Towards Effective Detection of Ponzi schemes on Ethereum with Contract Runtime Behavior Graph

RUICHAO LIANG, Wuhan University, China
JING CHEN*, Wuhan University, China
CONG WU, Nanyang Technological University, Singapore
KUN HE, Wuhan University, China
YUEMING WU, Nanyang Technological University, Singapore
WEISONG SUN, Nanyang Technological University, Singapore
RUIYING DU, Wuhan University, China
QINGCHUAN ZHAO, City University of Hong Kong, China
YANG LIU, Nanyang Technological University, Singapore

Ponzi schemes, a form of scam, have been discovered in Ethereum smart contracts in recent years, causing massive financial losses. Existing detection methods primarily focus on rule-based approaches and machine learning techniques that utilize static information as features. However, these methods have significant limitations. Rule-based approaches rely on pre-defined rules with limited capabilities and domain knowledge dependency. Using static information like opcodes for machine learning fails to effectively characterize Ponzi contracts, resulting in poor reliability and interpretability. Our research shows no significant difference between Ponzi and non-Ponzi contracts at the opcode level. Moreover, relying on static information like transactions for machine learning requires a certain number of transactions to achieve detection, which limits the scalability of detection and hinders the identification of *0-day* Ponzi schemes.

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 on CRBG, enhancing its overall effectiveness. The experiment results show that PonziGuard surpasses the current state-of-the-art approaches in the ground-truth dataset, achieving a precision of 96.9%, recall of 98.2%, and F1-score of 97.5%. It also exhibits the highest level of interpretability among the current tools. We applied PonziGuard to Ethereum Mainnet and demonstrated its effectiveness in real-world scenarios. Using PonziGuard, we identified 805 Ponzi contracts on Ethereum Mainnet, which have resulted in an estimated economic loss of 281,700 Ether or approximately $500 million USD. We also found *0-day* Ponzi schemes in the recently deployed 10,000 smart contracts.

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

*Jing Chen is the corresponding author.

Authors' addresses: Ruichao Liang, liangruichao@whu.edu.cn, Wuhan University, China; Jing Chen, chenjing@whu.edu.cn, Wuhan University, China; Cong Wu, cong.wu@ntu.edu.sg, Nanyang Technological University, Singapore; Kun He, hekun@whu.edu.cn, Wuhan University, China; Yueming Wu, wuyueming21@gmail.com, Nanyang Technological University, Singapore; Weisong Sun, weisong.sun@ntu.edu.sg, Nanyang Technological University, Singapore; Ruiying Du, duraying@whu.edu.cn, Wuhan University, China; Qingchuan Zhao, cs.qczhao@cityu.edu.hk, City University of Hong Kong, China; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore.

ACM 1557-7392/2024/12-ART
https://doi.org/10.1145/3707458

Additional Key Words and Phrases: Smart Contract, Ponzi Scheme, Flow Analysis, Graph Neural Networks

1 INTRODUCTION

With the popularity of Ethereum and the anonymity it provides, various scams have been discovered to implement themselves through smart contracts [43]. Ponzi schemes are typical scams found in Ethereum smart contracts [4, 6], 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 gas consumers on Ethereum, heightening already bad congestion and jacking up transaction fees [35].

Several approaches have been proposed [4, 7, 9, 10, 15–17, 19, 24, 28, 33, 42, 60] to detect Ponzi contracts on Ethereum. Rule-based approaches [4, 9, 42] 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 [10, 15–17, 24, 28, 29, 60, 63]. 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) provides a robust and theoretically grounded method for quantifying how one distribution diverges from another. We use KL divergence to measure the difference between two frequency distributions in Figure 1. It can be concluded 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 Ponzi contracts with none real transactions.

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, to effectively trigger the contract's runtime behavior pattern typical of a Ponzi scheme, we perform static analysis on the smart contract and use the obtained insights to generate transaction sequences that replicate typical Ponzi investment behaviors. We invoke the smart contract with these transaction sequences and conduct dynamic taint analysis during the contract's execution to gather runtime information. Then, we construct CRBG based on the contract runtime information 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 real-world scenarios. 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 and on-chain transaction. It can identify *0-day* Ponzi schemes before any economic losses occur. The dataset and experimental results are publicly available online[1].

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

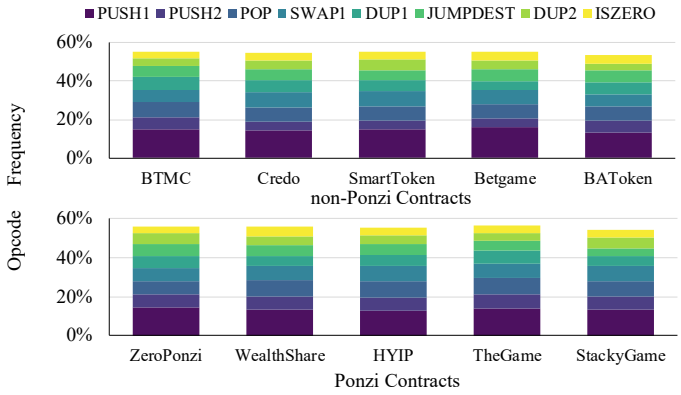

Fig. 1. Opcode Frequency Distributions. The KL divergence between Ponzi and non-Ponzi contracts ranges from 0.011 to 0.018, while the KL divergence between different Ponzi contracts ranges from 0.012 to 0.016.

- 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.
- We propose a strategy for generating contract invoke sequences based on function properties and data dependencies, enabling us to mimic typical investment behavior in a Ponzi scheme. We also design a dynamic taint engine to collect contract runtime behavior, which is essential for constructing CRBG.
- Experimental results show that PonziGuard outperforms the current state-of-the-art approaches on the ground-truth dataset and is also effective in real-world scenarios. We 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. We also found *0-day* Ponzi schemes in the recently deployed 10,000 smart contracts.

This paper is an extended version of our previous work [27] published in IEEE/ACM International Conference on Software Engineering (ICSE) 2024. We significantly enhanced the previous conference version in the following aspects: i) we added a static code analysis framework prior to generating transactions for collection of information such as data dependencies and function properties, to guide the generation of transaction sequences (§3.2); ii) we updated our transaction sequence generation algorithm, using information from static analysis and some heuristics to generate function invoke chains that mimic investment behavior of Ponzi schemes (§3.3); iii) we performed additional processing on CRBG including graph pruning and subgraph joining to achieve improved experimental results (§3.5); iv) we conducted the comparison experiment using a larger and more recent dataset, and added two SOTA works for comparison (§5.2); v) we conducted an interpretability experiment to visually explain how our CRBG works in the detection of Ponzi schemes (§5.3); vi) we conducted an ablation experiment to illustrate the impact of the graph pruning and joining processing that we integrated into CRBG (§5.4.2); vii) we explained the reasons behind the short lifetime of the 805 Ponzi contracts we identified, including how they were terminated (§5.5); viii) we conducted an experiment to detect newly deployed Ponzi schemes on Ethereum, showcasing our tool's ability to detect *0-day* Ponzi schemes (§5.5.2).

