TxLens: Scalable Real-Time Detection of Malicious Ethereum Transactions

Tamer Abdelaziz
NYU Abu Dhabi
tamer.m@nyu.edu

Karim Ali
NYU Abu Dhabi
karim.ali@nyu.edu

***Abstract*—Smart contracts enable decentralized applications through verifiable, self-executing logic on public blockchains. While their immutability and composability provide benefits, these properties also amplify risk: code flaws become permanent attack surfaces, and successful exploits often cause immediate, significant financial losses. Traditional security defense relies on pre-deployment code analysis (e.g., static analysis, symbolic execution) or rule-based monitoring of known attack patterns. These approaches, however, fail to address the dynamic, state-dependent, and novel exploit strategies that emerge during live contract execution, leaving a critical gap in runtime protection.**

To bridge this gap, we present TxLens, an ML-based tool for real-time detection of malicious Ethereum transactions. To reconstruct their full execution context, TxLens continuously monitors the mempool and simulates pending transactions against the latest chain state. From this simulation, TxLens extracts behavioral and structural features to classify each transaction as either benign or belonging to one of five high-impact security vulnerability classes.

We have evaluated TxLens on a large labeled dataset of real-world exploits, achieving a minimum F1-score of 98.9%. TxLens outperforms 14 state-of-the-art detection tools by an average of 29.4% in F1-score while maintaining low latency ($<$ 4 seconds for single-transaction analysis and $\approx$1 second per transaction in batch mode). TxLens provides a practical, proactive defense layer that operates before on-chain transaction inclusion and confirmation, enabling timely mitigation. We release TxLens to the community upon request to foster adoption and further research.

***Index Terms*—Blockchain Security, Machine Learning, Vulnerability Detection, Ethereum Transactions, Real-time Analysis.**

I. Introduction

Smart contracts have transformed blockchain platforms into engines for decentralized finance, digital assets, and automated agreements [1]. By enabling trustless, self-executing code on blockchain networks, they have catalyzed immense economic activity and fueled the proliferation of decentralized finance (DeFi) protocols [2] and non-fungible token (NFT) [3], driving unprecedented growth and inclusivity in global finance. The global blockchain market is undergoing rapid expansion, with its valuation expected to rise from approximately US$ 33 billion in 2025 to US$ 393 billion by 2030 [4]. However, the very properties that make smart contracts powerful (i.e., immutability and permissionless composability) also create severe security risks. A single vulnerability, once deployed, becomes a permanent attack surface. Exploits that manipulate contract state or leverage complex interactions between contracts have led to catastrophic financial losses exceeding $3.3 billion in 2025 alone, a 37% increase from the previous year, with Ethereum blockchain being the most targeted network [5].

Existing security defense paradigms fundamentally fail to mitigate runtime threats. As recent empirical studies demonstrate, pre-deployment code analysis tools (e.g., static analyzers and symbolic executors [6]–[10]) suffer from severe false-positive rates (up to 32.6%), excessive analysis runtimes (700+ seconds), and consequently, low developer trust [11]. Crucially, these tools cannot reason about dynamic execution contexts or multi-contract state interactions. Rule-based monitoring systems [12], [13] rely on static signatures, making them brittle against novel or obfuscated exploits and further exacerbating alert fatigue. While machine learning (ML) offers generalizability, existing applications either target account-level fraud [14], [15] or rely on opaque models that lack actionable intelligence for rapid intervention.

These shortcomings expose a critical defense gap: the inability to detect malicious transactions *before* on-chain confirmation. For Ethereum, the mempool [16] is the final window for such proactive defense. Bridging this gap necessitates a real-time tool that abandons static code inspection in favor of dynamic behavioral transaction analysis, delivering highly accurate, actionable threat classification to security responders.

In this paper, we introduce TxLens, an ML-based tool for real-time detection of malicious Ethereum transactions. TxLens addresses the runtime detection gap through two operational modes: (1) continuous monitoring of the mempool for any transaction targeting a user-specified contract address, and (2) on-demand analysis of any transaction by hash. In both modes, TxLens simulates transactions against the latest block to capture execution context, and extracts a rich set of structural and behavioral features. TxLens then employs its detection model, which we have trained on a labeled dataset of known exploits, to classify a given transaction as benign or as one or more of five vulnerability classes: *reentrancy*, *parity wallet hack 1*, *parity wallet hack 2*, *integer overflow/underflow*, and *unhandled exception*. This process completes in under four seconds, providing a security agent with an actionable alert and a vital window to mitigate the threat (e.g., by pausing the contract or invalidating the transaction—before it is finalized).

We have evaluated TxLens on a large, manually labeled benchmark of real-world exploits and benign transactions [17]. Across all targeted vulnerability classes, TxLens achieves a minimum F1-score of 98.9%. Compared to 14 state-of-the-

art analysis tools, TXLENS improves the average F1-score by 29.4%, demonstrating superior accuracy and robustness while maintaining the low latency required for live deployment. Consequently, TXLENS provides a practical, proactive layer of defense that complements pre-deployment audits and rule-based monitoring, equipping developers and security teams with a critical tool to safeguard live contracts against evolving threats.

II. BACKGROUND

A. Smart Contracts and Execution Context

Smart contracts are immutable, self-executing programs deployed on blockchains such as Ethereum [18]. They enable decentralized applications by automating agreements without intermediaries. After smart contract deployment, developers cannot alter their code, turning any code vulnerability into a permanent attack surface.

Transactions are the atomic units of interaction. An externally owned account (EOA) signs an external transaction and broadcasts it to the network to enter the *mempool* before being included in a block. The mempool is a public pool of pending transactions awaiting selection by validators [16]. During its mempool residence (typically seconds to minutes [19]), a transaction may be inspected without affecting on-chain state. This pre-inclusion window is the critical opportunity for real-time detection and intervention.

When executing, a transaction may trigger a series of internal transactions (or message calls) between contracts. The complete sequence, including all state changes, forms the transaction's execution trace. Leveraging this process, TXLENS simulates pending transactions against the latest chain state to reconstruct their full trace and analyze runtime behavior before block inclusion.

