Blockchain Large Language Models

Yu Gai†, Liyi Zhou‡, Kaihua Qin†, Dawn Song‡, and Arthur Gervais§
†University of California, Berkeley, ‡Imperial College London, §University College London,
Berkeley Center for Responsible, Decentralized Intelligence (RDI)

Abstract—This paper presents a dynamic, real-time approach to detecting anomalous blockchain transactions. The proposed tool, BLOCKGPT, generates tracing representations of blockchain activity and trains from scratch a large language model to act as a real-time Intrusion Detection System. Unlike traditional methods, BLOCKGPT is designed to offer an unrestricted search space and does not rely on predefined rules or patterns, enabling it to detect a broader range of anomalies.

We demonstrate the effectiveness of BLOCKGPT through its use as an anomaly detection tool for Ethereum transactions. In our experiments, it effectively identifies abnormal transactions among a dataset of 68M transactions and has a batched throughput of 2284 transactions per second on average. Our results show that, BLOCKGPT identifies abnormal transactions by ranking 49 out of 124 attacks among the top-3 most abnormal transactions interacting with their victim contracts. This work makes contributions to the field of blockchain transaction analysis by introducing a custom data encoding compatible with the transformer architecture, a domain-specific tokenization technique, and a tree encoding method specifically crafted for the Ethereum Virtual Machine (EVM) trace representation.

I. INTRODUCTION

With the increasing number of transactions per second processed by blockchains, a rich real-world dataset of user behavior and Decentralized Application (DApp) interactions has become accessible across the globe. For the first time in history, the information security community can access a transparent, timestamped, and non-repudiable dataset of transactions, including their dynamic smart contract execution traces. This dataset also contains attack transactions that have caused multi-million-dollar losses. Between April 30, 2018, and April 30, 2022, users, liquidity providers, speculators, and protocol operators in blockchain networks suffered a total loss of at least 3.24 billion USD [1]. These significant losses underscore the need for more generic, dynamic and scalable approaches to detect anomalous blockchain transactions, especially as the volume of transaction data continues to grow.

Real-time Intrusion Detection System (IDS) for blockchain transactions, however, remains challenging due to constraints on the search space and the substantial manual engineering efforts required. More specifically, State-of-the-Art (SOTA) works predominantly employ either (i) reward-based approaches, which focus on identifying and exploiting transactions that yield significant profits, or (ii) pattern-based techniques that depend on custom heuristics to deduce blockchain transaction intents and user address behavior (cf. Figure 2). However, the reliance on predefined rules, patterns, or profitable vulnerabilities may prevent these methods from capturing the full spectrum of anomalies. Consequently, there is an urgent need for more versatile and adaptive techniques that can effectively identify a wide range of anomalous transactions in real-time, enhancing the security of blockchain networks.

When exploring the different means to perform anomaly detection, it becomes apparent that a blockchain transactions’ trace outlines what a transaction does: the invoked smart contracts, the corresponding parameters, the order of invocations and storage information. Given that an attack transaction often executes a different path and exhibits different execution behaviors than normal transactions, we conjecture that an attack transaction portrays a different trace representation than benign transactions. Thus, we hypothesize that it is possible to train a model to learn representations of transaction execution traces without any prior knowledge of vulnerability patterns (e.g., reentrancy, price oracle manipulation, etc.) or information on transaction profitability. Certainly, there may also be limitations to such an approach, such as the number of false positives — we will also explore the efficacy of this approach with suitable detection or anomaly thresholds.

In this work, we leverage Ethereum as an open dataset to train a transaction anomaly ranking tool, BLOCKGPT. The main objective of BLOCKGPT is to elevate the art of blockchain security analysis into a comprehensive and real-time endeavor. By generating efficient trace representations of blockchain activity, BLOCKGPT trains a scalable large language model from scratch using an extended and aug-
mented dataset from a previous study, covering attacks within a four-year timeframe. This entire dataset comprises a total of 68M transactions, encompassing all transactions from 124 smart contracts that experienced an attack during the specified period. We evaluate the effectiveness of BLOCKGPT as a transaction anomaly ranking tool on the same dataset, assessing its performance in identifying anomalous transactions.

We evaluate BLOCKGPT using two simple metrics: (i) a percentage ranking alarm threshold; and (ii) an absolute ranking alarm threshold. BLOCKGPT successfully identifies abnormal transactions, ranking 49 out of the 124 attacks among the top-3 most abnormal transactions interacting with their corresponding victim contracts. Specifically, BLOCKGPT detects 20 adversarial transactions as the most abnormal, 20 as the second most abnormal, and 7 as the third most abnormal transaction associated with their victim contracts.

Regarding the practicality of BLOCKGPT, our analysis highlights its effectiveness when dealing with many transactions. For example, in DeFi applications with over 10,000 transactions and a 0.01% alarm threshold, BLOCKGPT successfully detects 24% of the attacks and maintains an average False Positive Rate (FPR) of 0.097% (cf. Figure 4). In the context of popular Decentralized Finance (DeFi) applications processing 100 transactions per day, a 0.1% FPR generates one alert approximately every 10 days. This demonstrates that BLOCKGPT is capable at providing a manageable number of alerts for further investigation, making it particularly suitable for high-volume transaction environments.

In terms of performance, BLOCKGPT showcases its real-time capabilities, achieving an average batched throughput of 2,284±289 transactions per second, and on average 0.16±0.3 seconds to rank a single transaction. This rapid detection of malicious blockchain transactions enables the triggering of a smart contract pause mechanism to prevent an attack as an Intrusion Prevention System. Approximately 50% of the attacked contracts we investigate already have such a pause mechanism deployed.

In summary, this work makes the following contributions to the field of blockchain transaction analysis:

- To our knowledge, this paper is the first to apply unsupervised/self-supervised learning for anomaly detection in smart contract transaction execution traces. We develop a large language model for Ethereum transaction anomaly detection, employing custom data encoding, domain-specific tokenization, and a tree encoding method tailored for EVM trace tree representation, capturing calls, function names, parameters, and storage modifications.
- We apply BLOCKGPT as an anomaly detection tool for Ethereum transactions to identify suspicious or malicious activities on the blockchain. We evaluate BLOCKGPT on a dataset of 124 attacks, consisting of a total of 68M transactions, spanning a period of 1,523 days, starting from block \([5,470,817]\) (19th April, 2018) and ending at block \([15,000,000]\) (21st June, 2022). We analyze the model across various dataset sizes and metrics, including two alarm thresholds, F1-score, F10-score, and CID score, demonstrating BLOCKGPT’s robustness and versatility. We benchmark BLOCKGPT against doc2vec and trace length heuristics, showcasing the superior effectiveness of BLOCKGPT.
- Evaluation results indicate BLOCKGPT effectively identifies abnormal transactions and can detect different types of malicious activities, as shown through a flash loan attack case study. With a throughput of 2284±289 transactions per second, our tool is a viable real-time IDS for blockchains.

II. BACKGROUND

A. Blockchain and DeFi

1) Blockchain: Since the inception of blockchains with Bitcoin in 2008 [2], it became apparent that their most well-suited use case is the transfer or trade of financial assets without trusted intermediaries [3]. A blockchain is considered permissionless when entities can join and leave the network at any time. Users authorize transactions through a public key signature and a subsequent broadcast on the blockchain peer-to-peer (P2P) network. Due to the openness of the P2P network, the information about a transaction becomes public, once a transaction is broadcast. For example, blockchain participants can observe which smart contract a pending transaction calls triggers along with the corresponding call parameters. Miners accumulate unconfirmed transactions and solve a Proof of Work (PoW) puzzle to append blocks to the blockchain. Various alternatives to PoW, such as Proof of Stake [4], [5] emerged. In addition to the block reward and transaction fees, Blockchain Extractable Value is a new miner reward source [6]. For a more thorough blockchain background, we refer the reader to SoKs [7]–[9].

2) Smart Contracts: While Bitcoin supports basic smart contracts through a stack-based scripting language, the addition of support for higher-level programming languages (e.g., Solidity) has resulted in widespread adoption. Note that SOTA blockchains generally require transaction fees as in to prevent Denial-of-Service attacks. Smart contracts are therefore only quasi Turing-complete because their execution can suddenly interrupt if the transaction fees exceed a predefined amount. Notably, blockchains do not store the human-readable source code, nor the application interface to interact with a smart contract (i.e., Application Binary Interface). Instead, SOTA blockchains only store the compiled bytecode on-chain.

The execution of a smart contract’s bytecode is triggered by blockchain transactions, which are then carried out within so-called virtual machines (e.g., EVM). Similar to traditional programming languages such as Java, the execution of smart contracts can be summarized with execution traces (also known as logs), which record the state transition at each step of the process. While full blockchain clients store the entirety of the historical blocks, intermittent states and execution traces are typically discarded due to excessive storage requirements. So-called archive nodes, however, store and provide an indexed database of historical smart contract executions.

3) DeFi: DeFi refers to an ecosystem of financial products and services built on top of permissionless blockchains. DeFi is currently experiencing a surge in popularity, with a peak
Despite incorporating basic functions inspired by traditional finance (e.g., lending, trading, derivatives and asset management), DeFi also introduces more novel designs (e.g., flash loans [10], automated market makers [11] and composable trading [12]) enabled by a blockchain’s atomicity property and DeFi’s composable nature. Understanding the semantics of transactions that trigger these novel DeFi designs presents a particular challenge because DeFi transactions typically are intertwined with multiple financial DApps.

B. Natural Language Processing (NLP)

Natural language models are designed to process and generate human-like text. They are used in a wide range of applications, including language translation, text summarization, language generation, and text classification.

1) NLP and Bytecode: There have been several approaches to applying natural language models to code, assembly, and bytecode. One approach is to treat the code or assembly as natural language text and input it into a natural language model. This can be useful for tasks such as code summarization, code generation, or code translation.

