Early Detection of Smart Ponzi Scheme Contracts Based on Behavior Forest Similarity

Weisong Sun, Guangyao Xu
Nanjing University
Nanjing, China
{weisongsun, xgy}@smail.nju.edu.cn

Zijiang Yang
Western Michigan University
Kalamazoo, Michigan, United States
zijiang.yang@wmich.edu

Zhenyu Chen*
Shenzhen Research Institute of Nanjing University
Shenzhen, China
zychen@nju.edu.cn

***Abstract*—Smart contracts empowered by blockchains often manage digital assets in a distributed and decentralized environment. People believe in smart contracts based on these new technologies. Unfortunately, malicious smart contacts, such as smart Ponzi scheme contracts (ponzitracts, for short), pose risk. Existing techniques detect ponzitracts by analyzing the code as well as a large amount of transaction data after time-consuming deployment. However, a conclusion based on transaction data can only be gotten after the damage has been caused. This paper proposes *PonziDetector*, a ponzitract detection technique that does not rely on transaction data. Behavior forest is introduced into *PonziDetector* to capture dynamic behaviors of smart contracts during interacting with them, which makes it possible to early detect ponzitracts. The empirical study demonstrates that *PonziDetector*, without transaction data, can improve the precision and the recall of the state-of-the-art to 94.6% and 93.0% respectively. This means that *PonziDetector* can avoid potential losses by early detecting ponzitracts.**

***Index Terms*—Smart contract, Smart Ponzi scheme, Behavior tree, Behavior forest, Ponzitract detection**

I. Introduction

The advent of Bitcoin makes it possible for anonymous participants to exchange value securely without the intermediation of trusted authorities [1]. The blockchain technology behind in Bitcoin enables decentralized financial activities and has become a hotspot in both academia and industry [2], [3]. After Bitcoin, Ethereum [4] ushered in the second-generation blockchain [5] that supports the execution of smart contracts. To make it possible, Ethereum implements Ethereum Virtual Machine (EVM) which provides a smart contract execution environment and defines a contract-oriented high-level scripting language called Solidity [6] to unambiguously describe smart contracts.

Smart contracts are self-executing programs with the terms of the agreement among stakeholders. Since a smart contract enables reliable transactions between mutually distrusting participants [7], it has become widely used for a broad spectrum of financial applications [8]. Unfortunately, malicious financial applications emerged to exploit technical innovations and people's trust in smart contracts. Among which is blockchain implementation of the notorious Ponzi scheme [9] — a classic scam originated in the offline world at least 150 years ago. A Ponzi scheme is a fraudulent investing and disguised as promising a high rate of returns to investors. The Ponzi scheme generates returns for early investors by acquiring new investors. This is similar to a pyramid scheme in that both are based on using new investors' funds to pay the earlier backers. Ponzi schemes eventually bottom out when the flood of new investors dries up and there isn't enough money to go around. At that point, the schemes unravel.

A smart contract implementation of the Ponzi scheme is called a ponzitract (smart Ponzi scheme contract) [9], [10]. Such implementation is very attractive because it inherits the inherent properties of smart contracts and blockchains: decentralized, anonymous and immutable. These properties are the cornerstones of many legit financial applications. But at the same time, they make smart Ponzi schemes more dangerous than traditional Ponzi schemes:

- The initiator of ponzitracts can stay anonymous, which apparently provides protection against incrimination of the people who started the scheme.
- The deployed ponzitracts are hard to modify and stop. Once an investor participates there is no easy way out. In order to avoid loss, the investor has to continually invite new investors. This exactly matches the malicious intention of the creators.
- Decentralization gives investors the illusion that ponzitracts are reliable and credible.

To the best of our knowledge, the first feasible detection technique of ponzitracts is proposed by Weili et al. [7]. They present a classification model trained with two kinds of features (i.e., "Account" and "Opcode" features) extracted from the transaction history data and operation codes (opcodes) of smart contracts to detect ponzitracts. "Account" features, such as *Known rate* (the proportion of receivers who have invested before payment) and *Balance* (the balance of the smart contract), are collected through analyzing the transaction history data. So their technique relies partly on transaction data which however are not available in the early days of ponzitract deployment. It means detection based on transaction data occurs only after the damage has been done. In the absence of transaction data, the existing approach gives a much less satisfactory result.

In this paper, we propose a novel ponzitracts detection technique according to our speculation that an accurate model

[*] Zhenyu Chen is the corresponding author.

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00047

of the opcodes will reveal the behavior of a smart contract and thus identify whether it is a ponzitract. Based on the hypothesis we have implemented *PonziDetector* that does not rely on transaction data. Behavior trees built on the Ethereum Testnet are introduced in *PonziDetector* to fully characterize the dynamic behaviors of smart contracts. Specifically, at first, *PonziDetector* employs the traditional testing techniques to mining the complete behaviors (i.e., opcodes) of smart contracts during interacting with them by using tests. We regard the opcodes as the behaviors of the smart contract, explained in detail in Section II-B. Secondly, these behaviors are leveraged to build behavior graphs, and we further use the pruning technology to eliminate the cyclic edges in that to produce comparable behavior trees. Then, the adapted All Path Tree Edit Distance (AP-TED) [11] algorithm is employed to calculate the similarities among weighted behavior trees. A behavior forest is constructed on all behavior trees in a smart contract. Behavior forest similarity is introduced to analyze the smart contact under detection. Finally, *PonziDetector* combines with the threshold analysis to detect ponzitracts. Note that the entire process does not rely on transaction data.