B. High-Impact Vulnerabilities

We target five historically significant vulnerability classes [20] that are exploitable through crafted transactions. These vulnerabilities represent critical failures in smart contract logic, state management, and inter-contract communication:

- *reentrancy*: A critical state-synchronization flaw occurring when a contract makes an external call to an untrusted destination before finalizing its internal state updates (e.g., deducting user balances). This allows an attacker to leverage a malicious fallback function to recursively call back into the vulnerable function, repeatedly draining funds or altering state before the initial execution context terminates (e.g., the 2016 DAO hack) [21], [22].
- *parity wallet hack 1*: Exploits the improper use of the `delegatecall` opcode combined with a lack of access controls on initialization functions. Because `delegatecall` executes a target library's code within the calling contract's state context, an attacker can invoke an unconstrained initialization function to overwrite critical storage variables and instantaneously seize ownership of the contract [23].
- *parity wallet hack 2*: Arises when a shared library contract is left uninitialized and lacks strict access controls. An attacker can claim direct ownership of the library itself and invoke the `selfdestruct()` opcode, permanently removing its bytecode from the blockchain. Consequently, any dependent proxy contracts relying on this library are rendered permanently inert, leading to the irreversible freezing of all associated assets [24].
- *integer overflow/underflow*: Prior to Solidity version 0.8.0, the EVM did not inherently revert arithmetic operations that exceeded or fell below fixed bit boundaries (e.g., `uint256`). Attackers exploit these silent integer overflows and underflows to bypass authorization checks, manipulate token balances, or corrupt the contract's intended execution flow [25].
- *unhandled exception*: Unlike standard function calls, low-level EVM operations (e.g., `call()`, `send()`) do not automatically propagate exceptions; instead, they return a boolean `false` upon failure. If the calling contract fails to explicitly verify this return value, its execution continues under the false assumption that the external interaction succeeded, leading to severe logical inconsistencies and potential financial loss [26].

Static analysis tools, which inspect source code or bytecode offline, struggle to detect these vulnerabilities when they arise from dynamic, cross-contract interactions at runtime [11], [27]. While rule-based monitors may flag known patterns, they fail against novel or obfuscated exploits. TXLENS addresses this gap by analyzing transaction behavior in real time.

III. OVERVIEW OF TXLENS

A. Design and Operation Modes

TXLENS shifts smart-contract security from reactive post-exploit detection to proactive pre-execution prevention. It operates in two distinct modes to accommodate different security workflows, with both modes centered on analyzing transactions before they are finalized on-chain.

- Mode 1: Continuous Mempool Monitoring. A user (typically a contract owner or security agent) provides one or more smart contract addresses that they aim to protect. TXLENS then continuously monitors the public Ethereum mempool [16]. For every pending transaction that targets a registered address, TXLENS immediately captures and analyzes it in real time.
- Mode 2: On-Demand Transaction Analysis. A user provides a specific transaction hash. TXLENS fetches, simulates, and analyzes this transaction regardless of its state, whether it is still pending in the mempool or already confirmed on-chain. This mode supports forensic investigation and the evaluation of historical transactions.

Both modes rely on the same core detection pipeline to assess transactions. The key innovation is that in Mode 1, the analysis happens *while the transaction is pending*, providing a window for intervention before block inclusion.

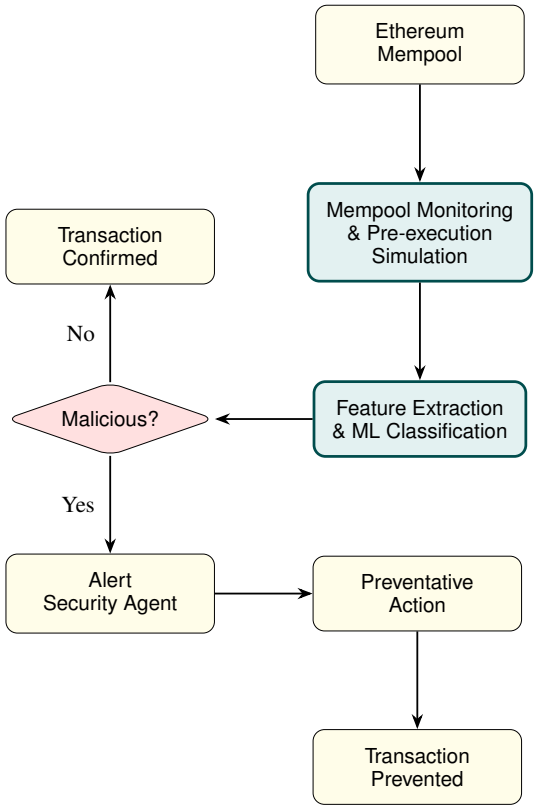

Fig. 1: TxLens workflow: continuous mempool monitoring and transaction simulation → behavioral feature extraction and real-time ML classification → alert security agent to take a prevention action before on-chain inclusion.


```
from_address:                "0xde7e47b63d49a2ff2a209807b1d57513a9968a5b"
to_address:                  "0x5cb073d82d28e76d38c21908fcd213c5cea3a20d"
value:                       "550000000000000000"
gas:                         "170550"
gas_price:                   "39000000000"
input:                       "0x9e5faafc"
receipt_cumulative_gas_used: "140937"
receipt_gas_used:            "107366"
block_timestamp:             "2017-02-11T01:05:58.000Z"
block_number:                "3160801"
logs:
  0:
    address:                 "0x59752433dbe28f5aa59b479958689d353b3dee08"
    block_number:            "3160801"
    block_hash:              "0xdd0ba6f0c0f6429653e234b77bddc048c80538b91ee3a7b61763
    block_timestamp:         "2017-02-11T01:05:58.000Z"
    data:                    "0x0000000000000000000000005cb073d82d28e76d38c21908fcd2
    log_index:               "0"
    transaction_hash:        "0x516e11f52cadd4820a782a548952836bd993f34d0171ec8ffdd3
    transaction_index:       "1"
    transaction_value:       "550000000000000000"
    topic0:                  "0xe1fffcc4923d04b559f4d29a8bfc6cda04eb5b0d3c460751c240
    decoded_event:
      label:                 "Deposit"
      signature:             "Deposit(address,uint256)"
      type:                  "event"
      params:                [ {…}, {…} ]
  1:                         { address: "0x59752433dbe28f5aa59b479958689d353b3dee08"
  2:                         { address: "0x59752433dbe28f5aa59b479958689d353b3dee08"
  3:                         { address: "0x59752433dbe28f5aa59b479958689d353b3dee08"
decoded_call:
  type:                      "function"
  label:                     "attack"
  signature:                 "attack()"
  params:                    []
transaction_fee:             "0.004187274"
internal_transactions:
  0:
    transaction_hash:        "0x516e11f52cadd4820a782a548952836bd993f34d0171ec8ffdd3
    block_number:            3160801
    block_hash:              "0xdd0ba6f0c0f6429653e234b77bddc048c80538b91ee3a7b61763
```


Listing 1: Sample output showing a simulated execution trace from the pre-execution stage.