Another approach is to first convert the code or assembly into a structured representation, such as an abstract syntax tree (AST), and then input the AST into a natural language model. This can allow the model to better understand the structure and meaning of the code, and can be useful for tasks such as code completion or code formatting.

Bytecode, which is a low-level representation of code that is typically executed by a virtual machine, can also be processed using natural language models. One approach is to convert the bytecode into a higher-level representation, such as assembly code, and then input it into a natural language model. Another approach is to treat the bytecode as a sequence of tokens and input it into a sequence-to-sequence natural language model, which can be useful for tasks such as bytecode translation or bytecode summarization.

2) Embeddings in NLP: An embedding is a dense, continuous-valued vector representation of a word or token. It is used to represent the meaning of the word or token in a numerical form that can be input into a machine learning model. Embeddings are commonly used in NLP to represent words or tokens in a way that captures the semantic meaning and relationships between the words.

There are several reasons why embeddings are useful in NLP. One reason is that they allow the model to handle large vocabularies more efficiently, as the model does not have to learn separate weights for each word in the vocabulary. Another reason is that embeddings can capture the relationships between words, such as synonymy and analogy, which can be useful for tasks such as language generation or translation. Finally, embeddings can also improve the generalization performance of the model by allowing it to handle out-of-vocabulary words or words that were not seen during training.

III. BlockGPT Overview

We begin by outlining the system and threat models before providing an overview of the key components of our proposed solution, which we refer to as BlockGPT.

A. System Model

Our system model considers a blockchain ledger that employs smart contracts and cryptocurrency assets, enabling traders and attackers to conduct transactions across various DeFi platforms such as exchanges, lending, leveraging. In this study, we specifically focus on EVM-based blockchains.

- **Transaction and State Transition**: A blockchain ledger functions as a state machine replication, with its state denoted by \( S \). Users define financial operations within a blockchain transaction, represented as \( tx \), to request state transitions on the blockchain. The transaction serves as a state transition function that alters the ledger’s state from \( S \) to \( S' \). In other words, \( S' = tx(S) \).

- **Smart Contract**: A smart contract is a piece of code that translates into one or more state transition functions, which can be activated by a transaction. Smart contracts can also trigger functions of other contracts.

- **Blockchain Nodes**: A blockchain node may be assigned to one or more tasks: (i) transaction sequencing, determining the order of transactions within a block; (ii) block generation; (iii) data verification; and (iv) data propagation. The two prevalent types of blockchain nodes are:
  - **Sequencer nodes**, also referred to as miners in PoW blockchains, validators in PoS blockchains, and block builders in PBS, encompass all four responsibilities mentioned above. Sequencers can insert, omit, and reorder transactions in the blocks they create within the boundaries set by the protocol.
  - **Ordinary nodes** solely handle blockchain data propagation and might also perform data verification.

- **Transaction Propagation**: There are primarily two methods for propagating transactions from the transaction generator to the sequencer nodes:
  - **Public Propagation**: Blockchain network protocols generally guide nodes on discovering and connecting to other nodes within the peer-to-peer (P2P) network. Transactions can be disseminated in the P2P network from the transaction generator, to the corresponding sequencer nodes, through ordinary nodes.
  - **Private Propagation**: Front-running as a Service (FaaS) services enable DeFi traders to submit transactions directly to sequencer nodes, bypassing a broadcast on the P2P network. FaaS may not be available on certain blockchains (e.g., Binance Smart Chain and Avalanche) and may be the only option on chains with a sole sequencer (e.g., Optimism). Ethereum presently supports both P2P and FaaS propagation.

- **Transaction Execution Trace**: A transaction execution trace documents the sequence of actions and state changes resulting from processing a transaction. Transaction traces
can be represented in various ways, such as showcasing low-level OP codes or high-level DeFi operations.

B. Threat Model

We consider a computationally bounded adversary (denoted by $A$) which is capable of executing transactions (i.e., performing actions) across a set of DeFi platforms. $A$ may exploit vulnerabilities, in an attempt to alter a DeFi protocol’s designed, expected state transition. Our threat model captures two different types of adversaries based on their capabilities:

- **Observable Adversaries**: These adversaries do not have the capability to hide their pending transactions from BLOCKGPT. Their transactions are observable to our system either by (i) broadcasting on P2P such that anyone can observe; or if (ii) BLOCKGPT is used by sequencer nodes, so even if the adversary uses private propagation (FaaS), the sequencer will still be able to observe adversarial transactions.

- **Hidden Adversaries**: These adversaries are capable of hiding their transactions from BLOCKGPT until the adversarial transactions are finalized. Hidden adversaries achieve this by using a FaaS system or similar method.

C. BLOCKGPT Overview

BLOCKGPT consists of three main components: a transaction tracer, a training module, and a detection module. The intuition behind this design choice is that normal transactions and attack transactions have different execution paths, and a powerful model can identify attack transactions based on their unusual execution paths.

1) The transaction tracer captures the execution trace of a transaction initiated by a user interacting with a DeFi protocol. The trace includes the sequence of smart contract function calls, the associated input and output data, and provides a detailed view of the execution path. This trace is used as input for the detection module.

2) The training module uses a dataset of historical transactions to train a model using unsupervised or self-supervised methods. The goal of the training phase is to learn a model that can differentiate between normal and abnormal transaction execution traces. This allows BLOCKGPT to identify anomalies in real-time transactions.

3) The detection module applies the trained model to new transaction traces to generate a score based on the likelihood of the trace. A ranking or threshold-based method is then used to raise alarms for transactions with abnormal scores. This allows BLOCKGPT to detect potential attacks on the blockchain in real-time and prevent them from causing harm to the DeFi platform.

D. Motivating Examples

We present two real-world examples that illustrate the potential benefits of using BLOCKGPT in the DeFi ecosystem.

- **Motivating Example 1 (Observable Adversary)**: Consider the attack on the Beanstalk project in April 2022. In a single transaction, the attacker borrowed one billion USD in cryptocurrency assets through Aave’s flash loan, exchanged the borrowed assets for a 67% stake in the Beanstalk project, and subsequently passed their proposal to withdraw the entire treasury. Etherscan received the adversarial transaction $0xc3d313ad7$ on the public P2P network approximately 30 seconds before its block confirmation, indicating that the attacker was an observable adversary.

Table 5 shows that our evaluation ranks the adversarial transaction as the most abnormal among all historical interactions with the Beanstalk victim contract. This suggests that, if Beanstalk had utilized BLOCKGPT alongside a well-connected blockchain node within the P2P network, it would have had a 30-second window to detect and respond to the attack. In reaction to such an attack, Beanstalk could preemptively counter the adversary by initiating an emergency withdrawal of user funds or enforcing an emergency pause. This example demonstrates how BLOCKGPT can enable DeFi protocols to proactively detect and prevent malicious activities, ensuring the security and integrity of users’ assets.

- **Motivating Example 2 (Hidden Adversary)**: Consider the attack on the Revest Protocol in March 2022, during which approximately two million USD worth of tokens were stolen in four transactions. The root cause of the attack was a reentrancy vulnerability in a minting contract. The attack comprised four transactions confirmed between block $14465357$ and block $14465427$, spanning roughly 70 blocks over a period of about 17 minutes. All transactions were propagated through a FaaS relay, specifically Flashbots.

Since the adversarial transactions were not visible on the P2P network until they were confirmed, the attacker would be considered a hidden adversary. In this scenario, BLOCKGPT could only act as a fast retrospective attack detection tool. If BLOCKGPT detected the first adversarial transaction upon receiving block $14465357$ and block $14465427$, it could have potentially prevented the other three attacks. Our evaluation found that BLOCKGPT ranked the first adversarial transaction in Revest Finance as the most abnormal transaction. This example underscores the importance of having a tool like BLOCKGPT, even if it is unable to observe pending transactions.

IV. BLOCKGPT DETAILS

We now proceed to elaborate on the details of BLOCKGPT.

A. BLOCKGPT Components

BLOCKGPT generates transaction representations for anomaly detection of blockchain transactions in real-time (cf. Figure 2). As such, BLOCKGPT takes as input a blockchain transaction (e.g., from the transaction pool), transforms it into a vector, extracts a vocabulary and applies a transformer in an effort to learn a probability distribution on an entire transaction’s blockchain trace. BLOCKGPT’s design consists of the following six components (cf. Section [IV] for details).

- **Intermediate Trace Representation (ITR) Construction**: Given a transaction, we construct a tree structured trace.
This trace captures not only the smart contract function call dependencies, but also finer granularity information such as state accesses and event logs. Each node in the ITR tree can be thought of as a step in a transaction execution process. An ITR node may consist of a stream of bytes, which are further split into ITR tokens in step 2 (cf. Fig. 1 for an example of an ITR construction).

**2 ITR Tokenization:** Given domain specific heuristics, we first split all ITR nodes into tokens, which then inform a custom vocabulary, drawn by the frequency distribution of those ITR tokens. An ITR token, for instance, can be a function name, event name, parameter type, parameter value, etc. To satisfy the requirements of step 3 and 4, we preprocess the ITR trace according to our custom grammar.

**3 Vocabulary Embedding:** Vocabulary embedding is a technique used to convert words or tokens in a text to numerical vector representations. It is commonly used in NLP to input text data into machine learning models. To perform vocabulary embedding in the given process, a one-hot encoding function is first applied to the ITR tokens, which converts the tokens into a binary vector representation of either 0, which converts the tokens into a binary vector representation.

The resulting vector representation of each token is the sum of three different embeddings: $E_{tok}$, $E_{tree}$, and $E_{ctx}$. $E_{tok}$ is the unique embedding of the token, which represents the token itself. $E_{tree}$ is an embedding that represents the position of the function call involving the token in the call graph, which forms a tree structure. $E_{ctx}$ is an embedding that represents the context in which the token appears, such as whether an address appears as a “from” or “to” address. By combining these three embeddings, the vocabulary embedding technique can capture both the unique characteristics of the tokens, their context within the trace, and their purpose. This can help the machine learning model to better understand the meaning and relationships between the tokens in the trace.

