# PonziGuard: Detecting Ponzi Schemes on Ethereum with Contract Runtime Behavior Graph (CRBG)


Ruichao Liang
Wuhan University, China
liangruichao@whu.edu.cn

Jing Chen*
Wuhan University, China
chenjing@whu.edu.cn

Kun He
Wuhan University, China
hekun@whu.edu.cn

Yueming Wu
Nanyang Technological University, Singapore
wuyueming21@gmail.com

Gelei Deng
Nanyang Technological University, Singapore
gdeng003@e.ntu.edu.sg

Ruiying Du
Wuhan University, China
duraying@whu.edu.cn

Cong Wu
Nanyang Technological University, Singapore
cong.wu@ntu.edu.sg



## ABSTRACT

Ponzi schemes, a form of scam, have been discovered in Ethereum smart contracts in recent years, causing massive financial losses. Rule-based detection approaches rely on pre-defined rules with limited capabilities and domain knowledge dependency. Additionally, using static information like opcodes and transactions for machine learning models fails to effectively characterize the Ponzi contracts, resulting in poor reliability and interpretability.

In this paper, we propose PonziGuard, an efficient Ponzi scheme detection approach based on contract runtime behavior. Inspired by the observation that a contract's runtime behavior is more effective in disguising Ponzi contracts from the innocent contracts, PonziGuard establishes a comprehensive graph representation called *contract runtime behavior graph* (**CRBG**), to accurately depict the behavior of Ponzi contracts. Furthermore, it formulates the detection process as a graph classification task, enhancing its overall effectiveness. We conducted comparative experiments on a ground-truth dataset and applied PonziGuard to Ethereum Mainnet. The results show that PonziGuard outperforms the current state-of-the-art approaches and is also effective in open environments. Using PonziGuard, we have identified 805 Ponzi contracts on Ethereum Mainnet, which have resulted in an estimated economic loss of 281,700 Ether or approximately $500 million USD.



## CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; **Malware and its mitigation**.

## KEYWORDS

Smart Contract, Ponzi Scheme, Flow Analysis, Graph Neural Networks



*Jing Chen is the corresponding author.





*ICSE 2024, April 2024, Lisbon, Portugal*





ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623318

**ACM Reference Format:**
Ruichao Liang, Jing Chen, Kun He, Yueming Wu, Gelei Deng, Ruiying Du, and Cong Wu. 2024. PonziGuard: Detecting Ponzi Schemes on Ethereum with Contract Runtime Behavior Graph (CRBG). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623318


## 1 INTRODUCTION

With the popularity of Ethereum and the anonymity it provides, various scams have been discovered to implement themselves through smart contracts [36]. Ponzi schemes are one of the typical scams found in Ethereum smart contracts [3, 5], namely Ponzi contracts, disguising as investment programs to lure users under the promise of high profits while users gain profits only if the investments made by subsequent users join the Ponzi schemes. Ponzi schemes have been one of the biggest consumers of gas on Ethereum, heightening already bad congestion and jacking up transaction fees [31].

Several approaches have been proposed [3, 6, 7, 13–15, 21, 25, 35, 45] to detect Ponzi contracts on Ethereum. Rule-based approaches [3, 6, 35] require domain knowledge on Ponzi schemes and can hardly cover all possible scenarios based on the existing known Ponzi contracts, which limits their capability to detect Ponzi contracts that fall outside the scope of the rules. Other detection approaches use static information such as opcode frequency and transactions for machine learning models to improve detection capabilities[7, 13–15, 21, 25, 45]. However, this static information has a low correlation with Ponzi schemes themselves, and these approaches fail to effectively characterize the Ponzi contracts, resulting in poor reliability and interpretability. For instance, Figure 1 shows the frequency distributions of some most frequently used operations in some Ponzi contracts and non-Ponzi contracts. These operations are predominantly stack operations and do not capture the characteristics of Ponzi contracts. The *Kullback-Leibler Divergence* (KL divergence) calculated from Figure 1 measures the difference between two frequency distributions. It can be concluded from the KL divergence that the distributions of opcode frequency exhibit low differences between Ponzi and non-Ponzi contracts, and no substantial similarities between different Ponzi contracts by comparison. Moreover, those approaches utilizing Ethereum transactions cannot detect *0-day* Ponzi contracts, i.e., having none real transactions.

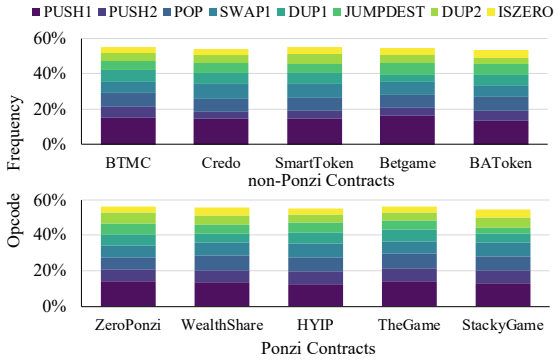


**Figure 1: Opcode Frequency Distributions. The KL divergence between Ponzi and non-Ponzi contracts ranges from 0.011 to 0.018, while the KL divergence between different Ponzi contracts ranges from 0.012 to 0.016.**

To address this gap, we delved deeper into the behaviors of Ponzi contracts at runtime and found that the contract runtime information provides more valuable insights into the unique characteristics of Ponzi contracts. We will discuss this insight in more detail in Section 2. Motivated by this observation, we propose a comprehensive graph representation called *contract runtime behavior graph* (**CRBG**) to characterize the runtime behaviors of Ponzi contracts.

In this paper, we propose PonziGuard, an effective Ponzi scheme detection approach based on CRBG. Specifically, we construct CRBG based on the runtime information of smart contratcs and empower Graph Neural Networks (GNNs) for CRBG analysis. We formulate the detection of Ponzi contracts as a graph classification task. We have experimentally validated the effectiveness of CRBG and conducted comparative experiments on a ground-truth dataset to evaluate the performance of PonziGuard. We further applied PonziGuard to Ethereum Mainnet to evaluate the effectiveness of our approach in open environments. The dataset and experimental results are publicly available online[1]. In summary, this paper makes the following contributions.

- We propose PonziGuard, an efficient approach for detecting Ponzi schemes on Ethereum. It does not require any domain knowledge on Ponzi schemes, and on-chain transaction data. It can pre-identify Ponzi contracts before the transactions occur.
- We introduce CRBG, a comprehensive graph representation for effectively characterizing the behaviors of Ponzi contracts . We model the detection of Ponzi contracts as a graph classification task and prove that CRBG is effective in disguising the Ponzi contracts from the innocent contracts.
- Experimental results show that PonziGuard outperforms the current state-of-the-art approaches on the ground-truth dataset and is also effective in open environments. We have found 805 Ponzi contracts using PonziGuard out of 14,000,000 Ethereum Mainnet blocks which have resulted in an estimated economic loss of 281,700 Ether or approximately $500 million USD.

[1]https://github.com/PonziDetection/PonziGuard

## 2 BACKGROUND AND INSIGHT

In this section, we introduce Ethereum smart contracts, explore the typical behavioral characteristics of Ponzi schemes, and outline objective criteria for identifying them. Additionally, we discuss our insight into utilizing the contract runtime behavior graph (CRBG) to detect Ponzi schemes.

### 2.1 Ethereum Smart Contracts

Ethereum smart contracts are programs running on top of Ethereum. They can be written in several programming languages, including Solidity, Viper, and Serpent. To deploy smart contracts on the blockchain, they need to be compiled into bytecode and then submitted to the blockchain with transactions. Once deployed on-chain, the contracts become immutable and the implementation of their logic relies on message calls from transactions. When invoked by a transaction, contracts will be executed in Ethereum Virtual Machine (EVM), a stack-based architecture [41]. There are three areas to store data in EVM:

- **Stack**: The stack is an object for basic stack operations in EVM. Data is pushed or popped from the top of the stack through instructions.
- **Memory**: The memory is a simple word-addressed byte array. It is used for temporary data storage, transfer of arguments and return values, and code copying [41]. The data in the memory comes from the stack or the external environments.
- **Storage**: Unlike the memory and stack that are volatile, the storage is non-volatile and maintained as part of the smart contract state. Variables in the storage region are called state variables, and they are persistent variables stored in the form of key-value pairs. Transactions can update the state variables of smart contracts by invoking the execution of contracts in EVM.