Figure 1 illustrates the core pipeline of TxLens, which consists of two sequential stages:

1) Pre-execution Simulation: When TxLens receives a target transaction, it first forks the latest canonical chain state. TxLens then executes the transaction in an isolated Ethereum Virtual Machine (EVM) environment using the standard `eth_call` RPC method. This simulation faithfully reconstructs the complete execution trace, including all internal calls, storage modifications, emitted events, and logs (as shown in Listing 1), without committing any state changes to the live chain.
2) Feature Extraction and Classification: TxLens extracts a rich set of structural and behavioral features from the simulated execution trace. TxLens's ensemble ML classifier then analyzes this feature vector to determine whether a given transaction exhibits malicious behavior associated with one or more of the five targeted vulnerability classes.

If its classifier deems a transaction benign, TxLens takes no further action (i.e., the pending transaction proceeds normally towards on-chain confirmation). If the classifier flags the transaction as malicious, TxLens surfaces this result to the user via its interface. This detection occurs in real time, providing a critical intervention window, typically several seconds, before a miner includes the transaction in a block. A contract owner or designated security agent monitoring the tool may then execute a predefined mitigation action, such as triggering an emergency pause function in the contract or broadcasting a transaction with a higher gas fee to invalidate the malicious one. By the time the malicious transaction is processed, the contract state may have already be altered by the mitigation action, rendering the exploit ineffective.

By performing high-fidelity simulation and real-time classification entirely within the pre-inclusion phase, TxLens delivers a practical, proactive defense layer that neutralizes exploits before on-chain finality. Future versions will automate the alerting process by immediately notifying security agents with detailed explanations of detected attacks.

B. Transactions Dataset

1) Data Collection: We build our detection model using a dataset of Ethereum transactions, which Torres et al. [28] manually label and release online [17]. This dataset focuses on real-world attack transactions rather than merely identifying vulnerable contracts. It covers 1,655 unique smart contracts, each associated with one of five prevalent vulnerability classes: *reentrancy*, *parity wallet hack 1*, *parity wallet hack 2*, *integer overflow/underflow*, and *unhandled exception*.

This dataset contains 129,863 annotated Ethereum transactions, categorized as either benign (122,830) or malicious

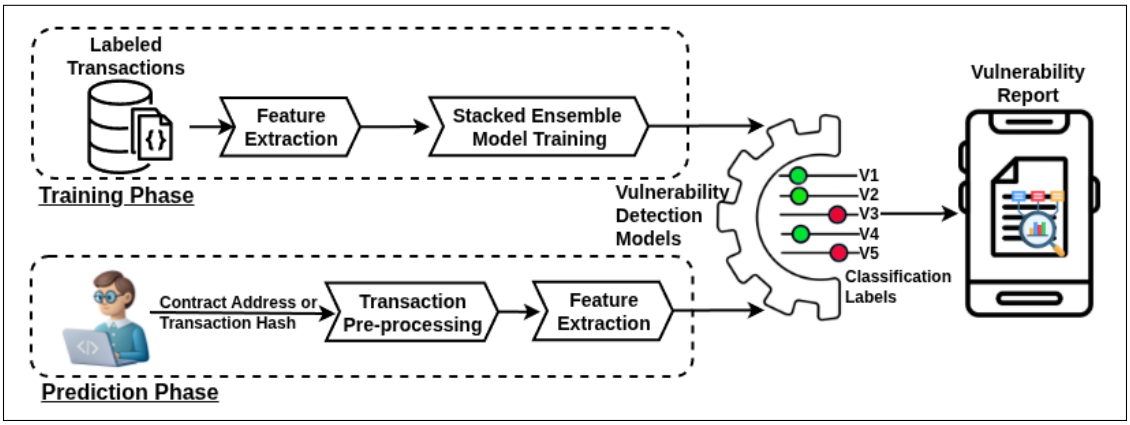

Fig. 2: Overview of the feature extraction and classification architecture of TxLens. In the training phase, the system extracts behavioral and structural features from a large labeled dataset to train a stacked ensemble model capable of detecting five vulnerability types. In the prediction phase, new transaction data is processed using the same feature set, allowing the model to classify transactions and assist developer decision-making.

(7,041). However, it provides only transaction hashes and binary labels. To enable our transaction-level analysis, we use the Moralis API [29] to retrieve full transaction traces and metadata for all 7,041 malicious transactions. To mitigate class imbalance while preserving representativeness, we randomly sample 9,609 benign transactions, resulting in a balanced and analyzable dataset of 16,650 samples for model development.

2) Feature Extraction: For each transaction, TxLens constructs a feature vector that captures transaction-level and account-level behavioral signals.

a) Transaction-level features: are extracted from the execution trace obtained via `eth_call`. They include opcode distribution, gas consumption patterns, depth and count of internal calls, specific function selectors invoked, storage slots accessed (read/written), and events emitted. These features characterize the precise runtime semantics and resource usage of the transaction.

b) Account-level features: model the broader interaction context. They include the pre- and post-execution balances of the originator, the recipient, and any intermediary contracts touched, historical nonce usage of the originator, the number of prior interactions between the involved parties, and aggregate token transfer volumes within the simulated call tree. These features help expose economic intent, privilege escalation patterns, and anomalous control-flow sequences that are invisible to static code analysis.

3) Processing Our Dataset: Following feature extraction, we identify and deduplicate 185 transactions that exhibit identical feature sets, ensuring dataset uniqueness. Following standard practice, we then partition the data for each vulnerability class by randomly allocating 80% of the data for training and reserving 20% for testing. Table I summarizes the final dataset composition. For each vulnerability class, the dataset includes all relevant malicious transactions (*Attack Txs*) and the same set of 9,609 benign transactions (*Benign Txs*), ensuring consistency in comparative analysis across classes.

