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, TxRank, 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, TxRank 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 TxRank 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, TxRank 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.**

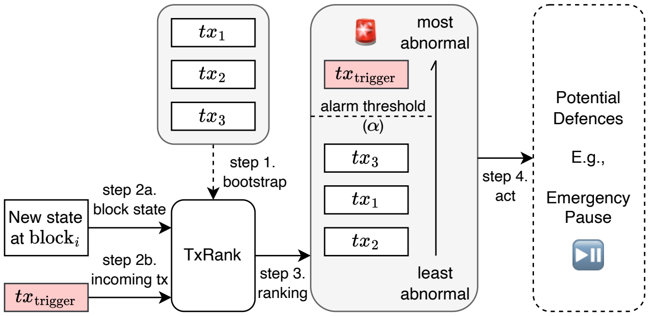

Fig. 1. High-level overview of the TxRank defense mechanism, which consists of the following four major steps. ❶ TxRank is bootstrapped by feeding in a dataset of historical transactions to train the model using unsupervised learning. ❷ Depending on the system and threat model, TxRank detects new block states, including already confirmed transactions, and pending transactions (cf. Section III). ❸ TxRank ranks transactions based on how abnormal their execution traces are (cf. Section IV). ❹ If an abnormal transaction is detected (cf. Section VI), TxRank triggers a defense mechanism such as a front-running emergency pause.

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 7). 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, TxRank. The main objective of TxRank is to elevate the art of blockchain security analysis into a comprehensive and real-time endeavor. By generating efficient trace representations of blockchain activity, TxRank trains a scalable large language model from scratch using an extended and augmented dataset from a

[*] Both authors contributed equally to this work.

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 then evaluate the effectiveness of TxRank as a transaction anomaly ranking tool on the same dataset, assessing its performance in identifying anomalous transactions.

We evaluate TxRank using two simple metrics: *(i)* a *percentage ranking alarm threshold*; and *(ii)* an *absolute ranking alarm threshold*. TxRank 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, TxRank 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 TxRank, 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, TxRank 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 TxRank 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, TxRank showcases its real-time capabilities, achieving an average batched throughput of $2,284 \pm 289$ 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 TxRank as an anomaly detection tool for Ethereum transactions to identify suspicious or malicious activities on the blockchain. We evaluate TxRank 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 TxRank's robustness and versatility. We benchmark TxRank against doc2vec and trace length heuristics, showcasing the superior effectiveness of TxRank.
- Evaluation results indicate TxRank 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, TxRank 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

Total Value Locked reaching $250B$ USD in December 2021. 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. TxRank 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 TxRank.

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 $\mathcal{A}$) which is capable of executing transactions (i.e., performing actions) across a set of DeFi platforms. $\mathcal{A}$ 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 TxRank. Their transactions are observable to our system either by *(i)* broadcasting on P2P such that anyone can observe; or if *(ii)* TxRank 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 TxRank until the adversarial transactions are finalized. Hidden adversaries achieve this by using a FaaS system or similar method.

C. TxRank *Overview*

TxRank 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 DApps. 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 TxRank 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 log-likelihood of the trace. A ranking or threshold-based method is then used to raise alarms for transactions with abnormal scores. This allows TxRank 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 TxRank 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 (0xcd31..3ad7) 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 TxRank 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 TxRank 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 relayer, specifically Flashbots.

  Suppose Revest Finance had deployed TxRank as an ordinary blockchain node without colluding with sequencers. 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, TxRank could only act as a fast retrospective attack detection tool. If TxRank detected the first adversarial transaction upon receiving block 14465357, it could have potentially prevented the other three attacks. Our evaluation found that TxRank ranked the first adversarial transaction in Revest Finance as the most abnormal transaction. This example underscores the importance of having a tool like TxRank, even if it is unable to observe pending transactions.

IV. TxRank Details

We now proceed to elaborate on the details of TxRank.

A. TxRank *Components*

TxRank generates transaction representations for anomaly detection of blockchain transactions in real-time (cf. Figure 2). As such, TxRank 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. TxRank's design consists of the following six components (cf. Section IV for details).

- ① *Intermediate Trace Representation (ITR) Construction*: Given a transaction, we construct a tree structured trace.

