SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts

WEIMIN CHEN, Beijing University of Posts and Telecommunications, China
XINRAN LI, Beijing University of Posts and Telecommunications, China
YUTING SUI, Beijing University of Posts and Telecommunications, China
NINGYU HE, Peking University, China
HAOYU WANG*, Beijing University of Posts and Telecommunications, China
LEI WU†, Zhejiang University, China
XIAPU LUO, The Hong Kong Polytechnic University, China

Ponzi schemes are financial scams that lure users under the promise of high profits. With the prosperity of Bitcoin and blockchain technologies, there has been growing anecdotal evidence that this classic fraud has emerged in the blockchain ecosystem. Existing studies have proposed machine-learning based approaches for detecting Ponzi schemes, i.e., either based on the operation codes (opcodes) of the smart contract binaries or the transaction patterns of addresses. However, state-of-the-art approaches face several major limitations, including lacking interpretability and high false positive rates. Moreover, machine-learning based methods are susceptible to evasion techniques, and transaction-based techniques do not work on smart contracts that have a small number of transactions. These limitations render existing methods for detecting Ponzi schemes ineffective. In this paper, we propose SADPonzi, a semantic-aware detection approach for identifying Ponzi schemes in Ethereum smart contracts. Specifically, by strictly following the definition of Ponzi schemes, we propose a heuristic-guided symbolic execution technique to first generate the semantic information for each feasible path in smart contracts and then identify investor-related transfer behaviors and the distribution strategies adopted. Experimental result on a well-labelled benchmark suggests that SADPonzi can achieve 100% precision and recall, outperforming all existing machine-learning based techniques. We further apply SADPonzi to all 3.4 million smart contracts deployed by EOAs in Ethereum and identify 835 Ponzi scheme contracts, with over 17 million US Dollars invested by victims. Our observations confirm the urgency of identifying and mitigating Ponzi schemes in the blockchain ecosystem.

CCS Concepts: • **Security and privacy** → **Software and application security**; **Intrusion/anomaly detection and malware mitigation**.

Additional Key Words and Phrases: Ethereum; smart contract; Ponzi scheme; symbolic execution

*Corresponding Author: Haoyu Wang (haoyuwang@bupt.edu.cn).
†Lei Wu is also with the Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province.

Authors' addresses: Weimin Chen, cwmjy1314@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Xinran Li, Beijing University of Posts and Telecommunications, Beijing, China, lxr_bupt@bupt.edu.cn; Yuting Sui, Beijing University of Posts and Telecommunications, Beijing, China, syt@bupt.edu.cn; Ningyu He, Peking University, Beijing, China, ningyu.he@pku.edu.cn; Haoyu Wang, Beijing University of Posts and Telecommunications, Beijing, China, haoyuwang@bupt.edu.cn; Lei Wu, Zhejiang University, Hangzhou, China, lei_wu@zju.edu.cn; Xiapu Luo, The Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk.

2476-1249/2021/6-ART26 $15.00
https://doi.org/10.1145/3460093

ACM Reference Format:
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. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 26 (June 2021), 30 pages. https://doi.org/10.1145/3460093

1 INTRODUCTION

Blockchain techniques (systems) and cryptocurrencies have been booming in recent years. However, a number of emerging scams have been exploiting blockchains and cryptocurrencies to make a profit. *Ponzi schemes*, a form of fraud that lures investors and pays profits to early investors, has a long history since the 18th century [3]. There is growing anecdotal evidence that this fraud has emerged in the blockchain ecosystem [4, 5, 22]. For example, a number of blockchain Ponzi schemes have been reported on Bitcointalks (the "Scam Accusation" board) [8], the most popular forum dedicated to the discussion of blockchain.

Due to this, blockchain Ponzi schemes have received attention from the research community. Marie Vasek and Tyler Moore [55] presented the first empirical analysis of Bitcoin-based scams in 2015, including Ponzi schemes (in the form of "high-yield investment programs"). After that, some papers have characterized blockchain Ponzi schemes by collecting known scam reports [20, 56]. The emergence of smart contracts, however, facilitates the spread of Ponzi schemes. Taking advantage of smart contracts, the initiator of a Ponzi scheme can stay anonymous. Further, the identity of the contract and the money extracted from the contract do not need to be disclosed after its establishment. Moreover, due to the nature of smart contracts, no one can terminate the program or return the victim's money.

State-of-the-art. A few recent studies have proposed techniques to detect ***Ponzi scheme smart contracts*** (*ponzitracts* for short). To the best of our knowledge, all existing efforts are taking advantage of machine learning techniques for the detection. They can be generally classified into two types according to their extracted features. The first line of work [42] relies on the *transaction behaviors* of smart contracts to perform detection. It assumes that the *Gini coefficient* [1] and other features could reflect the inequality of returns to investors (as seen in Ponzi schemes). Alternatively, the second line of work [32] takes the operation code (i.e., opcode) extracted from the smart contract bytecode as features to train a classifier. The basic idea is that the opcode frequency distribution should differ between Ponzi and non-Ponzi smart contracts.

However, state-of-the-art approaches face several limitations. On the one hand, *transaction behavior based detection requires a considerable number of transactions to learn the behaviors*. Thus, only popular ponzitracts that have lured a number of victims can be identified. On the other hand, *detection techniques based on the distribution of opcodes frequency lack interpretability*. Moreover, it is *prone to evasion techniques*, e.g., adding or removing some opcodes in the smart contracts will lead to its failure. Thus, *existing approaches may introduce some false positives and false negatives (see § 6), which are not applicable to analyze large scale smart contracts in the wild.*

This Work. In this paper, we present SADPonzi, a novel semantic-aware approach to detect Ponzi schemes in Ethereum smart contracts. Specifically, we first summarize the bytecode-level patterns of different kinds of Ponzi schemes according to our hands-on experience and observations (see **§ 3**). After that, we summarize the challenges of static analysis based Ponzi scheme detection (see **§ 4**), and propose a symbolic execution based system to perform the detection with semantic information (see **§ 5**). The proposed system can detect Ponzi schemes according to their bytecode (runtime code) by comparing the extracted semantic information with those summarized Ponzi scheme patterns. Our tool can address the aforementioned limitations of existing approaches, i.e., SADPonzi can identify new ponzitracts even if they have no transactions. We then perform extensive experiments to demonstrate the performance of SADPonzi and show that SADPonzi

has better robustness than the machine learning baselines when facing code obfuscation based evasion techniques (see **§ 6** and **§ 7**). We further discuss the limitations of SADPonzi, including the generalizability issues (e.g., cannot handle new kinds of Ponzi schemes and the combination of multiple Ponzi schemes) and the limitations inherited from symbolic execution techniques (see **§ 8**).

This work makes the following main research contributions:

- We propose a semantic-aware approach for ponzitract detection, which can achieve a balance between accuracy and scalability. It is accurate as the detection process keeps the semantic information, following the patterns of Ponzi schemes. To ensure its scalability, we further adopt a heuristic-guided symbolic execution technique. In addition, our approach has strong interpretability.
- We develop SADPonzi, a prototype system that works on Ethereum smart contract bytecode. When evaluating on a ground-truth dataset with 1,395 well-labelled samples, SADPonzi outperforms all the machine learning baselines and achieves 100% precision and 100% recall.
- We evaluate the robustness of SADPonzi with a set of evasion samples generated by the state-of-the-art code obfuscation techniques [13]. Experimental results show that SADPonzi can maintain the 100% detection accuracy when it has sufficient computational resources. We further generate adversarial examples against the benchmark dataset [24] [26], which can bypass the machine learning baselines with an average success rate of 84.1%.
- We apply SADPonzi to all 3.4 million smart contracts deployed by external owned accounts (EOAs) on Ethereum to measure the overall landscape and the impact of ponzitracts. SADPonzi identifies 835 ponzitracts, involving thousands of victims and a large amount of invested money (over $ 17 Million).

We have released the curated benchmark, and all the 835 Ponzi scheme smart contracts we identified, to the research community for future research at:

https://github.com/Kenun99/SADPonzi

2 BACKGROUND

2.1 Ethereum and Smart Contract

2.1.1 Ethereum Account. Ethereum supports two types of accounts, i.e., *External Owned Accounts* (EOA) and *Smart Contract Accounts* (COA). An EOA can be regarded as a type of wallet, which keeps its owner's asset; a COA can be created by an EOA or another smart contract. `Ether` is the default cryptocurrency in Ethereum. Each account has a unique random-like 160-bits address.

2.1.2 Ethereum Smart Contract and Bytecode. At present, smart contracts in Ethereum are primarily written in *Solidity* [9], a Turing-complete and object-oriented programming language maintained by Ethereum. Anyone can execute smart contract code by making a transaction request. In a smart contract, its developer can define the behaviors that will be performed when the predetermined conditions are met (like transferring money to a designated account), or that will be triggered by actions from other accounts.

A smart contract will be compiled into bytecode which will be executed in the Ethereum Virtual Machine (EVM)[1]. Specifically, once a contract is invoked, EVM will match a function by its first four bytes of the signature that is calculated by the Keccak256 hash function [34] (`sha3()` for short, which relies on the opcode `SHA3`)[2]. For example, the function `invest()` will be identified by

[1]Note that there are two types of compiled bytecode, i.e., *creation* code and *runtime* code. The runtime code will be initialized and deployed on Ethereum by the creation code. In this work, we only focus on the runtime code.
[2]Each contract has a entry point named `__dispathcher()__`, where EVM determines which functions to invoke.

`0xe8b5e51f`. Moreover, whenever a contract receives Ether with an empty input or cannot match a pre-defined function, its fallback function will be automatically executed.

2.1.3 Ethereum Transaction. A transaction is the atomic operation on Ethereum. It records the history of the entire network. Both EOAs and COAs can submit a transaction to transfer Ether or execute a smart contract.

Once a smart contract executes a transaction, a set of opcodes can realize the Ether transaction and provide information. `CALLVALUE` captures the amount of the caller's investment sent together with the current call, corresponding to *msg.value* in Solidity. `CALLER` pushes the 160-bit address of the immediate transaction's sender, i.e., the account that invokes the function, *msg.sender* in Solidity, into the stack. In contrast, `ORIGIN` refers to the original EOA that started the transaction, which is represented by *tx.origin* in Solidity.

The transactions invoked by COAs are achieved by internal calls with the following three opcodes[3]:

$$\text{CALL}(gas, callee, value, arg_O, arg_L, ret_O, ret_L)$$
$$\text{STATICCALL}(gas, callee, arg_O, arg_L, ret_O, ret_L)$$
$$\text{DELEGATECALL}(gas, callee, arg_O, arg_L, ret_O, ret_L)$$

Here 1) arg_O and arg_L represent the beginning location and offset in memory for input arguments of internal call (*calldata*); and 2) ret_O and ret_L represent the beginning location and offset in memory for the return value. For the opcodes that can call other contracts, EVM invokes an internal transaction for the callee with the *calldata* from the callers' memory (denoted as α, where $\alpha = mem[arg_O : arg_O + arg_L]$). Next, it switches runtime context to execute the callee if it is a smart contract. At the end of the execution of the callee, the opcode `RETURN(O,L)` returns the result from the callee's memory (denoted as β, where $\beta = mem[O : O + L]$). Finally, the caller sets the result in memory, i.e., $mem[ret_O : ret_O + ret_L] := \beta$, and continues the execution.

2.1.4 Persistent Data. The data in smart contracts could be stored in different places, including *calldata*, *stack*, *memory* and *storage*. The *calldata* is a read-only space for storing the input of the transaction. The data in *memory* and *stack* will be wiped out after the execution of the invoked transaction, while the *storage* data can be permanently recorded on the blockchain. The *storage* is a continuous data pool with 2^{256} 256-bits-width slots, where storage data consumes some bits according to its type. In EVM, `SLOAD(`σ`)` enables the smart contracts to load storage data from $S[\sigma]$. `SSTORE(`σ`,`τ`)` stores τ at $S[\sigma]$. $S[\sigma]$ means the storage content at position σ, where S is the *storage* (notations are defined in Table 1).