TABLE I: Overview of the Collected Transactions Dataset.

Vulnerability Type	Total Txs	Benign Txs	Attack Txs
reentrancy	11,546	9,609	1,937
parity wallet hack 1	11,222	9,609	1,613
parity wallet hack 2	9,847	9,609	238
integer overflow/underflow	10,050	9,609	441
unhandled exception	12,236	9,609	2,627

C. Detection Model of TxLens

Recent advancements in smart contract vulnerability detection highlight the effectiveness of ensemble learning techniques [30]–[32]. Building on this, we design a robust ensemble-based detection model for TxLens to identify the five targeted vulnerability classes. Our training pipeline involves three core steps: diverse model training, cross-validation, and stacked ensemble integration.

1) Diverse Model Training: We first train eight distinct ML classifiers: RandomForest [33], ExtraTrees [34], KNeighbors [35], XGBoost [36], CatBoost [37], LightGBM [38], a standard Neural Network (NeuralNet) [39], and a Neural Network via the FastAI library (NeuralNetFastAI) [40]. We select these classifiers for their diverse architectures and learning paradigms, enabling them to capture a broad spectrum of data distributions and vulnerability patterns. This diversity enhances the overall predictive performance of TxLens and its adaptability to different vulnerability classes.

2) Cross-Validation: We employ an 8-fold cross-validation strategy to improve model robustness and reduce variance. For each fold, we train a base model on the remaining 7 folds and use the held-out fold for evaluation. This process yields 8 instances of each classifier type, providing out-of-fold (OOF) predictions. We concatenate these OOF predictions to compute unbiased evaluation metrics against the ground truth. During inference, the individual models within each classifier type average their predictions to produce a final output.

3) Stacked Ensemble Integration: We apply a three-layer stacked ensemble to integrate the predictive strengths of all

classifiers. The first layer consists of the base models trained on the original features. Subsequent layers use the predictions from the previous layer as additional input features. The final layer employs a single meta-model, optimized using a greedy weighted ensemble algorithm [41], to combine all previous outputs and generate the ultimate prediction (benign or malicious). We repeat this entire process independently for each vulnerability class.

In total, our process involves training $5 \times (64 \times 2 + 1) = 645$ models (5 vulnerability types, 64 models per type from the cross-validation of 8 classifiers, 2 ensemble layers, and 1 final meta-model per type). This comprehensive ensemble approach enables TXLENS to achieve high accuracy and robustness, aligning with state-of-the-art methodologies for smart contract vulnerability detection.

IV. EVALUATION

A. Machine Setup

We have conducted the model training and evaluation on a machine with 128 GB of memory and a 16-core Intel® Xeon® W3-2435 CPU, running Ubuntu 24.04.1 LTS. Our software environment uses Python 3.12. The Moralis API [29] retrieves blockchain transaction data, and we use SmartBugs 2.0 [42] to run the comparative static analysis tools. The total wall-clock time to train all 645 models is 15 hours.

B. Research Questions

Our evaluation addresses two research questions:

- **RQ1**: How efficient is TXLENS in monitoring live Ethereum transactions at scale, in terms of throughput, latency, and resource cost?
- **RQ2**: How accurate is TXLENS compared to state-of-the-art analyzers for detecting different vulnerability classes?

C. Evaluation Benchmarks

We evaluate TXLENS using two distinct benchmarks, each with a different purpose and granularity.

*1) Transaction-Level Benchmark (*TXBENCH*):* This is the test set, comprising 20% of our collected data that is withheld during model training. TXBENCH provides the ground truth for malicious and benign transactions for the studied five vulnerability classes. We use TXBENCH to measure the classification performance of TXLENS (**RQ1**).

*2) Contract-Level Benchmark (*SCBENCH*):* Since most existing tools operate on the contract level, we curate this independent benchmark after model training to enable a fair comparison between TXLENS and state-of-the-art tools. SCBENCH consists of 644 smart contracts, each associated with an average of 54 transactions. These contracts are collected from diverse public repositories to ensure broad coverage. We establish the positive ground truth for four vulnerability classes—*reentrancy*, *parity wallet hack 2*, *integer overflow/underflow*, and *unhandled exception* —based on previously reported malicious contracts in the literature. For *reentrancy*, we collect 59 malicious contracts, including 34 from a historical reentrancy repository [43] and 25 from the study by Zhou et al. [44]. For *parity wallet hack 2*, we obtain 167 suicidal contracts from the study by Chen et al. [45]. Furthermore, we source 49 malicious contracts that demonstrate *integer overflow/underflow* from the study by Zhou et al. [44], and 47 malicious contracts that exhibit *unhandled exception* from the Smart Bugs Curated dataset [46]. We establish the negative ground truth using 322 audited, high-activity contracts from Etherscan [47], each accompanied by an audit report and having executed at least 100 transactions.

We retrieve transaction data for all contracts from the public Ethereum blockchain using the Moralis API [29]. Since the source code for all contracts is not necessarily available on Etherscan (e.g., due to the presence of self-destructed contracts), we supplement this dataset with bytecode that we obtain from Google BigQuery [48].

To ensure a balanced evaluation, we randomly distribute safe contracts across the vulnerability classes, resulting in equal representation of positive and negative ground truths. This approach ensures a robust and unbiased assessment of all analyzers performance. We use SCBENCH to compare TXLENS performance against 14 state-of-the-art analyzers (**RQ2**). We have made SCBENCH publicly available [49].

D. Metrics

Vulnerability classification outcomes are categorized into four groups: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). These outcomes form the basis for calculating performance metrics that assess the effectiveness of a classification model. Among these metrics, Precision, Recall, and the F1-Score are particularly important for balanced and imbalanced datasets, such as those involving vulnerable transactions/contracts.