We utilize a publicly available smart contract dataset [7] to evaluate our approach. The evaluation results show that the behaviors mined by *PonziDetector* are more completely, thereby can effectively improve the recall (increased by nearly 15%) of early detection of ponzitracts while maintaining precision is no less than that of a previous study [7].

In summary, we make the following contributions:

- *PonziDetector* detects ponzitracts from the perspective of the behavior described by the smart contracts themselves. It does not rely on anything else except for the smart contract itself and can detect the ponzitract that is just deployed.
- The experiments are conducted on a public dataset [1] and the experimental results show that *PonziDetector* can apparently improve the performance of ponzitracts detection, i.e., high precision and recall.
- We released all our experimental results and a behavior forest database of known ponzitracts at the website [2] for other researchers to conduct related research.

The rest of this paper is organized as follows. Section II describes two basic concepts. The details of our methodology are given in Section III. We introduce the experiment in detail in Section IV. Threats to validity and related work are given in Section V and Section VI respectively. The conclusion is in Section VII.

II. Basic concepts

In this section, we introduce two basic concepts on smart Ponzi scheme and program behavior.

A. Smart Ponzi Scheme

For ease of comprehension, we use a practical example to illustrate the smart Ponzi scheme. As shown in Fig.1, "StackyGame" [3] is a smart contract which implements the Ponzi scheme. It declares a participant array used to records all investors' information (line s_6). The function $enter()$ is a core carrier where the investment and Ponzi scheme occur (lines $s_{16} - s_{40}$). The $msg.sender$ and $msg.value$ are transaction properties [4] used to record the address and investment amount of the investor (lines $s_{23} - s_{24}$). When the investment amount of the new investor meets certain conditions (line s_{17}), "StackyGame" firstly collects some fees (lines $s_{25} - s_{30}$). Then, to encourage former investors to invite more new investors, it also automatically send high rewards to former investors (lines $s_{35} - s_{36}$) when the balance of the contract account (i.e., $balance$ declared at line s_9) is sufficient (line s_{31}). The number of rewards also depends on the former's investment amount (lines $s_{32} - s_{34}$), and the higher the investment, the higher the reward. Finally, the function $collectFees()$ which can only be accessed by the owner of "StackyGame" makes it easier to reap without spending a cent (lines $s_{41} - s_{45}$). Note that the former investors have income only based on new investors falling into the scam and all of them can not withdraw their investments by themselves. Obviously, it is a classic smart Ponzi scheme.

B. Program Behavior

There are many studies on program behavior related to program executions and operations [12]–[18]. A smart contract is a special program running on EVM. With the guide of the work [17], we employ opcodes to capture behaviors of the smart contract. In Ethereum, the opcode is the mnemonic form of EVM bytecode, and the type of operations corresponding to opcodes is fixed [4]. For example, the opcode corresponding to the bytecode '0x01' is 'ADD', i.e., representing addition operation. Therefore, the opcode is more readable while containing all the information contained in the bytecode.

For easy to understand, we use a simple smart contract with a loop instruction (i.e., *while* instruction), as shown in Fig. 2, to illustrate the smart contract behavior. After deploying the smart contract "SimpleSC" into Ethereum Testnet, we can interact with it by invoking the function "*rewardPay()*". For example, a testing input $\langle amount = 3, balance = 4\rangle$ can be used to interact with it, and then we can collect opcodes shown in Fig. 3-(a). Further, as shown in Fig. 3-(b), directed edges are used to connect these opcodes to represent continuous behaviors of the smart contract. To improve the efficiency of analysis, the cyclic edge would be pruned to produce a behavior tree, as shown in Fig. 3-(c). Formally, we define a behavior tree as follows.

DEFINITION 1 (Behavior Tree $\mathcal{BT}$): A behavior tree of a function f in a smart contract is a labeled tree defined by

$$\mathcal{BT}(f) =< v_0, V, E > \quad (1)$$

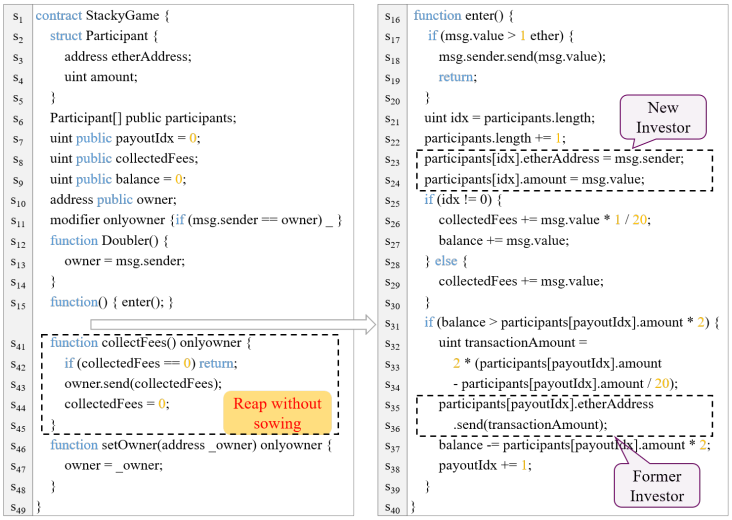
- v_0 is the entrance of f, as the root vertex of the tree;

[1] ibase.site/scamedb

[2] github.com/wssun/PonziDetector

[3] etherscan.io/address/0x8f13a1d43408b6434dd10e161361386f3952d665

[4] The msg variable is a special global variable that contains properties (e.g., $msg.sender$) which allow access to the blockchain.

