accomplish this, we define a binary decision function $f : T \to \{0, 1\}$, which classifies transaction t based on the model's predicted probability of malicious, denoted as $P(\text{abnormal} \mid t)$:

$$f(t) = \begin{cases} 1, & \text{if } P(\text{abnormal} \mid t) \geq \tau \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Where:

- T denotes the complete transaction set under consideration;
- $P(\text{abnormal} \mid t)$ represents the model's probabilistic prediction of transaction t being malicious;
- τ is a decision threshold to balance precision and recall in the detection task.

A transaction t is classified as malicious if $f(t) = 1$; otherwise, it is normal.

3.2 Technique Overview

To effectively detect malicious behaviors in Ethereum transactions, we propose a four-stage framework that transforms raw execution data into structured inputs suitable for large language model (LLM)-based analysis. The overall architecture consists of the following components as illustrated in Fig 2.

Trace Construction. We first extract and simplify multiple layers of Ethereum execution traces—including call traces, opcode traces, and VM traces—into a unified, structured format. This step involves operand extraction, depth annotation, gas cost tagging, and hierarchical call flattening, culminating in a linearized and semantically enriched sequence representation.

Trace Tokenization. We construct a custom tokenizer tailored for Ethereum semantics, incorporating dedicated vocabularies for opcodes, addresses, function signatures, numerical operands, and gas values. To enhance generalization and reduce sparsity, all numerical and gas-related tokens are quantized using multi-level bucketing strategies.

Trace Embedding. Tokenized traces are mapped into integer sequences and padded or split using a sliding window to conform to fixed-length input constraints. Additional structures, such as attention masks and sequence boundary markers, are added to support masked modeling and classification tasks.

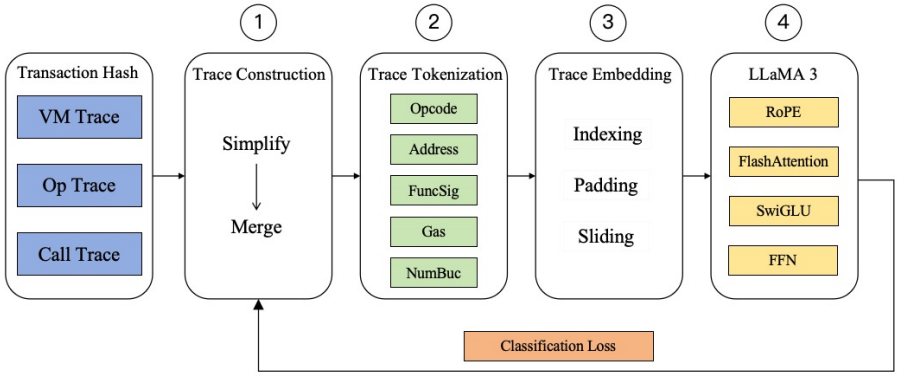

Fig. 2: System pipeline for malicious transaction detection. Starting from trace construction (①), we simplify and merge multiple trace types, tokenize the structured sequence (②), embed the result into a fixed-length vector input (③), and feed it into a fine-tuned LLaMA 3 model (④) for classification. A loop with supervised loss propagates gradients to guide learning.

Model Fine-tuning. We employ LLaMA 3, a decoder-only Transformer architecture, as the core detection engine. The model is fine-tuned on labeled Ethereum traces using a supervised classification objective. Its architectural features—including FlashAttention v2, SwiGLU activation, and rotary positional embeddings—enable efficient handling of long-range dependencies in execution sequences. To further improve adaptability, our design supports low-rank adaptation (LoRA) modules and gas-aware token fusion.

3.3 Technique Details

Due to space limitations, we have left the complete technical details in the appendix. Here, we introduce only a few key technologies.

Trace Construction. For each collected transaction (both normal and malicious), we retrieved three distinct execution-level traces from the blockchain: Call trace, op trace, and VM trace.

- **CallTrace**: captures high-level statistical metadata for each transaction, such as the addresses of senders and recipients, transferred amounts, and smart contract calls. Each call trace entry documents an interaction, including the call type, input and output data, and any nested internal calls.
- **OpTrace**: captures the precise opcode execution sequence. It includes opcode types, gas details, and stack contents at each execution step.
- **VMTrace**: provides an operational view of opcode-level execution within the EVM. It records the state changes during execution, particularly memory operations, stack manipulations, and interactions with persistent storage.

These traces are cleaned, simplified, and hierarchically merged to produce a unified representation. The resulting trace is formatted as a linearized sequence of opcodes and arguments, suitable for downstream language modeling. Details, examples, and formatting rules refer to Appendix A.

Trace Tokenization To support effective modeling, we construct a structured vocabulary encompassing opcodes, Ethereum addresses, function signatures, numerical operands, and gas values. These tokens are designed to preserve EVM semantics while reducing redundancy and improving interpretability.