Precision measures the proportion of correctly identified positive instances out of all instances predicted as positive. We define it as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall, also known as Sensitivity or True Positive Rate, measures the proportion of correctly identified positive instances out of all actual positive instances. We define it as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

The F1-Score is the harmonic mean of Precision and Recall, providing a single metric that balances both. We define it as:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

E. Performance (RQ1)

We evaluate TXLENS's classification accuracy on the transaction-level benchmark (TXBENCH). Table II presents detailed results across the five vulnerability types. TXLENS achieves perfect F1-scores (100%) for *reentrancy*, *parity wallet hack 1*, and *parity wallet hack 2*. For *integer overflow/underflow* and *unhandled exception*, it achieves F1-scores of

TABLE II: Summary of TXLENS Performance on TXBENCH. The columns include the vulnerability type, the number of transactions analyzed (**Total Txs**), true positives (**TP**), false positives (**FP**), true negatives (**TN**), and false negatives (**FN**). Performance metrics include Precision, Recall, F1-score, and analysis time per transaction in seconds (**Inference Time**).

Vulnerability Type	Total Txs	TP	FP	TN	FN	Precision (%)	Recall (%)	F1-score (%)	Inference Time (sec)
reentrancy	2,310	388	0	1,922	0	100.0	100.0	100.0	0.00067
parity wallet hack 1	2,245	322	0	1,922	1	100.0	100.0	100.0	0.00157
parity wallet hack 2	1,970	48	0	1,922	0	100.0	100.0	100.0	0.00016
integer overflow/underflow	2,010	76	10	1,912	12	98.9	98.9	98.9	0.00354
unhandled exception	2,448	521	13	1,909	5	99.3	99.3	99.3	0.00138

98.9% and 99.3%, respectively. The minimum F1-score across all classes is 98.9%, demonstrating high precision and recall.

We measure efficiency by timing TXLENS's pipeline. In its primary *single-transaction mode* (simulating real-time mempool monitoring), the end-to-end process—from fetching a pending transaction to issuing a classification—completes in under 4 seconds on average. This includes pre-processing (≈1 second), model loading (≈2.5 seconds), and inference (≈0.0014 seconds). This latency offers a practical mitigation window, as transaction inclusion times vary from several seconds to several minutes depending on network conditions. As reported by [19], transactions are processed in a median of 57 seconds, and 90% are processed within 8 minutes. For forensic analysis of multiple transactions, TXLENS batch-loads models once and processes transactions concurrently, reducing the average per-transaction analysis time to 1.1 seconds.

We design TXLENS to operate efficiently on modest computational resources, and it does not require expensive hardware to run. We deploy TXLENS on the free tier of the Hugging Face Spaces platform [50], which offers 2 vCPUs, 16 GB of RAM, and 1 GB of storage. TXLENS runs reliably within these constraints, demonstrating that it achieves its functionality without heavy computation or large memory footprints. This low resource requirement translates into very low operational cost and allows users to host and experiment with TXLENS even on entry-level infrastructure such as free cloud tiers or small local machines.

> TXLENS achieves a minimum F1-score of 98.9% with low latency (under 4 seconds per transaction in single mode and 1.1 seconds in batch mode), providing a mitigation window of several seconds to minutes before on-chain inclusion and confirmation. TXLENS also runs efficiently on modest hardware (2 vCPUs, 16 GB RAM), enabling practical live deployment.

F. Comparison to State-of-the-Art (RQ2)

We benchmark TXLENS against 14 state-of-the-art analyzers on the contract-level benchmark (SCBENCH). The analyzers include static analysis and ML-based tools: CONKAS [51], DLVA [52], DLVAX [53], ETHAINTER [54], ETHOR [55], HONEYBADGER [56], MADMAX [57], MAIAN [58], MYTHRIL [9], OSIRIS [59], OYENTE [8], PAKALA [60], SECURIFY [10], TEETHER [61], and VANDAL [6]. Since these tools analyze contract bytecode and

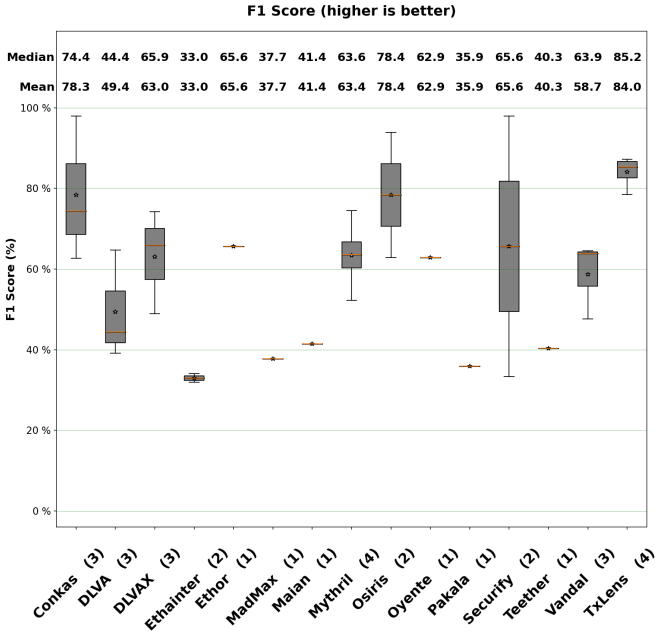

Fig. 3: F1 scores of 15 analyzers on SCBENCH. TXLENS achieves the highest average F1-score (84.0%), outperforming 14 state-of-the-art tools by 29.4% on average.

output a set of vulnerability labels to each contract, we adapt TXLENS's output to ensure a fair comparison. Specifically, we label a contract as vulnerable if TXLENS flags any of its historical transactions as malicious for a given vulnerability type. Otherwise, the contract is considered safe. We execute each tool with a timeout of 1,200 seconds per contract to complete the vulnerability analysis.

We summarize the high-level results of our competitor benchmarking in Figure 3 and Figure 4. The x-axis shows the tools, and in parenthesis the number of vulnerability types that we include in the benchmark for that tool (i.e., not every tool handles all vulnerability types).

Across the four vulnerability types in SCBENCH, TXLENS achieves an average F1-score of 84%, outperforming the average F1-score of the 14 competitors (54.6%) by 29.4%. In terms of speed, TXLENS is the third fastest tool, analyzing a contract (i.e., all its transactions) in 1.4 seconds on average. Nevertheless, TXLENS's speed remains competitive when compared to the other 12 tools.