### 2.2 Ponzi Schemes

A Ponzi scheme is an investment fraud that involves the payment of purported returns to existing investors from funds contributed by new investors [34]. It is a classic fraud that originated at least 150 years ago and now appears on blockchains [3]. Leveraging smart contracts, Ponzi schemes become more threatening and stealthy than ever and have grabbed a huge amount of profits on the blockchain [5].

**Code Example.** Listing 1 shows a code snippet of a typical Ponzi contract. The snippet comprises two functions, namely `enter()` and `pay()`, where `enter()` is responsible for receiving Ether from investors and `pay()` handles the redistribution of Ether. This contract promises investors very high return rates (Line 13) in exchange for their initial investment. The promised returns are paid out of new investments to attract additional investors until the scammers close up their scam and abscond with the illicit profits. Without legitimate earnings, a Ponzi scheme needs a steady stream of new investors to keep it running, otherwise, it will inevitably collapse and let the vast majority of participants bear the loss [2].

**Criteria.** Based on some previous studies [3, 6, 34] and our analysis of known Ponzi contracts, we have developed explicit criteria for objectively identifying Ponzi contracts in our study. Our proposed criteria include:

```
1   function enter(){
2     if(msg.value <1/100 ether){
3       msg.sender.send(msg.value);
4       return;}
5     uint amount = msg.value;
6     uint idx = persons.length;
7     persons.length += 1;
8     persons[idx].etherAddress = msg.sender;
9     persons[idx].amount = amount;}
10
11  function pay(){
12    while(this.balance > persons[payoutIdx].amount /
          100 * 500){
13      uint transactionAmount = persons[payoutIdx].
            amount / 100 * 500;
14      persons[payoutIdx].etherAddress.send(
            transactionAmount);
15      payoutIdx += 1;} }
```

**Listing 1: A code snippet of Ponzi contract.**

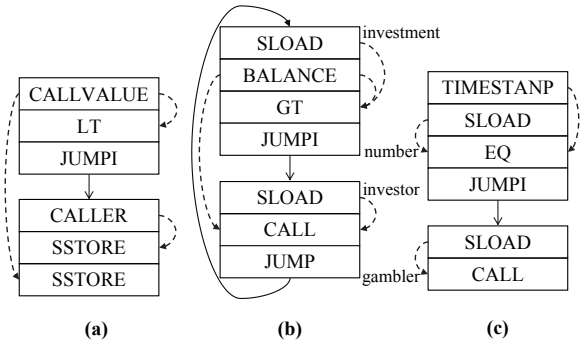


**Figure 2: Contract Runtime Behaviors. (a) and (b) are from the Ponzi contract in Listing 1, and (c) is from a gambling contract (non-Ponzi contract).**

- A Ponzi contract must incorporate at least two explicit behavioral logics: investment and reward. This criterion excludes contracts that receive cryptocurrency but provide users with assets through external markets such as real-world trades or auctions that utilize cryptocurrency for payments.
- The assets of a Ponzi contract must come from a multitude of investors rather than a specific source. This means that Ponzi contracts have no sources of income other than attracting investments. This criterion excludes contracts specifically designed to fulfill certain functions, such as enterprises distributing incentives to employees.
- In a Ponzi contract, all the investors are promised rewards that are typically expected to exceed their initial investment, although the implementation of these rewards is contingent upon attracting further investments. In other words, as long as there are constant new investments, everyone can theoretically reap the rewards. This criterion excludes the contracts that are likely to be mistaken for Ponzi contracts, such as gambling and puzzle contracts. In such contracts, not all users are promised rewards as they would be in a Ponzi contract. (There are always losers in gambling or puzzle games.)

## 2.3 Contract Behaviors and Our Insight

Through our proposed criteria, we can observe that the most crucial distinction between Ponzi contracts and benign contracts lies in their behavioral characteristics, such as the investment and reward logics and the flow of Ether, rather than specific transaction or instruction-level statistics. Therefore, in this section, we explore the contract runtime behaviors, trying to find an effective representation of these behaviors.

We invoke the smart contracts, gather their runtime information, and construct graphs based on this information, as depicted in Figure 2. It is important to note that the graphs in Figure 2 have been intentionally simplified to highlight the core logic of the contracts for the sake of clarity. Figure 2(a) depicts the investment behavior of the `enter()` function within the Ponzi contract shown in Listing 1. This contract first utilizes the `CALLVALUE` and `LT` operations to compare the Ether amount provided by investors (corresponding to Line 2 in Listing 1), and then utilizes `SSTORE` to store the investment amount and the address of the investors (Line 8 and Line 9). As it only relies on the comparison of the investment amount as the condition for receiving Ether, the source of Ether for the contract is not restricted to a specific address but encompasses all investors, which aligns with our second criterion for Ponzi contracts. Figure 2(b) depicts the reward behavior of the `pay()` function within the Ponzi contract shown in Listing 1. It first uses `SLOAD` to load the investment amount of the investor and calculates the promised reward (corresponding to Line 12 in Listing 1). If the contract balance is deemed sufficient to cover the reward, as determined through the comparison using `BALANCE` and `GT`, it proceeds to load the investor's address and completes the transfer (Line 14). Since this reward process iterates in a loop where the only condition for transferring Ether to investors is a sufficient contract balance, it can be inferred that every investor can potentially receive a reward as long as there is a continuous influx of investors, which aligns with our third criterion. The behaviors in Figure 2(a) and Figure 2(b) combined also satisfy our first criterion for Ponzi contracts. For comparison, consider Figure 2(c), which represents the reward behavior of a gambling contract[2]. In this case, the contract only rewards the gambler whose pre-selected number precisely matches the current timestamp. While this gambling contract fulfills the first and second criteria, it falls short of meeting our third criterion for Ponzi contracts.

In conclusion, these graphs depict the behaviors of the Ponzi contract as reflected in its source code and fulfill the criteria we have proposed, distinguishing it from benign contracts. This demonstrates that the graphs we constructed have the capability to effectively reveal the distinctive behavioral traits of Ponzi contracts. We refer to these graphs as contract runtime behavior graphs (CRBG). The illustrated graphs in Figure 2 serve as a preliminary illustration for clarity, while a more comprehensive description of CRBG can be found in Section 3.4.

# 3 PONZIGUARD

We first give an overview of PonziGuard. Then, we describe each step in detail.

[2] 0x4f9048d95616dbf7acc16fc4179f5ac6ee37bce6

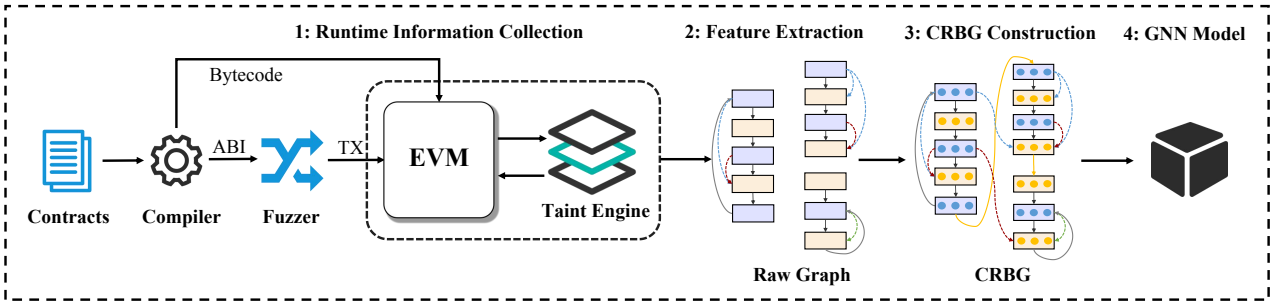


**Figure 3: Overview of PonziGuard.**

## 3.1 Overview

As shown in Figure 3, PonziGuard takes the smart contract as input and collects contract runtime information. Then, PonziGuard constructs a raw graph to depict the contract runtime behaviors. Next, PonziGuard preprocesses the raw graphs to construct the CRBG, which is better suited for model training due to improved node embeddings and more comprehensive behavioral information. Finally, PonziGuard trains a graph neural network using the CRBG, modeling the identification of Ponzi contracts as a graph classification task.