Solidity is a strongly typed language supporting several kinds of data, such as state data, dynamical-size array (DSA for short) and mapping data. Most data is saved as state data, such as type<address>, type<[u]int*>, type<bytes>, and short type<string>. According to the type size and the declared order in the source code, Solidity optimizes the allocation of storage variables for reducing space consumption [10]. Each variable is saved in parts of an entire slot, which can be represented by a tuple referring to $\theta(\sigma, \epsilon, o)$, where σ represents the slot position of data, ϵ indicates the shift of data from the low end of the slot, and o refers to the length of type<data>. A storage data can be expressed as $S[\sigma][\epsilon : \epsilon + o]$. Especially, each DSA and mapping variable takes a unique slot as a mark to address its elements, e.g., DSA takes a 256-bits slot for base address, and mapping data takes it as a placeholder. In brief, we can confirm that two parameters come from the same variable by parsing θ. We will discuss more details of slots in § 5.3.1.

[3]Another opcode `CALLCODE` has been deprecated, and it will not be discussed in this paper.

2.2 Ponzi Scheme

2.2.1 Definition. U.S. Securities and Exchange Commission gives a definition for general Ponzi scheme frauds [52], "*... a Ponzi scheme is an investment fraud in which so-called returns are paid to existing investors through funds provided by new investors. Organizers of Ponzi schemes often attract new investors by promising investment and claiming they can generate high returns with little risk. With little or no legitimate income, a Ponzi scheme requires a steady stream of new investment ...*".

In Ethereum, the Ponzi scheme's initiator can remain anonymous by taking advantage of smart contracts. In this paper, we emphasize that the most significant characteristic of a Ponzi scheme smart contract is to redistribute the profit to the remaining players, i.e., ***a Ponzi scheme contains a redistribution scenario to divide new investments among its previous participants, and thus the money flow is pyramid-shape.***. This characteristic reveals that ponzitracts intend to maintain the information of participants to distribute dividends accordingly. As we plan to identify ponzitracts based on their semantic information, it is worth noting that the *data structure that stores players' information* and the *pyramid redistribution strategies* must be taken into account. Otherwise, only identifying redistribution behaviors will introduce many false positives, as they can be seen in many other kinds of contracts (e.g., non-Ponzi gambling games).

2.2.2 The general workflow of ponzitract. A ponzitract has two kinds of actions: 1) investing action (*I*): a player invokes a transaction to invest Ether, and the ponzitract stores user information, e.g., the recipient address (introduced by *msg.sender* or *tx.origin*) and the investment of the player (depending on *msg.value*); 2) rewarding action (*R*): the ponzitract pays a bonus for the participated players with a pyramid strategy, which is achieved by CALL. Only when enough victims fall for the scam could the Ponzi scheme have enough money to reward participants. Thus, a ponzitract has to maintain participating players' information during the lifecycle to perform reward actions.

A pyramid distribution strategy is associated with the achievement of *I* and *R*. We further categorize ponzitracts into four types, including *handover-scheme*, *chain-scheme*, *tree-scheme* and *withdraw-scheme*, according to their redistribution strategies and the data structures used to store user information. This categorization follows existing work [17], which provides a high-level description of transaction behavior, and we work out the corresponding bytecode-level Ponzi patterns in § 3. This taxonomy is comprehensive enough, as it covers all kinds of topology data structures (linear, tree-shaped, and handover), and withdraw methods (manually or automatically) that can be used by Ethereum Ponzi contracts.

3 MOTIVATING EXAMPLES AND PRELIMINARY EXPLORATION

We next briefly explain the logic of ponzitracts and summarize their bytecode-level patterns.

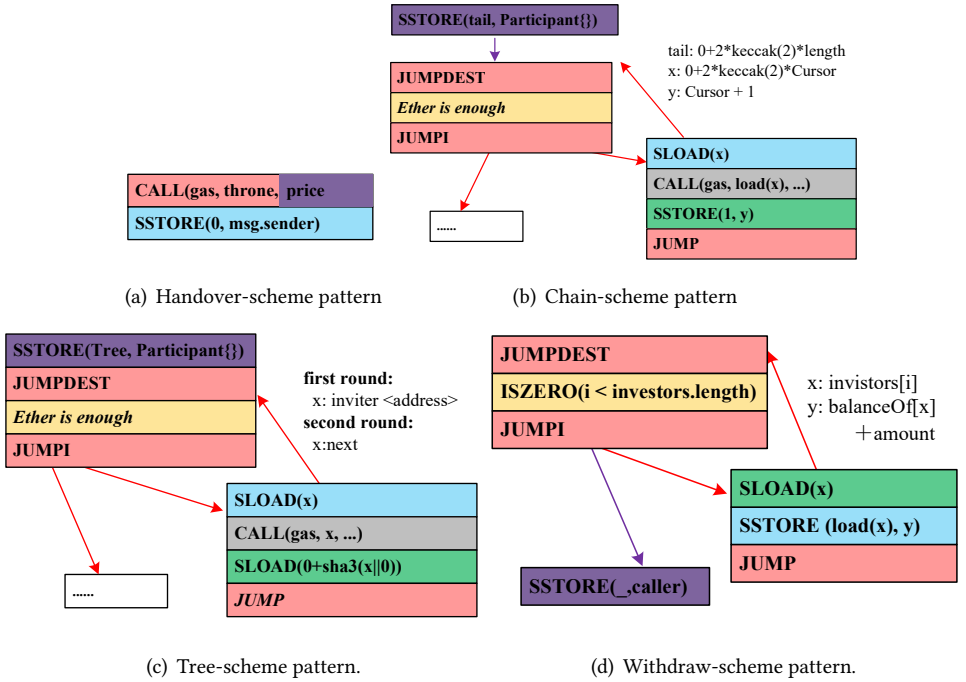
```
1. contract HandoverPonzi {
2. address public throne ;
3. uint public price = 1 ether;
4. uint fee = 0;
5.  function(){//fallback function
6.   if(msg.value < price) throw;
7.   fee += msg.value * 0.1;
8.   throne.transfer(msg.value*0.9);//reward
9.   throne = msg.sender;//invest
10.  price = price * 2;
11. }
12. function changeOwner
        (address adr) public{
13.  if (adr == owner)
14.    owner = adr;
15. }
16. function collectFee() public{
17.  if (msg.sender == owner)
18.    msg.sender.transfer(fee);
19. }
20.}
```

Fig. 1. An example of a handover-scheme contract.

3.1 Handover scheme

3.1.1 Example. In this scheme, the latest player will get all the money carried by the fresh participant (indicated by *msg.sender*). Figure 1 represents a typical case, where the state variable `throne` records the only address of the player and rewards him/her immediately when the condition is met. Specifically, when the investment (*msg.value*) is not less than the price set in advance (L6), the player can participate in the game, and 90% of the investment will be rewarded to the previous participant indicated by `throne` (L8). After that, the `throne` will be handed over to *msg.sender* (L9). Then the amount of `price` is multiplied by two (L10). During the investment event, the contract owner extracts 10% of investment and can withdraw the `fee` by invoking *collectFee()* at any time.

3.1.2 Bytecode-level Patterns. There is always only one investor in a pyramid distribution for a handover scheme: 1) once a new player has invested, the recipient's address is stored in a state variable, to which the contract will pay for her; 2) after the rewarding action, it sets the recipient variable to the caller (i.e. *msg.sender*), representing that the contract hands over the privilege to the new investor.



(a) Handover-scheme pattern (b) Chain-scheme pattern

(c) Tree-scheme pattern. (d) Withdraw-scheme pattern.

Fig. 2. Bytecode level patterns of four kinds of Ponzi schemes.

To detect handover-scheme Ponzi behaviors, we identify their investing actions and reward actions corresponding to the bytecode-level patterns (see Figure 2(a)). For investing actions, the handover-scheme ponzitract saves the addresses of players in storage with `SSTORE`. In the runtime code of this contract, two `SSTORE`s store the address in storage (denoted as i_{true} and i_{false}). However, we need further analysis to identify valid investing actions and filter out infeasible paths which are only exposed to the owner of the contract. i_{false} is used at the *changeOwner()* (L14) function, where the constraint at L13 requires that the caller must be the contract owner. This constraint reveals that

i_{false} cannot be executed by the players, hence should be filtered out. In contrast, for the i_{true} at the `fallback` function (highlighted in blue), the only constraint at L6 can be satisfied as long as the caller invests enough Ether. To be in-depth, the stack parameter of i_{true} indicates that the player's information is $\theta(0, 0, 160)$, indicating that the 160-bits sized address data is at slot 0, and the offset is 0. Next, the `CALL` (highlighted in red) performs a reward action in the handover-scheme ponzitract. L8 shows the reward action achieved by an external transfer. At the bytecode-level, it is executed by `CALL(gas,throne,msg.value*0.9)`. The second parameter indicates `CALL`'s recipient, saving the last investor's address. Based on this semantic information, we can confirm a single-layer pyramid money flow.

3.2 Chain Scheme

3.2.1 Example. A chain-scheme ponzitract uses a linear data structure to maintain the information of investors. Figure 3 illustrates a concrete example. First, a new player will be appended to the chain-like data structure with their address and the payout due, which is three times the original investment (L10). The participant sequence will designate her a sequence number (typically it is incremented by the order of the participation). Then, in the reward stage, the investment will be redistributed according to the sequence number. The `Cursor` variable denotes the distribution order (L12-L17), which always starts from the very beginning of the participant sequence (L3). A participant will only be paid off when the remaining contract balance is larger than her `payout` (L12). Generally speaking, in this scheme, the earlier you join, the more profit you will earn.

```
1. contract ChainPonzi {
2.   uint balance = 0;
3.   uint Cursor = 0;
4.   struct Participant {
5.     address etherAddress;
6.     uint payout;
7.   }
8.   Participant[] participants;
9.   function() {
10.    participants.push(Participant
       (msg.sender, msg.value* 3));
11.    balance += msg.value ;
12.  while (balance > participants[Cursor].
                                  payout) {
13.    uint payoutToSend = participants
                        [Cursor].payout;
14.    participants[Cursor].etherAddress
           .send(payoutToSend);
15.    balance = participants
                        [Cursor].payout;
16.    Cursor += 1;
17.  }
18. }
19.}
```

Fig. 3. An example of a chain-scheme contract.

3.2.2 Bytecode-level Patterns. The linear data structure keeps a chain-shaped distribution: 1) the recipient variable is stored in a DSA, and will be selected by a cursor; and 2) The cursor is updated linearly after the transfer. Figure 2(b) shows the bytecode-level pattern of a chain-scheme contract. `SSTORE` (highlighted in purple) is the outcome of L10 at the level of sourcecode, which stores the player's information in `Participant[] participants;` (a DSA). The `CALL` highlighted in grey transfers a bonus to the participant, which is a DSA element with the key of state variable `Cursor`. This key also will be updated linearly (highlighted in green), making the recipient selected in the chain-shape. Furthermore, we find the `Cursor` is in a loop shown in the red blocks, leading to a pyramid-shaped money flow. As the ponzitract cannot know how many investors it needs to record ahead of time, the fixed-sized array is hardly used to maintain investors' information. Therefore, the proposed chain-scheme pattern intends to cover DSA structure-based ones.

```
1. contract TreePonzi {
2.   struct Participant {
3.     address inviter;
4.     address herself;
5.   }
6.   mapping(address => Participant) Tree;
7.   address top = 0xffff..ffff;
8.   function enter(address inviter) public {
9.     uint amount = msg.value;
10.    Tree[msg.sender] = Participant(
           {herself: msg.sender,
           inviter: inviter});
11.    address next = inviter;
12.    while (next != top) {
13.      amount /= 2;
14.      next.send(amount);
15.      next = Tree[next].inviter;
16.    }
17.  }
18.}
```

Fig. 4. The detail of a tree-scheme smart contract.