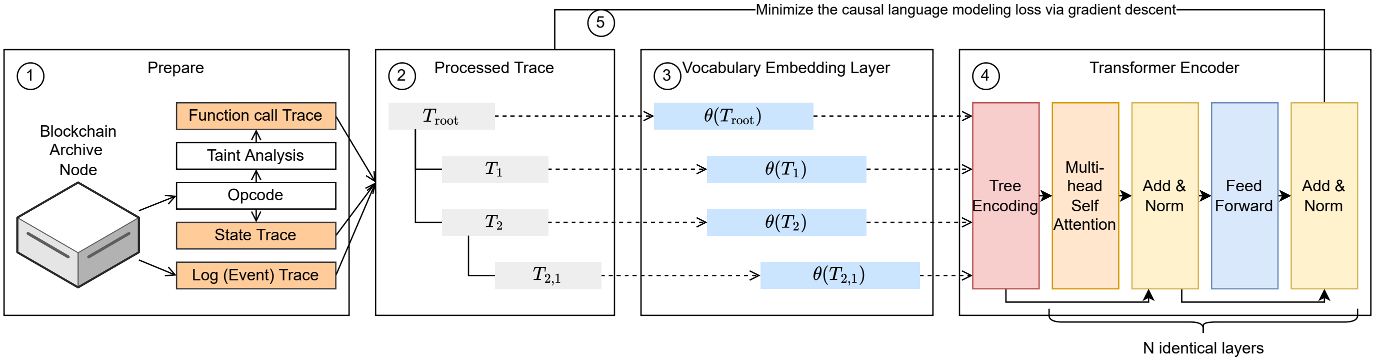

Fig. 2. TxRank System Architecture. In step ① we extract the transaction trace from a blockchain archive node and augment the trace with additional data. We then process the trace in a graph structure in step ② and apply a vocabulary embedding layer in step ③. In step ④ we apply a custom positional tree encoding, coupled with a loop to minimize the model loss in step ⑤.

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. 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 where each position corresponds to a unique token and the value is either 0 or 1.
  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 trace 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 TxRank, 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 α. Out of 124 attacks in the dataset, TxRank 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 $Tree_{\text{call}}(tx) = (T_{\text{call}}, E_{\text{call}})$ to represent a call trace. Each node $t \in T_{\text{call}}$ in $Tree_{\text{call}}(tx)$ corresponds to an executed function call, including its corresponding runtime encoded call data and return data in bytes. We use the notation $t^1_{\text{call}} \overset{e}{\to} t^2_{\text{call}}, e \in E_{\text{call}}$ to denote the edges in $Tree_{\text{call}}(tx)$, where a function t^2_{call} (i.e., callee) is triggered by another function t^1_{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 $Seqs_{state}(tx) = (T_{\text{state}}, E_{\text{state}})$. The first sequence $T_{\text{state}} = [t^1_{\text{state}}, \ldots, t^n_{\text{state}}]$ consists of state reads and writes details. If t^i_{state} is a read operation, then it consists of a value tuple $[\text{key}, \text{val}]$, meaning that a function reads value val from global storage (i.e., account storage) at position key during tx's execution. Similarly, if t^i_{state} 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 key with val during the execution of the function call. The second sequence $E_{\text{state}} = [t^1_{\text{call}} \overset{e}{\to} t^1_{\text{state}}, \ldots, t^n_{\text{call}} \overset{e}{\to} t^n_{\text{state}}]$ 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 $Seqs_{log}(tx) = (T^{tx}_{\text{log}}, E^{tx}_{\text{log}})$ to denote a log trace. The first sequence $T_{\text{log}} = [t^1_{\text{log}}, \ldots, t^n_{\text{log}}]$ contains all emitted smart contract events. Each $t^{tx}_{\text{log}} = [\text{Contract}, \text{Event Hash}, \text{Data (in bytes)}] \in T_{\text{log}}$ consists of a contract address, an event's hash identifier, and the corresponding encoded data in bytes. The second sequence $E_{\text{log}} = [t^1_{\text{call}} \overset{e}{\to} t^1_{\text{log}}, \ldots, t^n_{\text{call}} \overset{e}{\to} t^n_{\text{log}}]$ captures which function call emits each log.

Listing 1. An example of intermediate trace representation construction

```
CALL,from:0x99d...,to:0xe59...,data:c4f...
|- DELEGATECALL,from,0xe59...,to,0xe...,
    data,f39...
| |- READ, 0x95c..., 0x67a
| |- LOG1, 0x0b8..., 0x699
...
```

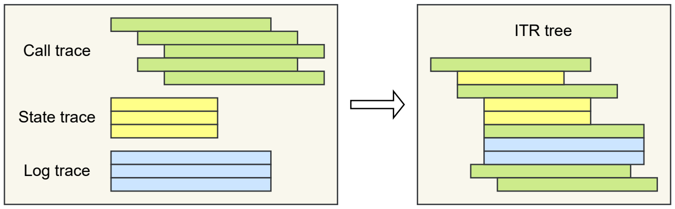

Fig. 3. Abstract example of a Trace construction in TxRank.

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 $Tree_{\text{call}}(tx)$ (cf. Figure 3).

$$\begin{aligned} Tree_{\text{ITR}}(tx) &= (T_{ITR}, E_{ITR}) \\ &= (T_{\text{call}} \bigcup T_{\text{state}} \bigcup T_{\text{log}}, E_{\text{call}} \bigcup E_{\text{state}} \bigcup E_{\text{log}}) \end{aligned}$$

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]).