TXLENS and MYTHRIL are the only two analyzers that

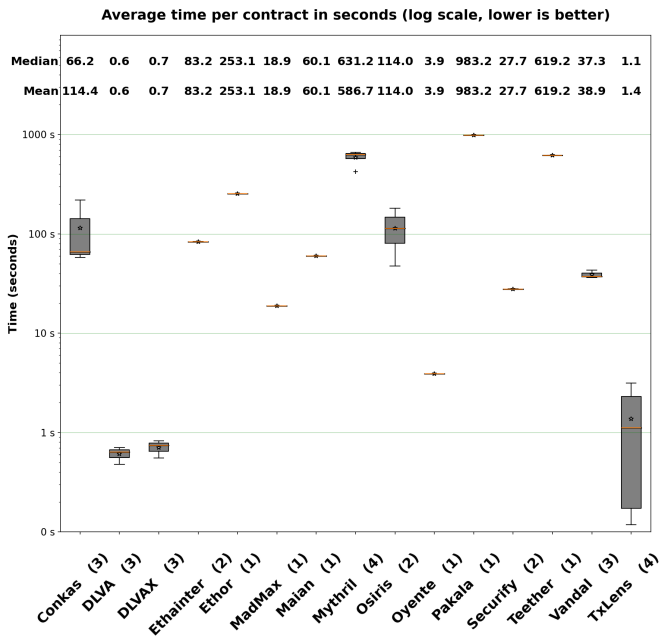

Fig. 4: Analysis time per contract (seconds) for 15 analyzers on SCBENCH. TXLENS (median: 1.1s, mean: 1.4s) is the third fastest tool, demonstrating low latency suitable for real-time deployment.

cover all four types of vulnerabilities in SCBENCH. Four analyzers cover three types, three analyzers cover two types, and six analyzers are only able to cover one type.

TXLENS and ETHAINTER are the only tools that complete all analyses without exceptions. Other tools fail on various contracts due to timeouts, unsupported opcodes, or memory errors. Specifically, ML-based tools such as DLVA and DLVAX experience exceptions during the construction of the Control-Flow Graph (CFG) of input contracts. ETHOR ranks fourth in generating exceptions, because it excludes contracts containing the `DELEGATECALL` or `CALLCODE` opcodes from its scope and occasionally fails to reconstruct the CFG. CONKAS ranks third due to exceeding the maximum recursion depth for complex contracts, which hinders its symbolic execution in identifying vulnerability traces. PAKALA ranks second, primarily due to numerous incomplete analyses and timeouts. Similar to ETHOR, its symbolic execution engine does not support contracts containing the `DELEGATECALL` or `CALLCODE` opcodes. TEETHER generates the most exceptions, often due to memory errors and timeouts.

> TXLENS outperforms the average F1-score of the 14 state-of-the-art tools by 29.4%, delivering high analysis speed with zero exceptions.

Looking at the results of this comparison, we recommend that smart contract developers use multiple analyzers together to minimize false positives. The choice of analyzers should depend on the processing time and efficiency of each tool and the available timeframe to mitigate vulnerabilities. For example, TXLENS may be used to monitor all contract transactions. If it reports a vulnerability, developers can then utilize OYENTE, DLVA, or/and DLVAX, which are relatively fast compared to other bytecode analyzers. If the source code is available, developers may also leverage SLITHER [7] or CONFUZZIUS [62], both of which are comparatively efficient.

G. Discussion

The high performance of TXLENS on transaction-level detection (**RQ1**) validates the effectiveness of simulating pending transactions and analyzing behavioral features. The superior contract-level accuracy (**RQ2**) demonstrates that our transaction-centric approach generalizes better than static code analysis, particularly for vulnerabilities that manifest through complex, state-dependent runtime interactions.

The variation in F1-scores across vulnerability types in SCBENCH stems from a fundamental distinction. TXLENS excels at detecting behavioral vulnerabilities such as *reentrancy* and *parity wallet hack 2*, which produce distinct, observable patterns in execution traces (e.g., recursive external calls, suicidal operations). Conversely, syntactic vulnerabilities such as *integer overflow/underflow* and *unhandled exception* are inherently tied to specific code patterns (e.g., missing arithmetic checks). For instance, *integer overflow/underflow*, as seen in the `batchTransfer` function (targeted on transaction 0xd97d2aa0... [63]), rely on unchecked arithmetic operations (e.g., `uint256 amount = uint256(cnt) * _value`). Detecting such flaws requires static analysis of the source code to identify the absence of safeguards such as SafeMath, because transaction data only includes function signatures and parameters, rather than internal computations.

This difference between behavioral and syntactic vulnerabilities highlights a key complementarity. TXLENS provides unmatched runtime threat detection, while static analyzers remain essential for finding code-level flaws pre-deployment. Therefore, we recommend a combined defense-in-depth strategy.

The performance gap between **RQ1** (98.9% F1) and **RQ2** (84% F1) arises from a fundamental difference in ground truth. In **RQ1**, TXBENCH contains only transactions where the exploit was actually executed, allowing our behavioral model to achieve near-perfect accuracy. In contrast, SCBENCH labels entire contracts as vulnerable based on the presence of flawed code, irrespective of whether an exploit transaction has ever occurred. Some contracts in this benchmark contain vulnerabilities (e.g., integer overflow) that are syntactic and latent (i.e., they exist in the code but have no corresponding malicious transactions in their recorded history). Since TXLENS analyzes transaction behavior and cannot inspect source code, it cannot flag a contract as vulnerable if no exploit transaction targets it. This scenario directly lowers recall in the contract-level evaluation. This result underscores the complementary roles of runtime transaction monitoring and static code analysis in a layered defense strategy.

V. Related Work

Smart-contract security research has evolved along two complementary axes: code analysis for pre-deployment vulnerability detection and transaction analysis for runtime threat identification. While pre-deployment tools dominate the literature [6]–[10], [51], [52], [54]–[62], [64], runtime approaches, particularly those focused on real-time transaction scrutiny, remain underexplored [65].