3.3 Tree Scheme

3.3.1 Example. The tree-scheme is similar to the chain-scheme, except that the data structure used to maintain players' balances is tree-shaped. Figure 4 shows a typical tree-scheme contract. Once an "old" player invites a new player by calling the function `enter` (L8), the newcomer will be recorded as a node linked to its `inviter` (L10), i.e., its father node in the tree-shaped data structure `Tree`. Then the investment will be rewarded to the participant denoted by `next` (L11), which will traverse the tree till the root node (L12-L16). After each distribution, the amount of payment will be reduced by half (L13). As a result, the more players you or your children nodes invite, the more profit you will make.

3.3.2 Bytecode-level Patterns. The tree scheme uses a tree-shaped data structure, which is usually a mapping type. The recipients are stored in a mapping and will be selected by a key (i.e., `next`). The key `next` can be set to its father node after the transfer. The pattern shows that the contract will pay for a participant based on a tree-shaped distribution strategy.

Figure 2(c) shows the bytecode-level pattern. In the investing phase (highlighted in purple), the `SSTORE` sets the `inviter` as the father node of caller and stores the relationship in the mapping variable `Tree`. In the reward phase, the ponzitract rewards `next` by the opcode `CALL` highlighted in grey. Note that, `next` indicates the inviter of the caller, which will be further used to choose the next father node (highlighted in green). In brief, players are maintained in a tree-shape, and the contract distributes Ether to the father nodes of players. The scheme can be detected by following patterns. If the recipient of the reward action is a mapping variable whose key is a type<address>, we flag it as a tree-scheme Ponzi contract.

3.4 Withdraw Scheme

3.4.1 Example. Typically, in the withdraw-scheme, a mapping variable `balanceOf` is managed. This variable maintains users' balances. Figure 5 shows an example. To take part in this game, the player has to transfer Ethers directly to the smart contract. After receiving transferred Ethers, the `invest()` function is invoked, which can reflect this scheme's essence. To be specific, from L15 to L17, Ether from the new incoming participant will be divided and distributed to other players according to their investment proportion. After that, the freshman will be appended as an investor (L19), and wait for the next players' investment. Moreover, through the `withdraw` function (L10), the investors can freely withdraw their profit. Note that, under this scheme, the profit accumulated in `balanceOf` cannot be automatically transferred to the beneficiaries. The players have to call the `withdraw` function to get money, according to the `balanceOf` mapping. In other words, the

```
1. contract PonziICO {
2.   uint public total;
3.   mapping (address => uint)
                    public invested;
4.   mapping (address => uint)
                    public balanceOf;
5.   address[] investors;
6.   address owner = 0x4f22...1e;
7.   function withdraw() public{
8.     uint amount = balanceOf[msg.sender];
9.     balanceOf[msg.sender] = 0;
10.    msg.sender.transfer(amount);
11.  }
12.
13.  function invest() private{
14.    uint dividend = msg.value;
15.   for(uint i=0;i<investors.length;i++) {
16.       uint amount = dividend * invested[in
                    vestors[i]] / total;
17.       balanceOf[investors[i]] += amount;
18.   }
19.   investors.push(msg.sender);
20.   invested[msg.sender] = msg.value;
21.   total += msg.value;
22. }
23. function() payable{ invest(); }
24.}
```

Fig. 5. An example of a withdraw-scheme ponzitract.

contract under this scheme is an Ether pool, which is enlarged by investment and divided to the participants.

3.4.2 Bytecode-level patterns. An investor can invoke the withdraw function to make a profit, meanwhile her investment is divided and distributed to other qualified participants, as follows:

- There is a `CALL(_,msg.sender,value,...)` instruction that supports any participant to withdraw Ethers (`value`).
- The `value` is an element of mapping, representing how many Ethers that an investor can harvest (i.e., `balanceOf` for short).
- Once the contract receives an investment, the value of `balanceOf` will be increased accordingly, which means current user's investment is distributed to others.

Figure 2(d) details the bytecode-level pattern. At the investing stage, the `SSTORE` highlighted in purple is invoked by L19 which records the caller's address in the DSA `investors` and the value invested in the `balanceOf`. A player can execute the rewarding action (L10) according to the `balanceOf`, where the recipient is *msg.sender*. Although L10 only transfers Ether to the caller, the value is read from `balanceOf` that records the investment of players. Alternatively, the green block loads the key of `balanceOf`, and the blue part updates `balanceOf[investors[i]]` by `SSTORE`. As a result, users can withdraw according to the value recorded in this data structure. In other words, the pyramid distribution can be detected by analyzing the increment of players' balance records, which is triggered by investing behavior.

4 TECHNICAL CHALLENGES

In this work, we aim to build a static-analysis system to effectively identify ponzitracts by applying our (optimized) symbolic execution technique. To be specific, the symbolic execution will generate semantic information within constraints along each feasible path. We can then distinguish investor-related behaviors based on the semantic information and find out the adopted pyramid distribution strategies. Additionally, by following Ponzi scheme's definition in § 2.2, we can classify these contracts accordingly in order to conduct further analyses. However, two challenges remain to be overcome to implement the system, including *performing internal calls* and *parsing storage variables*. In the following, we discuss these two challenges and our solutions.

4.1 Performing Internal Calls

There are two kinds of opcodes to achieve jumping operations that need to be handled: 1) executing direct or conditional jump instructions, i.e., `JUMP` and `JUMPI`; or 2) invoking an internal call, i.e.,

CALL, STATICCALL and DELEGATECALL. The CFG of a single contract can be constructed by jumping instructions, where existing researchers have made outstanding contributions. In contrast, as an inherent problem of symbolic execution, the internal call analysis remains a challenge of analyzing the smart contracts. Namely, the result generated by internal call (*retvalue* for short) may affect the caller's execution, which is according to the input and the context of the callee. Therefore, to perform a comprehensive symbolic analysis, the caller must consider the *retvalue*.

```
1. contract eth { // callee
2.   function getReceipt(addressadr)
3.     public returns (addr){
4.     return addr;
5.   }
6. }
7. contract Caller{//caller
8. address public throne;
9. uint public price = 1 ether;
10. function(){//fallback function
11.   oldEth eth1 = oldEth(0x00);
12.   if(msg.value < price) throw;
13.   throne.transfer(msg.value*0.9); //re
      ward
14.   throne = eth.getReceipt(msg.sender);
15.   price = price * 2;
16. }
17.}
```

Fig. 6. A smart contract that contains an internal call.

Some existing symbolic execution frameworks can generate symbolic context, such as MAIAN [51], Oyente [7], teEther [44], Vandal [21], MadMax [38] and Securify [54]. However, these frameworks cannot model internal calls properly. Specifically, they just create new symbols to balance the stack without analyzing the side effects. For example, `throne` is updated by the *retvalue* of `eth.getReceipt(msg.sender)` at L14 in Figure 6. If a new symbol is created without tracking the propagation of the argument *msg.sender*, the analysis will fail to find such a handover scheme.

Alternatively, Manticore [50] and Mythril [6] are two state-of-the-art tools that support inter call analysis, however, both of them have issues modeling internal calls [36]. Manticore sets all symbolic parameters to constant values and performs a concolic analysis. Mythril analyzes all possible paths of the callee, which may get stuck due to the path explosion. Hence, existing techniques for cross-contract analysis cannot satisfy our needs.

Our solution. We build cross-contract paths to analyze symbolic context (i.e. *stack*, *memory*, *storage*, constraints) and recover *retvalue*. To be specific, we locate the internal call and extract the caller's input to perform a symbolic cross-contract analysis. After performing the cross-contract analysis, we store the *retvalue* in the caller's symbolic memory and model the following execution in the caller's context. The details will be discussed in § 5.2.

4.2 Parsing Storage Variables

To effectively fetch data, the storage follows an address-value structure. During the symbolic execution, we need to identify variables and determine the data dependency according to the address, especially for the parameters utilized by some important opcodes. For example, to identify a chain-scheme ponzitract, we should get the second stack parameter of CALL and check whether the recipient is an array variable recording the investors' address. However, the optimization of the Solidity compiler brings a challenge for identifying variables at bytecode level.

As mentioned in § 2.1, $\theta(\sigma, \epsilon, o)$ can determine a storage data. For example, the variable `owner` (L6) in Figure 5 records the address of contract owner and can be expressed as $\theta(4, 0, 160)$, which is stored in the low-end 160 bits of the fourth storage slot, as the data type of address is 160 bits. The way to address fixed-length array's elements is similar to the state variable, as they can be

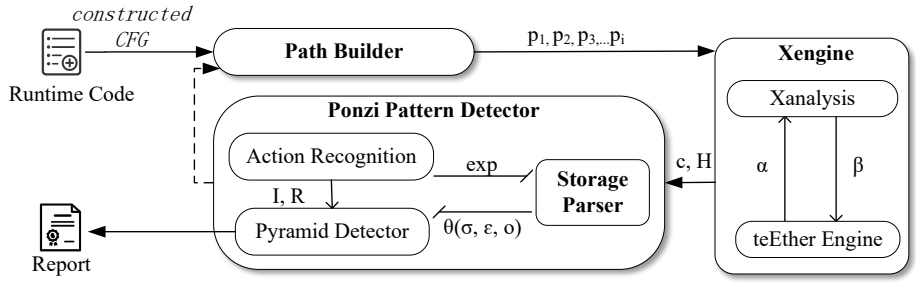


Fig. 7. The overall architecture of SADPonzi.

considered as several state variables stored closely in the *storage*. Due to their unpredictable size, DSA and mapping firstly consume an entire 256-bits slot as base address. Then the slot position of element is calculated by *sha3()* with θ. In short, it is challenging to locate a storage variable at bytecode level, as we are lack of concrete identifications (e.g., the variable names).

The complexity of the bytecode-level addressing makes it difficult to recognize variables. To find the behaviors of reading persistent data from storage and rewarding a participating player, we need to compare two symbolic expressions and determine whether they come from the same variable. To perform an accurate and practical analysis, we need to label the low-level variables.

Our solution. We model the storage variables with Z3 expressions and demonstrate the abstract syntax tree (AST for short) of variables in Figure 8(a). Based on the AST, the Z3 expression members in θ can locate storage variables (i.e., state, DSA and Mapping variable) according to the syntax shown in Figure 8(b). Furthermore, we design two algorithms to parse the storage variables according to the address model. The details will be discussed in § 5.3.1.

5 SADPONZI

We propose SADPonzi, a semantic-aware system to detect ponzitracts. Figure 7 shows the overall architecture of SADPonzi. Based on teEther [44], we design Xengine as the symbolic engine and add cross-contract analysis (Xanalysis for short). The workflow of our method is as follows. First, we construct a control flow graph (CFG) from runtime code (see § 2.1) and feed CFG to SADPonzi. Second, Path Builder generates paths that contain important opcodes, such as `CALL` and `SSTORE`. Next, Xengine generates symbolic context for the concrete paths. Once Xengine has handled an internal call operation, Xanalysis fetches the *calldata* of the callee (denoted as α) and performs a symbolic execution for the callee. The return value (or *retvalue*) from callee (denoted as β) is returned to the caller for generating a comprehensive context. After acquiring enough semantic information from Xengine, the Ponzi patterns detector (Detector for short) tries to match the patterns we discussed in § 3 and gives the final report.

Specifically, the Detector consists of three sub-modules, including *Storage Parser*, *Action Recognition* and *Pyramid Detector*. The Storage Parser handles each variables' AST and extracts the low-level information θ to recognize variables. The Action Recognition identifies the investing action and the rewarding action, denoted as I and R, respectively. At last, the Pyramid Detector analyzes the semantic information and control flow graph to match the pyramid-shape money flow of Ponzi scam and output the final result.