2 BACKGROUND AND INSIGHT

In this section, we introduce some necessary backgrounds and discuss our insight into utilizing the contract runtime behavior graph (CRBG) to detect Ponzi schemes.

```
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. Motivating Example

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, Vyper, 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 [49]. 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 [49]. 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 [38]. It is a classic fraud that originated at least 150 years ago and now appears on blockchains [4]. Leveraging smart contracts, Ponzi schemes become more threatening and stealthy than ever and have grabbed a huge amount of profits on the blockchain [6].

2.2.1 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,

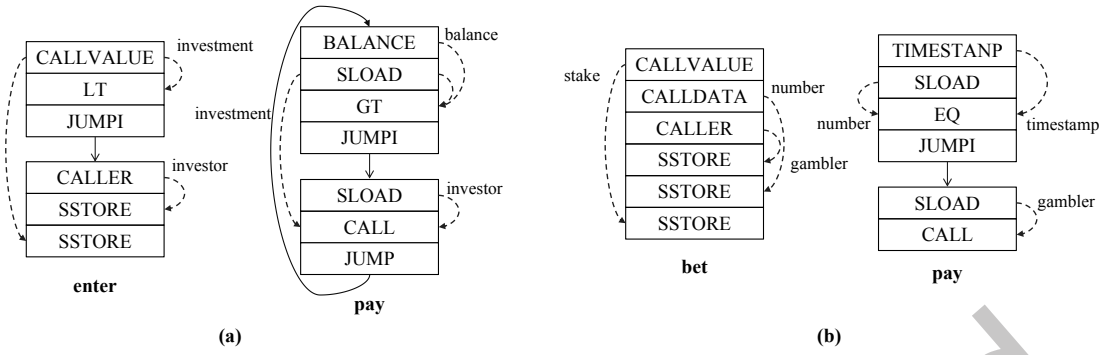

Fig. 2. Contract Runtime Behaviors. (a) is obtained from the Ponzi contract in Listing 1, and (b) is obtained from a gambling contract (non-Ponzi contract).

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].

2.2.2 Criteria. Based on some previous studies [4, 9, 38] and our analysis of known Ponzi contracts, we have developed explicit criteria for objectively identifying Ponzi contracts in our study. Our proposed criteria include:

- 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 non-ponzi 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. The left graph in 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

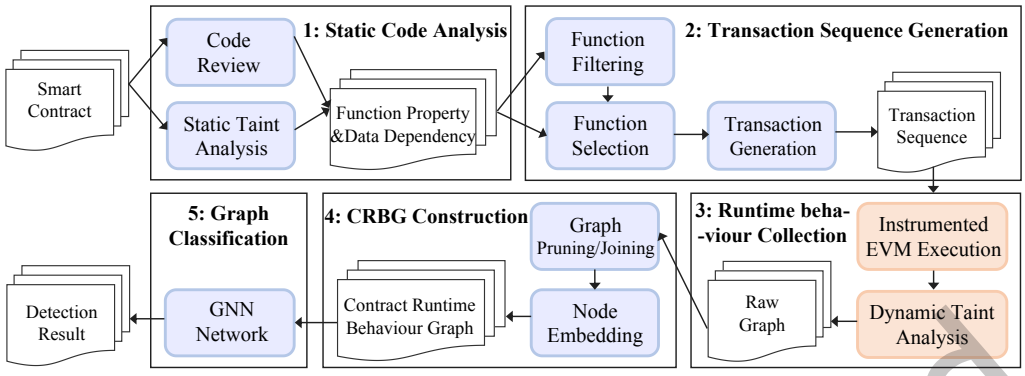

Fig. 3. Overview of PonziGuard.

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. The right graph in Figure 2(a) 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 presented by these two graphs of Figure 2(a) also satisfy our first criterion for Ponzi contracts. For comparison, consider Figure 2(b), which represents the bet and pay behaviors 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 non-ponzi 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 3.5.

3 PONZIGUARD

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

3.1 Overview

As shown in Figure 3, we perform code reviews to extract function properties and conduct static analysis to identify data dependencies across the smart contract. Contract source code and Application Binary Interface (ABI) are necessary for our analysis. Based on this gathered information, we conduct function filtering and selection to generate function invocation chains that mimic the investment behavior seen in a Ponzi scheme. Subsequently, we generate the corresponding transaction sequences to invoke the contract. Leveraging the dynamic taint engine, we collect runtime information during the contract execution within the instrumented EVM, creating

[2]0x4f9048d95616dbf7acc16fc4179f5ac6ee37bce6

raw graphs that encompass runtime control flow and data flow. After applying graph pruning and subgraph joining, we obtain the contract runtime behavior graph (CRBG). The CRBG serves as input for training a Graph Neural Network (GNN) designed to classify the graph. Consequently, we transform the Ponzi scheme detection into a graph classification task.

3.2 Static Code Analysis

To construct the CRBG that represents the behavior patterns of Ponzi schemes, the first step is to generate transactions that mimic the investment behavior typical of these schemes. However, randomly generating transactions, as is done in general fuzz testing [20], is inefficient for this purpose, as random tests are unlikely to precisely trigger the desired investment behavior. To address this, we perform static analysis on the smart contract before the detection. It is used to collect data dependencies, facilitating the generation of valuable transaction sequences that increase the likelihood of triggering Ponzi scheme behaviors. Specifically, in the code review procedure, we examine the contract's source code to extract essential function properties, including the function name, visibility, and whether they are declared payable (i.e., capable of receiving Ether). Then, we leverage SLITHER [18], a static analysis framework designed for smart contracts, to extract read and write operations of each function on state variables. It takes the Abstract Syntax Tree (AST) generated by the solidity compiler as input and converts it into Intermediate Representation (IR). Then it uses static single assessment (SSA) to facilitate the computation of various code analyses. Subsequently, we organize the data dependencies related to the functions within the contract. For instance, if function `A` writes to state variable α and function `B` reads from state variable β, where β is dependent on or exactly α, we include function `B` to the list of functions with a data dependency relationship on function `A`.

3.3 Transaction Sequence Generation

We utilize the extracted function properties and data dependencies to generate transactions, thereby simulating the investment behavior of the Ponzi contract and triggering its functionality. Due to the presence of state variables in smart contracts, prior transactions can influence the execution of subsequent ones. Therefore, achieving successful investment may necessitates multiple transactions. Accordingly, we generate transaction sequences to interact with the contract. The process of generating transaction sequences is outlined in Algorithm 1. Initially, we filter out functions that are irrelevant to the contract's (Ponzi) behavior, such as view functions and pure functions, and categorize specific functions into distinct groups For instance, in Algorithm 1, F_{kws} denotes functions whose names contain keywords (such as `invest, enter, init, deposit`); $F_{payable}$ encompasses payable functions, including the fallback function; F_w encompasses functions capable of altering state variables, while F_{all} encompasses all functions in the contract. We use heuristics to select the initial function to invoke (Lines 4-10). Specifically, we give priority to payable functions because the investment function is typically declared as payable. Additionally, we prioritize functions whose names contain specific keywords because they are more likely to encapsulate the logic of Ponzi contract investment. Once the first transaction is generated (Line 10), we proceed to complete the transaction sequences in a Read-After-Write order (Lines 11-22), which is a common practice to create meaningful function invocation chains. The function $randomChoose()$ selects a function randomly from its arguments (Line 5), while $generateTransaction()$ is responsible for creating a valid transaction based on the chosen function (Line 9). To achieve this, $generateTransaction()$ initially analyzes the contract's ABI (Application Binary Interface), then randomly selects values within the valid input range for fixed data types, such as uint256. For non-fixed data types like string, it determines a positive number as the data length and generates an input of that length. Additionally, for payable functions, $generateTransaction()$ employs a continuously increasing flow of Ether attached to transactions to facilitate the activation of specific behaviors

Algorithm 1: Transaction Sequences Generation.