- **Address Vocabulary:** Ethereum addresses are normalized to a fixed 20-byte hexadecimal format by padding truncated values and validating operand patterns. We extract candidate addresses from both call trace fields and opcode operands. The final address vocabulary contains 12,980 unique entries.
- **Function Signature Vocabulary:** Function selectors are derived from the first 8 hexadecimal characters of the calldata `input` field. To standardize them, we trim excessive leading zeros and pad short selectors when necessary. A total of 8,437 unique function signatures are included in the vocabulary.
- **Numerical Value Bucketing:** Operands not identified as addresses or signatures are treated as numerical values. These are converted from hexadecimal to normalized decimal, then discretized into a two-level scheme of buckets and blocks. The number of buckets serves as an adjustable hyperparameter. Empirically, 90 buckets and 100 blocks yield good performance.
- **Gas Value Bucketing:** Opcode-level gas costs are log-transformed and bucketed to capture computational characteristics. We tune the bucket resolution using entropy-based selection and represent each gas value with a symbolic token (e.g., `[GAS_x]`). Each gas value g is mapped to a discrete bucket index b using the following transformation:

$$b = \min\left(\lfloor \log(g+1) \cdot c \rfloor, b_{\max}\right) \tag{2}$$

  Where c is a scaling constant that controls bucket resolution, and $b_{\max}$ denotes the maximum allowable bucket index to prevent outlier explosion. To select an appropriate logarithmic base for bucketing, we empirically evaluate the entropy of bucket distributions under different bases. The base yielding the highest entropy is chosen to be the representation of gas costs.

The vocabulary consists of special tokens (10), EVM opcodes (149), function signatures (8,437), addresses (12,980), numerical buckets (90), blocks (100), and gas values (300). Compared to general-purpose tokenizers (e.g., BPE), our custom tokenizer can leverage domain knowledge to preserve EVM semantics, reduce vocabulary size and sequence length by avoiding sub-token fragmentation, and enable interpretable token-to-behavior mapping, beneficial for analysis.

Trace Embedding To align with the domain-specific vocabulary described above, we map each token in the trace into a structured integer sequence. The

tokenizer is implemented as a deterministic mapping function tokenizer : $\mathcal{V} \to \mathbb{Z}^+$, where $\mathcal{V}$ denotes the constructed vocabulary and each token is assigned a unique integer ID. The tokenization process consists of three primary stages:

- **Vocabulary Indexing:** All elements in the final vocabulary are indexed sequentially. Special tokens are assigned reserved positions (e.g., `PAD` = 0, `UNK` = 1), followed by opcodes, function selectors, addresses, numeric bucket-block combinations, and gas numbers.
- **Trace Parsing:** The tokenizer identifies instruction types.
- **Token Mapping:** Tokens are mapped to their corresponding indices. If a token is not found in $\mathcal{V}$, it is assigned to the `UNK` token.

All tokenized traces are processed to a fixed sequence length. Short traces are right-padded with the `PAD` token, while long traces are truncated or split using a sliding window strategy. Special tokens such as `[START]`, `[CONTINUE]`, and `[END]` are also added to indicate sequence boundaries during pretraining.

Model Fine-tuning In this work, we adopt LLAMA 3 [10], an open-source large language model released by Meta AI, as the backbone for malicious transaction detection. LLAMA 3 supports input sequences up to 4096 tokens, making it well-suited for long and complex transaction traces observed in Ethereum.

Given the variable length of traces, we implement a sliding window chunk-level mechanism to divide each trace into overlapping segments of fixed size. For long transactions, the window slides with a stride to preserve contextual continuity. We also conduct zero-shot experiments without fine-tuning, as detailed in Appendix B.

We fine-tune LLAMA 3 using supervised learning on labeled transactions from our curated dataset. The model is trained to minimize cross-entropy loss over the prediction probability $P(\text{abnormal}|t)$ for each input trace t:

$$\mathcal{L} = -y \cdot \log P(\text{abnormal}|t) - (1-y) \cdot \log(1 - P(\text{abnormal}|t))$$

where $y \in \{0, 1\}$ is the ground-truth label. The objective of fine-tuning is to adjust the pre-trained model weights so that the attention layers and embedding mechanisms specialize in recognizing malicious patterns in transaction traces.

4 Experimental Setup

4.1 Dataset

To evaluate our proposed detection framework, we construct a labeled dataset, including both normal and malicious Ethereum transactions. We collect all transactions by running a full Ethereum node locally, which allows direct access to raw data, including transaction traces and execution metadata.

We deploy the local node on a high-performance workstation equipped with a 12th Gen Intel(R) Core i9-12900KF processor (24 cores), an NVIDIA RTX

4080 SUPER GPU with 16 GB of memory, 64 GB of DDR4 RAM, and an 8 TB NVMe SSD. The system runs on Ubuntu 20.04.06 LTS.

Our dataset contains 240 malicious and around 3,000 normal transactions sampled across one year. The data is split using a random seed into 70% training, 10% validation, and 20% testing. We train our model on the labeled dataset.

Normal Transactions To ensure the diversity of transaction behaviors, we sample normal transactions from user interactions with a set of widely used Ethereum-based decentralized finance (DeFi) applications. These applications include token exchanges, lending protocols, liquidity pools, and other financial primitives deployed across the Ethereum mainnet.