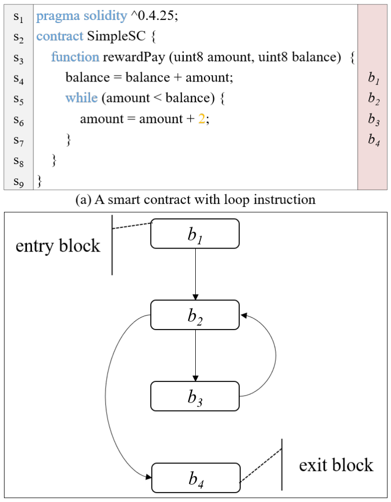


```
s1  contract StackyGame {
s2      struct Participant {
s3          address etherAddress;
s4          uint amount;
s5      }
s6      Participant[] public participants;
s7      uint public payoutIdx = 0;
s8      uint public collectedFees;
s9      uint public balance = 0;
s10     address public owner;
s11     modifier onlyowner {if (msg.sender == owner) _ }
s12     function Doubler() {
s13         owner = msg.sender;
s14     }
s15     function() { enter(); }

s41     function collectFees() onlyowner {
s42         if (collectedFees == 0) return;
s43         owner.send(collectedFees);
s44         collectedFees = 0;
s45     }
s46     function setOwner(address _owner) onlyowner {
s47         owner = _owner;
s48     }
s49 }
```

```
s16 function enter() {
s17      if (msg.value > 1 ether) {
s18         msg.sender.send(msg.value);
s19         return;
s20     }
s21     uint idx = participants.length;
s22     participants.length += 1;
s23     participants[idx].etherAddress = msg.sender;
s24     participants[idx].amount = msg.value;
s25     if (idx != 0) {
s26         collectedFees += msg.value * 1 / 20;
s27         balance += msg.value;
s28     } else {
s29         collectedFees += msg.value;
s30     }
s31     if (balance > participants[payoutIdx].amount * 2) {
s32         uint transactionAmount =
s33             2 * (participants[payoutIdx].amount
s34             - participants[payoutIdx].amount / 20);
s35         participants[payoutIdx].etherAddress
s36             .send(transactionAmount);
s37         balance -= participants[payoutIdx].amount * 2;
s38         payoutIdx += 1;
s39     }
s40 }
```

Fig. 1: Example of smart Ponzi scheme

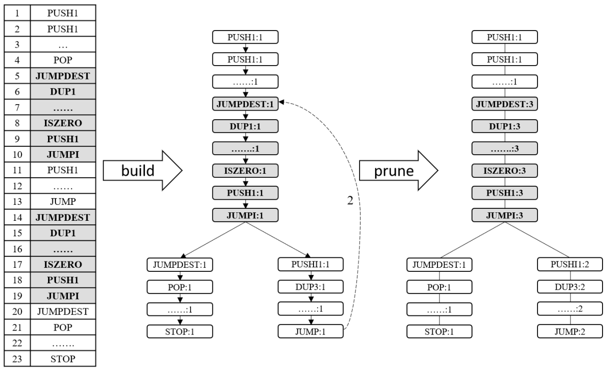


```
s1  pragma solidity ^0.4.25;
s2  contract SimpleSC {
s3      function rewardPay (uint8 amount, uint8 balance)  {
s4          balance = balance + amount;                        b1
s5          while (amount < balance) {                         b2
s6              amount = amount + 2;                           b3
s7          }                                                  b4
s8      }
s9  }
```

(a) A smart contract with loop instruction

(b) A corresponding control flow graph

Fig. 2: A simple smart contract



1	PUSH1
2	PUSH1
3	…
4	POP
5	**JUMPDEST**
6	**DUP1**
7	**……**
8	**ISZERO**
9	**PUSH1**
10	**JUMPI**
11	PUSH1
12	……
13	JUMP
14	**JUMPDEST**
15	**DUP1**
16	**……**
17	**ISZERO**
18	**PUSH1**
19	**JUMPI**
20	JUMPDEST
21	POP
22	…….
23	STOP

(a) Opcodes (b) Behavior Graph (c) Behavior Tree

Fig. 3: Building a behavior tree

- V is a non-empty finite set of vertices, in which each vertex has two labels: opcode and execution times.
- $E \subseteq V \times V$ is a set of unlabeled undirected edges.

All directed edges of smart contract execution is simplified as undirected edges in a tree. For each function, a behavior tree is constructed. In practice, we only construct the behavior tree for the function that is explicitly claimed in smart contract. In Fig. 2-(a), four basic blocks 'b_1–b_4' are remarked: b_1 at line s_4, b_2 at line s_5, b_3 at line s_6, and b_4 at line s_7. In particular, b_1 is the "entry block", b_4 the "exit block". The Fig. 2-(b) shows a control flow graph (CFG) [19] that consists of these four blocks. Fig. 2-(b) and Fig. 3-(c) are compared to illustrate the difference between two ways. $\mathcal{BT}$ not only contains the syntax (logic structure) the program that CFG mainly focuses on but also takes the semantics of the smart contract into account. The state-of-art [7] experimental results show that opcodes can be more critical and valuable for detecting ponzitracts. These inspires us to introduce $\mathcal{BT}$ to detect ponzitracts.

III. Methodology

Our methodology *PonziDetector*, as shown in Fig. 4, has the following parts: generating simulation data for transaction properties, generating tests by fuzzing, constructing behavior trees using tests, calculating behavior forest similarity and detecting ponzitracts using behavior forest similarity. The solid blue line (labeled '3' in Fig. 4) indicates the deployment of smart contracts, and the other solid black lines represent the flow of various data.

A. Data Generation

Given a smart contract X, before interacting with it, we firstly need to prepare data for transaction properties. The property $msg.sender$ is an ***address*** type and refers to the sender of the message (current caller). In our Ethereum Testnet, there are many simulated account addresses that can be used as external callers of X, i.e., senders of the message.