A. Code Analysis

Code-centric tools analyze smart-contract source code or bytecode offline to identify vulnerabilities before deployment. Static analyzers such as SLITHER [7] use abstract interpretation to detect issues including *reentrancy* and code inefficiencies, whereas symbolic execution tools such as OYENTE [8] and MYTHRIL [9] explore execution paths to uncover unsafe states. Pattern-based approaches, including SECURIFY [10] and VANDAL [6], rely on rule matching and constraint solving to achieve broad coverage. Although many ML-based methods have been proposed for smart-contract vulnerability detection, only a limited number of studies have released their tools to the community, which hinders reproducibility and practical adoption [66]. Despite their effectiveness in analyzing individual contracts, these tools cannot capture runtime exploits caused by dynamic interactions, state dependencies, or post-deployment compositions [65].

B. Transaction Analysis

Transaction-focused methods shift emphasis to on-chain behavior, subdivided into account-level profiling and per-transaction inspection, to detect active threats.

1) Account-Level Analysis: These approaches model blockchain interactions as graphs, analyzing aggregate account histories for anomalies. Phishing detectors [14], [67] use graph features and ML classifiers to flag scam accounts based on transaction volumes and structures. ETHERSHIELD [15] incorporates temporal patterns and contract metadata for scalable malicious-account identification, while BERT4ETH [68] applies transformer models to transaction sequences. However, by aggregating data over time, these methods overlook granular, individual-transaction exploits, limiting their utility for pre-inclusion intervention [69].

2) Transaction-Level Analysis: Targeting specific transactions addresses this limitation. HORUS [28] replays historical transactions in a modified Geth client to reconstruct state for vulnerability checks, enabling precise detection but at high computational cost. TXSPECTOR [70] instruments the EVM to generate execution flow graphs, applying Datalog rules for issues such as *reentrancy* and *integer overflow/underflow*, though requiring client modifications hampers deployability. DYNAMIT [71] employs ML on transaction features for *reentrancy* detection but lacks comprehensive benchmarks and tool availability.

C. Rule-based and Chain Monitors

Blockchain monitoring tools and rule engines (e.g., Forta bots [12], OpenZeppelin Defender [13]) provide near-real-time detection of known attack patterns by scanning mempool transactions and broadcasting alerts. They are effective for well-understood signatures but are brittle against novel or slightly modified exploits and can generate high false positive volumes without careful tuning.

VI. Tool Availability

To foster the continued development and validation of proactive defense strategies, we provide restricted access to the TXLENS interactive web interface for the academic and security research communities. Because the tool's advanced detection and execution-tracing capabilities could be misused by malicious actors, we employ a strict responsible disclosure and access policy [72]. Interested parties must submit an application using their official institutional credentials and sign the formal agreement at https://huggingface.co/spaces/blockchainsecurity/TxLens. The authors will review each request—verifying institutional affiliations and research backgrounds—to ensure responsible deployment and prevent adversarial exploitation.

VII. Conclusion

We present TXLENS, a tool that advances smart contract security from reactive post-exploit detection to proactive, real-time prevention. By bridging the gap between static pre-deployment audits and rigid rule-based monitoring, TXLENS addresses the need for dynamic, context-aware threat detection in live Ethereum environments.

TXLENS combines continuous mempool monitoring with pre-execution simulation to reconstruct the full execution context of pending transactions. This dual-engine design extracts granular behavioral features that are not observable through static analysis alone. Using a stacked ensemble learning model, TXLENS classifies transactions across five high-impact vulnerability classes and achieves a minimum F1-score of 98.9% on real-world transaction data. Our evaluation shows that TXLENS outperforms 14 state-of-the-art detection tools by an average of 29.4% in F1-score. The system also satisfies the low-latency requirements of live deployment, processing individual transactions in under 4 seconds.

By operating entirely in the pre-confirmation phase, TXLENS gives security agents a critical window to intervene before malicious transactions are finalized on-chain. This proactive design reduces financial risk and improves the resilience of decentralized applications.

Our ongoing work focuses on three directions: (1) extending detection to emerging vulnerability types and composite attacks; (2) developing automated response mechanisms that can trigger circuit breakers or pause contracts upon detection; and (3) extending support to other EVM-compatible chains and Layer-2 scaling solutions to strengthen security across the broader decentralized ecosystem.

REFERENCES

[1] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, no. 22-23, pp. 5–7, 2013.

[2] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 30–46.

[3] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token (nft): Overview, evaluation, opportunities and challenges," *arXiv preprint arXiv:2105.07447*, 2021.

[4] MarketsandMarkets, "Blockchain market by component, provider, type, organization size, application, and region - global forecast to 2030," https://www.marketsandmarkets.com/Market-Reports/blockchain-technology-market-90100890.html, 2025, accessed: 2025-10-15.

[5] CertiK, "Hack3d: *The Web3 Security Report 2025*," https://www.certik.com/resources/blog/hack3d-the-web3-security-report-2025, 2025.

[6] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[7] 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.

[8] 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.

[9] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, p. 54, 2018.

[10] 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.

[11] T. Abdelaziz, S. Alsaghir, and K. Ali, "Where do smart contract security analyzers fall short?" in *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*, 2026.

[12] F. Network, "Forta bot: Early attack detector beta research," Forta Foundation, 2024, accessed: May 2025. [Online]. Available: https://app.forta.network/bot/0x51133a1e4c99af29b56647aa3fe7a9a2d10763e6f66c07f2559b360418fb4f7b

[13] OpenZeppelin, "Openzeppelin defender docs: Monitor module," OpenZeppelin, 2024, accessed: May 2025. [Online]. Available: https://docs.openzeppelin.com/defender/module/monitor

[14] L. Chen, J. Peng, Y. Liu, J. Li, F. Xie, and Z. Zheng, "Phishing scams detection in ethereum transaction network," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 1, pp. 1–16, 2020.

[15] B. Pan, N. Stakhanova, and Z. Zhu, "Ethershield: Time-interval analysis for detection of malicious behavior on ethereum," *ACM Transactions on Internet Technology*, vol. 21, no. 1, pp. 1–30, 2024.

[16] Etherscan, "Pending transactions mempool," https://etherscan.io/txsPending, 2025, accessed: October 16, 2025. [Online]. Available: https://etherscan.io/txsPending

[17] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "Horus dataset," 2022, accessed: Jul 16, 2025. [Online]. Available: https://github.com/christoftorres/Elysium/tree/5ac506da715d2da9b422592f2c264d98201ef74c/evaluation/datasets/Horus

[18] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[19] M. Pacheco, G. Oliva, G. K. Rajbahadur, and A. Hassan, "Is my transaction done yet? an empirical study of transaction processing times in the ethereum blockchain platform," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–46, 2023.