5.1 Path Builder

Instead of generating semantic information for the whole CFG, we first generate concrete paths of investing actions and reward actions and then build a symbolic context for each path to perform detection for a lower resource cost. To be specific, to identify an investing action I, we start from a function entry and find a feasible path to the opcode `SSTORE`. For the reward action R, its path should end with `CALL`, which transfers value to an account. Note that we take an optimized strategy to prioritize analyzing the opcodes of the fallback function, as it can naturally respond to Ether transferring transactions, namely, the investment. This strategy can be achieved by filtering outing irrelevant functions through their signatures during the execution of `__dispathcher()__` (see § 2.1).

To ease the path explosion problem, we further propose heuristic strategies to limit the depth of the path, increased by jump-related instructions (i.e., JUMP and JUMPI). We confine the depth by adjusting the limit to prevent the analysis from being tangled in dealing with endless loop structures.

5.2 Xengine

After acquiring concrete paths, we need to generate symbolic information. To this end, SADPonzi has a symbolic execution engine named *Xengine*, which is based on teEther [44] with the Z3 [12] as the SMT backend. In teEther, fixed-size elements (i.e., stack) are modelled using fixed-size bitvector expressions[4], dynamic-size elements (i.e., *calldata*, *memory* and *storage*) are modelled using *Z3 Array* expressions[5]. Note that, to handle the hash result of opcode `SHA3`, teEther models the relationship between the argument and the result of `SHA3` with a key-value pair. All symbolic hashes are maintained in a dictionary, denoted as H.

To support the cross-contract analysis, *Xanalysis* is designed to model the following three opcodes: `CALL`, `STATICCALL` and `DELEGATECALL`.

Specifically, once handling an opcode that triggers an internal call, Xengine is suspended, and Xanalysis gets the parameters in the stack and generates calldata (α) of the caller from the symbolic memory. Next, Xanalysis initiates a new context for the callee and emulates the execution completely in the new context. Last, Xanalysis feeds the *retvalue* (β) to the caller and Xengine continues the execution.

5.3 Ponzi Pattern Detector

As previously mentioned, the Detector has three sub-modules: *Storage Parser*, *Action Recognition* and *Pyramid Detector*.

Notations. Formally, we define a set of variables in Table 1. In table 2, we define functions for formalization at CFG layout, Z3 layout and Bytecode layout:

- **CFG layout.** We define three functions for CFG analysis. The first one is $bb(i)$ referring to the basic block that contains the opcode i. The next function is $dcs(bb)$, which returns all descendant nodes of bb. There is a path where i_2 is executed after i_1, if $bb(i_2)$ is one of the descendant basic block of $bb(i_1)$, which can be formalized as $bb(i_2) \in dcs(bb(i_1))$. Third, we define $is\_cycle(\mathcal{G}, bb)$ to determine whether there is a cycle containing bb in the CFG $\mathcal{G}$.
- **Z3 layout.** Each expression (denoted as $expr$) contains an AST, and the details can be fetched with Z3 interfaces. $expr.decl$ means the operation of $expr$, for example, if $expr = expr_1 + expr_2$, then $expr.decl$ equals to +. We also introduce $get\_left\_tree(expr)$, $get\_right\_tree(expr)$ and $get\_trees(expr)$ to get the operands, e.g., $expr_1 := get\_left\_tree(expr)$.

[4]bitvector is the bit-vector object in Z3, which supports abstract calculation
[5]a symbolic array data in Z3

Table 1. The Notations of Variables.

Vars	Description	Vars	Description
S	*storage*	μ	player's address, i.e., *msg.sender* or *tx.origin*
σ	slot position, represented by tuple σ'	λ	player's investment, i.e., *msg.value*
$base$	basic slot position	r	a variable maintaining the player's address
τ	the value stored by SSTORE	v	a variable maintaining the player's investment
$expr$	Z3 symbolic expression	t	the recipient in CALL
c	symbolic constraints	b	the value in CALL
θ	a tuple that can locate the variable	$\mathcal{G}$	CFG
$sigs$	a collection containing all function signatures	f	the first function's signature of a concrete path

Table 2. The Notations of Functions.

CFG Layout	Z3 Layout	Bytecode Layout	
$bb(i)$	$get\_left\_tree(expr)$	$tag(expr)$	$is\_v(\tau)$
$dcs(bb)$	$get\_right\_tree(expr)$	$\mathcal{S}(\theta_1, \theta_2)$	$is\_r(\tau)$
$is\_cycle(\mathcal{G}, bb)$	$get\_trees(expr)$	-	-

- **Bytecode layout.** To locate the player's address (μ) and amount of investment (λ), we present $is\_r(\tau)$ and $is\_v(\tau)$ to analyze the persistent data τ stored by SSTORE(_, τ). Considering τ is a 256-bits window, we try to find a 160-bits sub-window, whose length equals to the μ's, by increasing the offset. If $is\_r(\tau)$ is $True$, r will be a valid address and can be retrieved by $\theta(\sigma, \epsilon, 160)$, where

$$\text{is\_r}(\tau) := \begin{cases} True, \text{if } \mu \in \{\tau[8 * \epsilon : 160 + 8 * \epsilon] \mid \epsilon \in \mathbb{N}, \epsilon \leq 12\} \\ False, \text{otherwise} \end{cases}$$

Recalling the discussion in § 2.2 that an investment v should be calculated from λ, thus the size of v is usually 256-bit. To this end, we get all the sub-expressions of τ by a Z3 API called get_vars() [11] and find if the result contains λ. If $is\_v(\tau)$ is $True$, v is equivalent to $\theta(\sigma, 0, 256)$, where

$$\text{is\_v}(\tau) := \begin{cases} True, \text{ if } \lambda \in \text{get\_vars}(\tau) \\ False, \text{ otherwise} \end{cases}$$

Apart from τ, we also define notations for parsing slot position σ. In our study, each σ can be calculated by several arguments (e.g., $base$) with different strategies depending on the type of variable (see § 5.3.1). With ability to calculate σ, we can parse variables in storage and obtain corresponding θ, and save it into the database. Moreover, $tag(\theta)$ is used to query the parsing result and get the type of θ, which includes tag_1, tag_2 and tag_3, representing the state variable, DSA variable and mapping variable, respectively. In order to confirm if the two parameters come from the same variable, we present $\mathcal{S}$ to tackle the problem. To be specific, for DSA and mapping variables, they save the array's length and a 256-bits placeholder at $S[base]$, respectively. Both of them will take a whole slot in the stack. Therefore, $base$ can be regarded as an unique identification, and we represent $base$ of θ by θ_{base}. For state variables, the shift and size should be taken into consideration. Hence, only if the $\theta_1 = \theta_2$, the $\mathcal{S}$ will be $True$. Formally, the $\mathcal{S}$ is defined as:

$$\mathcal{S}(\theta_1, \theta_2) := \begin{cases} True, \text{ if } tag(\theta_1) = tag(\theta_2) \wedge tag(\theta_1) = tag_1 \wedge \theta_1 = \theta_2 \\ True, \text{ if } tag(\theta_1) = tag(\theta_2) \wedge tag(\theta_1) \in \{tag_2, tag_3\} \wedge \theta_{1_{base}} = \theta_{2_{base}} \\ False, \text{ otherwise} \end{cases}$$

$$
\begin{aligned}
< func > &:= concat \mid extract \mid select \mid sha3 \\
< expr > &:= func(expr*) \mid expr_1\ [+ - */]\ expr_2 \\
< concat > &:= expr_1 \| expr_2 \\
< extract > &:= expr[p_h ... p_l] \\
< select > &:= S[expr] \\
< sha3 > &:= sha3(expr)
\end{aligned}
$$

(a) Context-free syntax.

$$
\begin{aligned}
\text{Var} &:= extract(\epsilon + o, \epsilon, select(S, \sigma)) \\
\sigma^{\text{state}} &:= base \\
\sigma^{\text{dsa}} &:= sha3(base) + width * key + offset \\
\sigma^{\text{mapping}} &:= offset + sha3(concat(0, key, base))
\end{aligned}
$$

(b) Syntax for locating three important types.

Fig. 8. Syntaxes of the Storage Variables.

5.3.1 Storage Parser. Storage Parser, which identifies the storage variables for the detector, is used to solve the challenge mentioned in § 4.2. Specifically, we delve into the storage layout of different types of variables and determine their addresses accordingly from Z3 expressions (see Figure 8). As a result, for a given storage variable, we can identify it according to its unique θ.

Variable Expression. Based on symbolic execution, we abstract storage variables in Z3 expressions (see Figure 8(a)). Specifically, the *concat* function concatenates two expressions and returns the result. *sha*3 is a keccak256 hash computation, which can emulate the opcode `SHA3`. *extract* gets the low-end $p_h - p_l$ bits from the input bitvector *expr*. As discussed in § 5.2, we use a *Z3 Array* to model the *storage*, hence $select(S, expr)$ refers to loading data from storage S with *expr* as the key, i.e., $S[expr]$. For example, if the `owner` at L6 of Figure 5 is loaded into the stack, the value should be $Extract(160, 0, Select(S, 4))$, equivalent to $\theta(4, 0, 160)$.

Address Semantics. As aforementioned, there are three important types of variables in storage: *state*, *mapping* and *DSA* variables. These three types adopt distinct strategies to determine the address in storage, especially the σ in θ. Figure 8(b) illustrates the detailed algorithms. Specifically:

- **State Variable.** The state variable has a numerical slot and is directly indexed by the slot number. Thus, σ^{state} equals to the *base*, we formalized the σ' as: $\sigma' = (base)$.
- **DSA Variable.** The σ' of DSA element is a tuple $(base, key, width, offset)$. Specifically, we can calculate the σ^{dsa} by $sha3(base) + width * key + offset$, where *key* indicates the DSA variable's position, *width* is the number of slots a single element occupies, and *offset* refers to the order of the element member. Considering the `uint256` variable `participants[1].payout` at L8 in Figure 3, its *base* is 2 according to Solidity. `participants[1]` is the second variable in the contract (indexed by 1) in which each element takes 2 slots. `payout` is the second member, hence the offset is 1. Thus `participants[1].payout` can be expressed as $\theta(sha3(2) + 2 * 1 + 1, 0, 256)$, where $\sigma' = (2, 1, 2, 1)$.
- **Mapping Variable.** The σ' of mapping variable is $(base, key, offset)$. And the $\sigma^{mapping}$ is calculated by a hash computation with *base* and *key*, i.e., $\sigma^{mapping} = offset + sha3(key \| base)$. Taking `uint amount = balanceOf[msg.sender]` at L8 in Figure 5 as an example, it retrieves data from the mapping structure that takes *offset* as 0, *msg.sender* as the *key* and *base* as 2. As a result, $\sigma^{mapping} = sha3(uint256(caller) \| uint256(2))$. Note that the input of *sha*3() is a 512-bits zero-padding string. After removing the padding, we can recover σ' as $(2, caller, 0)$.

We further present two algorithms to determine the address semantics. On the one hand, Algorithm 1 (see Appendix A.1) acquires the Z3 expression and the syntax template: $Var := extract(o + \epsilon, \epsilon, S[\sigma])$ to construct $\theta(\sigma, \epsilon, o)$. On the other hand, Algorithm 2 (see Appendix A.2) recovers the raw parameters of σ and recognizes the type of the variable. If σ contains a hash computation, we recover the corresponding input by querying H (see § 5.2) or directly exploit the

concrete value[6]. After that, according to the patterns of σ at Figure 8(b), we can parse σ and get its detailed structure. Moreover, Algorithm 2 can parse the σ in SSTORE(σ, _), which can illustrate the writing history of storage, enabling the tracking of variables storing the player's information.