- $f_{\text{token}}(T_{\text{call}})$ = [START], [CALL], from, to, function hash, gas, value, [INs], input1 type, input1 value, ..., [OUTs], output1 type, output1 value, ..., [END]
- $f_{\text{token}}(T_{\text{state}})$ = [START], [STATE], read / write, key, val, [END]
- $f_{\text{token}}(T_{\text{log}})$ = [START], [LOG], contract address, event hash, value1 type, value1, ..., [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 → 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, TXRANK 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 $E_{\texttt{[addr1]}} + E_{\text{root}} + E_{\text{src}}$ and $E_{\texttt{[addr2]}} + E_{\text{root}} + E_{\text{dst}}$ 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 $E_{\texttt{[func]}} + E_{\text{root}} + E_{\text{func}}$. 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 $O(2d)$ 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 $b_1, \dots, b_n \in \{L, R\}^n$ 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 $E_{1,L}, E_{1,R}, \dots, E_{D,L}, E_{D,R}$ where D is the maximum depth that we choose to permit. 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 $x_1, \dots, x_n$, we maximize as its objective the log-likelihood of the collection that is factorized as $\log p(x_1, \dots, x_n) = \sum_{i=1}^n \log p(x_i | x_{<i})$, where the context $x_{<i}$ is $x_1, \dots, x_{i-1}$ for $i > 1$, and the context $x_{<0}$ 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 nd matrix, the probability of a token [tok] occurring given the context $x_{<i}$ is given by

$$s = Az_{i-1} \in R^n$$

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

where #[tok] is an integer between 1 and n that is uniquely assigned to the token [tok]. The notation $s^{(j)}$ denotes the jth component of the vector s, and $s^{(\#\texttt{[tok]})}$ thus denotes the #[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 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 $z_1, \dots, z_n$, 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 $v_1, \dots, v_n$, 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:

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

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 a two layer neural network to each $\hat{z}_i$. The neural network consists of two linear layers interleaved with a nonlinear 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 ϕ is a nonlinear function. The resultant embedding is added to the original embedding:

$$z_i \leftarrow norm(z_i + FF(\hat{z}_i))$$

where $norm$ is a normalization operation that 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 θ iteratively as follows:

$$\theta \leftarrow \theta - \alpha \nabla_\theta f(\theta)$$

where α 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 θ 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

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

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 TxRank 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 $\log p(x_1|x_{<0}), \ldots, p(x_n|x_{<n})$, the sum of which is the log-likelihood of the trace. Given a DeFi application, TxRank 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 α is an adjustable parameter. Intuitively, increasing α 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 min-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 GPU 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 [20] 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 [18] with its default learning rate and momentum.

VI. Evaluation

This section presents an evaluation of TxRank as an IDS for detecting DeFi attacks. We begin by analyzing TxRank'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 TxRank 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 B).

A. Assumption

In the following we assume that for incidents involving multiple adversarial transactions, it is sufficient for TxRank 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 TxRank by analyzing various alarm threshold configurations. The alarm threshold is a system parameter that users must select for TxRank. 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

Dataset Size (the total number of transactions interacting with the vulnerable smart contract)	Percentage Ranking Alarm Threshold (%)					Absolute Ranking Alarm Threshold		
	$\leq 0.01\%$	$\leq 0.1\%$	$\leq 0.5\%$	$\leq 1\%$	$\leq 10\%$	top-1	top-2	top-3
0 - 99 txs (32 attacks, 28% of dataset)	-	-	-	-	5 (16%)	7 (22%)	20 (63%)	23 (72%)
Average false positive rate	-	-	-	-	8.18%	0%	14.8%	28.3%
Average number of false positives	-	-	-	-	5.1	0	1	2
100 - 999 txs (38 attacks, 33% of dataset)	-	-	8 (21%)	12 (32%)	28 (74%)	7 (18%)	12 (32%)	15 (39%)
Average false positive rate	-	-	0.24%	0.71%	9.65%	0%	0.46%	0.81%
Average number of false positives	-	-	1.5	3.5	39.4	0	1	2
1000 - 9999 txs (17 attacks, 15% of dataset)	-	6 (35%)	9 (53%)	11 (65%)	13 (76%)	4 (24%)	7 (41%)	7 (41%)
Average false positive rate	-	0.054%	0.45%	0.95%	9.96%	0%	0.049%	0.098%
Average number of false positives	-	1.4	11.5	23.7	324.5	0	1	2
10000 + txs (29 attacks, 25% of dataset)	2 (7%)	7 (24%)	16 (55%)	18 (62%)	21 (72%)	2 (7%)	3 (10%)	4 (14%)
Average false positive rate	0.007%	0.097%	0.50%	1%	10%	0%	0.004%	0.008%
Average number of false positives	2.5	120.1	429.9	819.6	7302.1	0	1	2
Overall	2 (2%)	13 (11%)	33 (28%)	41 (35%)	67 (58%)	20 (17%)	42 (36%)	49 (42%)
Average false positive rate	0.007%	0.077%	0.42%	0.90%	9.71%	0%	7.19%	13.5%
Average number of false positives	2.5	65.3	211.9	367.2	2368.5	0	1	2

Fig. 4. This table presents the performance of TxRank 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.