## 3.2 Runtime Information Collection

In order to obtain detailed insights into the contract's behavior, PonziGuard compiles the contract, generates transactions to invoke it, and then performs dynamic taint analysis to collect the contract runtime information.

**Compilation and Fuzzing.** We use *solc* [10] as the compiler to get the contract bytecode and ABI (Application Binary Interface). We select solc versions in descending order to find a compatible version. To invoke the contract, we employ the fuzzing [17] technique to generate transaction sequences for each contract. In order to trigger the behavior of the Ponzi contract with a higher probability, specific preferences are configured within the fuzzer. As is shown in Algorithm 1, given a contract $C$ as input, the fuzzer first gets all the function ABI from the compiler (Line 1) and then generates the transaction sequences in the loop (Lines 3-16). During the generation loop, the fuzzer applies priority rules to sort the ABI (Line 6). Payable functions (i.e., functions capable of receiving Ether) and functions with names containing strings such as "enter" or "deposit" are given higher priority. The selected ABI is analyzed to generate transaction elements, such as function arguments, Ether, and caller accounts (Line 7). To generate function arguments, the fuzzer randomly selects values within the valid input range for fixed data types, such as `uint256`, while for non-fixed data types like `string`, it first determines a positive number as the data length and then generates an input of that length. Regarding the Ether, we use a continuously increasing flow of Ether attached to transactions to facilitate the activation of specific behaviors of Ponzi contracts, such as investment and reward. Afterwards, the fuzzer integrates the transaction elements to generate a valid transaction, which is then added to the transaction sequence (Lines 8-9). If the transaction sequence successfully triggers the contract to receive and send

**Algorithm 1:** Transaction Sequences Generation.

```
   Input: Contract C
1: ABI_Pool ← GetABIFromCompiler (C)
2: g ← 0 #The number of generated transaction sequences
3: while g < MAX do
4:     Txs ← []
5:     while Txs.length < ABI_Pool.length do
6:         ABI ← SelectByPriority (ABI_Pool, Txs)
7:         elements ← Analyze (ABI, g)
8:         Tx ← Generate (ABI, elements)
9:         Txs ← Add (Tx, Txs)
10:    end
11:    res ← Run (C, Txs)
12:    if ReceiveEth (res) AND SendEth (res) then
13:        break
14:    end
15:    g ← g + 1
16: end
```

Ether outward (Line 12), or if the number of generated transaction sequences reaches the predefined limit (i.e., MAX) (Line 3), we stop invoking the contract.

It is worth noting that the fuzzer directly interacts with an independent instrumented EVM instead of a private Ethereum blockchain. The efficiency of contract execution on the independent EVM far surpasses that on the complete private chain due to the elimination of transaction packaging and block mining.

**Dynamic Taint Analysis.** Dynamic taint analysis is a popular technique to analyze the data flow in programming [33]. We have designed a taint engine for EVM to perform dynamic taint analysis and gather runtime details of smart contracts.

*Sources and Sinks:* As shown in Table 1, we have selected some operations as taint sources to introduce taint data. These operations will push some external data into the stack or memory, such as `CALLER`, `CALLVALUE`, `CALLDATALOAD`, `CALLDATACOPY`, which are related to the transaction sender and arguments, and `TIMESTAMP`, `BLOCKHASH`, which are related to the blockchain environment. In addition, we also consider some operations related to the contract itself, such as `BALANCE` and `ADDRESS`. Data derived from these sources

**Table 1: Taint Sources and Sinks.**

| Sources | Opcode Type |
|---|---|
| CALLVALUE/CALLDATASIZE/ CALLER/ORIGIN/ CALLDATALOAD/CALLDATACOPY | Transaction Related |
| TIMESTAMP/BLOCKHASH | Blockchain Environment |
| BALANCE/ADDRESS | Contract Related |
| **Sinks** | **Opcode Type** |
| EQ/LT/SLT/GT/SGT | Comparison |
| MSTORE/MSTORE8/MLOAD | Memory Related |
| SSTORE/SLOAD | Storage Related |
| CALL/CALLCODE/ DELEGATECALL/STATICCALL | Call |
| JUMPI | Jump |

is marked as tainted, while other data is marked as untainted. Regarding taint sinks, we select some meaningful operations as the location to check the flow of taint data. These operations either take taint data as their arguments (e.g., `GT`, `CALL` and `SSTORE`), or load taint data and push it into the stack (e.g., `MLOAD` and `SLOAD`).

*Taint Propagation:* To achieve the taint propagation, we implement the taint engine that encompasses the components such as a taint stack, a taint memory, and a taint storage. Each slot of the taint stack contains a taint that marks the corresponding slot in the EVM stack. Since the EVM memory is a byte array, each taint in the taint memory is responsible for a byte in the EVM memory. Both the taint stack and taint memory are volatile regions that are freed and allocated at the start of each new transaction. In contrast, the EVM storage is non-volatile and stores state variables in key-value pairs. In these key-value pairs, a 32-byte address calculated from the state variable is stored as the key, and the state variable is stored as the value. We maintain the taint storage in the same structure, with the address of the state variable as the key and the taint as the value. As storage is non-volatile, the taint storage is kept until all transactions are completed, as part of the Ethereum world state. In general, when one operand of an arithmetic operation is tainted, the result of the operation is also tainted regardless of the other operands. The implementation of the taint engine enables us to capture and trace the data flow throughout the contract execution.

## 3.3 Feature Extraction

We gather the information obtained in the contract execution and construct a raw graph that integrates the control flow and data flow of the contract runtime. The nodes of the graph are the operations executed during runtime, and we add control flow and data flow edges as the graph edges. The control flow edges are categorized into six types, with the most common type being the adjacent edge. This edge connects two operations whose program counters differ by only 1, indicating that they are executed in a successive manner.. The other types of control flow edges include the jump edge, which connects `JUMP(I)` and the operation executed after the jump, as well as the call, return, and creation edges that similarly connect the corresponding operation (e.g., `CALL`, `RETURN`, `CREATE`) and its successor. Regarding the data flow edges, we follow the principle of adding edges from taint sources to taint sinks, representing the

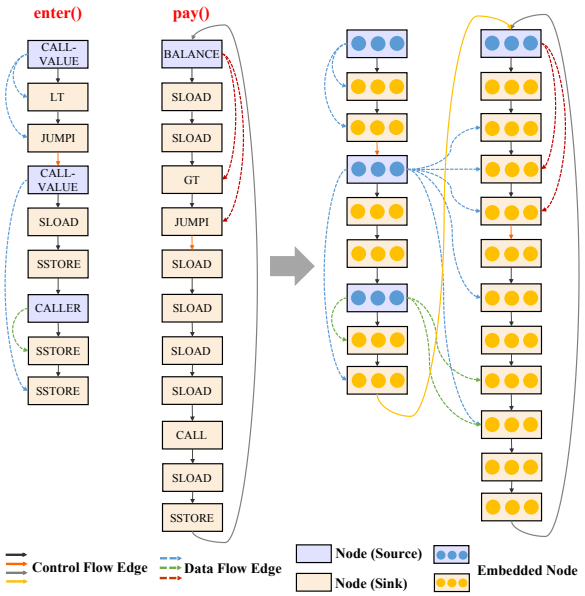


**Figure 4: CRBG Construction. (a) shows the raw graphs from feature extraction, each graph corresponds to a contract invoking. (b) is the CRBG with better node embeddings and more comprehensive runtime information. We simplify the graphs by keeping only the nodes representing the taint sources and sinks that capture the main logic of the functions for the convenience of display.**

propagation of taint data. There are eight kinds of data flow edges according to the taint sources in Table 1.