5.3.2 Action Recognition. As discussed in § 2.2, the two-phase recognition needs to be implemented.

Investing Action Recognition. Investing action is triggered by an Ether transferring transaction, which stores μ and λ in the storage by two SSTORE(_, τ). The investing action will be represented as $I := (i_r, i_v, r, v)$, where i_r and i_v store the player's address and investment in SSTOREs, respectively.

At first, we need to confirm that the investors can trigger the SSTOREs. We assume ponzitracts allow any users to participate, which means that the constraints c of the target SSTORE should not contain the verification against μ. Such a verification is usually adopted in validating the ownership of the contract, which requires μ be an address variable set by the creator (e.g., see L16 in Figure 3.1). To this end, we require c (the constraints of a path) should not contain a check between the μ and any 160-bits storage variable. Moreover, the target SSTORE should be triggered by transferring Ether (which means λ is larger than zero) or invoking the fallback function. Last, we need to analyze the stack parameters of SSTORE to find r, v, which can be identified by $is\_r(\tau)$ and $is\_v(\tau)$. As a result, the final recognition becomes:

$$i_r = \mathtt{SSTORE}_1(\_, \tau_1) \wedge is\_r(\tau_1) \wedge i_v = \mathtt{SSTORE}_2(\_, \tau_2) \wedge is\_v(\tau_2)$$
$$\wedge (\{\mu = S[*][8 * i : 160 + 8 * i] \mid i \in \mathbb{N}, i \leq 12\}) \not\subseteq c \wedge ((\lambda > 0 \in c) \vee f \notin sigs)$$

Rewarding Action Recognition. Rewarding action returns bonus to participants. Hence it can be expressed as $R := (i, t, b)$, where i is the corresponding opcode which transfers b Ether to address t. As CALL is the only opcode in Ethereum smart contracts that allows transferring Ether out, i is a CALL that rewards b to t, where t and b are the second and third stack parameter of i (from stack top to bottom), respectively. $R(i, t, b)$ can be found in two situations: if 1) the recipient is loaded from the investing action; or 2) the recipient is μ, meanwhile, the value is loaded from the investing action. Thus, the action can be formalized as:

$$i = \mathtt{CALL}(\_, t, b) \wedge ((t = I_r) \vee ((t = \mu) \wedge (b = I_v)))$$

5.3.3 Pyramid Detector. Based on I and R discovered in the previous phase, we perform the detection by analyzing the pyramid money flow for different Ponzi schemes. Specifically, if the recipient is the *msg.sender* or *tx.origin*, the contract may fall into the withdraw-scheme. While if it is a state variable, it may be a handover-scheme. Contracts with the DSA structures are more likely to be the chain-scheme. At last, we apply for the tree-scheme. Next we present four detectors to Ponzi schemes discussed at § 3.

- **Handover-scheme.** After rewarding to R_t, the contract updates R_t to μ in the investing action. During the procedure, I_{i_r} is executed after R_i finished, hence $bb(I_{i_r})$ must be the descendant basicblock of $bb(R_i)$. The formal description is shown below:

$$tag(I_r) = tag_1 \wedge I_r = R_t \wedge bb(I_{i_r}) \in dcs(bb(R_i))$$

- **Chain-scheme.** In this scheme, μ is stored at DSA where the contract linearly selects an element as the R_t by an index (named as key in our study and denoted as R_t^{key}). For a chain-shape distribution, R_t^{key} should be a state variable so that it can be updated incrementally; or

[6] As Solidity will pad the input of sha3() to 512 bits, we also need to remove the padding.

it is an immediate integer and R_i can be executed in a loop. Here is the formal description:

$$tag(I_r) = tag_2 \wedge I_r = R_t$$
$$\wedge\ (tag(R_t^{key}) = tag_1 \vee (type(R_t^{key}) = \text{integer} \wedge is\_cycle(\mathcal{G}, bb(R_i))))$$

- **Tree-scheme.** For these ponzitracts, μ is stored in a special mapping variable where the key is an <address>, e.g., `mapping(address=>address) usrs;`. In this structure, the ponzitract can use an address type element e to locate its father node by `usrs`$[e]$. Repeating the procedure, we can get the grandfather node of e by `usrs`[`usrs`$[e]$]. As discussed in § 2.1, the size of address type is 160 bits. We can identify such tree-shape variable whose key and corresponding value are all 160-bits.

$$tag(I_r) = tag_3 \wedge I_r = R_t \wedge I_r = \theta_{I_r}(\_, \_, 160)$$

  Where θ^{I_r} is the location information tuple of I_r.
- **Withdraw-scheme.** In this scheme, ponzitracts pay for μ with the bonus read from *balanceOf* which is updated in a loop structure when an investment comes.

$$R_r = \mu \wedge tag(R_b) = tag_2 \wedge I_v = R_b \wedge is\_cycle(\mathcal{G}, bb(I_{i_v}))$$

6 EXPERIMENTAL SETUP AND STUDY DESIGN

6.1 Implementation and Experimental Setup

SADPonzi is implemented on top of teEther [44], a widely used security analysis tool for EVM bytecode, because it adopts symbolic execution for traversing the feasible paths and uses the Z3 solver [12] as a SMT backend. We have implemented several modules: Xanalysis, Action Recognition, Pyramid Detector, and Storage Parser. Notably, we rename the teEther's engine as Xengine because we improve the cross-contract analysis ability with a new plugin called Xanalysis.

Our experiments are performed on a server running Ubutnu 18.04 with an i9-9900 CPU and 64 GB RAM. As mentioned in §5.2, we provide the configuration option of the *looplimit* to partially address the path explosion issue. During our experiments, we empirically set the looplimt from 3, 10, to 30 layers in sequence and the *timeout* as 20 minutes. We find this is enough to cover almost all the cases according to our evaluation (see § 7.1). By default, if SADPonzi gives a negative report, we activate Xanalysis and try again. Note that, these settings can be easily configured and customized to fulfil different requirements.

6.2 Research Questions

Our evaluation is driven by the following four research questions (RQs):

RQ1 **How accurate is SADPonzi in detecting Ponzi scheme smart contracts?** Considering that there are already several tools proposed for detecting ponzitracts, we seek to compare SADPonzi with state-of-the-art approaches using a reliable benchmark.

RQ2 **Is SADPonzi resilient to kinds of evasion techniques?** Considering that the scam developers may adopt advanced techniques (e.g., code obfuscation) to evade the detection of ponzitracts, it is necessary to evaluate whether SADPonzi is robust to evasion techniques.

RQ3 **How many smart contracts in the overall Ethereum ecosystem are Ponzi schemes?** No previous work has systematically measured the prevalence of Ponzi smart contracts in the overall ecosystem, as there are millions of smart contracts deployed. We want to measure the overall landscape of Ponzi schemes in the ecosystem and understand its evolution.

RQ4 **What is the impact of Ponzi smart contracts?** To understand the impact of Ponzi schemes, it is essential to measure how many investors are involved in and the overall economic scale.

Table 3. Overall Evaluation Results on the Benchmark.

Approach	Precision	Recall	F1-measure
TxML [42]	76.7%	57.9%	66.0%
OPCodeML [32]	92.0%	85.2%	88.5%
SADPonzi	**100%**	**100%**	**100%**

7 EXPERIMENTAL RESULTS

7.1 RQ1: The Effectiveness of SADPonzi

7.1.1 Baseline approaches. As previously mentioned, there are mainly two approaches for detecting Ponzi smart contracts in existing work: transaction-based machine learning classifiers (TxML for short) [33, 42] and machine learning classifiers based on the frequency of operation codes (OPCodeML) [32]. However, none of the existing tools have been released to the community. Thus, we make an effort to re-implement the tools based on the details provided by Jung et al. [42] and Chen et al. [32], respectively. Appendix B shows the details of how we build feature vectors and train the models.

7.1.2 Benchmark. Previous work [17] has collected a growing number of ponzitracts. The benchmark has been further adopted in [32, 33]. To enforce a fair comparison, our evaluation will perform on this benchmark. However, by manually analyzing the 184 labelled ponzitracts, we find that this benchmark is not as reliable as we expected. During their labelling process, they first manually analyze some ponzitracts, and then extend the dataset by identifying similar smart contracts using the normalized Levenshtein distance (NLD) [58], which inevitably introduces *non-Ponzi contracts* and *duplicated contracts*. Thus, the original benchmark may introduce bias (e.g., duplication will introduce an overfitting problem as the opcode features are identical, and non-Ponzi will introduce false positives). After removing 25 duplicates and 26 non-Ponzi contracts (51 in total), we have 133 ponzitracts left (see Table 8 for details). We believe the crafted benchmark is reliable, which has been released to the community. Moreover, we label 1,262 non-Ponzi DApp contracts from DAppTotal [2]. To enforce fair comparison, the proportion of ponzitracts and non-Ponzi contracts is constructed fully according to previous studies. *Overall, there are 1,395 contracts in the benchmark. All these samples are used to evaluate the effectiveness of SADPonzi. As for the ML baselines, we follow the previous works and split the dataset (1,395 contracts and 133 ponzitracts) into 80% for training and 20% for testing. The proportion of ponzitracts in the partition datasets is 9.5% (133/1395), which is inline with the entire dataset.*

7.1.3 Overall results. Table 3 shows the overall results. SADPonzi outperforms all existing approaches, with 100% precision and recall. *TxML* achieves the worst result, with only 76.7% and 57.9% for precision and recall, respectively. This result is inline with the evaluation in previous work [32]. Interestingly, our re-implementation of *OPCodeML* performs better than the one reported in their paper [32], with precision, recall and F-Measure of 92.0% (vs. 90%), 85.2% (vs. 80%) and 88.5% (vs. 84%), respectively. This might be because we removed some wrongly labelled contracts from the benchmark. It also indicates that we have implemented the models correctly.

7.1.4 Analysis. We then analyze the failure cases of these approaches to reveal their limitations.

TxML. Transaction based detection usually requires the smart contracts to have a considerable number of transactions (for learning their behaviors). However, we find that roughly 15% of smart contracts in the benchmark have less than ten transactions, and they are only active for less than one day. Thus, this approach cannot be used to detect new Ponzi smart contracts or less popular

ponzitracts. Actually, according to previous studies [31, 45], a large portion of the Ethereum smart contracts are less popular, which limits the real-world use of this approach.

OPCodeML. OPCodeML attains 92.0% precision and 85.2% recall on the benchmark. This approach only takes the frequency of opcodes as features, which do not have much semantic information. To further understand how it tells the difference between Ponzi and non-Ponzi smart contracts, we analyze the feature importance for each kind of opcodes, which is a score that indicates how useful or valuable each feature was in the construction of the boosted decision trees within the XGBoost model. Interestingly, XGBoost takes opcodes including `SSTORE`, `SLOAD` and `CALLDATALOAD` as the most important features. This is surprising as these opcodes are widely used in other types of smart contract, and they cannot represent the key features of a ponzitract. Based on this knowledge, the ponzitract developers can easily insert irrelevant opcode features to evade the detection of OPCodeML. We will further compare the robustness of OPCodeML and SADPonzi in §7.2.

Answer to RQ1: *Experimental results suggest that SADPonzi outperforms existing state-of-the-art machine learning approaches. We observe a number of limitations for existing techniques, which make them hard to apply to real-world scenarios. However, SADPonzi, preserving full semantic information of contract codes, achieves a good balance between accuracy and scalability.*

7.2 RQ2: The Robustness of SADPonzi

We next evaluate the robustness of SADPonzi against evasion techniques. Specifically, we consider the impact of *code obfuscation*, as it can affect all kinds of static analysis based detection. *We assume that the attacker can change the bytecode freely as long as the resulting bytecode maintains the original functionality and can be deployed at the mainnet.* To show the effectiveness of SADPonzi, we compare our tool with OPCodeML in various types of settings.