The sampling process spans over one year and follows a weekly strategy:

- For each week, we randomly select two Ethereum block numbers.
- For each selected block, all contained transaction hashes are retrieved.
- Using these hashes, we extract full transaction details and trace data directly from the local full node.

After initial collection, we perform manual curation to remove failed transactions or transactions unrelated to target DeFi applications. This cleaning step ensures the representativeness and relevance of normal samples while minimizing noise from spam or protocol-internal system calls. The resulting normal dataset comprises thousands of diverse and valid transaction records.

Malicious Transactions We collect malicious transactions from public repositories such as DeFiHackLabs, De.Fi, and Slowmist, which document well-known attacks on Ethereum DeFi protocols. Specifically, we extract 240 confirmed malicious transaction hashes that correspond to real-world exploits, including reentrancy attacks, flash loan-based arbitrage, price manipulation, and oracle abuse.

Trace Extraction For each malicious or normal transaction, we obtain three types of execution traces from the local Ethereum node:

- `Call Trace`: records the high-level contract invocation sequence;
- `Op Trace`: captures opcode-level EVM execution data and gas usage;
- `VM Trace`: contains detailed stack and memory operations during runtime.

All collected traces are processed according to the methodology described in Section 3.3, including trace simplification, trace merging, and trace tokenization. Appendix A will introduce the trace extraction demo.

4.2 Baselines

To comprehensively evaluate the effectiveness of our proposed LLaMA-based transaction malicious detection model, we compare it with two representative baselines from large language model (LLM)-based approaches.

BlockFound [20]: BlockFound utilizes masked language modeling (MLM) to identify patterns in normal transactions. It is based on the BERT architecture (300M parameters, hidden size 768, 12 layers, 12 attention heads), and trained using MLM over normal transactions. During inference, each transaction is partially masked, and the reconstruction error is computed as a malicious score. BlockFound benefits from bidirectional attention, allowing it to capture both past and future contextual information. However, it does not incorporate execution-level information, such as opcodes or call traces, limiting its capability in detecting subtle behavior-based malicious activity.

BlockGPT [9]: BlockGPT adopts a GPT-style causal transformer trained to predict the next token in a transaction trace. The underlying assumption is that malicious transactions deviate from the learned distribution of normal patterns. For each transaction, it computes the sum of conditional likelihoods of all tokens, treating a lower total likelihood as a higher malicious potential. Unlike BlockFound, BlockGPT only attends to previous tokens and does not consider future context. Moreover, it processes traces derived from persistent storage read/write operations only, without fine-grained opcode or call data, which could overlook necessary attack signatures. As no official code is available, we contacted the authors and implemented BlockGPT under their guidance.

4.3 Fine-tuning Model

We adopt the publicly released Meta-Llama-3.2-1B model [10] as the backbone for our malicious detection task. Compared to larger-scale LLMs, the 1 billion parameter variant offers a more efficient trade-off between inference cost and representation power. It supports a maximum context length of 4096 tokens, making it well-suited for modeling long transaction traces.

To ensure memory efficiency, we avoid loading the entire in-memory dataset. Instead, each transaction is sliced using a sliding window of 2048 tokens and a stride of 1024 tokens. Each chunk is framed by `[START]` and `[END]` tokens.

We utilize the Huggingface Trainer API for supervised fine-tuning of the model. The tokenizer is initialized from a custom vocabulary, and model parameters are loaded from the Meta-provided pre-trained checkpoint.

To reduce the memory footprint and accelerate training, we adopt Low-Rank Adaptation (LoRA) [11] for parameter-efficient fine-tuning. We inject trainable, rank-decomposed matrices into attention and projection layers while keeping the original model weights frozen. In our configuration, we set the LoRA rank to 8 and the scaling factor `lora_alpha` to 16. This lightweight adaptation mechanism significantly reduces the number of trainable parameters, enabling efficient fine-tuning even on consumer-grade GPUs without sacrificing detection performance.

5 Evaluation and Discussion

5.1 Quantitative Evaluation

Performance comparison across different models To assess the effectiveness of our proposed modeling approach, we compare its performance with four representative baselines in the blockchain malicious transaction detection:

- **BlockGPT** [9]: A GPT-style autoregressive transformer that takes persistent storage access traces as input and outputs a score of token-level log-likelihood. Malicious transactions are identified by lower sequence likelihood.
- **BlockFound** [20]: A BERT-based masked language model trained on normal transactions. During inference, it measures reconstruction error of masked tokens in opcode-level sequences to assign malicious scores.
- **GMM + Doc2Vec** [13]: A classical machine learning method where transactions are embedded using Doc2Vec to produce fixed-length vector representations, and a Gaussian Mixture Model (GMM) estimates the likelihood of each sample belonging to the distribution of normal transactions.
- **Rule-based (length)** [16]: A simple heuristic method that classifies transactions based on the total length of their sequences, under the assumption that malicious transactions tend to be longer.
- **BlockLens**: A supervised classification model based on LLAMA3 interleaves opcodes into a unified input sequence. The model outputs both a continuous probability score representing the likelihood of malicious behavior and a binary prediction indicating whether the transaction is malicious or normal.