**4 Transformer Encoder:** A processed transaction trace is initially converted using the prior vocabulary embedding layer, which associates the words or tokens in the trace with a dense vector representation. This representation is then fed into this step involving a transformer-based language model. As transformer-based models employ attention mechanisms to discern the relationships between input tokens, they necessitate a positional encoding to comprehend the relative location of each token in the input. It is important to consider that the input here is a transaction trace, which takes the form of a tree structure rather than a linear sequence of characters. Consequently, the graph structure is supplemented with positional tokens to preserve the tree information while providing the encoding to the transformer model. This enables the model to utilize its multi-head attention mechanism to learn the intricate relationships among the input tokens.

**5 Loss Minimization:** We apply the following steps to minimize the causal language modeling loss. We first define a causal language model loss function, which measures the difference between the predicted probability distribution of the next trace token in the sequence and the actual probability distribution of the next token. We then compute the gradient of the loss function with respect to the model parameters. We can then update the model parameters using the gradients and a learning rate with gradient descent. We repeat the above steps until the loss has reached a satisfactory minimum or a maximum number of iterations has been reached.

**6 Ranking-based intrusion detection** We adopt a ranking-based approach to intrusion detection. Given a DeFi application, our IDS ranks all transactions involving the application by the log-likelihood of their traces as computed by BLOCKGPT, and raises an alarm for the transactions with $\alpha\%$ lowest log-likelihood, i.e. the most abnormal transactions. The cost of running the IDS can be adjusted by controlling the parameter $\alpha$. Out of 124 attacks in the dataset, BLOCKGPT identified 20 transactions as the most abnormal, 20 transactions as the second most abnormal, and 7 transactions as the third most abnormal.
B. ITR Construction

It is difficult to develop a scalable, just-in-time security system using the typical approach of performing execution analysis directly with low-level opcode traces because of the large space and time cost involved [13]. Call traces and other high-level representations do not capture sufficient runtime information to allow for the generation of precise trace embeddings [14]. By creating a new trace tree, which we refer to as an ITR, we overcome the fundamental shortcomings of past techniques. ITR is a high-level function that combines the three traces that follow into a single tree structure.

1) During the execution of a transaction \( tx \), the call trace captures the function call dependencies that occur. When a smart contract function calls into another function, the call trace records such inter- and intra-contract calls within its trace. We use a directed tree \( \text{Tree}_\text{call}(tx) = (T\_\text{call}, E\_\text{call}) \) to represent a call trace. Each node \( t \in T\_\text{call} \) in \( \text{Tree}_\text{call}(tx) \) corresponds to an executed function call, including its corresponding runtime encoded call data and return data in bytes. We use the notation \( \text{t}_\text{call} \rightarrow \text{t}_\text{call}, e \in E\_\text{call} \) to denote the edges in \( \text{Tree}_\text{call}(tx) \), where a function \( \text{t}_\text{call} \) (i.e., callee) is triggered by another function \( \text{t}_\text{call} \) (i.e., caller).

2) Smart contracts can access and modify the volatile memory and persistent storage of a blockchain. A state trace records any read and write operations to the persistent storage that may occur during the execution of a transaction \( tx \). In particular, a state trace consists of two sequences, each of which captures state accesses and state changes during the execution of \( tx \).

We denote a state trace of a transaction \( tx \) with \( S\_\text{seq}_\text{state}(tx) = (T\_\text{state}, E\_\text{state}) \). The first sequence \( T\_\text{state} = [t^1, \ldots, t^n] \) consists of state reads and writes details. If \( t^i \) is a read operation, then it consists of a value tuple \([\text{key}, \text{val}]\), meaning that a function reads value \( \text{val} \) from global storage (i.e., account storage) at position \( \text{key} \) during \( tx \)’s execution. Similarly, if \( t^i \) is a write operation, then it consists of a two value tuple \([\text{key}, \text{val}]\), meaning that a function overrides the value at global storage position \( \text{key} \) with \( \text{val} \) during the execution of the function call. The second sequence \( E\_\text{state} = [t^1 \xrightarrow{\text{read}} t^1, \ldots, t^n \xrightarrow{\text{write}} t^n] \) captures which function call reads or writes the state.

3) A log trace is a sequence of variables that the smart contract developer chooses to expose at runtime. Logs help developers during contract debugging and extended data analytic tasks.

Similar to the state trace, we use \( S\_\text{seq}_\text{log}(tx) = (T\_\text{log}, E\_\text{log}) \) to denote a log trace. The first sequence \( T\_\text{log} = [t^1, \ldots, t^n] \) contains all emitted smart contract events. Each \( t^i \) consists of a contract address, an event’s hash identifier, and the corresponding encoded data in bytes. The second sequence \( E\_\text{log} = [t^1 \xrightarrow{\text{read}} t^1, \ldots, t^n \xrightarrow{\text{write}} t^n] \) captures which function call emits each log.

Construction: We construct the ITR tree by traversing through state and log traces. We convert each state and log trace element into a child leaf node, and append the leaf node to \( \text{Tree}_\text{call}(tx) \) (cf. Figure 3).

\[
\text{Tree}_\text{ITR}(tx) = (T\_\text{ITR}, E\_\text{ITR}) = (T\_\text{call} \cup T\_\text{state} \cup T\_\text{log}, E\_\text{call} \cup E\_\text{state} \cup E\_\text{log})
\]

C. ITR Tokenization

In NLP, tokenization is a common approach to identifying words that constitute a natural language sentence. Similarly, we apply tokenization to transform an ITR trace tree to token sequences. We formally define an ITR token as a string of arbitrary length. A token can, for example, represent a blockchain address, a function name, a log message, a storage key, a value, or a value type. We define our tokenization function \( f\_\text{token}(\cdot) \), taking as input either a call, state, or log ITR node. In our grammar, the first two tokens of every node must be [START] followed by either one of the following three injected tokens: ([CALL], [STATE], [LOG]). The last token of every node must be [END]. In addition, we differentiate the start of input and output parameters with two added tokens ([INs], [OUTs]).

- \( f\_\text{token}(T\_\text{call}) = [\text{START}], [\text{CALL}], \text{from}, \text{to}, \text{function hash}, \text{gas}, \text{value}, \text{[INs]}, \text{input1 type}, \text{input1 value}, \ldots, \text{[OUTs]}, \text{output1 type}, \text{output1 value}, \ldots, [\text{END}] \)
- \( f\_\text{token}(T\_\text{state}) = [\text{START}], [\text{STATE}], \text{read} / \text{write}, \text{key}, \text{val}, [\text{END}] \)
- \( f\_\text{token}(T\_\text{log}) = [\text{START}], [\text{LOG}], \text{contract address}, \text{event hash}, \text{value1 type}, \text{value1}, \ldots, [\text{END}] \)

We also choose to preprocess numerics to capture only the first two significant figures and the scale of numbers rather than the precise amounts (e.g., 1254 \( \rightarrow \) 1300). This is necessary to avoid vocabulary explosion because smart contracts frequently operate with big integers beyond 18 decimals.

a) Break-Down of the Tokens: We construct a vocabulary of 100k tokens, as further elaborated in the evaluation, out of
which 93,233 are Ethereum addresses and 6,759 are smart contract function signatures. Tokens which do not appear in our vocabulary are replaced with the [OOV] token.

D. Local Token Embedding

A local token embedding is the sum of three embeddings that respectively encode the identity, function call position, and local context of a token. For example, BlockchainGPT tokenizes the function call `addr1 -> addr2.func()` at the root of a call tree into three tokens: `[addr1]`, `[addr2]` and `[func]` (henceforth all tokens are enclosed by brackets). The local embeddings of the tokens `[addr1]` and `[addr2]`, both blockchain addresses, are \( E_{[addr1]} + E_{root} + E_{src} \) and \( E_{[addr2]} + E_{root} + E_{dst} \) respectively, where the embedding \( E_{root} \) indicates that the position of the function call is at the root of the call tree, and the embedding \( E_{src} \) and \( E_{dst} \) are two local context embeddings added to the local embeddings of all source and destination addresses, without which the transformer encoder cannot possibly distinguish the role of the two addresses in the function call. The local embedding of the token `[func]`, a function signature, is \( E_{[func]} + E_{root} + E_{func} \). The local context embedding \( E_{func} \) is hardly necessary, as a function signature in general can occur only in one place in each function call. We can similarly generate local embeddings for function call parameters. All embeddings so far are simply vectors retrieved from a lookup table.

In the example above, we use the embedding \( E_{root} \) from a lookup table to indicate that a function call occurs at the root of a call tree. To faithfully encode the position of each function call, we need to associate a unique embedding to each function call position in a call tree. Without loss of generality, we binarize all call trees into binary trees, with each function call being a node in the tree. As a binary tree of depth \( d \) has \( O(2^d) \) unique positions, it is infeasible to construct a lookup table that stores the embedding of each function call position. However, each node in a binary tree can be uniquely identified by the path that leads to the node starting from the root of the tree, which is equivalent to a sequence of binary actions whether to visit the left \((L)\) or right child \((R)\). The embedding of a node can thus be the sum of embeddings of the actions taken at each step. Mathematically, the embedding of the action sequence \( b_1, \ldots, b_n \in \{L, R\}^n \) is given by \( \sum_{i=1}^{n} E_{i,b_i} \), where

\[
E_{i,b_i} = \begin{cases} 
E_{i,L} & b_i = L \\
E_{i,R} & b_i = R 
\end{cases}
\]

Intuitively, \( E_{i,b_i} \) indicates that the action \( b_i \) is taken at step \( i \). This approach only requires a lookup table consisting of the embeddings \( E_{1,L}, E_{1,R}, \ldots, E_{D,L}, E_{D,R} \) where \( D \) is the chosen maximum depth. Empirically, we did not observe the summation leading to any numerical instability during training, likely due to layer normalization in the transformer encoder. Notably, the maximum depth of call trees that this embedding scheme can handle is not bounded by the embedding dimension, which differs from prior works [15].