```
   Input  : F_kws, F_payable, F_w, F_all, Max
   Output : TxSequences
1  g ← 0
2  TxSequences ← init()
3  while g <Max do
4      txs ← init()
5      func ← randomChoose(F_kws, F_payable, F_w)
6      if func is empty then
7          func ← randomChoose(F_all)
8      end
9      tx ← generateTransaction(func)
10     txs.add(tx)
11     while len(txs) <len(F_all) do
12         F_dep ← getDependency(txs)
13         if F_dep is not empty then
14             func ← randomChoose(F_dep, F_all)
15         end
16         else
17             func ← randomChoose(F_all)
18         end
19         tx ← generateTransaction(func)
20         txs.add(tx)
21     end
22     TxSequences.add(txs)
23     g ← g + 1
24 end
```


of Ponzi contracts, such as investment and reward. The function *getDependency*() retrieves functions that have data dependencies on the provided transactions (Line 12).

3.4 Runtime Behavior Collection

We use the generated transaction sequences to trigger contract execution, and collect the contract runtime behavior information. To achieve this, we design a taint engine for EVM to perform dynamic taint analysis and gather runtime details of smart contracts. Dynamic taint analysis, a widely-used program analysis technique [37], utilizes predefined taints to track program execution and observe runtime data flow and control flow. We also use taint analysis to compute data flow dependencies between different transactions.

3.4.1 Taint 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 is marked as tainted,

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

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`).

3.4.2 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.

3.4.3 Raw Graphs. 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 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 obtained from the dynamic taint analysis. 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).

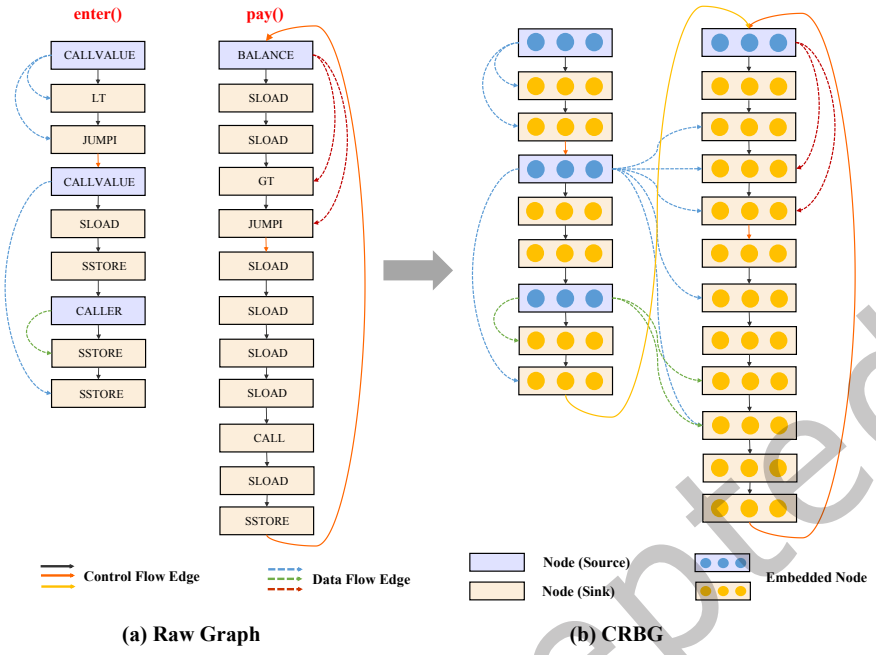

Fig. 4. CRBG Construction. (a) shows the raw graphs from dynamic taint analysis, each graph corresponds to a invocation. (b) is the CRBG with better node embeddings and more comprehensive runtime information. We simplify the graphs by keeping only the nodes representing the taint sources and sinks that capture the main logic of the functions for the convenience of display. The legend shown in this figure lists adjacent edge, jump edge, and connect edge in sequence, while data flow edges include callvalue edge, caller edge, and balance edge.

3.5 CRBG Construction

In this section, we illustrate why the raw graph obtained from the runtime behavior collection stage is not suitable for training an effective model and how we process these raw graphs to improve their representation of the behavioral characteristics of the Ponzi contract, making them suitable inputs for the graph neural network. The entire process is outlined in Figure 5.

3.5.1 Mitigating Graph Data Redundancy. As outlined in Section 3.4, we generate transactions to invoke the contract and execute its designated behavior patterns. Each transaction triggers the contract to execute once, generating a corresponding graph structure. To maximize the activation of Ponzi scheme behavior patterns, we generate multiple transaction sequences for each contract. Consequently, numerous redundant graphs occur, corresponding to repeated contract executions, failed executions due to contract assertions or invalid inputs, and executions unrelated to Ponzi scheme behavior patterns.

To mitigate graph data redundancy, we adopted two strategies to prune the raw graphs. The first strategy involves pruning based on contract behavior. This entails removing graphs from failed executions and executions that are not related to Ponzi scheme behavior patterns. Specifically, if an execution does not contain the opcode `CALLVALUE` and `CALLER`, it is unlikely to represent an investment operation of a Ponzi scheme. Similarly, if the execution lacks comparison (such as `LT`, `GT`, `EQ`) on the state variables, it is unlikely to be a reward operation of a Ponzi scheme. Therefore, we remove from the graphs any executions that are neither investment nor reward behaviors. Additionally, we remove graphs without the `SSTORE` opcode, as it is one of the most commonly used

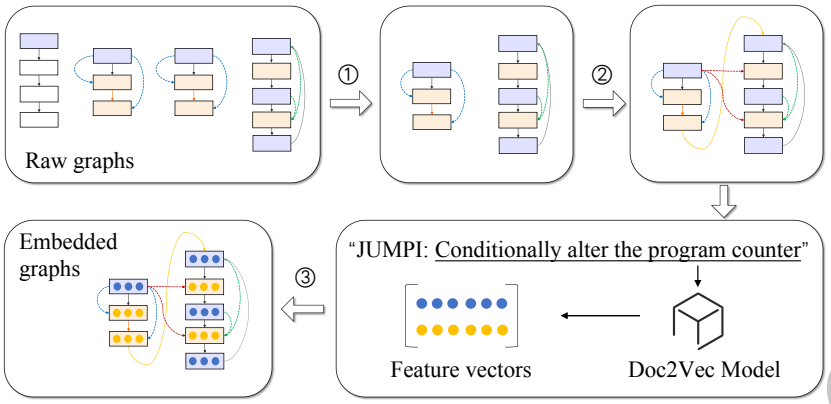

Fig. 5. Processing raw graphs. ① denotes the pruning of raw graphs. ② denotes the joining of independent subgraphs. ③ denotes the node feature embeddings.

opcodes to modify the contract state, and executions without SSTORE often indicate that they ended in failure. The second strategy involves pruning similar graphs. We compute the cosine similarity of node and edge features between each pair of graphs to assess their similarity. Graphs with a cosine similarity exceeding 0.8 are considered similar. This threshold is determined based on experimental observations, aiming to balance the trade-off between detection speed and accuracy. If the threshold is set too high, fewer graphs will be pruned, leading to more redundant data being retained. This not only slows down the detection process but also increases the risk of overfitting or noise in the model, which can degrade detection accuracy. If it is set too low, a larger number of graphs will be pruned. It could degrade detection accuracy by eliminating relevant information. Among similar graphs, we retain only one and discard the others.

3.5.2 Joining Independent Graphs and Enhancing Data Flow Integrity. After the invocation, a contract may correspond to multiple graphs, as each transaction can invoke the contract and generate a graph, as depicted in Figure 5. 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. Since smart contracts have persistent variables, there may also be data flow across transactions, which cannot be captured by individual graphs.

To address these issues, we connect all graphs of the same contract sequentially using a new type of edge called connection edge. These connection edges are established in the order of transaction execution to maintain the logical flow of the contract's operations. Specifically, each connection edge begins at the final node of the preceding graph, which is typically associated with the STOP instruction that marks the conclusion of a transaction. The connection then extends to the initial node of the subsequent graph, often represented by the PUSH instruction, which initiates the next transaction. This approach ensures a continuous and coherent representation of the contract's execution across multiple transactions, enabling a more accurate analysis of its behavior. 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.

Table 2. Introduction of JUMPI.

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

3.5.3 Enriching Node Features. 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 [49] as exemplified in Table 2. In Table 2, α represents the additional items placed on the stack, while δ represents the items removed from the stack [49]. 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. As shown in Figure 5, for each node, we use Doc2Vec [25], 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 α and δ 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.

3.6 Graph Classification

Deep learning excels at automatic feature extraction from raw data and achieves top performance in many fields [52, 54]. Unlike traditional deep learning models that primarily handle vector or matrix data [51], graph neural networks (GNNs) excel at modeling and processing graph-structured data [55]. In this section, we introduce our GNN solution to the Ponzi contract identification problem. As illustrated in Figure 6, our GNN model consists of three parts: graph input, graph embedding learning, and classification.