7.2.1 Code Obfuscation Method. We seek to obfuscate contracts at source-code level and compile to runtime code with the Solidity compiler (version *solc-0.4.24+e67f0147*). The obfuscated examples are generated by BiAn [13], which is a source-code level obfuscation tool developed for Solidity smart contracts. To the best of our knowledge, it is the only available tool for us to obfuscate Ethereum smart contract by the time of this study. BiAn can obfuscate contracts in three ways, including *Layout Obfuscation* (LAO for short), *Data flow Obfuscation* (DFO for short) and *Control Flow Obfuscation* (CFO for short). LAO does not change the logical design of source code but only affects the lexical analysis, such as replacing the names of variables and deleting comments. Namely, DFO and CFO change the data flow and control flow, respectively. Unfortunately, by the time of our study, the CFO module of BiAn cannot work properly, as acknowledged by the maintainers of BiAn. We assume a contract is obfuscated successfully when the obfuscated source code can be compiled into a legal contract. Hence, we only turn up LAO, DFO and LDO (combined with LAO and DFO) to generate evasive contracts. Since these obfuscations are not our contribution, we refer the readers to [13] for details. Besides, we find that most ponzitracts collected in the RQ1 dataset are optimised by *solc* (i.e., compiling with argument *–optimize-runs 200*), which may change the frequency of opcodes. To find whether the compiler's optimistic strategies can affect detection, we recompile existing contracts without any optimization. Therefore, based on the benchmark dataset, we generate four evasive scenarios to measure the robustness of SADPonzi (see Table 4). We fail to recompile some cases because not all of them are written in Solidity-0.4.24, and not all cases can be successfully obfuscated by BiAn.

7.2.2 Result. Table 4 shows the overall result. For the generated evasive cases, SADPonzi can accurately classify all of them. However, OPCodeML reports a number of FP and FN cases under

Table 4. A Comparison of the Robustness of SADPonzi and OPCodeML.

	# Recomplie		# LAO		# DFO		# LDO	
	FP	**FN**	**FP**	**FN**	**FP**	**FN**	**FP**	**FN**
OPCodeML	11/595	73/76	0/321	74/76	0/69	64/65	1/50	60/62
SADPonzi	0/595	**0/76***	0/321	**0/76***	0/69	**0/65**	0/50	**0/62**

* SADPonzi reports correctly as long as given larger computational resources.

different obfuscate conditions; we discuss them as follows. (1) *The impact of compiler optimization.* We successfully recompiled 595 non-ponzi contracts and 76 ponzitracts. To our surprise, OPCodeML reports 11 FPs and 73 FNs. However, the code recompiling does not affect SADPonzi. (2) *LAO obfuscation.* The variable names in source code are replaced with random strings, where 397 obfuscated contracts are successfully generated. OPCodeML reports 74 FNs. (3) *DFO obfuscation.* OPCodeML reports 64 FNs. None of the FPs are reported in the sub-dataset of LAO and DFO. In these two sub-datasets, BiAn obfuscates a contract, Fibonzi[7], which leads to the timeout of SADPonzi. However, by configuring the looplimits threshold of SADPonzi to 80, we successfully identify this obfuscated ponzitract. (4) *LDO obfuscation.* We combine LAO and DFO to generate 50 obfuscated non-ponzi contracts and 62 obfuscated ponzitracts (see the last column). OPCodeML has no way to be resistant to the obfuscated examples but reports 60 FNs. Our exploration suggests that OPCodeML is very sensitive to opcode frequency, while the detection model trained from the benchmark dataset cannot be transferred well to other dataset, let alone the obfuscated cases and adversarial attacks (note that we further provide the results of the adversarial attacks on OPCodeML models in Appendix C). Even recompiling the code can lead to the deterioration of its performance. SADPonzi is resilient to these kinds of evasion techniques, as it solely relies on the semantic information, which remains unchanged during code obfuscation.

Answer to RQ2: *SADPonzi is resilient to four types of obfuscation strategies that are commonly used today. However, the performance of OPCodeML decreases sharply when facing obfuscated and evasive smart contracts.*

7.3 RQ3: The Prevalence of Ponzi Smart Contracts in the Ecosystem

Dataset. To measure the Ponzi scheme's prevalence in the overall ecosystem, we have collected all the 3.4 million smart contracts deployed by EOA accounts on Ethereum from July 1st, 2015, the very beginning of Ethereum, to May 20th, 2020. They are used to measure the overall landscape and the evolution of ponzitracts. As previous work suggested that there are many duplicated smart contracts in Ethereum [40], we first identify the unique contracts by comparing the hash value of contract bytecode. At last, there are 83,269 unique contracts in total to be analyzed by SADPonzi. Note that for all the 83K smart contracts, it takes roughly 5 days for SADPonzi to analyze them on our server (see § 6.1) in a parallel manner.

Overall Results. Table 5 shows the overall result. Among the 83K unique smart contracts, we have identified 616 Ponzi schemes (see Column 2). Considering duplicated (identical) smart contracts, the number of identified Ponzi smart contracts has been extended to 835 (see Column 3). Specifically, the *Chain-scheme* accounts for a large portion of Ponzi schemes we identified, with 468 smart contracts in total (56.05%). Besides, 80 *Handover-scheme* contracts, 19 *Withdraw-scheme*

[7] 0xe8b55deaced913c5c6890331d2926ea0fcbe59ac

contracts and 268 *Tree-scheme* contracts are identified. Note that, as we evaluated in Section 7.1, SADPonzi is a robust detector to find contracts containing pyramid distribution strategies, hence it is not easy to report false positives. To further verify the identified real-world Ponzi schemes that are unknown to the community, we manually analyze the contacts with source code and find all of them can successfully pass the verification.

We have to admit that it is quite possible that we only find a subset of the active ponzitracts in the Ethereum ecosystem, and there remain some further works for finding more Ponzi schemes, which will be discussed in § 8.

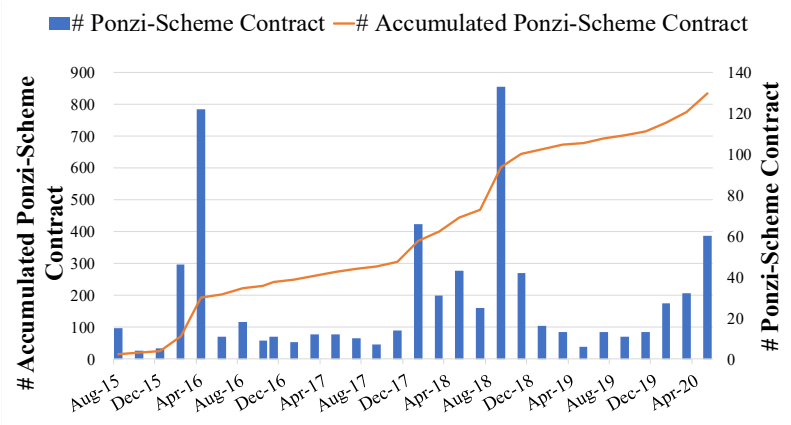


Fig. 9. The monthly distribution of ponzitracts.

The Creation Time of Ponzi Contracts. Figure 9 shows the monthly distribution of ponzitracts. Although ponzitracts were created from time to time, obviously, there are two peaks, as shown in Figure 9. The first peak appeared in April 2016, with 122 new ponzitract created. In the following two years, only a few Ponzi smart contracts were created. However, in 2018, with the growing popularity of Ethereum and the increasing price of Ether, it achieved another peak of ponzitracts in October, with 133 new ones created. The number of ponzitracts increased during the entire 1st quarter of 2020, which means many new people may fall for the scams.

The Creators of Ponzi Contracts. We further analyzed the attackers behind the Ponzi contracts. 444 EOAs created these 835 ponzitracts. Among them, 136 accounts have created more than one ponzitract. The most aggressive address[8] has created 17 ponzitracts. As the cost of contract creation is quite low in Ethereum, attackers can easily deploy a number of same/similar Ponzi contracts to attract unsuspecting users.

> **Answer to RQ3:** *By analyzing all the smart contracts in Ethereum, we have identified 835 Ponzi schemes created by 444 EOAs in total. These Ponzi contracts were created all the time within the span of 5 years, peaking in April 2016 and October 2018. This suggests that Ponzi schemes are prevalent in the ecosystem.*

7.4 RQ4: The Impact of Ponzi Smart Contracts

We have collected all the transactions related to the 835 Ponzi schemes (see Table 5), which are used to measure the involved users and the overall economic scale.

[8] 0xa2031ad9f8cd9577a830fb06e0a0d6f2455226ef

Table 5. The transactions related to Ponzi Smart Contracts.

Type	#Distinct Num	#Extend Num	Investment			
			# Victim	# Transaction	# Ether	USD
Chain	344	468	3,120	227,649	47,228.47	8,330,629.8233
Tree	212	268	14,565	99,612	52,073.8385	9,185,304.3730
Withdraw	13	19	174	824	463.8878	81,825.1690
Handover	47	80	148	2,603	624.6784	110,187.0230
Total	**616**	**835**	**17,870**	**330,688**	**100,390.87**	**17,707,946.38**

The number of transactions. In total, we have collected over 598.1K transaction records related to Ponzi schemes, with 330.7K incoming transactions. Each ponzitract has 396 incoming investment transactions on average. Although most of the ponzitracts attract less than 1 Ether, some of them have over 1K Ether. For example, one of the most popular Ponzi contracts[9] has received 2,082 incoming transactions. It suggests that some ponzitracts are quite popular in Ethereum.

The number of victims and the amount of money. We then analyzed the number of victims and the overall volume of money involved in ponzitracts. As shown in Table 5, roughly 17 thousand victims have sent over 100 thousand ETH (over 17 million US Dollars[10]) to the ponzitracts for investment. The most profitable ponzitract (same as footnote 2) has received 4,396 ETH (over $775K) from 558 victims. This result suggests that the overall market of Ponzi schemes is extremely huge, i.e., millions of US dollars were cheated from thousands of victims.

Answer to RQ4: *Our observation suggests the great impact of ponzitracts, i.e., thousands of victims were scammed of millions of US Dollars. It reveals the urgency to identify and mitigate blockchain Ponzi schemes.*

8 THREATS TO VALIDITY

Our work faces several limitations. First, the size of our crafted benchmark is limited, which may not fully represent all the variants of ponzitracts. This may result in generality issues when detecting some new ponzitracts, i.e., the combination of multi-scheme (see Appendix D). Nevertheless, we have tried our best to craft the benchmark and use it to compare our approach with existing techniques. Second, although we obtain enough semantic information, we did not investigate the deeper characteristics of each ponzitract (e.g., the dividend proportion), as there is no explicit general pattern at the bytecode level, which requires manual analyses case by case. After SADPonzi is released, new Ponzi schemes may get around our research and try to bypass the current detectors. Thus, it is important for SADPonzi to evolve as well. In terms of the provided semantic information of the ponzitracts' participants, as well as their behaviors, we believe it is feasible to extend the detectors to adapt to new Ponzi schemes. At last, although SADPonzi is resilient to four types of obfuscation strategies that are commonly used today, it cannot handle those evasion methods that can create serious path explosion (e.g., encrypting all storage data with DES and decrypting it when loading), as it inherits the limitations from symbolic execution techniques.

[9] 0x7cdfa222f37f5c4cce49b3bbfc415e8c911d1cd8

[10] As the price of Ether fluctuates, we estimate the profit using the price of Ether on October 2019 ($ 176.39/ETH).

9 RELATED WORK

9.1 Blockchain Ponzi Schemes

As Ponzi schemes are not a new topic, some previous works have discussed its feasibility [19], and characteristics [60]. With emerging of blockchain techniques, numerous opportunities are provided to scammers. Therefore, some researchers tried to characterize and identify them from various perspectives, like code similarity detection [18], stylometry technique [46] and mathematical ways [15]. Moreover, some proposed frameworks [32, 33, 42] are able to identify Ponzi scheme Ethereum smart contracts in favor of machine learning algorithms, which are compared with SADPonzi in § 6. Except for Ethereum, Ponzi schemes have also emerged on Bitcoin, and the identification of it has been studied by several works [17, 20, 56].