For the property $msg.value$, it is a ***uint*** type and refers to the number of wei sent with the message. Ether units consist of wei, $finney$, $szabo$ and $ether$, among which wei is the minimum unit [6]. To dig out the complete smart contract behavior as much as possible, it is necessary to generate reasonable data for $msg.value$. We employ the seeding strategy [20] to generate candidate data. Specifically, we firstly extract numeric values along with ether units from code statements related to $msg.value$. For example, the numeric *1 ether* would be extracted from the statement s_{17} in Fig. 1. We consider these seeds as boundaries. Further, with the guidance of mutation operation from genetic algorithms [20], for each boundary, we generate two new values (known as offsprings) that less and larger than the boundary respectively through two simplest mutation operators (i.e., $+$ and $-$). For example, given the boundary *1 ether*, we firstly transfer it to minimum unit ($1\ ether = 1 * 10^{18}\ wei$); then automatically generate two offsprings, e.g., $10^{18} \pm \boldsymbol{\Delta}$. In practice, $\boldsymbol{\Delta}$ usually set as the minimum disturbance. In our case, the input space of

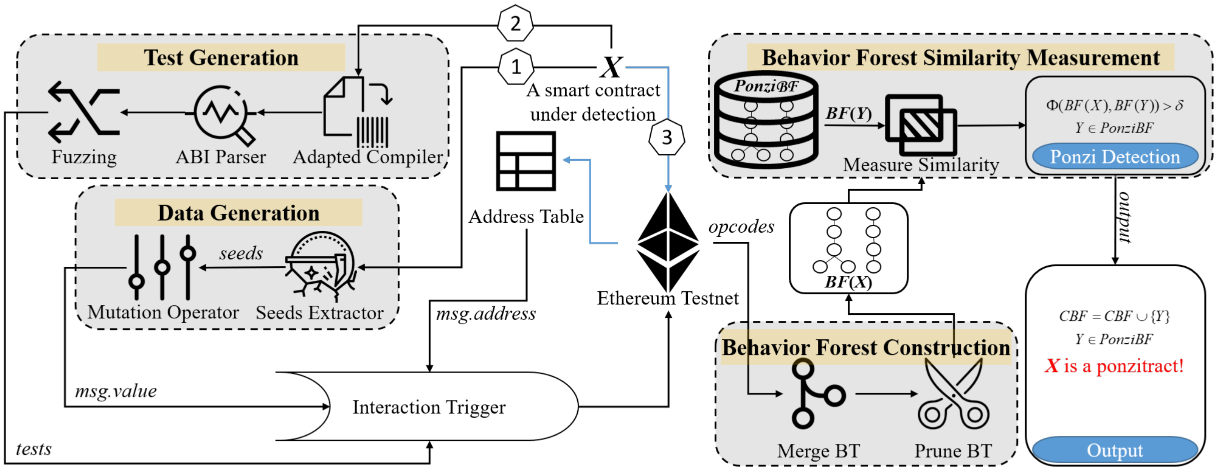


Fig. 4: Framework of *PonziDetector*

$msg.value$ is ***uint*** thereby *1* is the minimum disturbance. All seeds and offsprings are saved to a constant pool for subsequent interaction with X.

B. Test Generation

Given a smart contract X, the Adapted Compiler module integrating multiple versions of the *solc* compiler is responsible for compiling it. Some smart contracts specified the version number of the compiler in the first line of code by using the instruction "*pragma solidity* ***", and "***" will be replaced by the specific version number of *solc* compiler, such as the first line s_1 in Fig. 2. The Adapted Compiler automated scans the X's source code statically to find the best matching version of the *solc* compiler. If the target compiler version is not found, the Adapted Compiler will try to compile it using multiple versions of the compiler, from high to low versions until X is compiled successfully.

The ABI Parser module analyzes and extracts all functions information from the compilation results of X. For each function, the name and parameters of this function are extracted. This module employs the fuzzing [21] technique and aims at generating one set of candidate tests for each function that requires inputs. Specifically, the test generation module equipped with the tool ContractFuzzer [22] receipts types of parameters as inputs and outputs some specific values against specific types. ContractFuzzer [22] takes different strategies to generate inputs for fixed-size inputs (e.g., ***uint***) and non-fixed-sized inputs (e.g., ***string***). For example, in Fig. 2, the function "*rewardPay()*" with two ***uint*** parameters, our generator may generate tests like $\langle amount = 3, balance = 4\rangle$ for it. Considering the expensive runtime cost of the execution of test cases on smart contracts, our test generator creates 10 test cases for each input. Finally, all tests are saved to test set T, so that we can leverage them to interact with the more smart contracts in the next step.

C. Behavior Forest Construction

DEFINITION 2 (Behavior Forest $\mathcal{BF}$): A behavior forest of a smart contract $X = \{f_1, f_2, \cdots, f_m\}$ is defined by

$$\mathcal{BF}(X) = \cup_{i=1}^{m} \mathcal{BT}(f_i) \tag{2}$$

Algorithm 1 describes the construction process of behavior forest for a given smart contact. The inputs are a smart contract X with a set of functions $\{f_i\}$, a test set $T = \{t_j\}$ generated by fuzzing and a set of simulated account address $A = \{msg.address_k\}$ from Ethereum Testnet. The output is a behavior forest $\mathcal{BF}(X)$ with respect to T.