3.6.1 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 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 $\mathcal{G}$.

3.6.2 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.

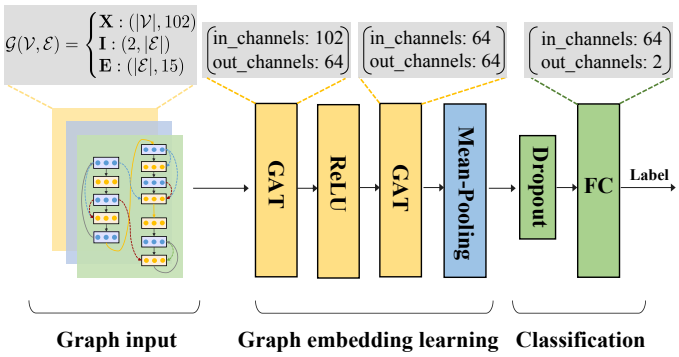

Fig. 6. GNN Model.

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, σ 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.

3.6.3 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.

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

Approach	Precision	Recall	F1-score
OpML[10]	89.0% (0.05)	77.8% (0.04)	83.0% (0.03)
TxML[60]	69.6% (0.06)	63.4% (0.02)	66.4% (0.07)
SADPonzi[9]	88.1%	64.5%	74.5%
MulCas[63]	95.1%	67.4%	78.9%
SourceP[29]	91.6% (0.04)	93.3% (0.03)	92.4% (0.02)
PonziGuard	**96.9%** (0.03)	**98.2%** (0.03)	**97.5%** (0.02)

4 IMPLEMENTATION

We leverage slither [18], a static analysis framework, to extract data dependencies of smart contracts. We instrumented the official Golang implementation of EVM (version 1.10.6) [13] 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 [32], and we employed Graph Attention Networks as the convolutional layers.

5 EXPERIMENTS

5.1 Research Questions

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. We conduct experiments to answer the following five questions.

- **RQ1:** How effective is PonziGuard in identifying Ponzi contracts compared to the existing tools?
- **RQ2:** How does CRBG work in detecting Ponzi contracts? Is the detection process interpretable?
- **RQ3:** How significantly do our CRBG designs contribute to PonziGuard's performance?
- **RQ4:** How does PonziGuard perform in real-world scenarios? Can it detect new Ponzi schemes?
- **RQ5:** What is the overhead of PonziGuard?

5.2 RQ1: Effectiveness of PonziGuard

5.2.1 Dataset. We utilize the dataset in XBlock [64] provided by Zheng et al. [63], obtained through crawling *Etherscan* [14] and manual cross-checking. This ground-truth dataset comprises 6,498 smart contracts, with 314 identified as smart Ponzi schemes. These smart contracts range from height 0 to height 7,500,000, and undergo a manual cross-check procedure to ensure the accuracy of their labels.

5.2.2 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: Precision = TP / (TP + FP). Recall measures the proportion of true positive predictions made by the approach out of all actual positive instances in the dataset: Recall = TP / (TP + FN). F1-score is the harmonic mean of Precision and Recall, providing a single measure of the approach's overall performance: F1-score = 2 × Precision × Recall / (Precision + Recall)

5.2.3 Tools Selection. We evaluated the effectiveness of PonziGuard and compared it with the studies of Chen et al. [10], Yu et al. [60], SADPonzi [9], MulCas [63] and SourceP [29]. Chen et al. [10] detect Ponzi contracts using XGBoost mainly based on the opcode frequency, and in this paper we refer to their work as OpML. Yu et

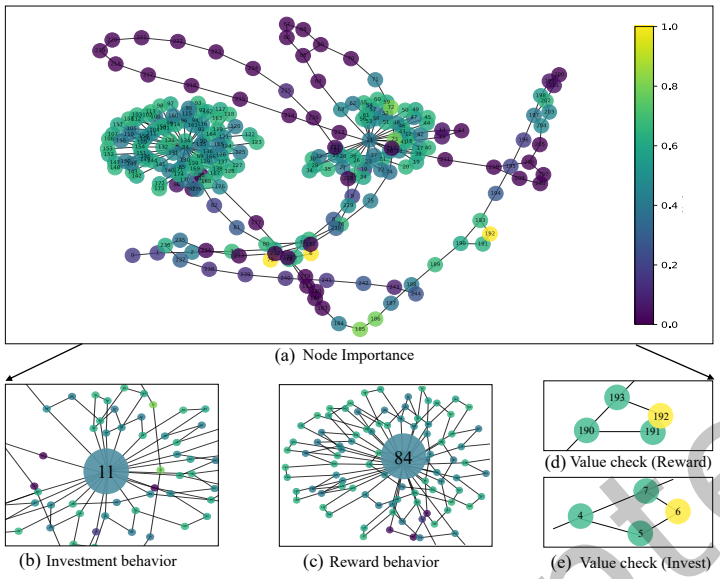

Fig. 7. Node Importance Heatmap.

al. [60] utilize the transactions on Ethereum to identify Ponzi contracts, and in this paper we refer to their work as TxML. SADPonzi detects Ponzi contracts based on symbolic execution. MulCas extracts contract features from multiple views and detects Ponzi schemes through multi-view training and ensemble. SourceP trains the classification model by converting contract source code into Abstract Syntax Trees (AST) to extract data flow information. [10, 29, 63] represent state-of-the-art (SOTA) operation/source code-based machine learning approaches. [60] represents SOTA transaction-based machine learning approaches, and [9] represents SOTA rule-based approaches.

5.2.4 Result and Analysis. In our approach, we generated 6,498 graphs from 6,498 contracts in the dataset for model training and testing. While we used the same dataset in the comparative experiment, different approaches processed the data differently. For example, in the approach of Chen et al. [10], we compiled the 6,498 contracts into bytecode and counted the opcode frequency as inputs for model training and testing. In the approach of Yu et al. [60], 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 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 96.9% precision, 98.2% recall, and 97.5% 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, TxML, MulCas and SourceP), and not all Ponzi contracts conform to the pre-defined behavior patterns (SADPonzi).

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.

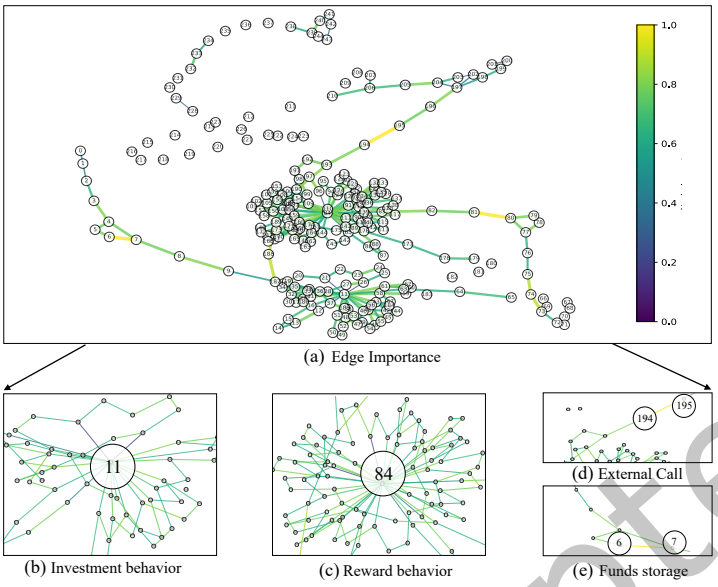

Fig. 8. Edge Importance Heatmap.

5.3 RQ2: Interpretability

We conduct an interpretability experiment to explain what role CRBG plays in detecting Ponzi schemes and to validate our insight into using CRBG as the key detection mechanism. Specifically, we selected a graph from the test set as input, and then calculated the gradient of the final classification decision for each node and edge in the input graph. Using these gradient values, we generated importance heatmaps to highlight the nodes and edges with great impact on the classification decisions, as shown in Figure 7 and Figure 8.

In Figure 7(a), some of the nodes with the greatest contribution form two clusters, and magnified details are shown in Figure 7(b) and Figure 7(c). In these two clusters, nodes 11 and 84 are the central nodes with the most connected nodes, and correspond to the opcode `CALLVALUE` and `BALANCE`, respectively. These two clusters represent the investment and reward behaviors of the Ponzi scheme. Specifically, the connections between node 11 and its surrounding nodes indicate that the contract continuously attracts new investors and records their investment amounts. The connections between nodes around node 84 represent the contract's cyclic operation of distributing rewards to investors. Several other important nodes such as nodes 6 and 192 shown in Figure 7(d) and Figure 7(e), correspond to comparison operations. They represent the value checks before both the investment and reward behaviors. Similarly, the most important edges in Figure 8 also form two clusters, same as those observed in Figure 7, delineating the control flow and data flow of the Ponzi scheme during the investment and reward processes. In particular, some of the most important edges such as (6, 7) and (194, 195) are presented in Figure 8(d) and Figure 8(e). As previously discussed, nodes 6 and 192, along with their adjacent nodes, represent the value check before the investment and reward. Subsequently, edges (6, 7) and (194, 195) represent the contract's behaviors subsequent to the completion of these checks. Specifically, edge (6, 7) represent the storage (`SSOTRE`) of investor's funds, while edge (194, 195) represents the external call (`CALL`) used for transferring rewards.

To compare the interpretability of PonziGuard with other approaches, we use the same contract as a case study and evaluate two key dimensions: i) the capture rate of essential Ponzi features and ii) the ability to explain behavioral patterns. We selected OpML [10] and SADPonzi [9] as representatives of static feature-based machine

Table 4. Comparison of interpretability.

Approach	Capture rate of key features	Behavioral pattern explanation capability
OpML[10]	3 out of 9 features are covered (33.3%)	○