9.2 Smart Contract Analysis

A number of techniques have been proposed to analyze the security vulnerabilities in Ethereum smart contracts [6, 21, 23, 25, 28–30, 35, 37, 39, 41, 47, 49, 50, 53, 54, 59]. For example, Oyente [49], Manticore [50], and sCompile [23] are all based on symbolic execution, which can extract semantic information hidden in bytecode to identify vulnerabilities. Zeus [43] leveraged both abstract interpretation and symbolic model checking to perform a sound analysis. ReGuard [47] was focused on detecting reentrancy vulnerabilities based on fuzzing techniques. Besides, several works also focused on the gas related issues [14, 38], code clone behaviors [40, 48], etc.

10 CONCLUSION

In this paper, we present SADPonzi, a semantic-aware approach for detecting Ponzi schemes in Ethereum smart contracts. Our approach is based on a heuristic-guided symbolic execution technique to model the execution of investor-related operations, which can achieve a balance between accuracy and scalability. Experimental results indicate that SADPonzi outperforms all the existing machine learning based techniques in accuracy and robustness. We further make efforts to apply SADPonzi to analyze over 3.4 million smart contracts in the Ethereum ecosystem and have identified 835 Ponzi schemes in total. The overall market of Ethereum Ponzi schemes is over $17 million, which is urgent to raise the attention from the whole community.

ACKNOWLEDGMENT

We sincerely thank our shepherd Prof. Heather Zheng (University of Chicago) and the anonymous reviewers for their valuable feedback and suggestions. We thank Prof. Gareth Tyson (Queen Mary University of London) and Liu Wang (Beijing University of Posts and Telecommunications) for their proofreading. This work was supported by the National Natural Science Foundation of China (grants No.62072046 and No.61702045), Hong Kong RGC Project (No. 152193/19E), and the Fundamental Research Funds for the Central Universities (K20210226).

REFERENCES

[1] Gini coefficient. [EB/OL]. https://en.wikipedia.org/wiki/Gini_coefficient Accessed April 4, 2020.
[2] DApp browser, Aug. 2020.
[3] Definition of Ponzi scheme, Aug. 2020.
[4] Inside a crypto 'ponzi': How the $6.5m banana.fund fraud unravelled, Aug 2020.
[5] Millions of people fell for crypto-ponzi schemes in 2019, January 2020.
[6] Mythril, security analysis tool for EVM bytecode, Aug. 2020.
[7] Oyente, static analyzer for Ethereum smart contract, Aug. 2020.
[8] Scam accusations - bitcointalk, Aug 2020.
[9] Solidity official document site, Aug. 2020.
[10] Solidity official site, Aug. 2020.

[11] Z3 prover, Aug. 2020.
[12] Z3 theorem prover, Aug. 2020.
[13] Bian is a source code level code obfuscation tool developed for solidity smart contracts., Jan. 2021.
[14] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125. Springer, 2020.
[15] Marc Artzrouni. The mathematics of ponzi schemes. *Mathematical Social Sciences*, 58(2):190–201, 2009.
[16] Emad Badawi and G. Jourdan. Cryptocurrencies emerging threats and defensive mechanisms: A systematic literature review. *IEEE Access*, 8:200021–200037, 2020.
[17] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *arXiv: Cryptography and Security*, 2017.
[18] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277, 2020.
[19] Olivier Blanchard and Philippe Weil. Dynamic efficiency, the riskless rate, and debt ponzi games under uncertainty. *The BE Journal of Macroeconomics*, 1(2), 2001.
[20] Yazan Boshmaf, Charitha Elvitigala, Husam Al Jawaheri, Primal Wijesekera, and Mashael Al Sabah. Investigating mmm ponzi scheme on bitcoin. *arXiv: Cryptography and Security*, 2019.
[21] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.
[22] Philippe Castonguay. FairWin, a Ponzi contract, Oct. 2019.
[23] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*, pages 286–304. Springer, 2019.
[24] Hongge Chen, Huan Zhang, Duane S. Boning, and Cho-Jui Hsieh. Robust decision trees against adversarial examples. *CoRR*, abs/1902.10660, 2019.
[25] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering (TSE)*, 2021.
[26] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17. Association for Computing Machinery, 2017.
[27] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.
[28] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. GasChecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020.
[29] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proceedings of IEEE/ACM International Conference on Software Engineering(ICSE)*, 2018.
[30] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.
[31] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In *Proceedings of IEEE INFOCOM*, 2018.
[32] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of World Wide Web Conference*, 2018.
[33] Weili Chen, Zibin Zheng, Edith C-H Ngai, Peilin Zheng, and Yuren Zhou. Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access*, 7:37575–37586, 2019.
[34] José R.C. Cruz. Keccak256 hash function, May. 2013.
[35] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.
[36] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *USENIX Security Symposium (USENIX Security)*, 2020.
[37] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019.
[38] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
[39] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang. Eosafe: Security analysis of eosio smart contracts. *USENIX Security Symposium*, 2021.

[40] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the ethereum smart contract ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 654–675, 2020.
[41] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.
[42] Eunjin Jung, Marion Le Tilly, Ashish Gehani, and Yunjie Ge. Data mining-based ethereum fraud detection. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 266–273. IEEE, 2019.
[43] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.
[44] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD, August 2018. USENIX Association.
[45] Xi Tong Lee, Arijit Khan, Sourav Sen Gupta, Yu Hann Ong, and Xuan Liu. Measurements, analyses, and insights on the entire ethereum blockchain network. In *Proceedings of The Web Conference 2020*, pages 155–166, 2020.
[46] Shlomi Linoy, Natalia Stakhanova, and Suprio Ray. De-anonymizing ethereum blockchain smart contracts through code attribution. *International Journal of Network Management*, page e2130, 2020.
[47] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *IEEE/ACM International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018.
[48] Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Eclone: Detect semantic clones in ethereum via symbolic transaction sketch. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 900–903, 2018.
[49] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
[50] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
[51] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of Annual Computer Security Applications Conference*, 2018.
[52] SEC. Definition of Ponzi scheme from SEC, Jul. 2019.
[53] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.
[54] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.
[55] Marie Vasek and Tyler Moore. There's no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams. In *International conference on financial cryptography and data security*, pages 44–61. Springer, 2015.
[56] Marie Vasek and Tyler Moore. Analyzing the ponzi scheme ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 101–112. Springer, 2018.
[57] Ian H Witten and Eibe Frank. Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, 31(1):76–77, 2002.
[58] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, 2007.
[59] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of USENIX Security Symposium*, pages 1371–1385, 2018.
[60] Anding Zhu, Peihua Fu, Qinghe Zhang, and Zhenyue Chen. Ponzi scheme diffusion in complex networks. *Physica A: Statistical Mechanics and its Applications*, 479:128–136, 2017.

Appendix A ALGORITHMS

A.1 Getting the Slot of Storage Variable

As discussed in § 4.2, each variable in storage is maintained in a part of the slot ($S[\sigma]$) which can be expressed in *Extract*$(o+\epsilon, \epsilon, S[\sigma])$. We present Algorithm 1 to analyze the syntax. Here are some details. At L4 and L5, Algorithm 1 gets ϵ and o from the parameters of extract-statement. At L6, Algorithm 1 gets the left-tree operand and generates $S[\sigma]$. $S[\sigma]$ is a Z3-expression loaded from *Z3 Array*. It takes σ as an index and can be expressed as a *Select* statement ($Select(S, \sigma)$). Therefore σ

can be fetched from the right-tree of *Select* statement (L9), and the final output of the algorithm is $\theta(\sigma, \epsilon, o)$.

Algorithm 1 Getting the Slot of Storage Variable

Input: symbolic expression: expr
Output: θ.

```
1:  if expr.decl ≠ extract then
2:      return
3:  else:
4:      //expr = Extract(ϵ + o, o, S[σ])
5:      o, ϵ ← expr.params // tuple (size, shift)
6:      expr ← get_left_tree(expr)
7:      if expr.decl = select then
8:          // expr = S[σ]
9:          σ ← get_right_tree(expr)
10:         return σ, ϵ, o
```

get_left_tree(expr): get the left trees of expr.
get_right_tree(expr): get the right trees of expr.

A.2 Storage Variables Recognition

Algorithm 2 parses σ and saves in a tuple *sigma'*. This algorithm is guided by the context-free syntax of storage variables shown in Figure 8(b). Before the analysis, we need to recover the hash computation caused by SHA3. We introduce two ways to exploit hash computations (like $hash := sha3(p)$) and get p. On the one hand, if *hash* is concrete, we present $get\_hash\_idx(hash)$ function to get p. This method mainly aims for DSA variables, where *hash* is the result of $sha3(base)$, and the *base* is a small number. By exploiting $\{\text{sha3}(i) \mid i \in \mathbb{N}\}$ [11], we can exploit p with little effort. On the other hand, as for the symbolic one, we recover p by querying H, which is a dictionary taking computation as a key and the parameter of sha3() as a value (see §5.2). Because Solidity will extend p to 512 bits with padding and then feed to sha3(), we need to remove the padding and recover the final result which can be got according to the syntax of $Extract$. Then we can parse σ with the accessing address syntax of storage variables shown in Figure 8(b). To be specific, if the input is a concrete number (L1), a state variable is identified. Otherwise, if the input contains a hash computation, we implement DSA and mapping patterns for the recognition. Specifically, if there are three items added altogether, we return the slot tuple from these items, where the base is the output of $get\_hash\_idx(items[0])$ at L7. Finally, we detect the mapping type if the number of operands is 2 (L13). The first operand of inputs is the $offset$, and the second is a symbolic computation which can be parsed by querying H (L17) or $get\_hash\_idx()$. The input of sha3() is a *Concat* statement whose operands are key and base (L19). Because Solidity will add padding for *key* and *base* to generate a 512 bits input for sha3(), we need to remove the padding and recover the final result (at L20-24).

Appendix B THE DETAILS OF ML BASELINES

There are two machine learning approaches used as baselines, which are OPCodeML based on frequency of operation code and TxML based on transaction features.

[11] i is smaller than 256 in our implementation.

Algorithm 2 Storage Variables Recognition

Input: the symbolic expression of slot: σ
Output: type of storage variable and the tuple of σ.

if $type(\sigma) = concrete$ **then**
  return $tag_1, (\sigma)$
else
  $items \leftarrow get\_trees(\sigma)$
  if $len(items) = 3$ **then**
    // DSA @ hash + width*key + offset
    $base \leftarrow get\_hash\_idx(items[0])$
    assert ($items[1].decl$ = bvmul)
    $width \leftarrow get\_left\_tree(items[1])$
    $key \leftarrow get\_right\_tree(items[1])$
    $offset \leftarrow item[2]$
    return $tag_2, (base, key, width, offset)$
  else:
    $offset, expr \leftarrow items$
    if $exp \in H$ **then**
      // mapping @ offset + sha3(key||base)
      $identitiy \leftarrow H[exp] \mid get\_hash\_idx(expr)$
      assert ($identitiy.decl$ = concat)
      $key, base \leftarrow identity[:256], identity[256:]$
      // removing padding
      if $key.decl$ = concat **then**
        $key \leftarrow get\_right\_tree(key)$
      if $base.decl$ = concat **then**
        $base \leftarrow get\_right\_tree(base)$
      return $tag_3, (base, key, offset)$

get_trees(expr): get the sub trees of input.
get_hash_idx(hash): get the concrete input of the keccak256.
computation according the hash value if possible.