Algorithm 1 Construction of $\mathcal{BF}$

```
Input: $X = \{f_i\}$, $T = \{t_j\}$, $A = \{msg.address_k\}$;
Output: $BT$;
 1: $Set\ \mathcal{BF} = \emptyset$;
 2: $Set\ V = simulateMV(X)$;
 3: $var\ comResult = compile(X)$;
 4: $var\ xAddr = deploy(comResult)$;
 5: $var\ funcList = ABIParser(comResult)$;
 6: for each function $f_i \in X$ do
 7:     $var\ \mathcal{BT}(f_i) = NULL$;
 8:     for each $msg.value$ $v_m \in V$ do
 9:         for each test $t_j \in T$ do
10:             $op = interact(xAddr, msg.address_k, v_m, f_i, t_j)$;
11:             $var\ \mathcal{BT} = buildBT(op)$;
12:             $\mathcal{BT}(f_i) = mergeAndPrune(\mathcal{BT}(f_i), \mathcal{BT})$;
13:         end for
14:         $\mathcal{BF}(X) = \mathcal{BF}(X) \cup \{\mathcal{BT}(X, f_i)\}$;
15:     end for
16: end for
17: output $\mathcal{BF}(X)$;
```

For a smart contract X, it is firstly statically analyzed to produce simulation data for the transaction property $msg.value$ (line 2). Then, it is compiled and deployed by invoking the interfaces of $compile()$ and $deploy()$ respectively (lines

3-4). Then, the function information is extracted from the compilation results and the opcodes are produced for each function f_i by invoking the interface $interact()$ (line 10). In essence, an interaction is a transaction. The elements required to complete a transaction include: address of X $xAddr$, address of the transaction initiator $msg.address_k$, transaction amount of $msg.value$ v_m, the function for executing the transaction f_i and its parameters t_j. The interface $buildBT()$ is used to generate a partial behavior tree (line 11). These partial behavior trees are merged and pruned to produce a behavior tree for each function (line 12). The merging operation is responsible for merging partial behavior trees generated under different v_m and t_j into a complete behavior tree or behavior graph. (as shown in Fig. 3-(b)). The pruning operation is responsible for eliminating the cyclic edge so that producing the easy-to-analyze behavior tree (as shown in Fig. 3-(c)).Finally, for each X, a behavior forest $\mathcal{BF}(X)$ is the output (line 17).

Behavior Tree Filtering: Obviously, some functions in a ponzitract present normal program behavior. Hence, we need to filter these normal behavior trees in a ponzitract. Given a ponzitract Y^P and a non-Ponzi smart contract Y^N, Algorithm 1 is used to generate two sets of behavior trees, $\mathcal{BF}(Y^P)$ and $\mathcal{BF}(Y^N)$, respectively. In order to improve the performance of ponzitract detection, $\mathcal{BF}(Y^N)$ is used to filter $\mathcal{BF}(Y^P)$. That is, we use $\mathcal{BF}(Y^P) - \mathcal{BF}(Y^N)$ for ponzitract detection in practice.

D. Behavior Forest Similarity Measurement

In order to introduce behavior forest similarity, we first define behavior tree similarity based on the tree edit distance [23], i.e., $\mathcal{D}(\mathcal{BT}(f), \mathcal{BT}(f'))$. The edit distance is the minimal-cost sequence of vertex edit operations that transforms one tree into another via insertion, deletion and relabeling operations. An efficient method AP-TED [11] is used for tree edit distance calculation in this paper. Please note that each vertex in a behavior tree has two labels: specific opcode and its execution times. For example, in Fig. 3-(c), there is a vertex with the label 'DUP1:3' where 3 is execution times of the opcode 'DUP1'. That is one label is counted into 0.5 for relabeling in our methodology.

Let $|\mathcal{BT}(f)|$ be the number of vertices in $\mathcal{BT}(f)$. It is not difficult to see that $\mathcal{D}(\mathcal{BT}(f), \mathcal{BT}(f'))$ is not larger than $\max\{|\mathcal{BT}(f)|, |\mathcal{BT}(f')|\}$. Intuitively, two trees are more similar if the tree edit distance is lower. In this paper, we use $\phi(\mathcal{BT}(f), \mathcal{BT}(f'))$ to represent the behavior tree similarity and $\phi \in [0, 1]$ is calculated by the following equation:

$$\phi(\mathcal{BT}(f), \mathcal{BT}(f')) = 1 - \frac{\mathcal{D}(\mathcal{BT}(f), \mathcal{BT}(f'))}{\max\{|\mathcal{BT}(f)|, |\mathcal{BT}(f')|\}} \quad (3)$$

Given two smart contracts $X = \{f_1, f_2, \cdots, f_m\}$ and $Y = \{f'_1, f'_2, \cdots, f'_n\}$, the behavior forests can be constructed from Algorithm 1 with filtering and denoted by $\mathcal{BF}(X)$= $\{\mathcal{BT}(f_1)$, $\mathcal{BT}(f_2)$, $\cdots$, $\mathcal{BT}(f_m)\}$, $\mathcal{BF}(Y)$= $\{\mathcal{BT}(f'_1)$, $\mathcal{BT}(f'_2)$, $\cdots$, $\mathcal{BT}(f'_n)\}$, respectively. The similarity of two sets is normally defined by the maximum similarity between two elements in the two sets. In order to balance each behavior tree $\mathcal{BT}(f_i)$ in a forest, a weight $\omega_i = \frac{|\mathcal{BT}(f_i)|}{\sum_{k=1}^{m}|\mathcal{BT}(f_k)|}$ is introduced. Formally, the behavior forest similarity is defined as follows.