Figure 4(a) shows examples of the output graphs from our feature extraction stage. The two graphs presented in Figure 4(a) are the result of invoking `enter()` and `pay()` within the contract shown in Listing 1. For the sake of clarity, only the nodes representing the taint sources and sinks that capture the main logic of the functions are included in these simplified graphs shown in Figure 4(a).

## 3.4 CRBG Construction

The raw graph obtained from the feature extraction stage may not be suitable for training an effective model as described below.

**Independent Graphs and Incomplete Data Flow:** As each transaction can invoke the contract and generate a graph in our previous steps, a contract with multiple functions may correspond to multiple graphs as shown in Figure 4(a). However, an individual graph may not be sufficient to fully capture the behavior of the contract. For instance, in Figure 4(a), each graph only depicts a single stage of the contract (i.e., the investment stage for `enter()` and the reward stage for `pay()`), and neither of these graphs alone can conclusively determine that it is a Ponzi contract. Moreover, the data flow of the contract is isolated among graphs as depicted in Figure 4(a). Since smart contracts have persistent variables, there may also be data flow *across transactions*, which cannot be captured by individual graphs.

**Table 2: Introduction of JUMPI.**

| Value | Mnemonic | $\delta$ | $\alpha$ | Description |
|---|---|---|---|---|
| 0x57 | JUMPI | 2 | 0 | Conditionally alter the program counter. $J_{\text{JUMPI}}(\mu) \equiv \begin{cases} \mu_s[0] & if\ \mu_s[1] \neq 0 \\ \mu_{pc}+1 & otherwise \end{cases}$ |

To address these issues, we connect all graphs of the same contract sequentially using a new type of edge called connection edge. This creates a connected graph that can better represent the behavior of the contract across multiple transactions. Furthermore, we complete the *across-transaction* data flow among previously independent graphs using taint storage. The taint storage records the taint status of variables at the end of each transaction, and this information is used to propagate taints to subsequent transactions. With these enhancements, we are able to capture data flow that spans multiple transactions and more accurately analyze the behavior of smart contracts.

**Inadequate Node Feature:** In the raw graph, nodes are distinguished by the type of operation they include. This results in each node being represented by a 139-dimensional one-hot vector (corresponding to 139 unique operations), with a single non-zero entry corresponding to the type of operation. However, one-hot vectors do not capture any information about the relationships between nodes in the graph, which are crucial for understanding its structure and properties. To improve the classification accuracy, we need better node embeddings that can capture these relationships.

We noticed that there is an introduction for each operation in the Ethereum Yellow Paper [41] as exemplified in Table 2. In Table 2, $\alpha$ represents the additional items placed on the stack, while $\delta$ represents the items removed from the stack [41]. The description section explains how the operation works in text, and shows how it operates the data in the EVM in the formula. To embed the nodes, we first remove the formula in the description section and keep only the text explanation to preserve the functional information of the operation. For each node, we use Doc2Vec [23], a model for generating embeddings of variable-length pieces of text, to convert the text explanation into a 100-dimensional vector, which we stitch together with $\alpha$ and $\delta$ to form the node feature. After that, we have completed the construction of CRBG which will be labeled for model training later. Figure 4(b) shows the constructed CRBG after the raw graphs in Figure 4(a) were preprocessed. The CRBG in Figure 4(b) has better node embeddings, more comprehensive contract runtime information, and can better characterize the contracts.

## 3.5 GNN Model

Unlike traditional neural network models that primarily handle vector or matrix data, graph neural networks (GNNs) excel at modeling and processing graph-structured data [42]. In this section, we introduce our GNN solution to the Ponzi contract identification problem. As illustrated in Figure 5, our GNN model consists of three parts: graph input, graph embedding learning, and classification.

**Graph input.** We use CRBG ($\mathcal{G}$) as the input graph which contains nodes $\mathcal{V} = \{1, ..., n\}$ and edges $\mathcal{E}$. The node features matrix $\mathbf{X}$ has a dimension of $(|\mathcal{V}|, 102)$, where each node is represented by a

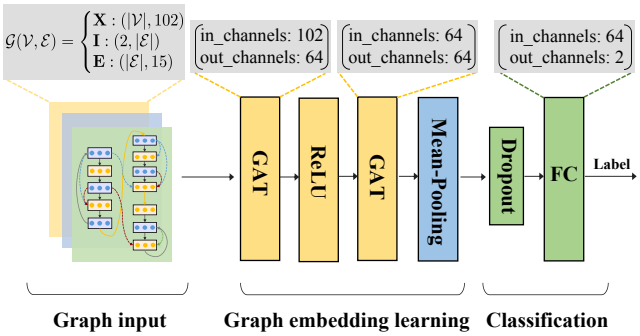


**Figure 5: GNN Model.**

102-dimensional feature vector. The edge index $\mathbf{I}$ has a dimension of $(2, |\mathcal{E}|)$, where each column corresponds to an edge and contains the indices of the nodes that the edge connects. The edge features matrix $\mathbf{E}$ has a dimension of $(|\mathcal{E}|, 15)$, where each edge is represented by a 15-dimensional feature vector. $|\mathcal{V}|$ and $|\mathcal{E}|$ represent the number of nodes and edges in $\mathcal{G}$.

**Graph embedding learning.** In graph embedding learning, we choose Graph Attention Networks (GAT) as the component of GNN convolutional layers. GAT performs the aggregation based on the self-attention mechanism, i.e., calculating the weights between nodes and edges through learnable weight matrices $\mathbf{W}$ and $\mathbf{W}_e$, so that each node can be weighted and aggregated according to the characteristics of its surrounding nodes. Since CRBG has multi-dimensional edge features, the attention coefficients $\alpha_{i,j}$ in the self-attention mechanism are computed as:

$$\alpha_{i,j} = \frac{\exp(\mathbf{e}_{i,j})}{\sum_{k \in \mathcal{N}_{i \cup i}} \exp(\mathbf{e}_{i,k})} \tag{1}$$

where $\mathbf{e}_{i,j}$ represents the attention score indicating the importance of node $j$'s features to node $i$, and $\mathcal{N}_{i \cup i}$ represents the set of adjacent nodes of node $i$. $\mathbf{e}_{i,j}$ is obtained by concatenating the feature vectors of node $i$ and node $j$ and performing linear transformation:

$$\mathbf{e}_{i,j} = \text{LeakyReLU}(\vec{\mathbf{a}}^T [\mathbf{W}\vec{\mathbf{h}}_i || \mathbf{W}\vec{\mathbf{h}}_j || \mathbf{W}_e \vec{\mathbf{m}}_{i,j}]) \tag{2}$$

where LeakyReLU represents the activation function, $\vec{\mathbf{a}}$ represents the weight vector, $||$ represents the concatenation operation, $\vec{\mathbf{h}}_i$ represents the feature vector of node $i$, and $\vec{\mathbf{m}}_{i,j}$ represents the multi-dimensional edge features between node $i$ and $j$.

By calculating the weight between nodes, the weighted sum of the adjacent nodes of node $i$ can be obtained:

$$\vec{\mathbf{h}}_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W}\vec{\mathbf{h}}_j\right) \tag{3}$$

where $\vec{\mathbf{h}}_i'$ represents the updated eigenvector of node $i$, $\sigma$ represents the activation function, $\mathcal{N}_i$ represents the adjacent node of node $i$.