E. Contextual Token Embedding and Generative Pre-Training

Given a collection of local token embeddings, a transformer encoder can yield a collection of contextual embeddings for each token, which can carry context-dependent meanings that depend on the objective that they are tuned to optimize. For a collection of tokens that is sequentially ordered as \( x_1, \ldots, x_n \), we maximize as its objective the log-likelihood of the collection that is factorized as \( \log p(x_1, \ldots, x_n) = \sum_{i=1}^{n} \log p(x_i | x_{<i}) \), where the context \( x_{<i} \) is \( x_1, \ldots, x_{i-1} \) for \( i > 1 \), and the context \( x_{<i} \) is a special symbol representing an empty sequence. Assuming that the local embeddings encode the position of each token, as is the case above, we can designate the contextual embedding of the token \( x_{i-1} \) to represent the context \( x_{<i} \), and transform it into a categorical distribution over the vocabulary, which gives the probability of the next token \( x_i \) occurring.

One way to transform an embedding into a categorical distribution over a vocabulary of size \( n \) is to apply a linear transform followed by soft-max. Let \( A \) be a \( n \times d \) matrix, the probability of a token \( [tok] \) occurring given the context \( x_{<i} \) is given by

\[
p([tok] | x_{<i}) = \frac{\exp(s(#[t tok]))}{\sum_{j=1}^{n} \exp(s(j))}
\]

where \( #[t tok] \) is an integer between 1 and \( n \) that is uniquely assigned to the token \([t tok]\). The notation \( s(j) \) denotes the \( j \)th component of the vector \( s \), and \( s(#[t tok]) \) thus denotes the \( #[t tok] \)th component of \( s \).

The factorization, however, mandates the transformer encoder to derive the contextual embedding of each token from a different context, namely, the token itself and all the tokens that precede the token in the sequence. Fortunately, with proper attention-masking, we can proceed as if computing the contextual embedding of each token with the entire sequence as context, which is crucial when applying transformers to large-scale datasets.

Since a transformer works with any collection of tokens with a sequential ordering, we can apply it to nodes in call trees by linearizing call trees with breadth-first traversal. We leave other linearization schemes for future work. It is worth emphasizing that linearization only generates sequential orderings for log-likelihood factorization; the local token embeddings represent the position of function calls in the actual, not the linearized, call trees, as described above.

F. Transformer Encoder

We briefly describe in this section the architecture of the transformer encoder and the technique of attention masking, and refer interested readers to the original paper [16].

On a high level, given a collection of vectors, a transformer encoder outputs a collection of vectors of equal size. This is done by applying \( n \) modules of identical architecture and independent parameters. Each module is the composite of two submodules: a multi-headed self-attention module and
a position-wise feedforward module. Given a collection of vectors \( z_1, \ldots, z_n \), a self-attention module generates a query vector \( q_i \), key vector \( k_i \), and value vector \( v_i \) for each vector \( z_i \) with linear transforms:

\[
q_i = Qz_i \quad k_i = Kz_i \quad v_i = Vz_i
\]

The self-attention module then generates a vector \( \hat{z}_i \) for each vector \( z_i \) by aggregating all value vectors \( v_1, \ldots, v_n \), weighted by the attention weights they receive from \( z_i \):

\[
\hat{z}_i = \sum_{j=1}^{n} \alpha_{i,j} v_j
\]

The attention weights are the inner products between key and query vectors, normalized by soft-max:

\[
\alpha_{i,j} = \frac{\exp(q_i^T k_j)}{\sum_{k=1}^{n} \exp(q_i^T k_k)}
\]

The soft-max normalization ensures that the attention weights for any \( z_i \) are summed to 1, i.e., forms a proper probability distribution. A multi-head self-attention module repeats the procedure multiple times with different parameters (\( Q, K, \) and \( V \)), and concatenates the resulting vectors into one.

The position-wise feedforward module applies two a layer neural network to each \( \hat{z}_i \). The neural network consists of two linear layers interleaved with a non-linear layer:

\[
\hat{z}_i \leftarrow A\phi(C\hat{z}_i + d) + b
\]

where \( A \) and \( C \) are matrices, \( b \) and \( d \) are vectors, and \( \phi \) is a nonlinear function. The resultant embedding is added to the original embedding:

\[
z_i \leftarrow \text{norm}(z_i + FF(\hat{z}_i))
\]

where the normalization operation ensures numerical stability during training, and \( FF \) denotes the position-wise feedforward network. The aforementioned operations are iteratively applied 8 times with different parameters for each layer.

G. Learning with stochastic gradient descent

Given a function \( f(\theta) \), we can minimize it by gradient descent by updating \( \theta \) iteratively as follows:

\[
\theta \leftarrow \theta - \alpha \nabla_{\theta} f(\theta)
\]

where \( \alpha \) is a learning rate set as a hyperparameter. In the case of generative pre-training, the function \( f(\theta) \) is the joint likelihood

\[
\sum_{x \in X} \log p_\theta(x) = \sum_{x \in X} \sum_{i=1}^{n_x} \log p_\theta(x_i | x_{<i})
\]

where \( X \) is the collection of transactions and \( \theta \) is the parameters of the transformer encoder. In practice, however, it is impractical to compute \( f(\theta) \) exactly since the collection \( X \), in our case, consists of nearly 68M transactions. We therefore approximate \( f(\theta) \) with

\[
\sum_{x \in X} \log p_\theta(x) = \sum_{x \in X} \sum_{i=1}^{n_x} \log p_\theta(x_i | x_{<i})
\]

where \( \hat{X} \) is a mini-batch of transactions randomly sampled from \( X \). Notably, if the samples are drawn independently at random from an identical distribution, the resulting approximation is an unbiased estimate of \( f(\theta) \), thereby guaranteeing the convergence of gradient descent. This technique is referred to as stochastic gradient descent. The gradient of this approximation to \( f(\theta) \) can be computed by deep learning frameworks such as PyTorch [17].

We can also accelerate the convergence of the iterative procedure by using second-order information of \( f(\theta) \) in addition to its gradient by, for example,

\[
\theta \leftarrow \theta - \alpha H_f(\theta)^{-1} \nabla_{\theta} f(\theta)
\]

where \( H_f(\theta) \) is the Hessian matrix of \( f(\theta) \). Although \( H_f(\theta) \) is impractical to evaluate in practice due to the large number of parameters in transformer encoders, multiple approximation techniques exist that utilize some of the information without prohibitive computation costs. In this paper, we use the AdamW optimizer [18] with its default learning rate and momentum.

V. DeFi Intrusion/Anomaly Detection System

In this work, we focus on the task of automated and real-time detection of abnormal transactions, such as DeFi attacks. Previous works [6], [19] rely on costly and time-consuming manual feature extraction and modeling, and then propose custom heuristics to identify specific attacks. As a result, without significant effort, existing approaches cannot be scaled and generalized across different types of DApps protocol designs on various blockchains. To overcome this limitation, we propose to automate the IDS process by performing transaction level anomaly detection. Intuitively, adversarial transactions should be semantically distinguishable from benign transactions within and across DApps.

A. Data

We extend a dataset comprised of all transactions that involve 124 previously compromised DeFi applications on Ethereum [1]. Our dataset shows an average increase in incident frequency from 3.1/month in 2020 to 8.5/month in 2022 (2.74×). The most common incident causes are smart contract Layer (42%, e.g., reentrancy attack), protocol layer (40%, e.g., price oracle manipulation), and auxiliary layer (30%, e.g., honeypot) vulnerabilities. Our dataset consists of a total of 68M transactions, spanning a period of 1,523 days, starting from block 5,470,817 (19th April, 2018) and ending at block [15,000,000] (21st June, 2022), and was constructed as outlined in Section IV.B and Section IV.C. We chose to use this dataset because it includes known instances of compromised DeFi applications, providing a ground truth for our analysis. Our approach is unsupervised/self-supervised learning, which allows us to evaluate the effectiveness of BLOCKGPT just using this dataset with previously exploited platforms. We leave it as future work to enhance the dataset by including transactions from non-compromised applications, and focus on previously compromised applications in this paper.
We pre-trained the transformer encoder as in Section IV-E to maximize the joint likelihood of the traces of the 68M transactions. With the maximum likelihood as its sole objective, generative pre-training needs no other data apart from the traces of the transactions in the dataset. For example, we do not need labels of whether a transaction is benign or anomalous, as in a supervised learning setting.

When evaluating the intrusion detection capability of the pre-trained transformer encoder (the IDS), we used all transactions in our dataset as the ground truth, assuming that only transactions tagged as malicious were malicious, while the rest of the transactions in the dataset are considered benign. It is important to note that our dataset has a limitation in that it is possible that other undiscovered attacks may have occurred prior or after the known attacks, which may affect the overall evaluation of the IDS performance. Additionally, it is worth noting that each attack may involve more than one malicious transaction, some of which may appear benign but in fact prepare for the attack. Therefore, our approach may not be able to identify any anomalies based on the transaction traces of these transactions, which is another limitation of our study.

We evaluate the IDS for each smart contract independently, as described below.

B. Detection Methodology

We adopt a ranking-based approach to intrusion detection. Recall that given a linearized transaction trace \( x_1, \ldots, x_n \), a transformer encoder coupled with a linear transform and soft-max can yield a sequence of conditional log-likelihoods \( \log p(x_1 | x_{<1}), \ldots, p(x_n | x_{<n}) \), the sum of which is the log-likelihood of the trace. Given a DeFi application, BLOCKGPT ranks all transactions involved with the application by the log-likelihood of their traces, and raises an alarm for the \( \alpha \% \) most abnormal transactions, where \( \alpha \) is an adjustable parameter. Intuitively, increasing \( \alpha \) makes it more likely for the IDS to detect attacks, but also raises the likelihood of false positives.