SADPonzi[9]	4 out of 9 features are covered (44.4%)	◐
PonziGuard	9 out of 9 features are covered (100%)	●

●= Fully captures and explains Ponzi scheme behavioral patterns.
◐= Partially captures Ponzi scheme behavioral patterns.
○= Does not capture or explain Ponzi scheme behavioral patterns.

learning and rule-based methods, respectively, for comparison. The results are presented in Table 4. For the first dimension, we define the opcodes `CALLVALUE`, `BALANCE`, `CALL`, `SSTORE`, `SLOAD`, `LT`, `GT`, `EQ`, and `CALLER` as the core set of instructions associated with Ponzi schemes, based on the criteria outlined in Section 2. Specifically, the first criterion requires that a Ponzi contract must include both investment and dividend logic. In the investment logic, the contract should compare the investor's investment (`CALLVALUE`) using comparison operations (`LT`, `GT`, `EQ`) and store (`SSTORE`) the investor's address (`CALLER`). In the reward logic, the contract should retrieve (`SLOAD`) stored investors and the current balance (`BALANCE`), and then make an external call (`CALL`) to distribute the reward. Thus, coverage of these opcodes is a necessary but not sufficient condition, for identifying a Ponzi contract. Based on this consensus, if a detection tool fails to incorporate these operations when determining Ponzi contracts, its conclusions may lack interpretability and rigor. For PonziGuard, we assess the importance of these core opcodes through heatmaps. If these opcodes are assigned high importance, the model is considered to have effectively captured the key features. By analyzing nodes with an importance score exceeding 0.7 in the heatmap, PonziGuard achieves 100% coverage of the defined key features. For OpML, we identified the top 8 opcodes based on XGBoost model feature importance. However, only 3 of these opcodes (`SSTORE`, `SLOAD`, `CALLER`) match the defined Ponzi-related opcodes, resulting in a 33.3% coverage. For SADPonzi, we analyzed the bytecode segments that trigger its detection oracle (specifically, chain-scheme patterns in this case) and examined their coverage of Ponzi-related opcodes. This method captured 4 key features (`SSTORE`, `LT`, `CALL`, `SLOAD`), leading to a coverage of 44.4%. For the behavioral pattern explanation capability, we evaluate whether the detection method explicitly incorporates the behavioral patterns characteristic of Ponzi schemes. For PonziGuard, as discussed earlier, the model successfully captures all key behavioral patterns of Ponzi schemes, including both control flow and data flow dynamics. This allows the model to explain the behavior of Ponzi schemes at a structural level. In contrast, OpML bases its classification solely on the frequency of opcodes. It does not capture or explain specific behavioral patterns of Ponzi schemes, focusing instead on the statistical occurrence of certain opcodes. As a result, it lacks any insight into the structural or functional dynamics of Ponzi schemes. For SADPonzi, the chain-scheme pattern oracle offers a partial explanation of Ponzi scheme behaviors. It captures some control flow aspects, such as conditional logic and function calls, which are relevant to Ponzi operations. However, it does not explicitly model the complete data flow, such as the transfer and storage of funds.

Answer to RQ2: The experiment showcases that CRBG effectively reflects the characteristic behaviors of Ponzi schemes, and these characteristics greatly contribute to the classification decisions. This underscores CRBG's effectiveness in Ponzi scheme detection. Additionally, the experiment highlights that CRBG can provide valuable interpretability for our tool.

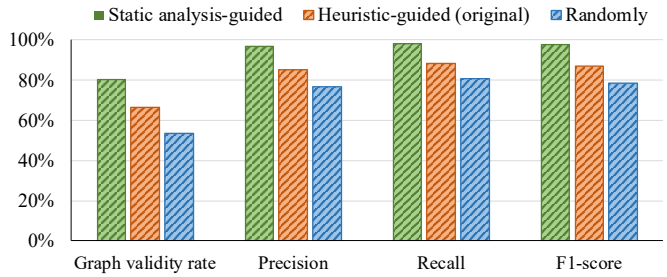

Fig. 9. Performance of PonziGuard without Static Analysis and Transaction Sequence Generation.

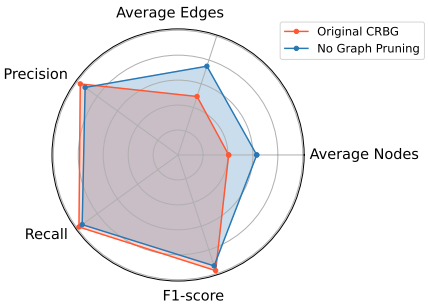

Fig. 10. Comparing CRBG with ablation 1.

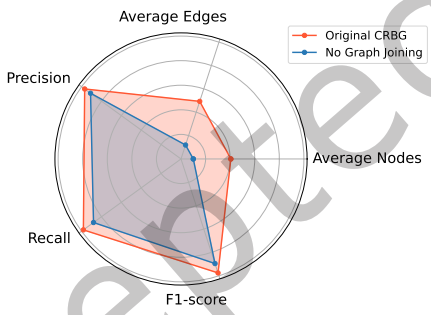

Fig. 11. Comparing CRBG with ablation 2.

5.4 RQ3: Ablation Study

In this section, we evaluate the effects of various processes applied to CRBG through a series of ablation studies, in order to determine how effective is CRBG compared to the raw graph obtained directly from runtime.

5.4.1 Effectiveness of static analysis framework-based transaction sequence generation. To evaluate the effectiveness of our static analysis-based transaction sequence generation for mimicking Ponzi scheme investment behavior, we conduct an ablation experiment. In this experiment, we remove the static analysis framework and transaction sequence generation algorithm, replacing them with: i) the original heuristic-based transaction generation algorithm, and ii) the unguided black-box fuzzer, ContractFuzzer [23], to randomly generate transactions and trigger contract execution. We then re-run the experiment on the RQ1 dataset using the ablation settings, and the results are presented in Figure 9. The graph validity rate measures the proportion of graphs retained after the pruning process described in Section 3.5.1. A higher rate indicates better quality in the generated transactions. PonziGuard with the static analysis framework achieves a validity rate of 80.1%. In contrast, the ablation settings where the static analysis framework and transaction generation algorithm are replaced by either a heuristic-based algorithm or random fuzzing, result in a decrease to 66.5% and 53.7%, respectively. Specifically, for the heuristic-based ablation, precision, recall, and F1-score are reduced by 11.8%, 9.7%, and 10.73%, respectively. For the random fuzzing ablation, precision, recall, and F1-score drop by 20%, 17.7%, and 18.8%, respectively.

5.4.2 Effectiveness of graph pruning and joining. To assess the impact of graph pruning and graph joining on CRBG performance, we design two ablation studies. In Ablation 1, we omit the graph pruning step, and in Ablation 2, we omit the graph joining step. We then test these ablations using the same dataset from RQ1, recording the average number of edges and nodes in the generated CRBG, along with the final experimental results. The outcomes are compared to the original PonziGuard in Figures 10 and 11.

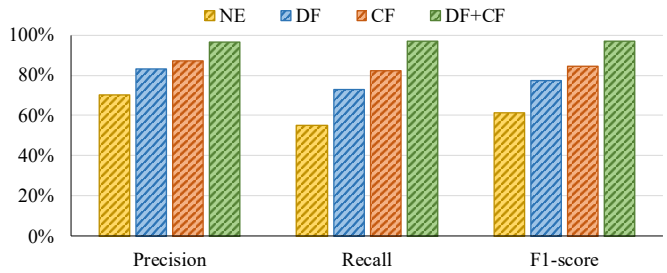

Fig. 12. Performance of different ablation configurations.

For the CRBG without the graph pruning operation, as shown in Figure 10, the average number of nodes and edges per graph increased by nearly 30%. This added significant redundant information, leading to lower precision, recall, and F1-score compared to the original CRBG. Additionally, the larger graph sizes increased the model's training time. For the CRBG without the subgraph joining operation, as shown in Figure 11, the average number of nodes and edges per graph was significantly reduced to nearly one-fifth of the original CRBG. However, as discussed in Section 3.5, these smaller subgraphs failed to fully capture the contract's behavior, resulting in a decrease in performance metrics: precision, recall, and F1-score dropped by 5.9%, 10.2% and 8.0%, respectively.

5.4.3 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 all data flow and control flow edges from the graph, leaving only the edges that connect the bytecode instructions in their sequential order. As a result, the **NE** graph only reflects the linear sequential order of the bytecode, ignoring any data dependencies and control flow logic like jumps or conditional branches. For example, consider instructions A(0x10): `JUMPI`, B(0x11): `PUSH`, and C(0x20): `JUMPDEST`. In the original control flow graph, since A is a conditional jump, it might have two outgoing edges: one leading to B and another to C. However, in **NE** mode, we ignore the conditional jump logic and only connect A to B (the next sequential instruction) while disregarding the jump to C. The mode **DF+CF** is native PonziGuard which includes both control and data flow.