We set two GAT layers and use ReLU in the middle for nonlinearly transforming the node features in order to better handle the nonlinear relationship of data and increase the expressiveness of the network. We utilize mean-pooling to aggregate the node features and obtain the global feature representation of the graph.

**Classification.** The classifier comprises a dropout and a fully connected layer (FC). The dropout randomly sets a fraction of the output of neurons to zero, which helps prevent overfitting and improves the model's generalization ability. The purpose of a fully connected layer is to learn non-linear combinations of the features in the input data, allowing the model to make more accurate predictions. We input the global feature representation into the classifier and obtain the predicted class label for the graph.

# 4 IMPLEMENTATION

We modified ContractFuzzer [20], a smart contract fuzzer, to integrate our transaction sequences generation algorithm. We instrumented the official Golang implementation of EVM (version 1.10.6) [11] to collect contract runtime information. We implemented our dynamic taint engine in Golang (version 1.16.6) to cooperate with the instrumented EVM and construct the CRBG. Our GNN model was implemented using Pytorch [29], and we employed Graph Attention Networks as the convolutional layers.

# 5 EXPERIMENTS

## 5.1 Research Questions

We conduct experiments to answer the following four questions.

- **RQ1:** How effective is PonziGuard in identifying Ponzi contracts compared to the existing tools?
- **RQ2:** How effective is the CRBG compared to the raw graph obtained directly from runtime?
- **RQ3:** How does PonziGuard perform in open environments?
- **RQ4:** What is the overhead of PonziGuard?

## 5.2 Experiment Setup

Our test environment is comprised of a server with a 16-core Intel(R)-Xeon(R)-Gold-5218 CPU @2.30 GHz, 340GB of RAM, and the Ubuntu 18.04 LTS operating system.

## 5.3 RQ1: Effectiveness of PonziGuard

*5.3.1 State-of-the-art.* We evaluated the effectiveness of PonziGuard and compared it with the studies of Chen et al. [7], Yu et al. [45] and Chen et al. [6]. Chen et al. [7] detects Ponzi contracts using XGBoost mainly based on the opcode frequency. Yu et al. [45] utilizes the transactions on Ethereum to identify Ponzi contracts. Chen et al. [6] detects Ponzi contracts based on symbolic execution. We used [7] as the baseline for opcode-based machine learning approaches, [45] as the baseline for transaction-based machine learning approaches, and attempted to replicate their models based on the descriptions in their papers and conducted experiments on our own dataset. We used [6] as the baseline for rule-based approaches which is open-sourced on github [22], and we applied it directly to our dataset.

*5.3.2 Dataset.* We obtained 184 Ponzi contracts from Bartoletti et al. [3]. After manually inspecting the contracts, we excluded 12 non-Ponzi contracts and 33 contracts without source code. This resulted in a set of 139 ground-truth Ponzi contracts. We obtained 1300 non-Ponzi contracts from an open-sourced smart contract dataset [27]. These contracts are real-world contracts running on Ethereum, which have already been labeled as non-Ponzi contracts and manually inspected by us. We added the 12 previously excluded non-Ponzi contracts to this dataset. As a result, we obtained a ground-truth dataset of 139 Ponzi contracts and 1312 non-Ponzi contracts. On average, the code length of this dataset is 407, with 1078 contracts having a length below 500 and 373 contracts having a length above 500.

**Table 3: Overall Evaluation Results. Values in parentheses represent the standard deviations across the K-fold.**

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| OpML[7] | 88.2% (0.03) | 75.0% (0.05) | 81.0% (0.04) |
| TxML[45] | 75.2% (0.06) | 57.9% (0.06) | 65.2% (0.05) |
| SADPonzi[6] | 91.3% | 71.9% | 80.4% |
| PonziGuard | **96.5%** (0.02) | **97.1%** (0.03) | **96.8%** (0.02) |

*5.3.3 Evaluation Metrics.* We use the following evaluation metrics to measure the effectiveness of our approach.

Precision measures the proportion of true positive predictions made by the approach out of all positive predictions:

$$\textbf{Precision} = \frac{TP}{TP + FP} \tag{4}$$

Recall measures the proportion of true positive predictions made by the approach out of all actual positive instances in the dataset:

$$\textbf{Recall} = \frac{TP}{TP + FN} \tag{5}$$

F1-score is the harmonic mean of Precision and Recall, providing a single measure of the approach's overall performance:

$$\textbf{F1-score} = 2 \times \frac{\textbf{Precision} \times \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}} \tag{6}$$

*5.3.4 Result and Analysis.* While we used the same dataset in the comparative experiment, different approaches processed the data differently. In our approach, we generated 1451 graphs from 1451 contracts in the dataset for model training and testing. In the approach of Chen et al. [7] (OpML), we compiled the 1451 contracts into bytecode and counted the opcode frequency as inputs for model training and testing. In the approach of Yu et al. [45] (TxML), we collected the transactions of these contracts on Ethereum and performed a random selection process to obtain a transaction network as input. The contract bytecode could be directly applied by the symbolic execution tool of Chen et al. [6] (SADPonzi). For the machine learning-based approaches, we randomly divided the dataset into 5 folds and performed K-fold cross-validation. The mean values of the evaluation metrics across the K models, as well as their corresponding standard deviations, were calculated to measure the average performances. As shown in Table 3, PonziGuard outperformed all the baselines on the test set, achieving 96.5% precision, 97.1% recall, and 96.8% F1-score. We believe that the poor performance of the state-of-the-art approaches can be attributed to the fact that static information cannot characterize the Ponzi contracts (OpML and TxML), and not all Ponzi contracts conform to the pre-defined behavior patterns (SADPonzi).

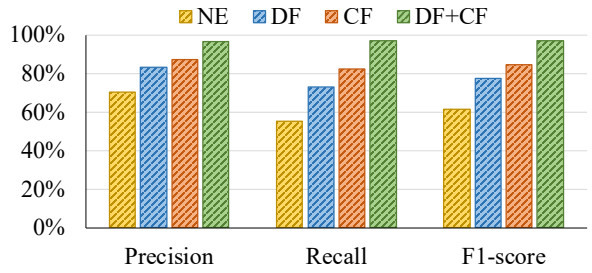


**Figure 6: Ablation study.**

In order to evaluate the effectiveness of our design choices for the fuzzing component, we replaced it with the native ContractFuzzer [20] while keeping the remaining setup unchanged and conducted the same evaluation on the dataset. It achieved a precision of 72.4%, recall of 75%, and F1-score of 73.9%, indicating that the fuzzing component, without our transaction sequences generation algorithm, performed poorly in triggering the specific behavior of Ponzi contracts.

**Answer to RQ1:** PonziGuard outperforms the state-of-the-art approaches in the comparative experiment, demonstrating the effectiveness of our approach in identifying Ponzi contracts.

## 5.4 RQ2: Effectiveness of CRBG

In Section 3.4, we constructed CRBG by incorporating both control flow and data flow into the raw graph and adopting better node embeddings. In this section, we demonstrated the effectiveness of CRBG compared to the raw graph by experimentally evaluating the contributions of control and data flow, as well as the node embeddings we have adopted.

*5.4.1 Effectiveness of runtime data flow and control flow in CRBG.* To gain a better understanding of the effectiveness of control and data flow in CRBG, we performed an ablation study by configuring PonziGuard in four distinct modes: data flow only (**DF**), control flow only (**CF**), both (**DF+CF**), and neither (**NE**). In **DF** mode, we removed control flow edges and only kept data flow edges in CRBG. On the contrary, in **CF** mode, we removed data flow edges and only kept control flow edges. In **NE** mode, we removed both data flow edges and control flow edges in CRBG. The mode **DF+CF** is the native PonziGuard which includes both control and data flow.