Victim Name	Victim Contract	Application Categories	Damage (in USD)
Beanstalk	0xc1e0..24c5	Stablecoin	181,500,000
MonoX	0x66e7..ee63	DEX	31,133,333
PopsicleFinance	0xd63b..3546	Yield farming	20,700,000
PrimitiveFinance	0x9dae..f2f9	Derivatives	13,000,000
PunkProtocol	0x929c..49d6	Others	8,950,000
VisorFinance	0xc9f2..14ef	Others	8,200,000
DAOMaker	0xd6c8..b1ec	Others	4,000,000
DAOMaker	0x933f..2a13	Others	4,000,000
DODO	0x051e..a2b6	DEX	3,800,000
DODO	0x509e..41fb	DEX	3,800,000
CheeseBank	0x833e..743d	Digital Bank	3,300,000
dydx	0x5377..ba2c	Derivatives	2,211,000
RevestFinance	0xe952..1659	Others	2,005,000
BTFinance	0x3ec4..8af0	Yield farming	1,600,000
VisorFinance	0x65bc..054f	Others	975,720
WildCredit	0x7b3b..c6ca	Lending	650,000
SharedStake	0xa231..7ef5	Others	500,000
88mph	0x2165..b0a6	Lending	100,000
SanshuInu	0x35c6..7810	Others	100,000
KlondikeFinance	0xacbd..e747	Synthetic assets	22,116

Fig. 5. The 20 attacks ranked by TxRank IDS as the most abnormal transaction that interacted with the respective victim contract. TxRank successfully identifies the most abnormal transaction for 18 unique DeFi protocols across various application categories, with the total damage value amounting to over 276 million USD.

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.

C. Effectiveness

To provide a comprehensive analysis, we evaluate the effectiveness of TxRank 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, TxRank 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. TxRank 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, TxRank 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 TxRank captures. For example, consider the case of smart contracts with 1000-9999 transactions. With an alarm threshold of 0.1%, TxRank 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 TxRank is positively correlated with the size of the dataset. For instance, 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 TxRank 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 TxRank when handling a large number of transactions. For instance, TxRank 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, TxRank 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, TxRank 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 TxRank can provide a manageable number of alerts for further investigation, making it particularly suitable for high-volume transaction environments.

E. Performance

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

F. Precision, Recall and F_1 Scores

This section aims to visualize traditional machine learning metrics for our tool, TxRank, 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 ($\frac{\text{TP}}{\text{TP+FP}}$) is the fraction of adversarial transactions among alarms raised.

$$\text{Prec} = \frac{\text{number of adversarial txs TxRank captures}}{\text{number of alarms TxRank raises}}$$

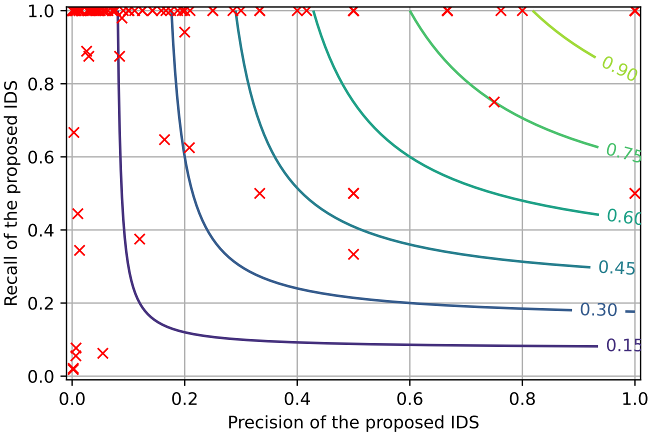

Fig. 6. The figure presents a detailed evaluation of the performance of the proposed TxRank IDS by analyzing its F_1 score for each previously compromised DeFi application in our dataset. Each data point represents the precision and recall achieved by the TxRank IDS for a specific DeFi application, thereby highlighting its effectiveness in detecting various types of attacks. The contour lines depict the F_1-score as a bivariate function of the precision and recall of the IDS, offering a comprehensive view of the model's performance across different applications and showcasing its ability to provide real-time intrusion detection for blockchains such as Ethereum.