And all models are evaluated on the same test set using a *ranking-based* approach. Each transaction receives a predicted malicious score, and transactions are sorted in descending order of score. We then compute evaluation metrics based on the top-k% most suspicious transactions, which simulates real-world alert systems where only the highest-risk samples are reviewed manually.

Table 1 and 2 present the False Positive Rate (FPR), Precision, Recall, and F1-score at various detection thresholds (Top-5%, 10%, 15%, and 20%).

Table 1: Performance comparison at Top-5% and Top-10% thresholds.

Model	Top 5%				Top 10%			
	FPR	Precision	Recall	F1	FPR	Precision	Recall	F1
BlockGPT	1.43%	73.33%	45.83%	56.41%	4.46%	58.33%	72.92%	64.81%
BlockFound	1.07%	80.00%	50.00%	61.54%	4.29%	60.00%	75.00%	66.67%
GMM + Doc2Vec	2.14%	60.00%	37.50%	46.15%	6.79%	36.67%	45.83%	40.70%
Rule-based (length)	1.61%	70.00%	43.75%	53.65%	6.07%	43.33%	54.17%	48.11%
Our BlockLens	0.89%	83.33%	52.08%	63.93%	3.75%	65.00%	81.25%	72.28%

Table 2: Performance comparison at Top-15% and Top-20% thresholds.

Model	Top 15%				Top 20%			
	FPR	Precision	Recall	F1	FPR	Precision	Recall	F1
BlockGPT	9.82%	39.56%	75.00%	51.80%	17.35%	32.23%	82.12%	33.46%
BlockFound	9.46%	41.76%	79.17%	54.68%	14.29%	33.88%	85.42%	48.52%
GMM + Doc2Vec	11.25%	30.77%	58.33%	40.33%	15.71%	27.27%	68.75%	39.00%
Rule-based (length)	10.54%	35.16%	66.67%	46.25%	15.18%	29.75%	75%	42.46%
Our BlockLens	8.92%	45.05%	85.42%	58.92%	13.75%	36.36%	91.67%	52.00%

Across all thresholds, our BlockLens consistently demonstrates strong recall performance, which is essential in security applications where missing malicious cases can be far more costly than raising a few false alarms.

At the Top-5% threshold, BlockLens achieves a recall of **52.08%**, surpassing both BlockGPT (45.83%) and BlockFound (50.00%), while maintaining a substantial precision of 83.33%.

At the Top-10% threshold, recall of BlockLens improves to **81.25%**, outperforming BlockGPT (72.92%) and BlockFound (75.00%).

At the Top-15% threshold, BlockLens reaches a recall of **85.42%**, exceeding BlockGPT (75.00%) and BlockFound (79.17%). While precision drops to 45.05%, the F1 score remains competitive, indicating a well-balanced trade-off.

At the Top-20% threshold, BlockLens attains the highest recall of **91.67%**, outperforming both BlockGPT (82.12%) and BlockFound (85.42%). This performance demonstrates the model's robustness in identifying nearly all malicious transactions when a broader detection margin is acceptable.

We further examine how varying the score threshold affects detection performance. As shown in Table 3, lower thresholds (10–20) yield perfect recall (100.00%) but suffer from extremely high false positive rates (up to 92.68%). In contrast, higher thresholds (80–90) lead to excellent precision (up to 100.00%) but miss a large portion of malicious samples, with recall dropping below 40%.

At intermediate thresholds (50–70), the model achieves a more favorable trade-off. For instance, at threshold 70, we observe a recall of 81.25% and a precision of 57.35%, resulting in the highest F1-score (67.14%). This setting strikes a balance between false alarms and detection effectiveness, making it suitable for real-world deployment.

To further illustrate the trade-off between precision and recall under varying malicious score thresholds, we plot a precision–recall (PR) curve using empirical data points from our threshold tuning experiments. As shown in Figure 3, each black point corresponds to a specific threshold setting, while the blue curve represents a polynomial regression fit that models the precision–recall relationship.

This curve demonstrates the inverse relationship between recall and precision—higher recall rates often come at the cost of reduced precision. The regression fit provides a smooth estimate of this trend, enabling more effective threshold selection strategies and improved interpretability in real-world deployments.

Thres	FPR	Precision	Recall	F1-score
10	92.68%	8.46%	100.00%	15.61%
20	86.78%	8.99%	100.00%	16.50%
30	58.39%	12.57%	97.92%	22.33%
40	40.18%	16.97%	95.83%	28.92%
50	27.32%	22.34%	91.67%	35.99%
60	11.07%	40.38%	87.50%	55.33%
70	5.18%	57.35%	81.25%	67.14%
80	0.53%	85.71%	37.50%	52.00%
90	0.00%	100.00%	16.67%	28.57%

Table 3: Performance at different malicious score thresholds.