Figure 6 shows the performance of these four modes after 5-fold cross-validation on the same dataset. Compared to the native PonziGuard baseline, there were drops of 9.3%, 15%, and 12.2% in the evaluation metrics of the **CF** mode. In the **DF** mode, the evaluation metrics decreased to a greater extent compared to the native PonziGuard baseline, with a drop of 13.5%, 24.2%, and 19.3%, respectively. Undoubtedly, **NE** mode exhibited the worst performance, with a significant drop of 26.4%, 42.1%, and 35.3%, respectively. The main reason for the poor performance of the **CF** mode is that, with control flow only, PonziGuard cannot capture the flow of investors' investments in contracts. Therefore, some contracts with Ether redistribution logic may be misreported. On the other hand, the lack of control flow in the **DF** mode results in the loss of contract context information, such as the functions and order in which variables are used. This ablation study highlights that both control flow and data flow are crucial in capturing the behavioral patterns of Ponzi contracts, and the gathering of this runtime information significantly improves the performance of PonziGuard.

**Table 4: Comparing model performance between different node embedding settings.**

| Test | Node Embeddings adopted in Test Set | Model | Precision | Recall | F1-score |
|---|---|---|---|---|---|
| 1 | One-hot vectors | MOE | 88.5% | 82.1% | 85.2% |
| 2 | Enhanced | MEE | 96.4% | 96.4% | 96.4% |
| 3 | Variant 1 | MEE | 96.3% | 92.9% | 94.5% |
| 4 | Variant 2 | MEE | 96.2% | 89.3% | 92.6% |

*5.4.2 Effectiveness of node embeddings adopted in CRBG.* In Section 3.4, we utilized Doc2Vec to enhance node embeddings in CRBG based on the operation descriptions from the Ethereum Yellow Paper. We believe that the semantic information conveyed in these descriptions is representative and can capture the relationships between the nodes in CRBG. We conducted a comparative experiment to evaluate the efficacy of the node embeddings we enhanced. In this comparative experiment, one model was trained on the dataset described in Section 5.3.2 using our enhanced node embeddings (Model with enhanced node embeddings, abbreviated as **MEE**). In contrast, another model was trained on the same dataset, but replacing our node embeddings with one-hot vectors (Model with one-hot vectors as node embeddings, abbreviated as **MOE**). We used 80% of the dataset for training and 20% for testing. As shown in Table 4, compared with the one-hot vectors as node embeddings (Test 1), the node embeddings based on the operation description (Test 2) performed better in the evaluation metrics. It demonstrates that the node embeddings generated from operation descriptions capture the underlying semantics, leading to a better understanding of the graph's structure and properties, which accounts for this performance improvement.

To ascertain that the performance improvement is primarily attributed to the semantics themselves, rather than the way the semantics are described, we conducted additional analysis. We rewrote the operation descriptions in the Ethereum Yellow Paper with two principles while retaining the original semantics. The first principle is using synonyms substitution. By substituting words with their synonyms, we retained the original semantics while using a different expression. Another principle is changing sentence structure or grammar. This can be done by using different sentence patterns, altering the word order, or adjusting the placement of clauses. For instance, the description for `JUMPI` in Table 2 can be rewritten as "*Conditionally change the instruction pointer*" and "*Alter the program counter based on the condition*" according to these two principles. Based on these two alternative description rewriting principles, we re-extracted the node embeddings and created two variants of the test set (Variant 1 and Variant 2). Then, we evaluated our model (**MEE**, the model trained with enhanced node embeddings) on these two variant test sets. As shown in Table 4,

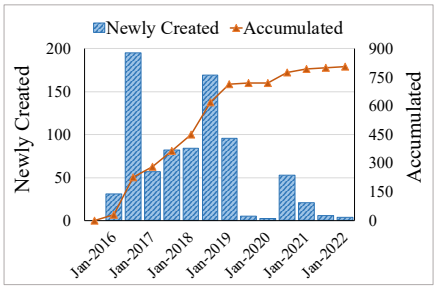


**Figure 7: Creation time of Ponzi contracts.**

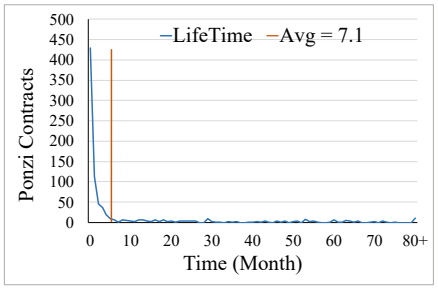


**Figure 8: Lifetime/Average Lifetime of Ponzi Contracts.**

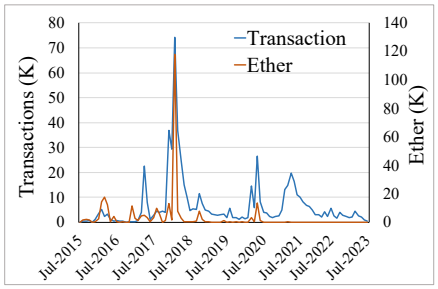


**Figure 9: Transaction and Ether Flow of Ponzi contracts.**

our model exhibited similar performance on the variant test sets (Test 3 and 4) compared to the native test set (Test 2), suggesting that the improvement in model performance is primarily attributed to the semantic information rather than the specific way in which the semantics are conveyed.

> **Answer to RQ2:** Through the evaluation of runtime control flow and data flow, as well as the enhanced node embeddings, the CRBG is proven to be effective and greatly improves the detection precision, recall, and F1-score.

## 5.5 RQ3: Performance in open environments

*5.5.1 Experiment Description.* In order to evaluate the effectiveness of PonziGuard in open environments and roughly estimate the number of Ponzi contracts on Ethereum, we conducted a preliminary experiment on the Ethereum Mainnet. Firstly, we ran the *Geth* client with the option: *sync-mode-full* to synchronize with the Ethereum Mainnet. The number of smart contracts has experienced explosive growth in recent years (about one million per quarter [1]), which is a significant amount for our approach based on runtime information. Therefore, we set the synchronization time until January 2022 (approximately 14,000,000 blocks), only as a preliminary experiment to verify the performance of PonziGuard in open environments. Then, we replaced the native EVM with our instrumented EVM and integrated the dynamic taint engine. We re-executed every transaction on the synchronized blockchain from the genesis block, which is a time-consuming process. Finally, we fed the generated graphs into our GNN model for prediction.

As a result, PonziGuard successfully identified 805 Ponzi contracts on Ethereum Mainnet, out of which 497 contracts have accessible source code on *Etherscan* [12]. We randomly selected 50 contracts[3] from these 497 contracts and conducted a manual examination through Remix [28], a solidity IDE, to ensure they meet our predefined criteria for Ponzi contracts, resulting in a 100% true positive rate. To gain deeper insights into these 805 Ponzi contracts, we conducted further analysis using the data collected from Etherscan in the remainder of this section.

*5.5.2 Creation Time of Ponzi Contracts.* Figure 7 shows the distribution of these 805 Ponzi contracts. Ponzi schemes started appearing on Ethereum as early as 2015. Subsequently, the rapid development of Ethereum led to a significant growth of Ponzi contracts during the years 2016-2019. Then, we witnessed a brief recession in Ponzi schemes possibly linked to the impact of the COVID-19 pandemic [24]. The global crypto mining boom in 2021 [9] resulted in another minor peak in Ponzi schemes. With the increasing popularity of various tokens on Ethereum, ERC-20 Tokens for instance, we anticipate another peak in Ponzi schemes on Ethereum in the near future.

*5.5.3 Lifetime of Ponzi Contracts.* We regard the time from the creation of a Ponzi contract to its last transaction as its lifetime. We investigated the lifetime of these 805 Ponzi contracts, as shown in Figure 8. While some of these contracts remain active in 2023[4], the majority of Ponzi contracts have a lifetime of less than three months, and their average lifetime is about seven months. These findings indicate that Ponzi contracts are likely to collapse within a short period of time, and most of their users will not be unable to reclaim their promised returns.