Figure 12 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.

5.4.4 Effectiveness of node embeddings adopted in CRBG. In Section 3.5, we utilized Doc2Vec to enhance node embeddings in CRBG based on the operation descriptions from the Ethereum Yellow Paper. We believe that the

Table 5. 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%

Table 6. Variants of opcode description.

Value	Mnemonic	
0x57	JUMPI	**Original Description:** *"Conditionally alter the program counter."*
		Synonyms substitution: *"Conditionally change the instruction pointer."*
		Changing sentence structure or grammar: *"Alter the program counter based on the condition."*

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 5.2.1 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 5, 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, as shown in Table 6, 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 5, 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.

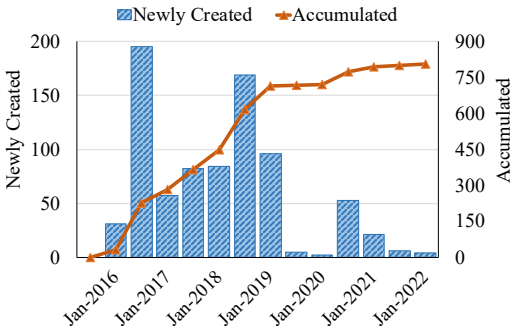

Fig. 13. Distribution of Ponzi schemes creation.

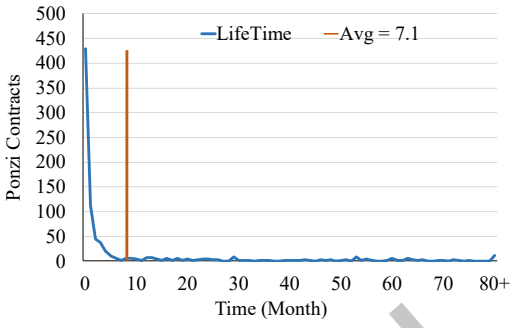

Fig. 14. Lifetime/Average Lifetime of Ponzi schemes.

Answer to RQ3: CRBG proves effective compared to the raw graph obtained directly from runtime. And ablation studies further confirm that our processes on CRBG such as graph pruning, subgraph joining, and enhanced node embeddings significantly enhance detection performance.

5.5 RQ4: Performance in real-world scenarios

To evaluate the effectiveness of PonziGuard in real-world scenarios, we conducted two experiments on the Ethereum Mainnet. The first is a large-scale transaction replay on the Ethereum historical blocks, to assess the number and economic impact of Ponzi contracts over the past few years. The second involved collecting recently deployed contracts to detect new Ponzi schemes.

5.5.1 Historical Transaction Replay. 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 real-world scenarios. 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* [14]. We randomly selected 50 contracts[3] from these 497 contracts and conducted a manual examination through Remix [31], 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.

Creation Time of Ponzi Contracts. Figure 13 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 [26]. The global crypto mining boom in 2021 [12] resulted in another minor peak in Ponzi schemes. With the increasing popularity of various tokens on

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

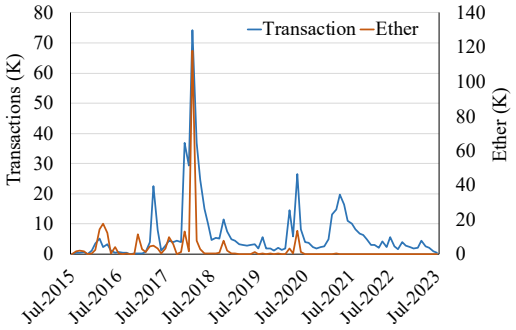

Fig. 15. Transaction flow and Ether flow of Ponzi schemes over time.

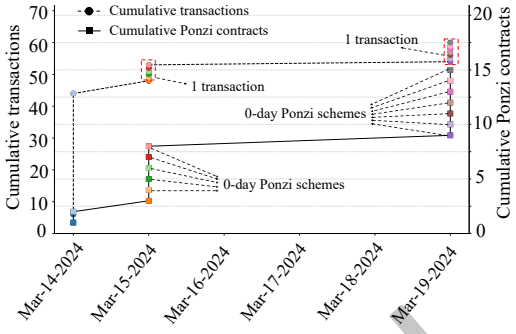

Fig. 16. *0-day* Ponzi schemes in 10,000 recently deployed smart contracts.

Ethereum, ERC-20 Tokens for instance, we anticipate another peak in Ponzi schemes on Ethereum in the near future.

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 14. 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. As for the short lifetime of the Ponzi, some ended because the scam failed to attract new investors, resulting in its collapse, exemplified by the scam `theultimatepyramid`[5]. Others were ended because the scam owner intentionally triggered the self-destruct function of the contracts and absconded with the funds, exemplified by the scam `TheGame`[6]. These findings indicate that Ponzi contracts are likely to collapse within a short period of time, and most of their users will not be able to reclaim their promised returns.

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 15 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[7].

5.5.2 Detecting new Ponzi Schemes. To ascertain whether new Ponzi schemes still keep emerging and whether our tool can detect them effectively, we collected the source code of about 10,000 smart contracts recently deployed in March 2024 from Etherescan [14] and employed PonziGuard for detection. After manual confirmation, we identified 15 Ponzi schemes among these 10,000 recently deployed contracts, and their address are available at our GitHub repository[8]. As shown in Figure 16, most of these Ponzi contracts were newly deployed and have

[4]For instance: 0xa90be2201bfed97587a2a17949e8624eafe51d13 and 0xf8f04b23dace12841343ecf0e06124354515cc42
[5]0x0F26c26318872E8Fa85DEe5d30cbA45eD53B3D3E
[6]0x3e84512f277A5081B9209831C51bCe665035D9DB
[7]Evidenced by the example of 0xb3836d31d43d315ba74c21aad3818f9378256152
[8]https://github.com/PonziDetection/SmartPonzi/tree/main/dataset/Result

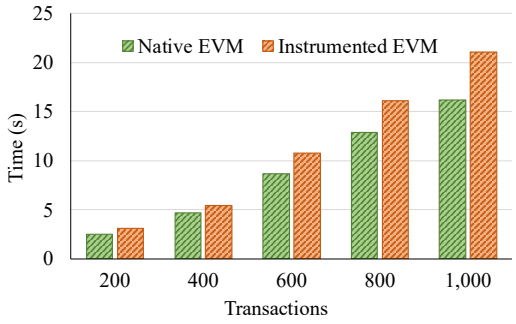

Fig. 17. Overhead on the ground-truth dataset.

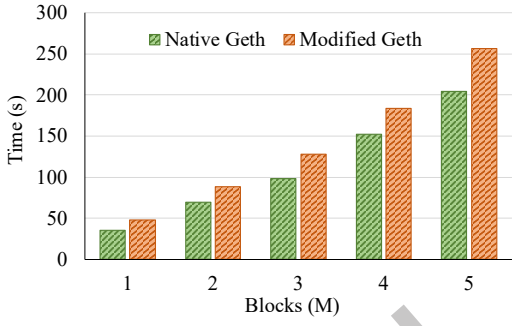

Fig. 18. Overhead in real-world scenarios.

only one transaction (creation transaction) on them, showcasing the ability of PonziGuard to uncover new Ponzi schemes.

Answer to RQ4: PonziGuard successfully identified 805 Ponzi contracts in the Ethereum historical blocks and found 15 latest 0-day Ponzi contracts deployed within a month, demonstrating the effectiveness of PonziGuard in real-world scenarios. 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 RQ5: 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 5.2, 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 17, 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 Real-World Scenarios. 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 18, 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 RQ5: 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. [4], 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. [42] who leverage behavior forest similarity to detect Ponzi contracts, and Chen et al. [9] 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, 10, 16, 19, 24, 28, 29, 33, 60, 63] use static information like opcode or transactions for machine learning models to improve detection capabilities. For example, Chen et al. [10] train an XGBoost model based on the frequency of opcode and transactions to identify Ponzi contracts. Galletta et al. [19] leverage transaction statistics information to train a classifiers for Ponzi detection. 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 [3, 21, 23, 30, 45, 56, 62]. ContractFuzzer [23] is a black-box fuzzer for Ethereum smart contracts to detect security bugs such as gasless send and timestamp dependency. Some grey-box fuzzers [21, 30, 40, 45, 56–58, 62] 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 [39, 41, 47]. There have been studies that leverage taint analysis to help analyze smart contract such as Osiris [46], Sereum [34] and EthPloit [62]. Osiris [46] is an integer bug detection framework that combines taint analysis and symbolic execution. Sereum [34] leverages taint analysis to protect smart contracts with re-entrancy vulnerabilities from being exploited. EthPloit [62] 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 subset of deep learning techniques that have shown remarkable effectiveness across various domains, including user authentication [50, 53] and mitigating vulnerabilities [65]. GNNs are designed to process and learn from data that is structured in the form of graphs [55]. They have been shown highly effective in various tasks, such as node classification [22, 44], link prediction [59, 61], and graph classification [5, 48].