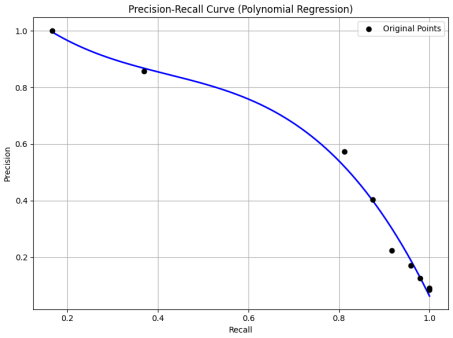

Fig. 3: Precision–Recall curve fitted over threshold-tuned data points.

5.2 Qualitative Evaluation

Case studies of correctly and incorrectly classified transactions To better understand the model's decision-making process and evaluate its strengths and limitations, we present several case studies. These examples cover both successful and failed predictions.

Correctly Classified Malicious Transaction. One case involves a transaction exploiting a `DELEGATECALL` vulnerability. The opcode trace includes a sequence of `CALLER`, `SLOAD`, `DELEGATECALL`, and `JUMPI` instructions, with unusually high gas usage clustered around memory manipulation. Our model correctly identified this segment with a high score (>95), emphasizing the contribution of both control-flow patterns and gas cost peaks.

Correctly Classified normal Transaction. Another example is a common ERC-20 token transfer, which utilizes standard opcodes such as `CALLVALUE`, `TRANSFER`, and `RETURN`, resulting in low and steady gas consumption. The model assigned a low score (<5), correctly identifying it as normal.

False Positive: Normal Transaction Misclassified as Malicious. In some cases, complex but legitimate smart contract patterns—such as multi-signature wallet operations or DeFi position adjustments—are mistakenly flagged as high-risk. For instance, a transaction with nested `SWAP` and `DUP` instructions, along with multiple storage writes, appeared structurally similar to attack patterns. Despite being legitimate, it received a high score, showing the model's over-sensitivity to intricate opcode compositions.

False Negative: Malicious Transaction Missed. A particularly challenging false negative case involved a long transaction with over 2000 chunks. While a few chunks exhibited high scores (>90) with suspicious use of some opcodes, the remaining chunks were normal. When aggregated using mean-based scoring, the overall transaction score fell below the detection threshold.

5.3 Ablation study in Impact of Opcode and Gas

Unlike prior works that often rely solely on high-level transaction metadata or simplified traces, BlockLens directly operates on execution sequences composed of low-level opcode tokens and aligned gas cost.

To evaluate the contribution of each component, we perform an ablation study by selectively removing gas or specific opcodes from the input sequence. Results are shown in Table 4.

Table 4: Ablation study: effect of gas embedding and opcode removal on detection performance.

Model Variant	**Top-10%**			**Top-15%**		
	FPR	Precision	Recall	FPR	Precision	Recall
BlockLens	3.75%	65.00%	81.25%	8.92%	45.05%	85.42%
w/o Gas	4.29%	60.00%	75.00%	9.11%	43.96%	83.33%
w/o POP	3.93%	48.33%	60.42%	10.36%	36.26%	68.75%
w/o JUMP	6.61%	38.33%	47.92%	11.07%	31.89%	60.42%

We observe that removing key opcodes such as `POP` or `JUMP` results in a substantial drop in both precision and recall. In particular, removing `JUMP` severely impairs detection at Top-10%, highlighting the critical role of opcode semantics.

Moreover, removing gas embedding leads to consistent performance degradation, particularly in terms of precision. This confirms that gas usage patterns provide meaningful signals for distinguishing malicious behaviors.

Model interpretability To enhance interpretability and support actionable security analysis, we integrate a local outlier detection strategy into our framework. We exploit the natural alignment between sliding-window chunk-level and token-level granularity in LLMs. Each transaction is divided into multiple overlapping segments (*chunks*) via a sliding window, and each chunk is independently passed through the model to produce a local malicious score.

These local scores can be aggregated using simple strategies such as *max*, *mean*, or *voting* to produce a final transaction-level decision. They also offer a natural mechanism for explainability: individual high-risk chunks highlight specific regions of the trace that contribute most to the malicious prediction.

This approach is efficient for detecting localized attacks, such as flash loan exploits or price manipulation, where the malicious behavior is confined to a short segment within a long and otherwise normal transaction. In these weak-signal scenarios, global averaging may dilute malicious signals, whereas chunk-level prediction preserves sensitivity to local patterns.

Moreover, from an engineering standpoint, this chunk decomposition provides direct answers to a common practical question in blockchain security: *"Why*

is this transaction considered malicious?" By inspecting which chunks receive high malicious scores, e.g., token ranges `[0, 2048]`, `[8192, 10240]`, security analysts can pinpoint specific operations (such as a suspicious `CALL` or `SSTORE`) that contributed to the model's decision.

This localized interpretability enhances model transparency and supports downstream tasks such as automatic alert prioritization, rule extraction, or transaction-level mitigation strategies.