- Recall ($\frac{\text{TP}}{\text{TP+FN}}$) is the fraction of attacks that are detected.

$$Recall = \frac{\text{number of adversarial txs TxRank captures}}{\text{total number of adversarial txs}}$$

- F_1-score is the harmonic mean of precision and recall.

$$F_1 = \frac{2 \cdot \text{Prec} \cdot \text{Recall}}{\text{Prec} + \text{Recall}}$$

Figure 6 presents the best F1-score that TxRank 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 weighs 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_β-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 β is thoroughly justified quantitatively. For a more quantitative method to evaluate

TxRank, we recommend referring to Appendix B, where we discuss the Intrusion Detection Capability Score (CID score).

G. Impact of Flash Loans on TxRank's Efficiency

We proceed to examine the effectiveness of TxRank 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 TxRank 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 TxRank's performance on these attacks as follows:

1) In attacks involving a flash loan, TxRank 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, TxRank ranks the adversarial transaction among the top-3 most abnormal transactions for 31 out of 90 cases (34%).
3) The current results indicate that TxRank has a higher success rate in detecting flash loan attacks while also exhibiting its capability to detect non-flash loan attacks.

TxRank 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 TxRank 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, TxRank 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 TxRank. Overall, TxRank ranks 49 attacks in our attack dataset as top-3 abnormal transactions, demonstrating enhanced performance compared to the heuristic-based baseline. Importantly, TxRank 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 TxRank 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, TxRank 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 [23]–[28].
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 [27], [29]–[39].
5) Formal verification involves using mathematical methods to prove that a smart contract meets certain security properties [27], [48]–[50]. 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

Technique	Assumed Prior Knowledge	Searchspace Unrestricted From Vulnerability Patterns	Real-Time Capable	Application Agnostic
Rank based – the goal is to find all unexpected execution patterns, implicitly capturing vulnerabilities				
TxRank (this paper)	All historical transactions	Unrestricted	●(0.16s)	●
Reward based – the goal is to extract financial revenue, implicitly capturing vulnerabilities				
APE [21]	N/A	Only profitable patterns	●(0.07s)	●
Naive Imitation [6]	N/A	Only profitable patterns	●(0.01s)	●
DeFiPoser [12]	DApp models	Only profitable patterns + Limited by the DApp models	●(5.93s)	○
Pattern based – the goal is to match / classify predefined known vulnerability patterns with rules (including machine learning methods)				
Pattern based dynamic analysis [19], [22]	Rule	Limited by the rule	●	◐
Pattern based fuzzing [23]–[28]	Rule + ABI / DApp models	Limited by the rule	◐	◐
Pattern based symbolic execution [27], [29]–[39]	Rule + Source code / Bytecode	Limited by the rule	N/A	◐
Pattern based static analysis [22], [34], [40]–[47]	Rule + Source code / Bytecode	Limited by the rule	N/A	◐
Proof based – the goal is to prove that a set of smart contracts meet specific security properties				
Formal verification [27], [48]–[50]	Formal security properties + Source code / DApp models	Limited by the security properties	N/A	◐

Fig. 7. Systematization of intrusion detection / prevention techniques. Unlike reward-based approaches, TxRank employs an unrestricted search space, enabling it to identify unexpected execution patterns instead of focusing solely on profitable vulnerabilities. In contrast to pattern-based techniques (dynamic analysis, fuzzing, symbolic execution, and static analysis), TxRank does not rely on predefined rules or patterns, which allows it to detect a broader range of anomalies. Furthermore, TxRank is capable of real-time analysis, a feature not present in pattern-based symbolic execution or static analysis methods.

control flow. Static analysis can be used to detect various types of vulnerabilities, such as uninitialized variables and unsafe function calls [22], [34], [40]–[47].

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 environments [22].

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 [51].

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 [52] 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.* [53] 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 TxRank, an innovative transaction anomaly ranking tool for Ethereum-based blockchains. By analyzing a rich dataset spanning four years, TxRank demonstrated its ability to accurately identify attack transactions amidst a highly imbalanced dataset of benign and adversarial transactions. Our results indicate that TxRank effectively detects abnormal transactions, ranking 49 out of 124 attacks among the top-3 most abnormal transactions interacting with their corresponding victim contracts.

TxRank showcases its real-time capabilities with an average batch throughput of $2,284 \pm 289$ 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.

References

[1] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) incidents," *arXiv preprint arXiv:2208.13035*, 2022.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] K. Wüst and A. Gervais, "Do you need a blockchain?" in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 2018, pp. 45–54.

[4] F. Saleh, "Blockchain without waste: Proof-of-stake," *The Review of financial studies*, vol. 34, no. 3, pp. 1156–1190, 2021.

[5] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[6] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 198–214.

[7] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "Sok: Research perspectives and challenges for bitcoin and cryptocurrencies," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 104–121.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[9] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[10] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the defi ecosystem with flash loans for fun and profit," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 3–32.

[11] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 428–445.

[12] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," *arXiv preprint arXiv:2103.02228*, 2021.

[13] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. De Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, 2009.