DEFINITION 3 (Behavior Forest Similarity): Given two behavior forests $\mathcal{BF}(X)$ and $\mathcal{BF}(Y)$, the behavior forest similarity Φ is defined as,

$$\Phi(\mathcal{BF}(X), \mathcal{BF}(Y)) = \sum_{i=1}^{m} \omega_i \times \max_{j \in [1,n]} \{\phi(\mathcal{BT}(f_i), \mathcal{BT}(f'_j)\} \quad (4)$$

It is not difficult to see that $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y)) \in [0, 1]$ for any X and Y, because each $\phi \in [0, 1]$ and the result is the average. Please note that $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y))$ is asymmetrical, i.e, $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y)) \neq \Phi(\mathcal{BF}(Y), \mathcal{BF}(X))$ in some cases. In our methodology, X is a smart contract under detection and Y is a known ponzitract. The asymmetric can work well on the one-way comparison in ponzitract detection.

Ponzitract Detection: Given a smart contract X under detection, Algorithm 1 is used to construct a behavior forest $\mathcal{BF}(X)$. Please note that a large number of known ponzitracts and non-Ponzi smart contracts have been collected. All the behavior forests will be constructed in advance to generate a $Ponzi\mathcal{BF}$ database. For each ponzitract $Y \in Ponzi\mathcal{BF}$, we calculate the behavior forest similarity between X and Y, i.e., $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y))$. If $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y)) > \delta$, X will be classified as a ponzitract, in which δ is a threshold, which will be discussed in Section 4. All Y with $\Phi(\mathcal{BF}(X), \mathcal{BF}(Y)) > \delta$ will also be collected for manually inspecting X in further.

IV. Evaluation

In this section, we present the details of our experiments as well as the results analysis. To evaluate the performance of *PonziDetector*, we applied it to a public smart contracts dataset and investigated the following research questions:

- **RQ1**: How effective is *PonziDetector* for ponzitract detection?
- **RQ2**: How efficient is *PonziDetector* for ponzitract detection?
- **RQ3**: What prompts *PonziDetector* to employ the behavior forest similarity (behavioral similarity for short) instead of common textual similarity to detect ponzitract?

A. Experimental Setup

Experimental Environment. All the experiments were performed on 64-bits Ubuntu 16.04.4 desktop with Intel i5-3470 CPU and 8GB of memory. We setup a private blockchain as the Ethereum Testnet with the geth client version 1.8.21. We used web3.js in our node.js program to interact with the geth client and used JSON-RPC to gain the results of runtime opcodes. In order to deploy smart contracts with different versions, we built a solc.js compiler with binaries of soljson from v0.1.1-nightly to v0.6.0-nightly.

Experimental Design. To answer our three research questions, we designed the following two experiments:

The first experiment is to evaluate the effectiveness and efficiency of *PonziDetector* by comparing the precision and recall of ponzitract detection achieved by *PonziDetector* with a previous work [7] under the same dataset of smart contracts.

The second experiment is to compare the textual and behavioral similarity among smart contracts, aiming at proving that, in the application scenario of ponzitract detection, it is more accurate to use the behavioral similarity rather than the textual similarity to represent the similarity among smart contracts. In this comparison, we used Difflib [24] to compare two smart contracts' codes, then calculated the textual similarity.

B. Experimental Subjects and Evaluation Metrics

In this paper, we utilized a public dataset of smart contracts released by [7] as our experimental subjects, and downloaded 3,972 smart contracts with verified source code from *ibase.site*, of which 132 were labeled as ponzitracts. 129 ponzitracts were deployed successfully. Three ponzitracts deployed failed for two reasons: (1) two of them failed due to a wrong compilation of the source code; (2) the remaining one prompted "*This contract does not implement all functions and thus cannot be created.*". Then, we tried to interact with 129 ponzitracts, one of which [5] can't be interacted with because its code specifies two specific addresses that do not exist in our Ethereum Testnet. In this study, we temporarily ignore this ponzitract, and will explore this issue further in future research.

We randomly split 128 ponzitracts into two groups, one group containing 96 ponzitracts is used to build a $Ponzi\mathcal{BF}$ database, and the remaining 32 ponzitracts is used to verify the performance of *PonziDetector* (corresponding to the SC^P column in Table I). Furthermore, we randomly selected 192 out from 3,840 non-Ponzi smart contracts, and all of them can be successfully deployed and interacted with. 96 of 192 non-Ponzi smart contracts were used to filter behavior trees (i.e., normal program behaviors) that are also appearing in the behavior forest database of non-Ponzi smart contracts from $Ponzi\mathcal{BF}$ (corresponding to the G1's SC^N column in Table I). The remaining 96 non-Ponzi smart contracts were randomly divided into three groups. All groups were combined with 32 ponzitracts respectively to get three test datasets (corresponding to G2, G3, G4 in Table I). The ratio of ponzitracts to non-Ponzi smart contracts in each group is 1:1. It should be noted that we used the same 32 ponzitracts in G2, G3, and G4, that is, among which ponzitracts are identical, but the other 32 non-Ponzi smart contracts are entirely different. In summary, four groups of datasets as shown in Table I are used in our experiments.

To evaluate the performances of *PonziDetector*, three measure metrics: precision ($P = TP/(TP + FP)$), recall ($R = TP/(TP + FN)$), and F_1-measure ($F_1 = 2 * P * R/(P + R)$) are employed. The TP and FP values are the number of ponzitracts and non-Ponzi smart contracts among the detected results respectively; and the TN and FN values are the number of non-Ponzi smart contracts and ponzitracts among the rest results respectively. The precision corresponds to the proportion of ponzitracts in all smart contracts detected by the threshold, which indicates how useful the detected results are. The recall corresponds to the proportion of ponzitracts detected by the threshold in all ponzitracts labeled manually, which indicates how complete the detected results are. Since both precision and recall are important in ponzitract detection, we further use the F_1-measure to evaluate *PonziDetector*. The F_1-measure is the harmonic average of the precision and recall, where an F_1-measure reaches its best value at 1 (perfect precision and recall) and worst at 0.