C. Implementation Details

We pack traces into mini-batches to speed up training. Each mini-batch consists of 32 traces, each with 512 tokens. The mini-batch size is chosen to maximize GPU utilization (we use an A100 SXM4 GPUs with 40 GB memory). As most traces in the dataset yield far less than 512 tokens after tokenization, we concatenate as many short traces as possible into one trace that is no longer than 512, and apply proper attention masking to ensure that the log-likelihood of each trace is computed as if it is the only trace. Each mini-batch therefore contains a variable number of traces.

The dataset contains 5M unique Ethereum addresses and 5M unique Ethereum function signatures. It is infeasible to store all their embeddings on a CPU due to GPU memory constraints. We therefore store the embeddings of the most frequent 100K tokens in the GPU, and the embeddings of the rest in RAM. Embeddings in the GPU are updated synchronously as the rest of the parameters in the transformer encoder. Embeddings in RAM are updated by a parameter server using asynchronous stochastic gradient descent. This architecture enables us to store significantly more embeddings. This approach is highly efficient as the embeddings of rare tokens are rarely accessed. Without further optimization, the proposed IDS attains a batched throughput of 2284 ± 289 transactions per second on a single NVIDIA A100 GPU. We use the AdamW optimizer with its default learning rate and momentum.

VI. Evaluation

This section presents an evaluation of BLOCKGPT as an IDS for detecting DeFi attacks. We begin by analyzing BLOCKGPT’s performance under various alarm threshold configurations. We then employ common evaluation metrics such as Precision, Recall, and F-score to further assess the system’s effectiveness. Additionally, we compare BLOCKGPT against several benchmark approaches to demonstrate its capability in detecting a diverse range of attacks. For readers interested in a more advanced, IDS-specific metric for comparison with related works, the Intrusion Detection Capability Score is evaluated in the appendix (cf. Appendix D).

A. Assumption

In the following we assume that for incidents involving multiple adversarial transactions, it is sufficient for BLOCKGPT to detect just one transaction in the sequence, rather than all of them. This is based on the premise that, upon detecting an abnormal transaction, DeFi protocol operators can take immediate action to prevent further harm, such as activating an emergency pause on the entire protocol. Research indicates that approximately 50% of attacked DeFi protocols already have such a pause mechanism in place [1].

B. Alarm Threshold and Metrics

We assess the performance of BLOCKGPT by analyzing various alarm threshold configurations. The alarm threshold is a system parameter that users must select for BLOCKGPT. It determines the sensitivity of an IDS and is defined as the likelihood of a transaction being deemed abnormal. Transactions that fall below this threshold will generate an alarm. As an example, when the alarm threshold is set to 1%, the IDS will raise alarms for the 1% least likely, or most abnormal, transactions interacting with a smart contract.

We consider two different types of alarm thresholds in this paper: (i) Percentage ranking alarm threshold, which is expressed as a proportion of transactions in the dataset; and (ii) Absolute ranking alarm threshold, which corresponds to a fixed number of top-ranked transactions.

Note that the detection thresholds granularity depends on the dataset sizes. For example, if a smart contract has 100 interacting transactions, the alarm threshold would be 1% even when inspecting a single transaction, which is the best possible scenario. This limitation arises due to the inherent constraints of the dataset size and should be considered when interpreting the results in Figure 4.
need to select a suitable threshold based on the specific attributes of the smart contract under investigation. Attacks, albeit at the cost of an increased false positive rate. Notably, the efficacy of the alarm threshold varies across different dataset sizes, emphasizing the need to select a suitable threshold based on the specific attributes of the smart contract under investigation.

Fig. 4. This table presents the performance of B\textsuperscript{LOCK}GPT under various alarm threshold configurations, organized by the number of transactions interacting with the vulnerable smart contracts. For example, with an alarm threshold of $\leq 0.01\%$, our method detects 24% of the attacks within the 10000+ transaction range, with an average false positive rate of 0.097%. The results indicate that using a lower alarm threshold enables the detection of a higher percentage of attacks, albeit at the cost of an increased false positive rate. Notably, the efficacy of the alarm threshold varies across different dataset sizes, emphasizing the need to select a suitable threshold based on the specific attributes of the smart contract under investigation.

<table>
<thead>
<tr>
<th>Dataset Size (the total number of transactions interacting with the vulnerable smart contract)</th>
<th>Percentage Ranking Alarm Threshold (%)</th>
<th>Absolute Ranking Alarm Threshold</th>
<th>Absolute Number of False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 99 txs (32 attacks, 28% of dataset)</td>
<td>$\leq 0.01%$</td>
<td>$\leq 0.1%$</td>
<td>$\leq 0.5%$</td>
</tr>
<tr>
<td>Average false positive rate</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Average number of false positives</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100 - 999 txs (38 attacks, 33% of dataset)</td>
<td>$\leq 0.01%$</td>
<td>$\leq 0.1%$</td>
<td>$\leq 0.5%$</td>
</tr>
<tr>
<td>Average false positive rate</td>
<td>-</td>
<td>-</td>
<td>8 (21%)</td>
</tr>
<tr>
<td>Average number of false positives</td>
<td>-</td>
<td>-</td>
<td>1.5</td>
</tr>
<tr>
<td>1000 - 9999 txs (17 attacks, 15% of dataset)</td>
<td>$\leq 0.01%$</td>
<td>$\leq 0.1%$</td>
<td>$\leq 0.5%$</td>
</tr>
<tr>
<td>Average false positive rate</td>
<td>-</td>
<td>0.054%</td>
<td>0.45%</td>
</tr>
<tr>
<td>Average number of false positives</td>
<td>-</td>
<td>1.4</td>
<td>11.5</td>
</tr>
<tr>
<td>10000 + txs (29 attacks, 25% of dataset)</td>
<td>$\leq 0.01%$</td>
<td>$\leq 0.1%$</td>
<td>$\leq 0.5%$</td>
</tr>
<tr>
<td>Average false positive rate</td>
<td>2 (7%)</td>
<td>7 (24%)</td>
<td>16 (55%)</td>
</tr>
<tr>
<td>Average number of false positives</td>
<td>2.5</td>
<td>120.1</td>
<td>429.9</td>
</tr>
<tr>
<td>Overall</td>
<td>$\leq 0.01%$</td>
<td>$\leq 0.1%$</td>
<td>$\leq 0.5%$</td>
</tr>
<tr>
<td>Average false positive rate</td>
<td>2 (2%)</td>
<td>13 (11%)</td>
<td>33 (28%)</td>
</tr>
<tr>
<td>Average number of false positives</td>
<td>2.5</td>
<td>65.3</td>
<td>211.9</td>
</tr>
</tbody>
</table>

Fig. 5. The 20 attacks ranked by B\textsuperscript{LOCK}GPT IDS as the most abnormal transaction that interacted with the respective victim contract. B\textsuperscript{LOCK}GPT successfully identifies the most abnormal transaction for 18 unique DeFi protocols across various application categories, with the total damage value exceeding 276 million USD. In more detail, B\textsuperscript{LOCK}GPT ranked 20 adversarial transactions as the most abnormal transactions involving their victim contracts, 22 as the second least likely, and 7 as the third least likely. The top-2 and top-3 most abnormal transactions can be found in Appendix E.

C. Effectiveness

To provide a comprehensive analysis, we evaluate the effectiveness of B\textsuperscript{LOCK}GPT across various alarm threshold settings (i.e., 0.01%, 0.1%, 0.5%, 1%, 10% for percentage ranking alarm threshold, and top-1, top-2, top-3 for absolute ranking alarm threshold) and different dataset sizes (i.e., 0-99, 100-999, 1000-9999, and 10000+ transactions). Figure 4 presents the performance results for each alarm threshold level, including the following metrics: (i) The number of detected attacks (and the corresponding percentage of total attacks); (ii) The average false positive rate; and (iii) The average number of false positives.

As demonstrated in Figure 4, B\textsuperscript{LOCK}GPT is proficient at identifying abnormal transactions, ranking 49 out of the 124 attacks (42%) among the top-3 most abnormal transactions interacting with their respective victim contracts. In particular, the top-1 most abnormal transactions and the associated damages incurred by their victims are displayed in Figure 5. B\textsuperscript{LOCK}GPT effectively detects the most abnormal transaction for 18 unique DeFi protocols across various application categories, with the total damage value exceeding 276 million USD. In more detail, B\textsuperscript{LOCK}GPT ranked 20 adversarial transactions as the most abnormal transactions involving their victim contracts, 22 as the second least likely, and 7 as the third least likely. The top-2 and top-3 most abnormal transactions can be found in Appendix E.

Note that our data suggests that there is a trade-off between the false positive rate and how many attacks B\textsuperscript{LOCK}GPT captures. For example, consider the case of smart contracts with 1000-9999 transactions. With an alarm threshold of 0.1%, B\textsuperscript{LOCK}GPT detects 35% of attacks while maintaining a false positive rate of 0.054%. Increasing the threshold to 0.5% improves the detection rate to 53% with a corresponding false positive rate of 0.45%. Although the detection rate has increased by 18 percentage points, the false positive rate has also increased by approximately 8.33 times. This demonstrates that there is a trade-off between improved detection capabilities and higher false positive rates.

Our findings indicate that the performance of B\textsuperscript{LOCK}GPT is positively correlated with the size of the dataset. For instance,
Recall of the proposed IDS

0.0 0.2 0.4 0.6 0.8 1.0
Precision of the proposed IDS

0.0 0.2 0.4 0.6 0.8 1.0
Recall of the proposed IDS

0.15 0.30 0.45 0.60 0.75 0.90
False Positive Rate

0.0 0.2 0.4 0.6 0.8 1.0
False Positive Rate

at the same alarm threshold of 0.5%, our model demonstrates better performance on larger transaction sets. It detects 55% of the attacks within the 10000+ transaction range, 53% for the 1000 – 10000 range, and only 21% for the 100 – 1000 range. This observation suggests that BLOCKGPT may benefit from larger datasets, as it allows for more accurate identification of attacks while maintaining an acceptable false positive rate.