*5.5.4 Financial Impact.* We analyzed the financial impact of the 805 Ponzi contracts identified by PonziGuard on Ethereum Mainnet by aggregating their transactions and the inflow of Ether. Figure 9 shows their monthly distribution, revealing a positive correlation between the inflow of Ether and the number of transactions of the contracts. The peak was in February 2018, when a total of 117,953 Ether flowed into Ponzi contracts, equivalent to $108 million at the exchange rate of that time. From January 2015 to July 2023, 615,483 transactions, totaling 281,700 Ether, flowed into Ponzi contracts. At the current exchange rate, the value of these tokens can reach as high as $500 million. It is also evident that, in recent years, the involvement of Ether may not be substantial in a Ponzi scheme, as some of them began adopting ERC tokens for investments and rewards. However, it is important to note that such kinds of Ponzi contracts still meet the criteria outlined in Section 2.2, and our method remains effective in identifying them[5].

> **Answer to RQ3:** Our preliminary experiment on Ethereum Mainnet successfully identified 805 Ponzi contracts with a

[3]https://github.com/PonziDetection/PonziGuard/tree/main/dataset/Result/verified

[4]For instance: 0xa90be2201bfed97587a2a17949e8624eafe51d13 and 0xf8f04b23dace12841343ecf0e06124354515cc42

[5]Evidenced by the example of 0xb3836d31d43d315ba74c21aad3818f9378256152

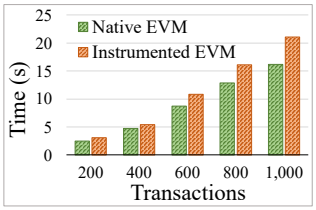


**Figure 10: Overhead on the Dataset.**

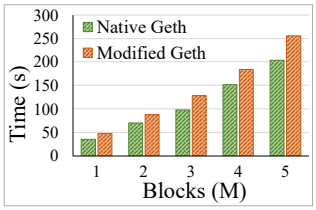


**Figure 11: Overhead in Open Environments.**

high accuracy rate, demonstrating the effectiveness of PonziGuard in open environments. These contracts have resulted in significant financial losses, amounting to millions of USD, which emphasizes the severity of Ponzi contracts on Ethereum and the urgency of identifying them effectively.

## 5.6 RQ4: Overhead of PonziGuard

In PonziGuard, we instrument the EVM and build a dynamic taint engine to obtain contract runtime information, which introduces a certain amount of time overhead compared to the native smart contract execution environment. We conducted experiments to evaluate this overhead.

*5.6.1 Ground-Truth Dataset.* For the experiment described in Section 5.3, the contracts were executed in an independent instrumented EVM with the taint engine. To evaluate the time overhead, we generated 1,000 transactions for a contract and sent them along with the contract to both the native EVM and the instrumented EVM separately. To accurately assess the time overhead, we repeated the process 10 times and recorded the average time it took to process these transactions. The results, shown in Figure 10, indicate that when the processing of 1,000 transactions was completed, the average overhead of the instrumented EVM reached a maximum of approximately 30.2%.

*5.6.2 Open Environments.* In the experiment described in Section 5.5, we conducted re-execution of historical transactions on the synchronized blockchain. To evaluate the time overhead, we re-executed the transactions of the first 500,000 blocks on the synchronized blockchain using both the native *Geth* and the *Geth* modified by PonziGuard separately. To accurately assess the time overhead, we repeated the process 10 times and recorded the average time it took to complete the re-execution. As shown in Figure 11, when it comes to 500,000 blocks, the time overhead amounts to 25.5%, which is smaller than the overhead on the ground-truth dataset. This can be attributed to the fact that re-execution involves additional reading and verification operations on the blockchain, in addition to the time consumed by contract execution.

**Answer to RQ4:** The time overhead introduced by our taint engine and the modification of EVM is an acceptable compromise to obtain contract runtime information.

# 6 RELATED WORK

In this section, we first describe the previous studies about Ponzi schemes on Ethereum. Then, we describe the studies related to the techniques we use.

## 6.1 Ponzi Scheme on Ethereum

Bartoletti et al. [3], the first to study Ponzi schemes on Ethereum, use the Normalized Levenshtein Distance (NLD) to measure the similarity of contract bytecode. Similarly, the rule-based approaches have been developed by Sun et al. [35] who leverage behavior forest similarity to detect Ponzi contracts, and Chen et al. [6] who use symbolic execution for detection. These approaches require a comprehensive summary of existing Ponzi schemes and expert experience. However, it is challenging to cover all possible scenarios based on the existing known Ponzi contracts, which limits their capability to detect Ponzi contracts that fall outside the scope of the summarized rules. Additionally, other approaches [7, 14, 21, 25, 45] use static information like opcode or transactions for machine learning models to improve detection capabilities. However, these approaches suffer from the limitation that static information cannot well distinguish Ponzi contracts from other contracts, and transaction-based machine learning approaches cannot detect *0-day* Ponzi schemes.

## 6.2 Smart Contract Fuzzing

Fuzzing has been proven to be effective to exploit vulnerabilities in smart contracts [18, 20, 26, 38, 43, 47]. ContractFuzzer [20] is a black-box fuzzer for Ethereum smart contracts to detect security bugs such as gasless send and timestamp dependency. Some grey-box fuzzers [18, 26, 38, 43, 47] have also been proposed for smart contracts. These methods are designed to exploit vulnerabilities in smart contracts, while PonziGuard uses fuzzing to invoke contracts and obtain their runtime information.

## 6.3 Taint Analysis

Taint analysis is an effective technique to analyze the data flow in programs. There have been studies that leverage taint analysis to help analyze smart contract such as Osiris [39], Sereum [30] and EthPloit [47]. Osiris [39] is an integer bug detection framework that combines taint analysis and symbolic execution. Sereum [30] leverages taint analysis to protect smart contracts with re-entrancy vulnerabilities from being exploited. EthPloit [47] adopts taint analysis to generate exploit-targeted transaction sequences, in order to make the contract fuzzing process more efficient. Those studies are orthogonal to this paper: they aim to uncover security vulnerabilities in smart contracts, while our tool is designed specifically for for identifying malicious contracts.

## 6.4 Graph Neural Network

Graph Neural Networks (GNNs), are a class of neural networks that are designed to process and learn from data that is structured in the form of graphs [42]. GNNs have been shown to be highly effective in a variety of tasks, such as node classification [19, 37], link prediction [44, 46], and graph classification [4, 40]. In this paper, we leverage GNNs for CRBG analysis and formulate the detection of Ponzi contracts as a graph classification task.

```
1  Origin:
2  uint index = Calculator.length + 1;
3  Calculator[index].ethereumAddress = msg.sender;
4  Modified:
5  uint index = Calculator.length;
6  Calculator.length += 1;
7  Calculator[index].ethereumAddress = msg.sender;
```

**Listing 2: Snippet of squareRootPonzi**

## 7 DISCUSSION

We note that some static analysis tools [8, 16, 32] can obtain the *Static* control and data flow with lower overhead, which also reflect the contract behavior to some extent. In this section, we provide the explanation for our decision to use *Runtime* information rather than *Static* information to construct our CRBG.

Firstly, static analysis is inherently imprecise following the principle of over-approximation. This conservative approach preserves all "could happen" or "could exist" cases, which is useful for capturing program errors and vulnerabilities but inappropriate for characterizing a program's behavior. For instance, squareRootPonzi[6] is a false positive case in the previous study [3], and its code snippet is shown in Listing 2. This contract appears to follow the logic of a typical Ponzi scheme, however, the incorrect assignment to the variable `index` will cause the typical *IndexError* during its runtime. Consequently, the contract will always exit with an error. The correct code is demonstrated in Line 5 and Line 6. However, static analysis tools cannot recognize this invalid execution path due to the lack of runtime information, and following the principle of over-approximation. If we utilize the static information to characterize the contract behaviors, it is likely to misreport it as a Ponzi scheme.