5.4 Ablation study in Effectiveness of fine-tuning.

To evaluate the necessity of task-specific fine-tuning, we compare our model with some LLM baselines under zero-shot inference. We want to answer a question: *Can a sufficiently large, general-purpose LLM detect malicious transactions without any fine-tuning?* We conduct experiments using the following models:

- **LLaMA 3.2-1B (base)**: a non-instruction-tuned model. We use structured prompts to guide the model to produce meaningful anomaly scores.
- **LLaMA 3.1-8B-Instruct**: an instruction-tuned version of LLaMA 3 that supports chat-style interactions and better prompt following.
- **GPT-4o (OpenAI)**: a state-of-the-art LLM accessed via API for evaluating practical zero-shot inference with strong general reasoning capabilities.

Since most transaction traces are too long to fit into a single model input, we adopt a sliding window strategy to segment each trace into chunks. Each chunk is independently scored, and results are aggregated using max, mean, or majority voting. Each prompt follows this format:

Prompt and Example

You are an expert blockchain security analyst tasked with evaluating a blockchain transaction for potential anomalous or malicious behavior.
The following input is a transaction execution trace, including opcodes, operands, gas usage, addresses, and function signatures. Analyze the opcode sequence and execution trace provided. Identify if the opcode pattern matches known vulnerability or attack signatures (e.g., Reentrancy, Delegatecall exploits, Infinite loops, or unusual control flows).
Below is chunk {chunk_id} of a transaction. Provide your assessment logically. Assign an anomaly score ranging from 0 to 100, where:0 means completely normal, 100 means highly malicious. Clearly articulate the reasoning behind your assessment, structured logically.
Format your answer like this:
"Score: <number>"A numerical anomaly score between 0 and 100.
"Reason: <short explanation>step-by-step explaining the analysis."
In addition, we provide few-shot examples such as:
Code: CALLCODE DELEGATECALL SLOAD.....
Score: 70. Reason: Use of delegatecall with external input suggests reentrancy risk.

We also tested instruction-following models, including LLaMA 3.1-8B-Instruct and GPT-4o, by prompting them via a chat-style interface. Despite their advanced reasoning capabilities, both models exhibited similar over-sensitivity, frequently misjudging normal segments. Nearly all sequences received high scores (70–90) with explanations referencing generic features like `JUMP`, `SWAP`/`DUP` usage, or memory manipulation, even in completely normal transactions.

Zero-shot performance exhibited two major weaknesses. First, the models lacked domain knowledge. Despite detailed prompts and examples, they had not been trained on transaction data and were unable to distinguish between normal and malicious behaviors. Second, the models showed a tendency to overgeneralize, often misclassifying long transactions as malicious based on surface-level features, while missing deeper semantic cues. Due to the absence of instruction tuning, they frequently failed to follow the format.

5.5 Scalability and Deployment Considerations

Real-time Feasibility of LLAMA-based Fraud Detection To assess the practicality of deploying our model in real-world blockchain monitoring systems, we evaluate the end-to-end latency of our detection pipeline.

Upon receiving a transaction hash, the system immediately queries an EVM simulation backend to retrieve execution traces (`op trace`, `VM trace`, `call trace`). Our preprocessing pipeline, including trace simplification, tokenization, and chunking, is efficient and completes within milliseconds per transaction.

We benchmark our LLAMA-based detector on a single NVIDIA RTX 4080 SUPER GPU (16 GB memory). The prediction time for a single 2048-token chunk is approximately **0.27 seconds**, including embedding lookup, attention computation, classification, and decompilation for high-risk samples.

Given that most transactions can be processed in 1–4 chunks, we achieve near-real-time throughput, making it feasible to integrate into on-chain monitoring pipelines for high-frequency, high-stakes scenarios such as DeFi protocols.

Memory and Latency Constraints for Transaction Monitoring To evaluate the feasibility of deploying models for real-time Ethereum transaction monitoring, we conduct a series of system-level measurements, including inference latency and GPU memory usage under varying input and batch sizes. The experiments are conducted on a single NVIDIA RTX 4080 SUPER (16 GB) card.

Table 5: Latency(sec) and Memory Usage(MB) under Different Chunk Lengths

Chunk Length	Latency	Memory Usage
512	0.0168	4017.35
1024	0.0289	4300.61
1536	0.0416	4714.87
2048	0.0564	5259.13
4096	0.1177	8635.18

Latency vs. Chunk Length The inference latency increases approximately linearly with the input token length, as shown in Table 5. This is expected due to the quadratic complexity of attention operations in Transformer models.

Memory Usage vs. Chunk Length Table 5 illustrates how GPU memory usage grows with input length. For large-scale monitoring, selecting appropriate chunk lengths is crucial to prevent Out-of-Memory (OOM) errors.

6 Conclusion

In this paper, we present BlockLens, a novel LLM-based framework for detecting malicious transactions on the Ethereum blockchain. Our method introduces a new tokenization strategy that integrates opcode semantics and execution costs into a unified flattened modeling input. By fine-tuning a decoder-only LLaMA 3.2-1B model with Low-Rank Adaptation (LoRA), we enable efficient classification over long and complex transaction traces. Through evaluation against established baselines, our approach demonstrates strong recall and precision.

Furthermore, the chunk-level prediction architecture not only improves detection granularity but also enhances interpretability, allowing security analysts to localize and inspect specific segments contributing to malicious scores. Our results underscore the value of opcode-level modeling and domain-specific token engineering in transaction-level malicious detection.