[14] K. Wang, "Learning scalable and precise representation of program semantics," *arXiv preprint arXiv:1905.05251*, 2019.

[15] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," *Advances in neural information processing systems*, vol. 32, 2019.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[18] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[19] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He, and K. Ren, "Defiranger: Detecting price manipulation attacks on defi applications," *arXiv preprint arXiv:2104.15068*, 2021.

[20] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[21] K. Qin, S. Chaliasos, L. Zhou, B. Livshits, D. Song, and A. Gervais, "The blockchain imitation game," 2023.

[22] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," 2019.

[23] C. Ferreira Torres, A. K. Iannillo, A. Gervais *et al.*, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *European Symposium on Security and Privacy, Vienna 7-11 September 2021*, 2021.

[24] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[25] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[26] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[27] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[28] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[29] https://github.com/nveloso/conkas.

[30] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*. Springer, 2018, pp. 513–520.

[31] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1591–1607.

[32] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[33] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[34] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[35] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[36] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[37] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.

[38] T. D. Nguyen, L. H. Pham, and J. Sun, "Sguard: Smart contracts made vulnerability-free," 2021.

[39] S. So, S. Hong, and H. Oh, "Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[40] A. Ghaleb, J. Rubin, and K. Pattabiraman, "etainter: Detecting gas-related vulnerabilities in smart contracts," 2022.

[41] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[42] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[43] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 127–137.

[44] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[45] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[46] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on*

Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, 2019, pp. 8–15.

[47] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[48] S. Azzopardi, J. Ellul, and G. J. Pace, “Monitoring smart contracts: Contractlarva and open challenges beyond,” in *International Conference on Runtime Verification*. Springer, 2018, pp. 113–137.

[49] J. Frank, C. Aschermann, and T. Holz, “{ETHBMC}: A bounded model checker for smart contracts,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2757–2774.

[50] I. Grishchenko, M. Maffei, and C. Schneidewind, “Ethertrust: Sound static analysis of ethereum bytecode,” *Technische Universität Wien, Tech. Rep*, pp. 1–41, 2018.

[51] M. Eshghie, C. Artho, and D. Gurov, “Dynamic vulnerability detection on smart contracts using machine learning,” in *Evaluation and Assessment in Software Engineering*, 2021, pp. 305–312.

[52] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” *arXiv preprint arXiv:1802.05365*, 2018.

[53] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[54] C. Rijsbergen, “Information retrieval 2nd ed buttersworth,” *London [Google Scholar]*, 1979.

[55] A. A. Cárdenas, J. S. Baras, and K. Seamon, “A framework for the evaluation of intrusion detection systems,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*. IEEE, 2006, pp. 15–pp.

[56] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skorić, “Measuring intrusion detection capability: An information-theoretic approach,” in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, 2006, pp. 90–101.

[57] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[58] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” *arXiv preprint arXiv:1310.4546*, 2013.

[59] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[60] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the em algorithm,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[61] A. A. Neath and J. E. Cavanaugh, “The bayesian information criterion: background, derivation, and applications,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 2, pp. 199–203, 2012.

[62] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” *arXiv preprint arXiv:1607.06450*, 2016.

APPENDIX A
F_{beta} AND $F_1 0$

F_β-score is a generalization of the F_1-score, which “measures the effectiveness of retrieval regarding a user who attaches β times as much importance to recall as precision” [54]. Figure 8 visualizes the best $F_1 0$ score that TXRANK can achieve and the corresponding precision and recall, which attaches 10 times as much importance to recall as precision.

APPENDIX B
INTRUSION DETECTION CAPABILITY (C_{ID})

Traditional metrics such as false alarm rate, detection rate, and F-score may not provide a complete picture when the dataset is imbalanced and the cost of false negatives is high [55]. 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} [56], 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.

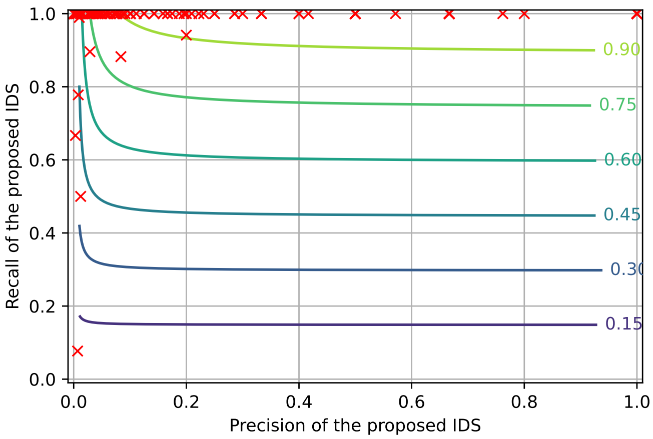

Fig. 8. Identical to Fig. 6, except with the F_1-score replaced with the F_{10} score (F_β score with $\beta = 10$). Instead of weighing the precision and recall of the IDS equally like the F_1 score, the F_{10} score attaches 10 times as much importance to recall as precision, reflecting the fact that an undetected attack costs significantly more than a false alarm for DeFi applications.