TABLE I: Division of experimental subjects

Groups	SC^P	SC^N
G1	96	96
G2	32	32
G3	32	32
G4	32	32
Total	128	192

C. Experimental Results and Analysis

This part of the experiment study is mainly used to answer **RQ1** and **RQ2**.

PonziDetector employs threshold analysis to classify smart contracts. When the behavior forest similarity between the smart contract under detection X and the ponzitract Y is greater than a given threshold δ (i.e., $\Phi(BF(X), BF(Y)) > \delta$), we classify X as the ponzitract. Therefore, the performance of the *PonziDetector* is affected by the choice of threshold. To find an optimal threshold or a suitable threshold interval, we experimented with several different thresholds. Specifically, the initial threshold was set to 0.05, and each time the increase was 0.05 to the end of 1.00, a total of 20 experiments were performed.

Table II lists the experimental results of *PonziDetector* on test datasets G2, G3 and G4 with different thresholds. Due to the limited space, for the experimental results on the $Ponzi\mathcal{BF}$ database after filtering, we only list its F_1-measure (corresponding to the F_1' column in the G2, G3, and G4 columns) and average of each metric results (corresponding to the P', R' and F_1' columns in "Average" column) to represent the performance of *PonziDetector*, without showing precision and recall details. When the threshold is 0.75, *PonziDetector* has achieved a precision of 1.0 on every test dataset, so we didn't list the experimental data after 0.75.

1) PonziDetector*'s Effectiveness:* In order to better illustrate the effectiveness of *PonziDetector*, we directly compared the experimental results of *PonziDetector* with that shown in the paper [7] reported by Weili et al. The reason for comparing with them are: (1) the smart contract dataset we used is public by them; (2) we all tried to solve the same problem, that is, ponzitract detection; and (3) the opcodes of smart contracts were used in both two methodology. The best performance of their methodology presented in their paper as follows: the precision is 0.94, the recall is 0.81, and the F_1-measure is 0.86. Compared with their methodology, as shown in Table II,

[5] etherscan.io/address/0x723dff0e27cc38b80556f5e05dfdbdcb721654d7

TABLE II: The experimental results on test datasets G2, G3, and G4

	G2				**G3**				**G4**				**Average**		
δ	P	R	F_1	F_1'	P	R	F_1	F_1'	P	R	F_1	F_1'	P'	R'	F_1'
0.05	0.525	0.969	0.681	0.681	0.585	0.969	0.729	0.729	0.544	0.969	0.697	0.697	0.551	0.969	0.702
0.10	0.525	0.969	0.681	0.681	0.585	0.969	0.729	0.729	0.544	0.969	0.697	0.697	0.551	0.969	0.702
0.15	0.526	0.938	0.674	0.674	0.577	0.938	0.714	0.714	0.577	0.938	0.714	0.714	0.560	0.938	0.701
0.20	0.526	0.938	0.674	0.674	0.600	0.938	0.732	0.732	0.588	0.938	0.723	0.723	0.564	0.938	0.704
0.25	0.588	0.938	0.723	0.723	0.652	0.938	0.769	0.769	0.625	0.938	0.750	0.750	0.622	0.938	0.747
0.30	0.625	0.938	0.750	0.750	0.714	0.938	0.811	0.811	0.682	0.938	0.789	0.789	0.674	0.938	0.783
0.35	0.698	0.938	0.800	0.800	0.769	0.938	0.845	0.845	0.714	0.938	0.811	0.811	0.727	0.938	0.819
0.40	0.789	0.938	**0.857**	**0.870**	0.811	0.938	0.870	0.870	0.732	0.938	0.822	0.822	0.784	0.938	0.854
0.45	0.857	0.938	0.896	0.896	0.882	0.938	0.909	0.909	0.811	0.938	0.870	0.870	0.850	0.938	0.891
0.50	0.938	0.938	0.938	0.938	0.909	0.938	0.923	0.923	0.909	0.938	0.923	0.923	0.919	0.938	0.928
0.55	**0.968**	**0.938**	**0.952**	**0.952**	**0.968**	**0.938**	**0.952**	**0.952**	**0.938**	**0.938**	**0.938**	**0.938**	**0.958**	**0.938**	**0.947**
0.60	0.967	0.906	0.935	0.935	0.967	0.906	0.935	0.935	0.967	0.906	0.935	0.935	0.967	0.906	0.935
0.65	0.966	0.875	**0.918**	**0.900**	0.966	0.875	0.918	0.900	1.000	0.875	0.933	0.915	0.976	0.844	0.905
0.70	0.963	0.813	0.881	0.862	1.000	0.813	0.897	0.877	1.000	0.813	0.897	0.877	0.987	0.781	0.872
0.75	1.000	0.813	0.897	0.877	1.000	0.813	0.897	0.877	1.000	0.813	0.897	0.877	1.000	0.781	0.877

PonziDetector achieves the best performance on the G2, G3, G4 test datasets when the threshold $\delta = 0.55$. At this time, except for the precision on the G4 is less than 0.94, the values of other metrics are higher than their methodology.

In addition, if only the "Opcode" features are considered, regardless of the "Account" features, the performance of their methodology is as follows: the precision is 0.90, the recall is 0.80, the F_1-measure is 0.84. It is worth noting that there may not be enough "Account" features available in the early stages of a ponzitract release. Fortunately, *PonziDetector* does not rely on the "Account" features, so it is not limited by how long the contract has been deployed on Ethereum blockchain and can be used for early detection of ponzitracts.