Secondly, the output of static analysis includes all possible execution paths and data flows of the contract, making it challenging to determine which information should be pruned. Constructing this information into a graph structure can result in a significant increase in size and contain redundant data, which is not efficient for model training.

## 8 CONCLUSION

In this paper, we propose PonziGuard, an approach for identifying Ponzi schemes on Ethereum based on the *contract runtime behavior graphs* (CRBG). The experimental results demonstrate that PonziGuard is effective on both the ground-truth dataset and open environments with acceptable overhead. Moreover, our preliminary experiment conducted on Ethereum Mainnet has identified 805 Ponzi contracts that have caused millions of USD in financial losses. This highlights the severity of Ponzi contracts on Ethereum and the pressing need to effectively identify them.

## ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers for their constructive comments. This research was supported in part by the National Key R&D Program of China under grant No. 2021YFB2700200, the Fundamental Research Funds for the Central Universities under grants No. 2042022kf1195, the National Natural Science Foundation of China under grants No. 62172303, 62076187, the Key R&D Program of Hubei Province under grant No. 2022BAA039, and Key R&D Program of Shandong Province under grant No. 2022CXPT055. This project is also supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

[6] 0x8ea6c8077d6316b46e449aec8fb60a606cf50eea

## REFERENCES

[1] Alchemy. 2022. ethereum-statistics. Retrieved November 17, 2022 from https://www.alchemy.com/overviews/ethereum-statistics
[2] Marc Artzrouni. 2009. The mathematics of Ponzi schemes. *Mathematical Social Sciences* 58 (2009).
[3] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2020. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. *Future Generation Computer Systems* 102 (2020).
[4] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. 2022. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45 (2022).
[5] Chainalysis. 2022. The Chainalysis 2022 Crypto Crime Report. Retrieved March 20, 2023 from https://go.chainalysis.com/2022-Crypto-Crime-Report.html
[6] Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, and Xiapu Luo. 2021. SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
[7] Weili Chen, Zibin Zheng, Jiahui Cui, Edith C. H. Ngai, Peilin Zheng, and Yuren Zhou. 2018. Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In *Proceedings of International World Wide Web Conferences (WWW)*.
[8] ConsenSys. 2023. Mythril. Retrieved April 22, 2023 from https://github.com/ConsenSys/mythril/
[9] Dimitris Drakopoulos. 2021. Crypto Boom Poses New Challenges to Financial Stability. Retrieved July 3, 2023 from https://www.imf.org/en/Blogs/Articles/2021/10/01/blog-gfsr-ch2-crypto-boom-poses-new-challenges-to-financial-stability
[10] ethereum. 2023. solc-bin. Retrieved September 19, 2022 from https://github.com/ethereum/solc-bin
[11] ethereum foundation. 2023. Official Go implementation of the Ethereum protocol. Retrieved October 20, 2022 from https://geth.ethereum.org/
[12] Etherscan. 2023. Etherscan.io. Retrieved March 20, 2023 from https://etherscan.io/
[13] Shuhui Fan, Shaojing Fu, Haoran Xu, and Xiaochun Cheng. 2021. Al-SPSD: Anti-leakage smart Ponzi schemes detection in blockchain. *Information Processing and Management* 58 (2021).
[14] Shuhui Fan, Shaojing Fu, Haoran Xu, and Chengzhang Zhu. 2020. Expose Your Mask: Smart Ponzi Schemes Detection on Blockchain. In *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN)*.
[15] Shuhui Fan, Haoran Xu, Shaojing Fu, and Ming Xu. 2020. Smart Ponzi Scheme Detection using Federated Learning. In *Proceedings of IEEE International Conference on High Performance Computing and Communications (HPCC)*.
[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*.
[17] Justin E. Forrester and Barton P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*.
[18] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of Conference on Computer and Communications Security (CCS)*.
[19] Sergei Ivanov and Liudmila Prokhorenkova. 2021. Boost then Convolve: Gradient Boosting Meets Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.
[20] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of International Conference on Automated Software Engineering (ASE)*.
[21] Eunjin Jung, Marion Le Tilly, Ashish Gehani, and Yunjie Ge. 2019. Data Mining-Based Ethereum Fraud Detection. In *Proceedings of IEEE International Conference on Blockchain (Blockchain)*.

[22] Kenun99. 2022. SadPonzi. Retrieved October 25, 2022 from https://github.com/Kenun99/SADPonzi

[23] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*.

[24] Richard Lehman. 2021. Ponzi schemes dropped in 2020. Retrieved July 3, 2023 from https://www.ponzitracker.com/home/ponzi-schemes-dropped-in-2020-but-this-may-not-be-a-silver-lining

[25] Yincheng Lou, Yanmei Zhang, and Shiping Chen. 2020. Ponzi Contracts Detection Based on Improved Convolutional Neural Network. In *Proceedings of International Conference on Services Computing (SCC)*.

[26] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of International Conference on Software Engineering (ICSE)*.

[27] polarwolf. 2021. Ponzi scheme contracts on Ethereum. Retrieved January 10, 2023 from https://www.kaggle.com/datasets/polarwolf/ponzi-scheme-contracts-on-ethereum?resource=download

[28] Remix Project. 2023. Remix. Retrieved June 12, 2023 from https://remix-project.org/

[29] pytorch. 2023. Pytorch. Retrieved June 12, 2023 from https://pytorch.org/

[30] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[31] Nivesh Rustgi. 2020. Ethereum's Top Gas Guzzlers are Ponzi Schemes. Retrieved March 26, 2023 from https://cryptobriefing.com/ethereums-top-gas-guzzlers-ponzi-schemes/

[32] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (SIGSAC)*.

[33] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*.

[34] U.S. SEC. 2019. U.S. Securities and Exchange Commission (SEC) Website. Retrieved February 7, 2023 from https://www.sec.gov/spotlight/enf-actions-ponzi.shtml

[35] Weisong Sun, Guangyao Xu, Zijiang Yang, and Zhenyu Chen. 2020. Early Detection of Smart Ponzi Scheme Contracts Based on Behavior Forest Similarity. In *Proceedings of International Conference on Software Quality, Reliability and Security (QRS)*.

[36] Nick Szabo. 1994. Smart Contracts: Building Blocks for Digital Markets.

[37] Shantanu Thakoor, Corentin Tallec, Mohammad Gheshlaghi Azar, Mehdi Azabou, Eva L. Dyer, Rémi Munos, Petar Velickovic, and Michal Valko. 2022. Large-Scale Representation Learning on Graphs via Bootstrapping. In *The Tenth International Conference on Learning Representations (ICLR)*.

[38] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *Proceedings of European Symposium on Security and Privacy (EuroS&P)*.

[39] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*.

[40] Lanning Wei, Huan Zhao, Zhiqiang He, and Quanming Yao. 2023. Neural Architecture Search for GNN-Based Graph Classification. *ACM Trans. Inf. Syst.* (2023).

[41] Gavin Wood. 2022. Ethereum: A secure decentralised generalised transaction ledger Berlin version. Retrieved January 26, 2023 from https://ethereum.github.io/yellowpaper/paper.pdf

[42] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32 (2020).

[43] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[44] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware graph neural networks. In *International conference on machine learning (ICML)*. 7134–7143.

[45] Shanqing Yu, Jie Jin, Yunyi Xie, Jie Shen, and Qi Xuan. 2021. Ponzi Scheme Detection in Ethereum Transaction Network. In *Blockchain and Trustworthy Systems (BlockSys)*. https://doi.org/10.1007/978-981-16-7993-3_14

[46] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. 2021. Labeling trick: A theory of using graph neural networks for multi-node representation learning. *Advances in Neural Information Processing Systems* 34 (2021).

[47] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.