```
1   Origin:
2   uint index = Calculator.length + 1;
3   Calculator[index].ethereumAddress = msg.sender;
4   Calculator[index].name = ``masterly calculated'';
5
6
7   Modified:
8   uint index = Calculator.length;
9   Calculator.length += 1;
10  Calculator[index].ethereumAddress = msg.sender;
11  Calculator[index].name = ``masterly calculated'';
```

Listing 2. Snippet of squareRootPonzi

In this paper, we leverage GNNs for CRBG analysis and formulate the detection of Ponzi contracts as a graph classification task.

7 DISCUSSION

We note that some static analysis tools [8, 11, 18, 36] 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[9] is a false positive case in the previous study [4], 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 8 and Line 9. 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.

Another point to note is that when historical transactions are unavailable, our tool requires the contract's source code for detection. This is because the static analysis, as well as transaction generation and contract deployment, rely on the source code to perform data dependency analysis, compile the contract, generate the ABI, and deploy the bytecode on EVM. Detecting Ponzi scheme contracts without source code and ABI will be the focus of our future research.

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 real-world scenarios with acceptable overhead. Moreover, our preliminary experiment conducted on Ethereum Mainnet has identified 805 Ponzi contracts that have caused millions of USD

[9] 0x8ea6c8077d6316b46e449aec8fb60a606cf50eea

in financial losses. We also found *0-day* Ponzi schemes in the recently deployed 10,000 smart contracts. This highlights the severity of Ponzi contracts on Ethereum and the pressing need to effectively identify them.

ACKNOWLEDGEMENT

This research was supported in part by the National Key R&D Program of China under grant No. 2021YFB2700200, the National Natural Science Foundation of China under grants No. 62172303, 62472323, the Key R&D Program of Hubei Province under grant No. 2022BAA039, 2024BAB018, and Key R&D Program of Shandong Province under grant No. 2022CXPT055. This research was also supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the National Research Foundation, Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) programme. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not reflect the views of the National Research Foundation Singapore or the Cyber Security Agency of Singapore.

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] Imran Ashraf and W. K. Chant. 2022. An Empirical Study on the Effects of Entry Function Pairs in Fuzzing Smart Contracts. In *IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*.
[4] 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).
[5] 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).
[6] Chainalysis. 2022. The Chainalysis 2022 Crypto Crime Report. Retrieved March 20, 2023 from https://go.chainalysis.com/2022-Crypto-Crime-Report.html
[7] Shibao Chen and Fei Li. 2024. Ponzi scheme detection in smart contracts using the integration of deep learning and formal verification. *IET Blockchain* 4 (2024).
[8] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2020).
[9] 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)*.
[10] 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)*.
[11] ConsenSys. 2023. Mythril. Retrieved April 22, 2023 from https://github.com/ConsenSys/mythril/
[12] 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
[13] ethereum foundation. 2023. Official Go implementation of the Ethereum protocol. Retrieved October 20, 2022 from https://geth.ethereum.org/
[14] Etherscan. 2023. Etherscan.io. Retrieved March 20, 2023 from https://etherscan.io/
[15] 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).
[16] 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)*.
[17] 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)*.
[18] 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)*.

[19] Letterio Galletta and Fabio Pinelli. 2024. Explainable Ponzi Schemes Detection on Ethereum. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing (SAC)*.
[20] google. 2022. american fuzzy lop. Retrieved July 27, 2024 from https://github.com/google/AFL
[21] 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)*.
[22] Sergei Ivanov and Liudmila Prokhorenkova. 2021. Boost then Convolve: Gradient Boosting Meets Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.
[23] 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)*.
[24] 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)*.
[25] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*.
[26] 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
[27] 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 *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*.
[28] 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)*.
[29] Pengcheng Lu, Liang Cai, and Keting Yin. 2024. SourceP: Detecting Ponzi Schemes on Ethereum with Source Code. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.
[30] 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)*.
[31] Remix Project. 2023. Remix. Retrieved June 12, 2023 from https://remix-project.org/
[32] pytorch. 2023. Pytorch. Retrieved June 12, 2023 from https://pytorch.org/
[33] Kun Qian, Jinping Jia, Zhao Zhang, Xiang Li, Yanqin Yang, and Cheqing Jin. 2024. DS-Ponzi: Anti-jamming Detection of Ponzi Scheme on Ethereum Utilizing Dynamic-Static Features of Smart Contract Codes. In *Database Systems for Advanced Applications*, Makoto Onizuka, Jae-Gil Lee, Yongxin Tong, Chuan Xiao, Yoshiharu Ishikawa, Sihem Amer-Yahia, H. V. Jagadish, and Kejing Lu (Eds.).
[34] 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)*.
[35] 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/
[36] 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)*.
[37] 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)*.
[38] 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
[39] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. 2024. Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis. In *Proceedings of International World Wide Web Conferences (WWW)*.
[40] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[41] Cong Sun, Yuwan Ma, Dongrui Zeng, Gang Tan, Siqi Ma, and Yafei Wu. 2023. μDep: Mutation-Based Dependency Generation for Precise Taint Analysis on Android Native Code. *IEEE Transactions on Dependable and Secure Computing* 20 (2023).
[42] 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)*.
[43] Nick Szabo. 1994. Smart Contracts: Building Blocks for Digital Markets.
[44] 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)*.
[45] 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)*.
[46] 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)*.

[47] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting Flow of Sensitive Data in Mini-Programs with Static Taint Analysis. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[48] Lanning Wei, Huan Zhao, Zhiqiang He, and Quanming Yao. 2023. Neural Architecture Search for GNN-Based Graph Classification. *ACM Trans. Inf. Syst.* (2023).
[49] Gavin Wood. 2022. Ethereum: A secure decentralised generalised transaction ledger Berlin version. Retrieved January 26, 2023 from https://ethereum.github.io/yellowpaper/paper.pdf
[50] C. Wu, H. Cao, G. Xu, C. Zhou, J. Sun, R. Yan, Y. Liu, and H. Jiang. 2024. It's All in the Touch: Authenticating Users with HOST Gestures on Multi-Touch Screen Devices. *IEEE Transactions on Mobile Computing* (2024).
[51] Cong Wu, Jing Chen, Kun He, Ziming Zhao, Ruiying Du, and Chen Zhang. 2022. EchoHand: High Accuracy and Presentation Attack Resistant Hand Authentication on Commodity Mobile Devices. In *Proceedings of Conference on Computer and Communications Security (CCS)*.
[52] Cong Wu, Kun He, Jing Chen, Ruiying Du, and Yang Xiang. 2020. CaIAuth: Context-Aware Implicit Authentication When the Screen Is Awake. *IEEE Internet of Things Journal* 7 (2020).
[53] Cong Wu, Kun He, Jing Chen, Ziming Zhao, and Ruiying Du. 2020. Liveness is Not Enough: Enhancing Fingerprint Authentication with Behavioral Biometrics to Defeat Puppet Attacks. In *Proceedings of USENIX Security Symposium*.
[54] Cong Wu, Kun He, Jing Chen, Ziming Zhao, and Ruiying Du. 2022. Toward Robust Detection of Puppet Attacks via Characterizing Fingertip-Touch Behaviors. *IEEE Transactions on Dependable and Secure Computing* 19 (2022).
[55] 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).
[56] 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)*.
[57] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2024. xFuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21 (2024).
[58] Mingxi Ye, Yuhong Nan, Zibin Zheng, Dongpeng Wu, and Huizhong Li. 2023. Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[59] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware graph neural networks. In *International conference on machine learning (ICML)*. 7134–7143.
[60] 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
[61] 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).
[62] 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)*.
[63] Zibin Zheng, Weili Chen, Zhijie Zhong, Zhiguang Chen, and Yutong Lu. 2023. Securing the Ethereum from Smart Ponzi Schemes: Identification Using Static Features. *ACM Trans. Softw. Eng. Methodol.* (2023).
[64] zhijie. 2023. Ponzi Contract Dataset. Retrieved May 3, 2024 from https://xblock.pro/#/dataset/25
[65] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting Deep Learning-based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Trans. Softw. Eng. Methodol.* 30 (2021).

A APPENDIX