Acknowledgment

We thank the contributors of open-source Ethereum datasets that supported our data collection and preprocessing. We also acknowledge the maintainers of the Meta LLaMA model [17] and the Huggingface ecosystem for enabling efficient model fine-tuning. We further appreciate the availability of the BlockFound [20] open-source codebase, which greatly supported our development.

References

1. Aldaham, T., HAMDI, H.: Enhancing digital financial security with lstm and blockchain technology. International Journal of Advanced Computer Science & Applications **15**(8) (2024)
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International conference on principles of security and trust. pp. 164–186. Springer (2017)
3. Brown, T., Mann, B., Ryder, N.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
4. Buterin, V.: Ethereum white paper: A next generation smart contract & decentralized application platform (2013), https://github.com/ethereum/wiki/wiki/White-Paper
5. Chowdhery, A., Narang, S., Devlin, J.: J. Mach. Learn. Res. **24**(1) (Jan 2023)

6. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. arXiv preprint arXiv:1904.05234 (2019)
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (Jun 2019). https://doi.org/10.18653/v1/N19-1423, https://aclanthology.org/N19-1423/
8. Ding, Y., Steenhoek, B., Pei, K., Kaiser, G., Le, W., Ray, B.: Traced: Execution-aware pre-training for source code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3608140, https://doi.org/10.1145/3597503.3608140
9. Gai, Y., Zhou, L., Qin, K.: Blockchain large language models (2023), https://arxiv.org/abs/2304.12749
10. Grattafiori, A., Dubey, A., Jauhri, A.: The llama 3 herd of models (2024), https://arxiv.org/abs/2407.21783
11. Hu, E.J., Shen, Y.: Lora: Low-rank adaptation of large language models (2021)
12. Ince, P., Luo, X., Yu, J., Liu, J.K., Du, X.: Detect llama - finding vulnerabilities in smart contracts using large language models. In: Information Security and Privacy: 29th Australasian Conference, ACISP 2024, Sydney, NSW, Australia, July 15–17, 2024, Proceedings, Part III. p. 424–443. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-981-97-5101-3_23, https://doi.org/10.1007/978-981-97-5101-3_23
13. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32. p. II–1188–II–1196. ICML'14, JMLR.org (2014)
14. Lu, S., Guo, D., Ren, S.: Codexglue: A machine learning benchmark dataset for code understanding and generation (2021), https://arxiv.org/abs/2102.04664
15. Mark Chen, J.T., Jun., H.: Evaluating large language models trained on code. ArXiv **abs/2107.03374** (2021), https://api.semanticscholar.org/CorpusID:235755472
16. Risse, N., Böhme, M.: Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection (2024), https://arxiv.org/abs/2408.12986
17. Touvron, H., Lavril, T., Izacard, G.: Llama: Open and efficient foundation language models. CoRR **abs/2302.13971** (2023), http://dblp.uni-trier.de/db/journals/corr/corr2302.html#abs-2302-13971
18. Vaswani, A., Shazeer, N., Parmar, N.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
19. Yang, L., Dong, X., Xing, S.: An abnormal transaction detection mechanim on bitcoin. In: 2019 International Conference on Networking and Network Applications (NaNA). pp. 452–457 (2019). https://doi.org/10.1109/NaNA.2019.00083
20. Yu, J., Wu, X., Liu, H.: Blockfound: Customized blockchain foundation model for anomaly detection (2024), https://arxiv.org/abs/2410.04039

Appendix A: Trace Extraction Demo

Trace Cleaning

There are three execution-level traces from the blockchain: Call trace, op trace, and VM trace.


```
"Call Trace":[
    {"from":"0x838b...8270","gas":"0x1835a","gasUsed":"0x2591","to":"0xcf2a...10f2","
 input":"0x69d48074","output":"0x0000...a6972cde3b","type":"STATICCALL"}, ...]

"Op Trace":[
        {"pc":0,"op":"PUSH1","gas":35577,"gasCost":3,"depth":1,"stack":[]}, ...]

"VM Trace":[
    {"cost":3,"ex":{"mem":null,"push":["0x80"],"store":null,"used":116472},"pc":0,"sub
 ":{},"op":"PUSH1","idx":"327-0"}, ...]
```


Trace Simplification

Op Trace Simplification. We begin by sorting all op trace entries by their program counter values to ensure temporal consistency. Operand extraction is guided by opcode semantics: `POP` retrieves the top stack value (i.e., the last element), `JUMPI` captures two parameters—`counter` and `condition`—from the top two elements, and `JUMP` uses only the top element as the `counter`. Arithmetic and logical operations such as `ADD`, `MUL`, `SUB`, `DIV`, `MOD`, `EXP`, `AND`, `EQ`, `LT`, and `GT` consume the top two stack elements, while three-operand instructions like `ADDMOD` and `MULMOD` use the top three. Single-operand opcodes, including `NOT`, `ISZERO`, `CALLDATALOAD`, `SLOAD`, and `TLOAD`, retain only the top element. For all other opcodes, operand extraction is deferred to the trace merging phase. Additionally, we annotate each opcode with its corresponding execution gas cost (`gasCost`), as gas patterns are useful, such as a malicious trace may contain unusually expensive instructions like a high-cost `CALL`, or repetitive low-cost operations such as `SLOAD`.


```
{"pc": 882, "op": "GAS", "depth": 1, "gasCost": 2}
{"pc": 883, "op": "CALL", "depth": 1, "gasCost": 5180926}
{"pc": 0, "op": "PUSH1", "depth": 2, "gasCost": 3}
{"pc": 2, "op": "PUSH1", "depth": 2, "gasCost": 3}
```