D. Practicality

Our analysis highlights the practicality of BLOCKGPT when handling a large number of transactions. For instance, BLOCKGPT demonstrates its best performance on historical data for DeFi applications with a transaction history ranging from 1,000 to 9,999. Using a 0.01% alarm threshold, BLOCKGPT captures 35% of the attacks while maintaining an average false positive rate (FPR) of 0.054%, resulting in only 1.4 false positive transactions on average. Additionally, in DeFi applications with over 10,000 transactions and a 0.01% alarm threshold, BLOCKGPT successfully detects 24% of the attacks with an average FPR of 0.097%, corresponding to 120.1 false positive transactions on average.

In the context of popular DeFi applications processing 100 transactions per day, a 0.1% FPR generates one alert approximately every 10 days. This showcases that BLOCKGPT can provide a manageable number of alerts for further investigation, making it particularly suitable for high-volume transaction environments.

E. Performance

BLOCKGPT exhibits real-time capabilities, achieving an average batch throughput of 2,284 ± 289 transactions per second and taking an average of 0.16 ± 0.3 seconds to rank a single transaction.

F. Precision, Recall and F1 Scores

This section aims to visualize traditional machine learning metrics for our tool, BLOCKGPT, and provide insights into its performance. We begin by providing definitions for the metrics and key terms:

- True Positives (TP): Number of correctly identified adversarial transactions.
- False Positives (FP): Number of non-adversarial transactions incorrectly classified as adversarial.
- False Negatives (FN): Number of adversarial transactions incorrectly classified as non-adversarial.
- Precision = True Positives / (True Positives + False Positives)
- Recall = True Positives / (True Positives + False Negatives)
- F1-score = 2 · Precision · Recall / (Precision + Recall)

Figure 6 presents the best F1-score that BLOCKGPT can achieve, and the corresponding precision and recall, for each previously compromised DeFi application in our dataset. We observe that 112 (96%) of our data exhibits a recall above 0.8, while 64 (55%) has a precision of below 20%. The average F1 score is only 37.6%. However, it is important to note that while the traditional metrics may not appear as impressive, it does not imply that our system is not useful. The discrepancy can be attributed to the data imbalance issue and the so-called base rate fallacy problem.

Consider a scenario with a dataset containing one million transactions and only one adversarial transaction. An IDS with an alarm triggering threshold of 0.01% (raises in total 100 alarms) can only attain a precision of 1%. In this case, the recall would be 100% because the attack is detected. The F1-score, which equally weights precision and recall, would be approximately 1.96%. Given that the cost of an attack is tremendous, in such cases, a high detection rate is desirable as it indicates that the IDS is better at detecting intrusions, even if it comes at the cost of a higher false alarm rate.

An alternative approach for visualization is using the $F_{\beta}$-score to place greater emphasis on recall than precision (refer to the $F_{10}$ score in Appendix A). However, this approach is subjective unless the value of $\beta$ is thoroughly justified quantitatively. For a more quantitative method to evaluate BLOCKGPT, we recommend referring to Appendix B where we discuss the Intrusion Detection Capability Score (CID score).
G. Impact of Flash Loans on BLOCKGPT’s Efficiency

We proceed to examine the effectiveness of BLOCKGPT in detecting specific attack types, particularly focusing on whether the transaction utilizes a flash loan or not. Gaining a more profound understanding of this aspect is crucial for assessing the efficacy of BLOCKGPT in identifying attacks employing various strategies. We find that out of the 124 attacks in our dataset, approximately 34 (27%) employed a flash loan. We can break down BLOCKGPT’s performance on these attacks as follows:

1) In attacks involving a flash loan, BLOCKGPT ranks the adversarial transaction among the top-3 most abnormal transactions for 16 out of 34 cases (47%).
2) In attacks without a flash loan, BLOCKGPT ranks the adversarial transaction among the top-3 most abnormal transactions for 31 out of 90 cases (34%).
3) The current results indicate that BLOCKGPT has a higher success rate in detecting flash loan attacks while also exhibiting its capability to detect non-flash loan attacks.

BLOCKGPT performs slightly better on flash loan attacks (47% vs 34%), suggesting it considers other factors in the trace, not just naively classifying flash loans as attacks.

H. Benchmark 1 — Comparison with a Doc2Vec Model

To evaluate BLOCKGPT against a naive baseline, we implement a transaction ranker using a combination of doc2vec and Gaussian mixture models. The doc2vec model treats the flattened transaction trace as a document for analysis. This baseline ranks 8 attacks in our attack dataset as the most abnormal transactions interacting with the victim contracts, 11 as the second least likely, and 5 as the third least likely. In contrast, BLOCKGPT ranks 20 attacks as the most abnormal transactions, 22 as the second least likely, and 7 as the third least likely, showcasing its superior performance. For a detailed explanation of the technical aspects, we refer interested readers to Appendix C, where the Gaussian mixture model parameters are estimated using Expectation-Maximization, and the number of clusters is chosen by minimizing the Bayesian information criteria.

I. Benchmark 2 — Comparison with a Trace Length Heuristic

Based on the observation that malicious transactions often have abnormally long traces, we develop a heuristic-based IDS that ranks transactions interacting with a contract according to their trace lengths and flags those with the longest traces. This heuristic system identifies 20 attacks within the top-3 ranked transactions in our dataset, among which 18 are also ranked in the top-3 by BLOCKGPT. Overall, BLOCKGPT ranks 49 attacks in our attack dataset as top-3 abnormal transactions, demonstrating enhanced performance compared to the heuristic-based baseline. Importantly, BLOCKGPT maintains its ability to detect these attacks even if their traces are deliberately shortened to evade the baseline system, as it only examines the first 512 elements of each trace.

J. Discussion

The effectiveness of BLOCKGPT is influenced by the choice of the detection threshold. Selecting an appropriate detection threshold should consider not only the performance of the IDS, but also the risk appetite and cost-benefit trade-off for the DeFi protocol operator. In practice, DeFi protocol operators ought to balance their risk tolerance and cost-benefit considerations when choosing a suitable detection threshold. They may also contemplate employing multiple thresholds for different growth stages, such as implementing a higher threshold for a contract’s initial transactions and a lower threshold for transactions involving substantial asset amounts. We leave it to future work to automatically suggest alarm thresholds based on applications and growth stages. Furthermore, BLOCKGPT can be combined with orthogonal security measures, including smart contract auditing and whitelisting, establishing a comprehensive security framework for DeFi protocols.

VII. RELATED WORKS

A. Smart Contract Intrusion Prevention and Detection

Intrusion detection and prevention are essential components in the realm of decentralized finance security research. Table 7 compares our proposed method with alternative approaches.

1) Qin et al. [6], [21] introduce two generalized imitation attack methods utilizing dynamic program analysis to automatically observe, copy, and synthesize profitable transactions from the P2P network. While these methods find profitable transactions, they do not determine whether these profitable transactions are abnormal or not.
2) DeFiPoser [12] uses logical DeFi protocol models and a theorem prover (e.g., Z3) or the Bellman-Ford-Moore algorithm to create profitable DeFi transactions. It operates in real-time or offline. However, it also does not distinguish attacks and relies heavily on provided protocol models.
3) Fuzzing involves providing generated inputs to a smart contract to uncover vulnerabilities. The idea is to test the contract’s behavior in unexpected situations, which can reveal potential security issues. Fuzzing has been shown to be effective in detecting vulnerabilities in smart contracts, but it has limitations, such as the lack of coverage of all possible code paths [24, 29].
4) Symbolic execution involves evaluating a smart contract’s code with symbolic inputs, rather than concrete values. The goal is to explore all possible code paths and uncover potential vulnerabilities. Symbolic execution has been used to detect various types of vulnerabilities, such as reentrancy attacks and integer overflow/underflows [28, 30–40].
5) Formal verification involves using mathematical methods to prove that a smart contract meets certain security properties [28, 49–51]. The idea is to formally prove that the contract’s code is correct, which can provide a higher level of assurance than testing.
6) Static analysis involves analyzing a smart contract’s code without executing it. The goal is to uncover potential vulnerabilities by examining the contract’s structure and dependencies.
control flow. Static analysis can be used to detect various types of vulnerabilities, such as uninitialized variables and unsafe function calls [22], [35], [41]–[48].

7) Dynamic analysis involves executing a smart contract and monitoring its behavior. The goal is to uncover potential vulnerabilities by observing the contract’s behavior in a real-world environment [19], [22], [23].

While the aforementioned techniques are effective in identifying vulnerabilities, they are not typically considered as real-time IDS / IPS due to performance limitations. Various techniques and approaches have been proposed in the literature to improve real-time smart contract security in the DeFi ecosystem. One approach is the use of rule-based methods, which rely on predefined rules and patterns to detect and prevent smart contract vulnerabilities in real-time. Related work explored a rule-based approach to detect and prevent price oracle manipulation attacks [19] as well as machine learning to detect reentrancy attacks [52].

It is worth noting that while the above-mentioned methods are effective in identifying specific vulnerabilities, they may not cover all possible types of vulnerabilities. Our proposed work aims to detect any anomaly in transaction trace in real-time while being protocol-agnostic.

B. Embedding Techniques In NLP

ELMo: ELMo [53] is a deep contextualized model that represents characteristics of word use (e.g., syntax and semantics) across linguistic contexts and captures context-dependent aspects of word meaning. ELMo takes the entire sentence as the input of a bidirectional LSTM (biLSTM) model, thus effectively encoding the contextualized sentence information.

BERT: By following ELMo, Devlin et al. [54] propose a deep pre-trained embedding model called BERT, which applies bidirectional training of Transformer with an attention mechanism that learns contextual dependency between words. Moreover, BERT can be fine-tuned for a wide variety of NLP tasks by adding just one output layer to the core model.

VIII. Conclusion

In this work, we introduced BLOCKGPT, an innovative transaction anomaly ranking tool for Ethereum-based blockchains. By analyzing a rich dataset spanning four years, BLOCKGPT demonstrated its ability to accurately identify attack transactions amidst a highly imbalanced dataset of benign and adversarial transactions. Our results indicate that BLOCKGPT effectively detects abnormal transactions, ranking 49 out of 124 attacks among the top-3 most abnormal transactions interacting with their corresponding victim contracts.