Table 7. Variants of opcode description

Mnemonic	Original Description	Synonyms substitution	Changing sentence structure or grammar
STOP	Halts execution.	Terminates execution.	Execution is halted.
ADD	Addition operation.	Performs addition.	Execute an addition operation.
MUL	Multiplication operation.	Carries out multiplication.	Perform a multiplication operation.
SUB	Subtraction operation.	Performs subtraction.	Execute a subtraction operation.
DIV	Integer division operation.	Conducts integer division.	Perform an integer division operation.
SDIV	Signed integer division operation (truncated).	Carries out signed integer division (truncated).	Perform a truncated signed integer division operation
MOD	Modulo remainder operation.	Calculates modulo remainder.	Perform a modulo operation to find the remainder.
SMOD	Signed modulo remainder operation.	Computes signed modulo remainder.	Perform a signed modulo operation
ADDMOD	Modulo addition operation.	Performs modulo addition.	Execute an addition operation with modulo applied
MULMOD	Modulo multiplication operation.	Carries out modulo multiplication.	Execute a multiplication operation with modulo applied
EXP	Exponential operation.	Performs exponentiation.	Execute an exponential operation.
SIGNEXTEND	Extend length of two's complement signed integer.	Extends the length of a two's complement signed integer.	Extend the size of a two's complement signed integer.
LT	Less-than comparison.	Performs less-than comparison.	Compare if one value is less than another.
GT	Greater-than comparison.	Performs greater-than comparison.	Compare if one value is greater than another.
SLT	Signed less-than comparison.	Performs signed less-than comparison.	Compare if one value is less than another
SGT	Signed greater-than comparison.	Performs signed greater-than comparison.	Compare if one value is greater than another
EQ	Equality comparison.	Checks for equality.	Determine if two values are equal.
ISZERO	Simple not operator.	Performs a simple not operation.	Check if a value is zero.
AND	Bitwise AND operation.	Executes bitwise AND operation.	Perform an AND operation at the bit level.
XOR	Bitwise XOR operation.	Executes bitwise XOR operation.	Perform an XOR operation at the bit level.
OR	Bitwise OR operation.	Executes bitwise OR operation.	Perform an OR operation at the bit level.
NOT	Bitwise NOT operation.	Performs bitwise NOT operation.	Execute a NOT operation at the bit level.
BYTE	Retrieve single byte from word.	Retrieves a single byte from a word.	Fetch a specific byte from a word, counting from the left.
SHA3	Compute Keccak-256 hash.	Computes the Keccak-256 hash.	Generate the Keccak-256 hash.
ADDRESS	Get address of currently executing account.	Obtains the address of the currently executing account.	Get the address associated with the executing account.
BALANCE	Get balance of the given account.	Retrieves the balance of a specified account.	Get the balance for a given account.
ORIGIN	Get execution origination address.	Gets the execution origin address.	Retrieve the address of the original transaction's sender.
CALLER	Get caller address.	Gets the caller's address.	Retrieve the address of the account responsible for the current execution.
CALLVALUE	Get deposited value by the instruction/transaction responsible for this execution.	Retrieves the value deposited by the responsible instruction/transaction.	Get the deposited value associated with this execution.
CALLDATALOAD	Get input data of current environment.	Loads the input data of the current environment.	Fetch the input data provided by the current message call or transaction.
CALLDATASIZE	Get size of input data in current environment.	Gets the size of input data in the current environment.	Retrieve the length of input data in this environment.
CALLDATACOPY	Copy input data in current environment to memory.	Copies input data from the current environment to memory.	Transfer input data from the current environment into memory.
CODESIZE	Get size of code running in current environment.	Retrieves the size of the executing code.	Determine the size of the code that is currently running.
CODECOPY	Copy code running in current environment to memory.	Copies the executing code to memory.	Transfer the running code to memory.
GASPRICE	Get price of gas in current environment.	Retrieves the current gas price specified by the transaction.	Obtain the gas price as set by the originating transaction.
EXTCODESIZE	Get size of an account's code.	Retrieves the size of the code for an account.	Determine the size of the code associated with an account.
EXTCODECOPY	Copy an account's code to memory.	Copies the code of an account into memory.	Transfer the code of an account from storage to memory.
BLOCKHASH	Get the hash of one of the 256 most recent complete blocks.	Retrieves the hash of one of the last 256 complete blocks.	Obtain the hash of a recent block among the last 256 blocks.

Mnemonic	Original Description	Synonyms substitution	Changing sentence structure or grammar
COINBASE	Get the block's beneficiary address.	Retrieves the address of the block's beneficiary.	Obtain the address of the recipient of the block reward.
TIMESTAMP	Get the block's timestamp.	Retrieves the timestamp of the block.	Obtain the timestamp for the current block.
NUMBER	Get the block's number.	Retrieves the number of the block.	Obtain the block number for the current block.
DIFFICULTY	Get the block's difficulty.	Retrieves the difficulty level of the block.	Obtain the difficulty of the current block.
GASLIMIT	Get the block's gas limit.	Retrieves the gas limit of the block.	Obtain the gas limit set for the current block.
POP	Remove item from stack.	Pops an item off the stack.	Remove the top item from the stack.
MLOAD	Load word from memory.	Loads a word from memory.	Retrieve a word from memory.
MSTORE	Save word to memory.	Stores a word into memory.	Save a word to memory.
SLOAD	Load word from storage.	Retrieves a word from storage.	Load a word from the storage.
SSTORE	Save word to storage.	Stores a word into storage.	Save a word to storage.
JUMP	Alter the program counter.	Change the instruction pointer.	Modify the program counter.
JUMPI	Conditionally alter the program counter.	Conditionally change the instruction pointer.	Alter the program counter based on a condition.
PC	Get the value of the program counter prior to the increment corresponding to this instruction.	Retrieves the value of the instruction pointer before the increment.	Obtain the value of the program counter before the current instruction increment.
MSIZE	Get the size of active memory in bytes.	Retrieves the size of the active memory in bytes.	Obtain the size of the active memory.
GAS	Get the amount of available gas.	Retrieves the available gas amount.	Obtain the remaining gas amount.
JUMPDEST	Mark a valid destination for jumps.	Designates a valid jump destination.	Mark a location as a valid jump destination, with no effect on the machine state.
PUSH1	Place 1 byte item on stack.	Push a 1-byte item onto the stack.	Add a 1-byte item to the stack.
DUP1	Duplicate 1st stack item.	Duplicates the top stack item.	Copy the top item of the stack.
SWAP1	Exchange 1st and 2nd stack items.	Swaps the top two stack items.	Swap the positions of the top two items on the stack.
LOG0	Append log record with no topics.	Creates a log record without topics.	Append a log record to the blockchain with no associated topics.
CREATE	Create a new account with associated code.	Generates a new account and attaches code.	Instantiate a new account with its associated code.
CALL	Message-call into an account.	Perform a message-call to an account.	Execute a message call to another account.
CALLCODE	Message-call into this account with an alternative account's code.	Performs a message call to the current account with a different account's code.	Make a message call to the current account using another account's code.
RETURN	Halt execution returning output data.	Stops execution and returns output data.	End execution and provide output data as a result.
SELFDESTRUCT	Halt execution and register account for later deletion.	Terminate execution and mark the account for deletion.	End execution and set the account for deletion at a later time.
DELEGATECALL	Message-call into this account with an alternative account's code	Perform a message-call with another account's code	Invoke a message call using the code of another account
REVERT	Halt execution reverting state changes but returning data and remaining gas.	Stop execution, undoing state changes but returning data and remaining gas.	End execution while reverting state changes, returning both data and unused gas.
RETURNDATACOPY	Copy output data from the previous call to memory.	Copies data from the previous call's output to memory.	Transfer the output data from the last call into memory.
RETURNDATASIZE	Get size of output data from the previous call from the current environment.	Retrieves the size of the output data from the previous call.	Obtain the size of the output data from the previous call.
STATICCALL	Static message-call into an account.	Perform a static message call to an account.	Execute a static message call to an account with no state modifications.
EXTCODEHASH	Get hash of an account's code.	Retrieves the hash value of an account's code.	Obtain the hash of the code belonging to an account.
SAR	Arithmetic (signed) right shift operation.	Performs an arithmetic right shift operation.	Execute a signed arithmetic right shift.
SHR	Logical right shift operation.	Executes a logical right shift.	Perform a logical right shift operation.
SHL	Left shift operation.	Executes a left shift operation.	Perform a left shift operation.