Virtual Machine Trace Simplification. Each trace entry is simplified to contain `pc`, `op`, stack manipulation (`push`) and persistent storage interactions (`store`).

To represent hierarchical contract calls, we annotate each opcode with a call `depth`, initialized to 1. The depth is incremented whenever encountering opcodes indicating nested calls (`CALL`, `STATICCALL`, `DELEGATECALL`, `CREATE`, or `CREATE2`). Within each nested call structure (`sub`), the depth increments accordingly, clearly preserving the hierarchical execution context. Once a nested call concludes, depth returns to the previous level. The final simplified format is:


```
{"ex":{"mem":[],"push":["0x504f39"],"store":[]},"pc":882,"sub":[],"op":"GAS","depth":1}
{"ex":{"mem":[],"push":["0x1"],"store":[]},"pc":883,"sub":[],"op":"CALL","depth":1}
{"ex":{"mem":[],"push":["0x80"],"store":[]},"pc":0,"sub":[],"op":"PUSH1","depth":2}
{"ex":{"mem":[],"push":["0x40"],"store":[]},"pc":2,"sub":[],"op":"PUSH1","depth":2}
```

Call Trace Simplification. We simplify it by flattening nested call structures into a sequential list. Each nested call is annotated with an execution depth, which is initialized to 1. For each level of nested calls, the depth increments by 1.

```
{"from": "0x81..", "to": "0x97..", "input": "0x63..", "type": "CALL", "depth": 0}
{"from": "0x97..", "to": "0xc1..", "input": "0x42..", "type": "CALL", "depth": 1}
{"from": "0xc1..", "to": "0x66..""input": "0x47..", "type": "DELEGATECALL", "depth": 2}
```

Trace Merging

Merge Op Trace and VM Trace. The merging process follows several heuristic rules to enrich opcode semantics with runtime information. For `PUSH` and `DUP`, we extract operand values from the VM trace's `push` field and record them as `value`. For arithmetic and logical operations such as `ADD`, `EQ`, and `LT`, only the computation result from the `push` field is retained as `result`. In the case of `SWAP`, we record the first and last stack elements from the VM trace `push` as `value1` and `value2` respectively. For `CALLDATALOAD`, both the offset (from the op trace) and the corresponding data value (`data[i]`) are stored. Opcodes that access blockchain state, such as `CALLER`, `BALANCE`, or `GAS`, have their associated address or numeric values extracted from the `push` field. Memory operations like `MSTORE` and `MLOAD` include values and memory offsets retrieved from `mem.data` and `mem.off`, with long hexadecimal strings simplified when possible. For storage operations such as `SSTORE` and `TSTORE`, both the `key` and `value` fields from the `store` field are preserved. All other opcodes are left in their original form without further augmentation. The result is as follows:

```
{"pc": 882, "op": "GAS", "depth": 1, "value": "0x504f39"}
{"pc": 883, "op": "CALL", "depth": 1}
{"pc": 0, "op": "PUSH1", "depth": 2, "value": "0x80"}
{"pc": 2, "op": "PUSH1", "depth": 2, "value": "0x40"}
```

Merge with Call Trace. To integrate high-level contract call semantics, we further align the merged_trace_1 with the flattened call trace to be the merged_trace_2. Each call-related opcode (`CALL`, `STATICCALL`, etc.) is matched sequentially with entries in call trace based on both opcode type and depth.

Format into Final Trace. To prepare the final trace format for model input, we remove structural metadata such as `pc` and `depth`. Field names are stripped, leaving only raw opcodes and operands arranged sequentially as plain text. For `JUMPDEST` instructions, we retain the original `pc` value as a label, e.g., `JUMPDEST 0x38`, to preserve jump targets. In arithmetic and logical operations, only the computation `result` is kept to reduce redundancy. For external calls, the `input` field is truncated to its first 10 bytes, corresponding to the function selector. In contrast, for contract creation instructions like `CREATE` and `CREATE2`, the `input` field is discarded entirely.

And the final trace ready for subsequent processing is presented below:

```
"transaction_id": "12345",
"input": "START FROM 0x81.. TO 0x97.. INs 0x63.. VALUE 0x0 OPTRACESTART .. POP 0x0 GAS 0
 x504f39 CALL 0x97.. 0xc1.. 0x42.. 0x0 PUSH1 0x80 PUSH1 0x40 MSTORE 0x80 64 .. END",
"label": "1"
```