BLOCKGPT showcases its real-time capabilities with an average batch throughput of 2,284±289 transactions per second, making it a viable real-time intrusion detection system for blockchain networks such as Ethereum. The proposed system can trigger smart contract pause mechanisms in response to malicious blockchain transactions, thus preventing attacks.

Our research contributes to the field of blockchain transaction analysis by being the first to employ unsupervised/self-supervised learning for anomaly detection of transactions. Additionally, we constructed a large language model specifically designed for this task, incorporating custom data encoding and domain-specific tokenization techniques. This work lays the foundation for further exploration of real-time, learning-based security analysis tools for blockchain networks.
datasets is imbalanced and the cost of false negatives is high [56]. In such cases, a high detection rate is desirable as it indicates that the IDS is better at detecting intrusions, even if it comes at the cost of a higher false alarm rate, lower F1, and lower precision. To address this, researchers have proposed advanced metrics such as $C_{ID}$ [57], which considers the operating environment and costs associated with false alarms and missed intrusions. In this paper, we propose to apply CID to evaluate the performance of our DeFi IDS, considering the imbalanced nature of the dataset and the high cost of false negatives. This aims to provide a more comprehensive evaluation and improve decision-making for the DeFi IDS’s deployment and maintenance.

On a high-level, the $C_{ID}$ is defined as the ratio of the mutual information between the IDS input and output to the entropy of the input, where $I$ and $H$ respectively denote the mutual information and the entropy.

$$C_{ID} = \frac{I(X;Y)}{H(X)}$$

The entropy $H(X)$ of a random variable $X \in \mathcal{X}$ is

$$H(X) = -\sum_{x \in \mathcal{X}} P_X(x) \log P_X(x)$$

where $P_X$ denotes the distribution of $X$. Intuitively, the entropy $H(X)$ quantifies the uncertainty in $X$. The mutual information $I(X;Y)$ between discrete random variable $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$ is given by

$$I(X;Y) = \sum_{x \in \mathcal{X}} P_X(x) \sum_{y \in \mathcal{Y}} P_{X,Y}(x,y) \log \frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)}$$

where $P_{X,Y}$ denotes the joint distribution of $X$ and $Y$, and $P_X$ and $P_Y$ denote the marginal distribution of $X$ and $Y$ respectively. Intuitively, the mutual information $I(X;Y)$ quantifies the amount of information about $Y$ that observing
$X$ yields. The Intrusion Detection Capability $C_{ID}$, being the ratio between the mutual information and the entropy, thus quantifies the proportion of uncertainty in a transaction being abnormal that is captured by the IDS.

Although the $C_{ID}$ accounts for attack distributions by normalizing the mutual information between attacks and alerts with the entropy of attacks, it fails to incorporate the cost of undetected attacks and false alarms. More generally, we can associate a cost $\gamma_{x,y}$ with each pair of outcomes $(x, y) \in X \times Y$. The cost-aware mutual information between $X$ and $Y$ is then

$$I_{\gamma}(X; Y) = \frac{1}{\gamma} \sum_{x \in X} \sum_{y \in Y} \gamma_{x,y} P_{X,Y}(x, y) \log \frac{P_{X,Y}(x, y)}{P_X(x)P_Y(y)}$$

$$\gamma = \sum_{x \in X} \sum_{y \in Y} \gamma_{x,y}$$

is the total cost of all outcomes. Multiplying each term by the cost associated with the outcome and dividing by the total cost biases the mutual information towards outcomes with high costs. With the entropy $H(X)$ of $X$ defined as above, the cost-aware Intrusion Detection Capability $C_{ID}^{(\gamma)}$ is given by

$$C_{ID}^{(\gamma)} = \frac{I_{\gamma}(X; Y)}{H(X)}$$

The advantage of the cost-aware $C_{ID}$ is that it only requires cost estimates that are easy to calculate. For an IDS, both $X$ (if a pending transaction is an attack) and $Y$ (if the IDS sends an alert for a pending transaction) are binary variables. The costs we need to estimate include:

1) $X$ and $Y$ are both positive: The IDS alerts the DeFi protocol operator about a potential attack. After verifying it’s an attack, the operator stops it. Assuming no cost to prevent attacks, the cost of this outcome is the cost of the operator inspecting a transaction.

2) $X$ is negative, $Y$ is positive (false positive): The IDS alerts the system about a possible attack, but it’s a benign transaction. The operator lets it proceed. The cost of this outcome is the cost of the operator inspecting the transaction.

3) $X$ is positive, $Y$ is negative (false negative): An attack succeeds without the operator noticing, causing a loss to the protocol. Our data shows that each attack costs around ten million USD on average.

4) $X$ and $Y$ are both negative: The transaction goes through with no cost.

To estimate the cost of a DeFi protocol operator inspecting a transaction, we use the statistics from a recruitment firm that DeFi security auditors can earn 400,000 USD per year in 2022. We estimate that inspecting a false positive transaction will take one hour or more, which is approximately 204 USD. It should be noted that DeFi protocol operators can choose to activate defense mechanisms such as emergency pause. These defenses may have an impact on the user experience, resulting in reputation damage and an implicit cost. We ignore the reputation damage in this paper. We estimate the distribution $P_X$, $P_Y$, and $P_{X,Y}$ using the frequency estimator.

![Fig. 9. The Intrusion Detection Capability ($C_{ID}$) of the proposed IDS. The x-axis corresponds to $P_{FA}$, the probability of the IDS raising false alarms (FA), while the y-axis corresponds to $P_D$, the fraction of detected attacks.]
is technically similar to CBOW while PV-DBOW is similar to SG. It is used to learn words’ representations of arbitrary length sequences. Concretely: Given a set of documents \( D = \{d_1, d_2, d_3, \ldots \} \) and a sentence \( s(d_i) = \{w_1, w_2, w_3, \ldots \} \) composed of words from document \( d_i \), it learns the embeddings \( \mathbf{d}_i \in \mathbb{R}^d \) and \( \mathbf{w}_i \in \mathbb{R}^d \). This is done by considering a word \( w_i \in s(d_i) \) and maximizing \( \sum_{i=1}^{d} \log P(w_i|d_i) \) where the probability of \( w_i \) occurring in the context \( d_i \) is \( P(w_i|d_i) = \frac{\exp \mathbf{d}_i \cdot \mathbf{w}_i}{\sum_{w \in V'} \exp \mathbf{d}_i \cdot \mathbf{w}} \). The key computational challenge to the approach is to compute the denominator \( \sum_{w \in V} \exp \mathbf{d}_i \cdot \mathbf{w} \), as the vocabulary size \( |V| \) can grow drastically as the corpus size grows. [60] proposes two solutions:

- **Hierarchical softmax.** Given a binary tree with each leaf associated with a word in the vocabulary, a hierarchical softmax classifier assigns a vector to each node of the tree. Let \( v_n \) denote the vector assigned to node \( n \). In the following we omit the context for brevity. All probabilities should be understood as conditional on the context. Given a word \( w \) and a path \( n_1, \ldots, n_k \) from the root of the tree to the leaf associated with the word \( w \), the log-likelihood of the word occurring is given by

\[
\log P[w] = \log P[n_1, \ldots, n_k|w] = \log \prod_{j=1}^{k} P[n_j|w] = \sum_{j=1}^{k} \log P[n_j|w] = \sum_{j=1}^{k} \log \sigma(v_{n_j}^{T} w)
\]

where \( \sigma \) is the sigmoid function given by

\[
\sigma(x) = \frac{\exp x}{\exp x + 1} \in (0, 1)
\]

- **Negative sampling.** Negative sampling reduces the computation cost of the denominator \( \sum_{w \in V} \exp \mathbf{d}_i \cdot \mathbf{w} \) by only summing over a random sample of the vocabulary \( V' \). Mathematically,

\[
P_r(w_i|d_i) = \frac{\exp \mathbf{d}_i \cdot \mathbf{w}_i}{\sum_{w \in V'} \exp \mathbf{d}_i \cdot \mathbf{w}}
\]

where \( V' \subset V \) is chosen randomly. Clearly, the efficiency of the solution depends on how to draw samples from the vocabulary. [60] proposes to draw samples from the vocabulary following the rule below:

\[
P(w) = 1 - \sqrt[|f(w)|]{t}
\]

where \( f(w) \) is the frequency of word \( w \) in a training corpus, and \( t \) is a hyperparameter. Intuitively, the sampling rule increases the likelihood of rare words being chosen compared to uniform sampling, which makes sense as rare words are more likely to be informative than frequent ones.

We experimented with both approaches and observed no significant performance difference in our settings. The baseline result reported in this paper was achieved using negative sampling, which provided marginally better results.

3) **Doc2Vec and gaussian mixture.** Mathematically, the log-likelihood of a trace with doc2vec embedding \( v \in \mathbb{R}^d \) under a Gaussian mixture model with parameters \( \pi_1, \ldots, \pi_C \in \mathbb{R} \), \( \mu_1, \ldots, \mu_C \in \mathbb{R}^d \), and \( \Sigma_1, \Sigma_C \in \mathbb{R}^{d \times d} \) is given by

\[
\log p(v|\pi, \mu, \Sigma) = \log \sum_{c=1}^{C} \pi_c \phi_{\mu_c, \Sigma_c}(v)
\]

where \( \phi_{\mu, \Sigma} \) is the multidimensional Gaussian probability density function with mean \( \mu \) and covariance \( \Sigma \), given by

\[
\phi_{\mu, \Sigma}(x) = \frac{(2\pi)^{-d/2}(\det \Sigma)^{1/2}}{\sqrt{\pi^d}} \exp \left( -\frac{1}{2} (x - \mu)^T \Sigma (x - \mu) \right)
\]

where \( \det \Sigma \) is the determinant of the covariance matrix \( \Sigma \).