[20] J. V. Behanan and Shashank, "OWASP Smart Contract Top 10," Web Page, OWASP (Open Web Application Security Project), 2025, accessed: 2025-10-20. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/

[21] S. Registry, "Reentrancy," 2020, accessed: Jul 16, 2025. [Online]. Available: https://swcregistry.io/docs/SWC-107/

[22] D. Siegel, "Understanding the dao attack," https://www.coindesk.com/learn/understanding-the-dao-attack, 2016.

[23] S. Registry, "Parity wallet hack #1," 2020, accessed: Jul 16, 2025. [Online]. Available: https://swcregistry.io/docs/SWC-100/

[24] ——, "Parity wallet hack #2," 2020, accessed: Jul 16, 2025. [Online]. Available: https://swcregistry.io/docs/SWC-106/

[25] ——, "Integer overflow and underflow," 2020, accessed: Jul 16, 2025. [Online]. Available: https://swcregistry.io/docs/SWC-101/

[26] ——, "Unhandled exception," 2020, accessed: Jul 16, 2025. [Online]. Available: https://swcregistry.io/docs/SWC-104/

[27] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and defi security tools: Do they meet the needs of practitioners?" in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[28] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of horus: Spotting and analyzing attacks on ethereum smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 33–52.

[29] "Moralis web3 documentation," https://docs.moralis.com/, 2025.

[30] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "A novel smart contract vulnerability detection method based on information graph and ensemble learning," *Sensors*, vol. 22, no. 9, p. 3581, 2022.

[31] T. Abdelaziz and A. Hobor, "Schooling to exploit foolish contracts," in *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, 2023, pp. 388–395. [Online]. Available: https://ieeexplore.ieee.org/document/10338924

[32] S. Wei, "Research on smart contract vulnerability detection technology based on vcs and ensemble learning," in *Proceedings of the 3rd International Conference on Bigdata Blockchain and Economy Management, ICBBEM 2024, March 29–31, 2024, Wuhan, China*, 2024.

[33] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[34] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[35] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[36] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[37] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," in *Advances in neural information processing systems*, 2018, pp. 6639–6649.

[38] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in neural information processing systems*, 2017, pp. 3146–3154.

[39] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[40] J. Howard and S. Gugger, "fastai: A layered api for deep learning," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3712236

[41] R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes, "Ensemble selection from libraries of models," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 18.

[42] M. Di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 2102–2105.

[43] GitHub, "Historical reentrancy," 2022, accessed: Jul 16, 2025. [Online]. Available: https://github.com/pcaversaccio/reentrancy-attacks

[44] S. Zhou, M. Möser, Z. Yang, B. Adida, T. Holz, J. Xiang, S. Goldfeder, Y. Cao, M. Plattner, X. Qin *et al.*, "An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2793–2810.

[45] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–37, 2021.

[46] GitHub, "Sb curated," 2022, accessed: Jul 16, 2025. [Online]. Available: https://github.com/smartbugs/smartbugs-curated

[47] Etherscan, "Safe contracts," 2025, accessed: Jul 16, 2025. [Online]. Available: https://etherscan.io/contractsVerified?filter=audit

[48] Google, "BigQuery bigquery-public-data.crypto_ethereum.contracts," Retrieved on 20 December 2024, https://console.cloud.google.com/bigquery.

[49] Tamer Abdelaziz and Karim Ali, "SCBENCH: Smart contract benchmark," https://bit.ly/SCBENCH, 2025.

[50] "Hugging face spaces," https://huggingface.co/spaces, accessed: 2025-10-11.
[51] N. Veloso, "Conkas: A modular and static analysis tool for ethereum bytecode," 2023.
[52] T. Abdelaziz and A. Hobor, "Smart learning to find dumb contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1775–1792.
[53] ——, "Usenix'23 artifact appendix: Smart learning to find dumb contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*.
[54] 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.
[55] 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.
[56] 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.
[57] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *Communications of the ACM*, vol. 63, no. 10, pp. 87–95, 2020.
[58] 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.
[59] C. F. Torres, J. Sch¨utte, 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.
[60] Pakala, "Pakala," Aug 2021, [Online]. Available: https://github.com/palkeo/pakala.
[61] J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.
[62] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.
[63] Etherscan, "Ethereum Transaction Details," https://etherscan.io/tx/0xd97d2aa04e7a92bfadf90a2a5f1a59c426640cfaa6023c06ca79166a83ca5199, 2025.
[64] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working conference on verified software: theories, tools, and experiments*. Springer, 2019, pp. 161–179.
[65] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security threat mitigation for smart contracts: A comprehensive survey," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–37, 2023.
[66] T. A. A. Mohamed, "Towards secure smart contracts: A deep learning approach for detecting security threats," Ph.D. dissertation, National University of Singapore (Singapore), 2023. [Online]. Available: https://scholarbank.nus.edu.sg/handle/10635/247301
[67] H. Wen, J. Fang, J. Wu, and Z. Zheng, "Transaction-based hidden strategies against general phishing detection framework on ethereum," in *2021 IEEE international symposium on circuits and systems (ISCAS)*. IEEE, 2021, pp. 1–5.
[68] S. Hu, Z. Zhang, B. Luo, S. Lu, B. He, and L. Liu, "Bert4eth: A pre-trained transformer for ethereum fraud detection," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2189–2197.
[69] J. Sun, Y. Jia, Y. Wang, Y. Tian, and S. Zhang, "Ethereum fraud detection via joint transaction language model and graph representation learning," *Information Fusion*, vol. 120, p. 103074, 2025.
[70] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "{TXSPECTOR}: Uncovering attacks in ethereum from transactions," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2775–2792.
[71] 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.
[72] R. Böhme, L. Eckey, T. Moore, N. Narula, T. Ruffing, and A. Zohar, "Responsible vulnerability disclosure in cryptocurrencies," *Communications of the ACM*, vol. 63, no. 10, pp. 62–71, 2020.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback and the SANAD Lab members for their insightful discussions. This work has been supported by NYU Abu Dhabi.