As for the training settings, we follow the previous work and split the benchmark dataset (1395 contracts and 133 ponzitracts mentioned in §7.1) into 80% for training and 20% for testing. The proportion of ponzitracts in the partition datasets is 9.5% (133/1395), which keeps the same with the full dataset. After that, in OPCodeML, we randomise the dataset and convert the bytecode to a vector. We first use EVM (1.9.25) to obtain an opcode sequence for each bytecode. With *Tokenlizer* API of Keras, we allocate a unique ID for each opcode and convert the sequence to a row vector, where each item is the frequency of one opcode. At last, the dimension of the entire dataset is 1395 * 73. In TxML, we first collect the corresponding transactions from Etherscan and calculate the behavior features stated by [42] to construct a vector for each contract. However, several features in [42] (Gini_amt_in, Gini_amt_out, overlap_addr, num_addr_in, num_addr_out) are not taken into account, because most contracts have insufficient transactions.

After finishing feature engineering, we adopt n-fold cross-validation to find the best parameters of the model. Using the existing ML framework, we can train comprehensive models with little effort. In OPCodeML, we implement the model under the XgBoost [27] framework and train the

model with 5-fold cross-validation to find the best parameters (see Table 6). In TxML, the model is trained by WEKA [57] using the command line: *$weka.classifiers.trees.RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1 -depth 6*. The parameters are tuned with dozens of 10-fold cross-validation.

Table 6. The Training Parameters Used by OPCodeML

Parameters	**Value**	**Parameters**	**Value**	**Parameters**	**Value**
max_depth	200	nthread	-1	colsample_bytree	1
learning_rate	0.05	gamma	0	colsample_bylevel	1
n_estimators	200	missing	None	reg_alpha	0
silent	true	seed	2020	reg_lambda	10
objective	binary:logistic	min_child_weight	20	scale_pos_weight	110
booster	gbtree	max_delta_ste	1	base_score	0.5
n_jobs	1	subsample	0.9	random_state	2

At last, our benchmark results stated in §7.1 are close to the original paper. For OPCodeML[32], three XgBoost models with three different features combination are trained. The best model uses transactions and opcode frequency, which is cited by the survey paper [16]. We follow the author's paper and train OPCodeML with opcode frequency only, which is even better than the opcode-only model in [32] (see Table 2). As for TxML, the performance is affected by the dataset. The original paper [42] trains the model with a biased dataset without removing duplication. Moreover, as we stated in §7.1, we found roughly 15% of smart contracts in the benchmark have less than ten transactions, and they were only active for less than one day. These transactions are filtered by the original paper but are taken into account in TxML's performance. The shortage of real-world transactions causes TxML's poor performance, not to mention generalizability issues.

Appendix C ADVERSARIAL ATTACKS ON OPCODEML

Table 7. The Results of Adversarial Attacks on OPCodeML.

	Ponzi				Non-Ponzi			
	Cnt#	Success Rate	Avg.L_2	Avg.Time	Cnt#	Success Rate	Avg.L_2	Avg.Time
ZOO attack	0/23	0%	-	48.34	186/250	74.40%	0.010	17.96
Kantchelian's attack	23/23	100.00%	0.01	0.13	250/250	100.00%	0.008	0.15

[1] AvgL_2 is the average of distribution added in the feature space.

To further explore the limitations of OPCodeML, we only generate adversarial examples against the testing dataset. Furthermore, *we can calculate the related code perturbation and insert a large portion of non-sensitive operation codes to the samples* (without invoking real functionalities), in order to obfuscate the distribution of opcodes in ponzitracts.

We seek to know *whether adding or removing some position-insensitive codes will make this approach fail*. Here, we generate adversarial examples by changing the feature matrix of the testing dataset. Our experiment presents three metrics to evaluate the experiment, including Success Rate, Avg.L_2 and Avg.Time. If an adversarial contract is flipped to a wrong label, we say it is exploited. The success rate is the proportion of exploited cases in the whole test data. The L_2 metric refers to the maximum opcode frequency changed of one exploited case. Considering the whole test data, the average L_2 (donated as $AvgL_2$) measures the attack's performance. Generally speaking, the

smaller $AvgL_2$ is, the less code we need to inject. Namely, the Avg.Time measures the effectiveness of the attack. Table 7 shows the final results.

1) **Kantchelian's attack** [24] is a white-box attack, where the attacker can get the details of the model, such as the weight of subtrees of XGboost. This scenario's parameters include guard value as 0.05 and the number of digits to round as 20. The experiment shows all ponzitracts, and non-Ponzi cases are successfully exploited with the average time of 0.13s and 0.15s for each case, respectively. The cost of Kantchelian's attack is low, and the Avg.L_2 is 0.01 and 0.008 for Ponzi exploiting cases and non-ponzi exploiting cases, respectively.

2) **ZOO attack** [26] is a black-box attack, where the attacker can only get outputs of the model (probabilities here) by uploading samples. In this scenario, the attacker adds perturbation to increase the feature value and then queries the target model to verify the result. Its performance mainly depends on the query count (iteration) and the minimum steps of adding perturbation (binary step). For a comprehensive analysis, our evaluation performs 500 iterations and sets 200 for binary search steps. The black-box algorithm converges worse than the white-box one as it lacks details of the model. The ZOO Algorithm flags 186/251 cases for non-ponzi cases, with a 74.4% success rate and 0.01 Avg.L_2. Unfortunately, none of the Ponzi cases are exploited because of the settings of black-box attack.

This preliminary exploration suggests that *due to a lack of semantic information, the opcodes based machine learning approach is likely to fail in ambiguous cases, which may introduce both false positives and false negatives.* Position-insensitive codes can change the frequency of specific opcodes without affecting the control flow of the smart contract. In contrast to OPCodeML, SADPonzi is naturally immune to such adversarial examples.

Appendix D CRAFTING THE BENCHMARK DATASET

The third column of Table 8 shows why we remove the 51 smart contracts from the original dataset. Apart from 25 duplications, there are 26 cases violating the definition of Ponzi Scheme smart contracts as introduced in Sec 2.2. Some of them never reward the participants but only reward their creators. For example, FirePonzi[12] was labeled as a chain scheme contract. In this contract, a cursor called `payoutCurost_Id` selects a participant to reward, however, the contract never increases this cursor but updates another variable called `payoutCurost_Id_`, which contains an additional underline. In other words, this contract is a financial scam that cannot distribute a bonus to the participants with a pyramid strategy, which is not a Ponzi scheme. Notably, if the smart contract is opaque (close-source), we analyze the data dependency of the stack parameters used in `CALL` from the decompiled code generated by *contract-library*[13]. This information can tell us whether a participant can receive money when a new investment come. Taking 0x7011f3edc7fa43c81440f9f43a6458174113b162 as an example, we remove it because all of its `CALL` can only be used to return extra investment (at most 1 Ether) to players or to collect fee for the creator.

Table 8. The removed 51 smart contracts from the original benchmark.

address	name	annotation
0x9758da9b	Etheramid1	duplicate, old version of 0xfeeb8a96
0x29430848	ShinySquirrels1	duplicate with 0x870fe80e
0xf835b307	Newponzi	duplicate with 0x7894ccf2
0x1368e088	ResetPonzi2	duplicate with 0xe861ad00
0x51170b18	CrystalDoubler	fake chain, never update $nr

[12] 0x062524205ca7ecf27f4a851edec93c7ad72f427b

[13] https://contract-library.com/

Table 8. The removed 51 smart contracts from the original benchmark.

address	name	annotation
0x687a2414	DepositHolder2	Non-Ponzi, the latter you participate, the more you won
0x49c3019b	SmartRevshare1	the test contract for ba69e7c9
0x06252420	FirePonzi	fake chain, never update $payoutCursor_Id
0x5a437d94	NotAnotherPonzi	duplicate with 0x7894ccf2
0xc357a046	EthFactory	fake chain, never update $index
0xcd6608b1	DepositHolder1	Non-Ponzi, the latter you participate, the more you won
0xa0f9fb21	DividendProfit1	fake handover, player cannot update $deployer
0x24ec083b	Tomeka	duplicate with 0xa9e4e3b1
0x4398a4a1	Timegame2	duplicate with 0xa864694c
0x7fcc7ed2	Timegame3	duplicate with 0xa864694c
0xf1aa63ad	EthVenturePlugin	fake handover, player cannot update $owner
0x99925cc9	BankWithInterest	Non-Ponzi, no pyramid scheme
0xc99b3615	BestBankwithInterest	duplicate with 0x245233bc
0x245233bc	BetterBankWithInterest	duplicate with 0xc99b3615
0xba3048b1	Copypaste	duplicate with 0x7894ccf2
0x0444f06a	ResetPonzi7	duplicate with 0x772cba2f
0x9d31ff89	AFreeEtherADay	non-Ponzi, only the creator can receive benefit
0x6adf9e66	FreeEtherADayFundsReturn	fake handover, never update $poorguy
0x9f1d916a	NanoPyramid1	duplicate with 0xe19e5f10
0x870fe80e	ShinySquirrels2	duplicate with 0x29430848
0x2464d1d9	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0xf767fca8	LuckyDoubler	Non-Ponzi, random lottery
0xda922e47	SwarmRedistribution1	Non-Ponzi no pyramid scheme
0x6846b938	SwarmRedistribution2	Non-Ponzi no pyramid scheme
0xf77ac34c	SwarmRedistribution3	Non-Ponzi no pyramid scheme
0x8e66ffe6	SwarmRedistribution4	Non-Ponzi no pyramid scheme
0xf184279e	SwarmRedistribution5	Non-Ponzi, never reward players
0x0f170120	SwarmRedistribution6	Non-Ponzi, never reward players
0x79c039d0	Unnamed	duplicate with 0xa9e4e3b1
0x827ce5d8	Unnamed	duplicate with 0xa39fcb48
0xf52ecc52	Unnamed	opaque Non-Ponzi, player cannot update $_topWizard
0xadd6f226	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0xd2dc2625	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0x787e3cef	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0xc2b9e482	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0x0d591957	Unnamed	opaque Non-Ponzi, player cannot receive bonus
0x827ce5d8	Unnamed	duplicate with 0xa39fcb48
0x6b62497f	Unnamed	duplicate with 0xa9e4e3b1
0x79c039d0	Unnamed	duplicate with 0xa9e4e3b1
0x15d03b33	Unnamed	duplicate with 0xc1824278
0x8153510d	Unnamed	duplicate with 0xc1824278
0xac2bb751	Unnamed	duplicate with 0xca7c390f
0xdf05fb3e	Unnamed	duplicate with 0xca7c390f
0xc07ec6b6	Unnamed	duplicate with 0xe8271920
0x3ae8b3ae	Unnamed	duplicate with 0xfd2487cc
0x7011f3ed	Unnamed	opaque Non-Ponzi, player cannot receive bonus

Appendix E EVASION EXAMPLE

Currently, SADPonzi cannot detect the combination of multi-schemes. We create a ponzitract on-the-fly with the features of both a tree-shape scheme and withdraw scheme (see Figure 10). The participant can call `invest(address parent)` to update the user information tree and call `withdraw()` to receive the bonus while allocating 20% of investment to her inviter. To identify such a scheme, we can first utilize the tree-scheme detector to get I and then parse R with the withdraw-scheme detector. We believe SADPonzi can evolve without much effort, so we can find more Ponzi schemes in the future.

```
1.  contract test {
2.    mapping (address => address) tree;
3.    mapping (address => uint256) balanceOf;
4.
5.    function invest(address inviter) public payable{
6.      tree[msg.sender] = inviter;
7.      balanceOf[inviter] = msg.value;
8.    }
9.    function withdraw() public payable{
10.     balanceOf[msg.sender] = 0;
11.     uint fee = balanceOf[msg.sender]*0.2;
12.     balanceOf[tree[msg.sender]] += fee;
13.     msg.sender.transfer(balanceOf[msg.sender]*0.8);
14.   }
15. }
```

Fig. 10. A new Ponzi scheme that can evade the detection of SADPonzi.

Received March 2021; revised April 2021; accepted April 2021