Gaussian mixture models can be interpreted as probabilistic clustering of vector embeddings, with the parameter \( C \) interpreted as the number of clusters, the parameters \( \pi_1, \ldots, \pi_C \) the probability of the vector embedding belonging to each cluster, and the parameters \( \mu_1, \ldots, \mu_C \) the centroid of each cluster. The parameters \( \pi, \mu, \) and \( \Sigma \) are estimated by maximizing the log-likelihood of historical transactions:

\[
\max_{\pi, \mu, \Sigma} \sum_v \log p(v|\pi, \mu, \Sigma)
\]

which is accomplished by an Expectation-Maximization (EM) algorithm [61].

The EM algorithm alternates between an E (Expectation) and M (Maximization) step until some convergence criterion is met. The E step assigns each observation to the cluster where its likelihood is maximized. Mathematically, the cluster \( c_e \) assigned to an observation \( x \) is given by

\[
C_x = \arg \max_{e=1, \ldots, C} \log p(x|c_e) + \log \phi_{\mu_e, \Sigma_e}(x)
\]

The M step that follows then update the parameters \( \pi_1, \ldots, \pi_C, \mu_1, \ldots, \mu_C \), and \( \Sigma_1, \ldots, \Sigma_C \) by maximizing the likelihood of all observations. The likelihood of each cluster \( \pi_1, \ldots, \pi_C \) is simply given by the proportion of observations that are assigned in the cluster. The mean and covariance of each cluster are given by maximizing the log-likelihood:

\[
\log \phi_{\mu_e, \Sigma_e}(\{x : c_e = e\}) = \log \prod_{x|c_e=e} \phi_{\mu_e, \Sigma_e}(x) = \sum_{x|c_e=e} \log \phi_{\mu_e, \Sigma_e}(x)
\]

In our experiments, we initialize the clusters with the K-Means algorithm, and leave experimenting other initialization methods for future work. More specifically, we first cluster the embeddings of traces in our training corpus with K-Means, and then set the cluster means \( \mu_1, \ldots, \mu_C \) to the centroids of the clusters. The likelihood of each cluster \( \pi_1, \ldots, \pi_C \) is set to the proportion of observations that are assigned to the cluster, and the covariance of each cluster is set to the sample covariance of each cluster. The objective of the K-Means algorithm is

\[
\arg \min S \sum_{i=1}^{K} \sum_{x \in S_i} |x - \mu_i|^2
\]
where \( S = (S_1, \ldots, S_K) \) is a tuple of \( K \) disjoint sets (clusters) such that \( \bigcup_{i=1}^{K} S_i = \{x_j\}_{j=1}^{N} \).

Multiple parameterization schemes for the covariance matrix exist. The one most expressive, yet most prone to overfit, is to use a full covariance matrix. The least expressive one is to reduce the covariance matrix to a single positive number. An approach that attempts to strike a balance between these extremes is to constrain the covariance matrix to be diagonal, i.e.

\[
\Sigma = \begin{bmatrix} \Sigma_1 & \cdots & \Sigma_d \\ \vdots & \ddots & \vdots \\ \Sigma_d & \cdots & \Sigma_1 \end{bmatrix}
\]

To avoid overfitting while maintaining some expressive power, we constrain the covariances \( \Sigma_1, \ldots, \Sigma_C \) to be diagonal.

The parameter \( C \) is chosen by minimizing the Bayesian information criteria (BIC) \[62\]

\[
\min_C -2 \sum_v \log p(v|\pi, \mu, \Sigma) + d(C) \log N
\]

where \( N \) is the number of historical transactions and \( d(C) \) is the number of parameters of a Gaussian mixture model, determined by the parameter \( C \). Intuitively, the BIC consists of two parts, a log-likelihood term and a regularization term

\[
\sum_v \log p(v|\pi, \mu, \Sigma), \quad d(C) \log N
\]

which penalizes mixtures with large number of parameters, e.g. mixtures with full covariance matrices. The regularization term grows as the number of observations grows to keep up with the scale of the log-likelihood term.

We find in preliminary experiments that diagonal covariance matrices generally maximize the BIC, likely due to a balance between their complexity and expressiveness. In our case, where all covariance matrices are constrained to be diagonal, the number of parameters \( d(C) \) is given by

\[
d(C) = C + C \cdot d + C \cdot d = C \cdot (2d + 1)
\]

Without any constraint on the covariance matrices, the number of parameters \( d(C) \) is given by

\[
d(C) = C + C \cdot d + C \cdot d^2 = C \cdot (d^2 + d + 1)
\]

When all covariance matrices are reduced to a single positive number, the number of parameters \( d(C) \) is given by

\[
d(C) = C + C \cdot d + C = C \cdot (d + 2)
\]

**APPENDIX D**

**NORMALIZATION IN TRANSFORMER ENCODER**

We find empirically that normalization is crucial to training the transformer encoder, particularly given our unique positional encoding scheme. As the sum of many embeddings, embeddings produced by our tree encoding can easily reach numerical scales that destabilize training. We find Layer Normalization (LayerNorm) \[63\] particularly helpful in this case. Given a \( d \)-dimensional vector \( v \), LayerNorm normalizes the vector as \( \hat{v} = \frac{v - \mu}{\sigma} \), where \( \mu \) is the mean of all components of \( v \), given by \( \mu = \frac{1}{d} \sum_{i=1}^{d} v_i \) and \( \sigma \) is the standard deviation of all components of \( v \), given by \( \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (v_i - \mu)^2} \). Clearly, the normalized vector \( \hat{v} \) satisfies the conditions \( \hat{\mu} = \frac{1}{d} \sum_{i=1}^{d} \hat{v}_i = 0 \) and \( \hat{\sigma} = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (\hat{v}_i - \mu)^2} = 1 \), which contributes to more stable training of the transformer encoder.

**APPENDIX E**

**SECOND AND THIRD MOST ABNORMAL TRANSACTIONS**

We present the results of our analysis on the transactions that \( \text{BLOCKGPT} \) identified as the second and third most abnormal. Figure 10 showcases the 20 attacks ranked by \( \text{BLOCKGPT} \) as the second most abnormal transaction that interacted with the respective victim contract.

![Fig. 10. The 20 attacks ranked by BLOCKGPT IDS as the second most abnormal transaction that interacted with the respective victim contract.](image)

<table>
<thead>
<tr>
<th>Victim Name</th>
<th>Victim Contract</th>
<th>Application Categories</th>
<th>Damage (in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexedFinance</td>
<td>0x5bd6..dec4</td>
<td>Others</td>
<td>16,000,000</td>
</tr>
<tr>
<td>ValueDeFi</td>
<td>0xddde..dd0d</td>
<td>Yield farming</td>
<td>7,200,000</td>
</tr>
<tr>
<td>DAOMaker</td>
<td>0xddd7..2b7d</td>
<td>Others</td>
<td>4,000,000</td>
</tr>
<tr>
<td>DAOMaker</td>
<td>0x6e70..2e22</td>
<td>Others</td>
<td>4,000,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x8eb2..1248</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>Vether</td>
<td>0x7557..abab</td>
<td>Others</td>
<td>900,000</td>
</tr>
<tr>
<td>Chainswap</td>
<td>0xc128..647f</td>
<td>Cross chain bridge</td>
<td>800,000</td>
</tr>
</tbody>
</table>

**Fig. 11. The 7 attacks ranked by BLOCKGPT IDS as the third most abnormal transaction that interacted with the respective victim contract.**

<table>
<thead>
<tr>
<th>Victim Name</th>
<th>Victim Contract</th>
<th>Application Categories</th>
<th>Damage (in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexedFinance</td>
<td>0x146e..e4bd</td>
<td>Others</td>
<td>27,000,000</td>
</tr>
<tr>
<td>InverseFinance</td>
<td>0x3981..db15</td>
<td>Lending</td>
<td>15,600,000</td>
</tr>
<tr>
<td>WarpFinance</td>
<td>0xae46..c8ef</td>
<td>Lending</td>
<td>7,800,000</td>
</tr>
<tr>
<td>DAOMaker</td>
<td>0xad4b..9289</td>
<td>Others</td>
<td>4,000,000</td>
</tr>
<tr>
<td>DAOMaker</td>
<td>0x29f6d..5940</td>
<td>Others</td>
<td>4,000,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x7755..4db8</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0xd361..1e1e</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x4265..ea66</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x2e410..1e12</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x90416..bd46</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0xe1cb..5e81</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0xe7481..909d</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0x8770..1b90</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0xe8ce2..0a19</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>GemSwap</td>
<td>0xe777..b02d</td>
<td>DEX</td>
<td>1,300,000</td>
</tr>
<tr>
<td>BasketDAO</td>
<td>0x462..3b88</td>
<td>DAO</td>
<td>1,200,000</td>
</tr>
<tr>
<td>Li.Finance</td>
<td>0x3a9f..bed1</td>
<td>DEX aggregator</td>
<td>600,000</td>
</tr>
<tr>
<td>BuildFinance</td>
<td>0x3157..c758</td>
<td>DAO</td>
<td>470,000</td>
</tr>
<tr>
<td>SashimiSwap</td>
<td>0xe44e..9410</td>
<td>DEX</td>
<td>200,000</td>
</tr>
<tr>
<td>Formation.Fi</td>
<td>0xc6b0a..a723</td>
<td>Yield farming</td>
<td>100,000</td>
</tr>
</tbody>
</table>

Of \( v \), given by \( \mu = \frac{1}{d} \sum_{i=1}^{d} v_i \) and \( \sigma \) is the standard deviation of all components of \( v \), given by \( \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (v_i - \mu)^2} \). Clearly, the normalized vector \( \hat{v} \) satisfies the conditions \( \hat{\mu} = \frac{1}{d} \sum_{i=1}^{d} \hat{v}_i = 0 \) and \( \hat{\sigma} = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (\hat{v}_i - \mu)^2} = 1 \), which contributes to more stable training of the transformer encoder.