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.

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

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

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

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 ration 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 $(x, y) \in \mathcal{X} \times \mathcal{Y}$. The *cost-aware* mutual information between X and Y is then

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

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

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 $C_{\text{ID}}^{(\gamma)}$ is given by

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

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 operator 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 a 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.

Figure 9 shows the distribution of C_{ID} for 124 attacks in relation to the probability of false alarms (P_{FA}) and the fraction of detected attacks (P_D). The figure allows for an easy comparison of the performance of the IDS across different smart contracts. By using an example of a C_{ID} of 0.3, we can see that if the false positive rate is 0.4, the probability of a true positive is 0.24. It is important to note that, despite a high probability of false alarms, 117 out of the 124 attacks achieved a C_{ID} score of more than 0.9. This is because the cost of a false negative is much higher than the cost of a false positive, which results in a higher weighting of P_D.

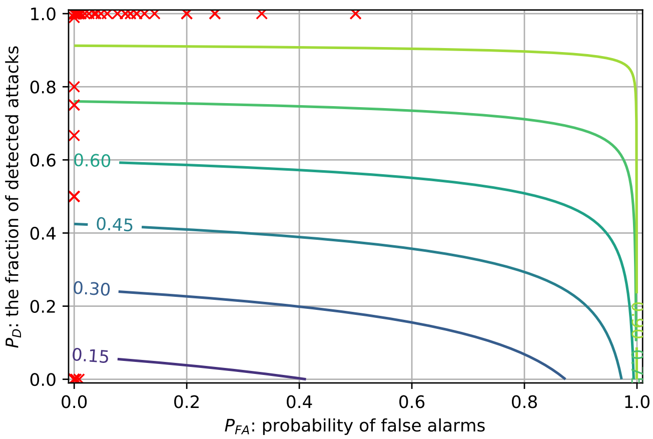

Fig. 9. The Intrusion Detection Capability (C_{ID}) of the proposed IDS. The x-axis corresponds to P_{FA}, the probability of the IDS raising false alarms (FA), while the y-axis corresponds to P_D, the fraction of attacks detected the IDS. We also plotted the contour lines of C_{ID} as a bivariate function of P_{FA} and P_D. For example, given a C_{ID} of 0.3, we find that if the false positive rate is 0.4, then the probability of a true positive is 0.24.

Appendix C
Baseline - doc2vec

To compare TxRank to a naive baseline, as a proof of concept, we prototyped a transaction ranker that estimates the likelihood of transactions using doc2vec and Gaussian mixture. The prototype ranks 8 as the least likely (most abnormal) transaction that interacted with their victim contracts, 11 as the second least likely, and 5 as the third least likely. Our results show that TxRank identifies abnormal transactions by ranking 49 out of 124 attacks among the top-3 most abnormal transactions interacting with their victim contracts.

1) Word2Vec: A prominent embedding approach to learn the distributed representation of words is Word2Vec. Mikolov *et al.* [57], [58] propose two models: the continuous bag-of-words (CBOW) model that predicts the center word given its surrounding context and the skip-gram (SG) model that predicts the surrounding context given a center word.

2) Doc2Vec: Motivated by Word2Vec, Le *et al.* [59] propose Doc2Vec that represents input documents as dense vectors. Doc2Vec also has two models: the distributed memory model of paragraph vectors (PV-DM) and distributed bag of words version of paragraph vector (PV-DBOW). PV-DM 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

$\mathbb{D} = \{d_1, d_2, d_3, ...\}$ and a sentense $s(d_i) = \{w_1, w_2, w_3, ...\}$ composed of words from document d_i, it learns the embeddings $\vec{d_i} \in \mathbb{R}^\delta$ and $\vec{w_i} \in \mathbb{R}^\delta$. This is done by considering a word $w_i \in s(d_i)$ and maximizing $\sum_{j=1}^{l} \log Pr(w_i|d_i)$ where the probability of w_i occuring in the context d_i is $Pr(w_i|d_i) = \frac{\exp \vec{d_i}.\vec{w_i}}{\sum_{w \in \mathbb{V}} \exp \vec{d}.\vec{w}}$. The key computational challenge to the approach is to compute the denominator $\sum_{w \in \mathbb{V}} \exp \vec{d}.\vec{w}$, as the vocabulary size $|\mathbb{V}|$ can grow drastically as the corpus size grows. [59] 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 occuring is given by

$$\begin{aligned}\log \Pr[w] &= \log \Pr[n_1, \ldots, n_k|w] = \log \prod_{j=1}^{k} \Pr[n_j|w] \\ &= \sum_{j=1}^{k} \log \Pr[n_j|w] = \sum_{j=1}^{k} \log \sigma(v_{n_j}^T w)\end{aligned}$$

  where σ 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 $\sum_{w \in \mathbb{V}} \exp \vec{d}.\vec{w}$ by only summing over a random sample of the vocabulary $\mathbb{V}$. Mathematically,

$$Pr(w_i|d_i) = \frac{\exp \vec{d_i}.\vec{w_i}}{\sum_{w \in \mathbb{V}'} \exp \vec{d}.\vec{w}}$$

  where $V' \subset V$ is chosen randomly. Clearly, the efficiency of the solution depends on how to draw samples from the vocabulary. [59] proposes to draw samples from the vocabulary following the rule below:

$$P(w) = 1 - \sqrt{\frac{t}{f(w)}}$$

  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 μ and covariance Σ, given by

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

where $\det \Sigma$ is the determinant of the covariance matrix Σ.

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 π, μ, and Σ 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 [60].

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_x assigned to an observation x is given by

$$c_x = \operatorname*{arg\,max}_{c=1,\ldots,C} \log \pi_c + \log \phi_{\mu_c,\Sigma_c}(x)$$

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:

$$\begin{aligned}\log \phi_{\mu_c,\Sigma_c}(\{x : c_x = c\}) &= \log \prod_{x:c_x=c} \phi_{\mu_c,\Sigma_c}(x) \\ &= \sum_{x:c_x=c} \log \phi_{\mu_c,\Sigma_c}(x)\end{aligned}$$

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

$$\operatorname*{arg\,min}_{S} \sum_{i=1}^{K} \sum_{x \in S_i} |x - \mu_i|^2$$

where $S = (S_1, \ldots, S_K)$ is a tuple of K disjoint sets (clusters) such that $\cup_{i=1}^{K} S_i = \{x_j\}_{j=1}^{n}$

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.

$$\Sigma = \begin{bmatrix} \Sigma_1 & & \\ & \ddots & \\ & & \Sigma_d \end{bmatrix}$$

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) [61]

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

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) [62] particularly helpful in this case. Given a d-dimensional vector v, LayerNorm normalizes the vector as $\hat{v} = \frac{v-\mu}{\sigma}$, where μ is the mean of all components of v, given by $\mu = \frac{1}{d} \sum_{i=1}^{d} v_i$ and σ 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.

Victim Name	Victim Contract	Application Categories	Damage (in USD))
SorbetFinance	0x14e6..e4bd	Others	27,000,000
InverseFinance	0x39b1..db15	Lending	15,600,000
WarpFinance	0xae46..c8cf	Lending	7,800,000
DAOMaker	0xa43b..9289	Others	4,000,000
DAOMaker	0x2fd6..5940	Others	4,000,000
GemSwap	0x7755..4dab	DEX	1,300,000
GemSwap	0xd361..11e1	DEX	1,300,000
GemSwap	0x4265..eab6	DEX	1,300,000
GemSwap	0xf3d1..0e12	DEX	1,300,000
GemSwap	0xa416..bdf6	DEX	1,300,000
GemSwap	0xefcb..5b8f	DEX	1,300,000
GemSwap	0x748f..190d	DEX	1,300,000
GemSwap	0x8770..fb90	DEX	1,300,000
GemSwap	0x8cc2..0a19	DEX	1,300,000
GemSwap	0xe777..b02d	DEX	1,300,000
BasketDAO	0x4622..21b8	DAO	1,200,000
Li.Finance	0x5a9f..6ed1	DEX aggregator	600,000
BuildFinance	0x3157..e758	DAO	470,000
SashimiSwap	0xe4fe..9410	DEX	200,000
Formation.Fi	0xcb6a..a723	Yield farming	100,000

Fig. 10. The 20 attacks ranked by TxRank IDS as the *second* most abnormal transaction that interacted with the respective victim contract.

Victim Name	Victim Contract	Application Categories	Damage (in USD))
IndexedFinance	0x5bd6..dee4	Others	16,000,000
ValueDeFi	0xddd7..1101	Yield farming	7,200,000
DAOMaker	0xdd57..2167	Others	4,000,000
DAOMaker	0x6e70..2e22	Others	4,000,000
GemSwap	0x8cc7..214e	DEX	1,300,000
Vether	0x7557..1d8b	Others	900,000
Chainswap	0xc518..647f	Cross chain bridge	800,000

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

Appendix E
Second and Third Most Abnormal Transactions

We present the results of our analysis on the transactions that TxRank identified as the second and third most abnormal. Figure 10 showcases the 20 attacks ranked as the second most abnormal transaction, while Figure 11 highlights the 7 attacks that were ranked as the third most abnormal transaction. These tables provide insights into the victim names, victim contracts, application categories, and damages (in USD) associated with each attack, further demonstrating the effectiveness of TxRank in detecting various types of malicious activities on the Ethereum blockchain.