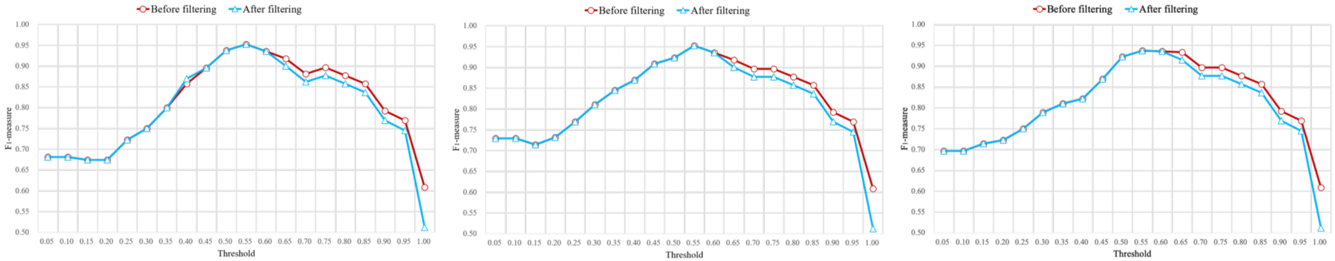
We suspect that some behavior trees in $Ponzi\mathcal{BF}$ may also exist in some non-Ponzi smart contracts. However, the existence of them may not only reduce the accuracy of ponzitract detection but also reduce the detection efficiency. To improve the effectiveness and efficiency of *PonziDetector*, we tried to filter the database $Ponzi\mathcal{BF}$. It is well known that a decrease in recall usually accompanies the increase in precision. In our scenario, when the threshold is 0.55, *PonziDetector* achieves the best performance, and the corresponding F_1 value is the largest. From Fig. 5 (a) - (c), combining with the detailed data in Table II, we can find two phenomena :

(1) when $\delta \in [0.00, 0.55)$, the F_1 and F_1' value will increase as the δ increases, and in the case of the same δ, the F_1' value is greater than or equal to the value F_1 (i.e., $F_1' >= F_1$);

(2) when $\delta \in (0.55, 1.00]$, the F_1 and F_1' value will decrease as the δ decreases, and in the case of the same δ, the F_1' value is less than or equal to the value F_1 (i.e., $F_1' <= F_1$).

The reason behind these phenomena is that in the case of the same δ, the number of results searched after filtering may be less than before filtering. And when $\delta \in [0.00, 0.55)$, the number of FP is reduced faster than TP, while $\delta \in (0.55, 1.00]$ is the reverse. In summary, the filter improves the accuracy of the behavioral similarity measure, making F_1 converge faster, which improves the effectiveness of *PonziDetector*.

To make *PonziDetector* gain better ability of generalization, we do not intend to set the similarity threshold δ to a fixed value of 0.55. Instead, we hope to find a value interval for the threshold (TVI) in which *PonziDetector* can perform satisfactorily. To find such an interval, we first need to find a satisfactory value interval for three metrics (i.e., precision, recall, and F_1-measure). In other words, *PonziDetector*'s performance would be satisfactory if the values of three metrics fall within this interval. The value interval of the threshold corresponding to the value interval of metrics (MVI) is which we expect to find. As we have described before, we consider that precision and recall are equally crucial for ponzitract detection. F_1-measure is the harmonic average of the precision and recall. Therefore, we set the lower bound of the value interval of metrics to the maximum between the precision and the recall. In summary, we set the upper and lower bounds of the value interval of metrics to 0.90 and 1.00, respectively, i.e., $MVI \in [0.90, 1.00]$. The value 0.90 is inspired by the maximum precision and recall that can be achieved by the methodology proposed by Weili et al. [7] in the absence of "Account" features. Fig. 6 shows the performance of *PonziDetector* (i.e., three metrics) with different thresholds. In Fig. 6, we can see a black division line (corresponding to the value of three metrics are 0.9), and the interval MVI corresponds the part above the division line. We can clearly see that the nodes (circles, diamonds, and triangles nodes) used to represent the values of the three metrics appear three times at the same time above the division line, and the corresponding thresholds are 0.5, 0.55, and 0.60, respectively. Therefore, we conclude that the upper and lower bounds of the value of the threshold are 0.5 and 0.6, respectively, i.e.,



(a) Filter on G2

(b) Filter on G3

(c) Filter on G4

Fig. 5: Comparison of the value of F_1-measure value before and after filtering the database $Ponzi\mathcal{BF}$

$TVI \in [0.50, 0.60]$.

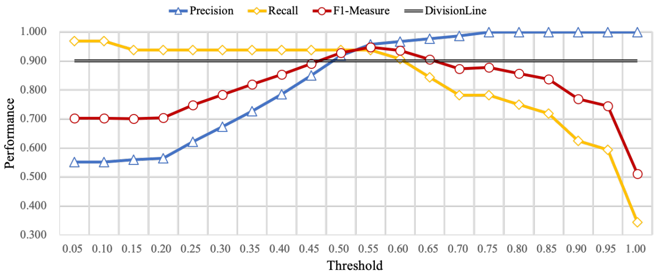


Fig. 6: The performance of *PonziDetector* with different thresholds

Moreover, to objectively evaluate the effectiveness of *PonziDetector* in our limited ponzitract set, we also adopted cross-validation. Cross-validation is any of various similar model validation techniques for assessing how the results of a statistical analysis will generalize to an independent data set [25]. Specifically, we use *k*-fold cross-validation to evaluate the effectiveness of *PonziDetector*. For experimental research, 30 subjects per group are often cited as the minimum [26]. 30 is a rule of thumb. Like all rules of thumb it only says something about reasonableness. Thus, to ensure statistical significance, we set the value of *k* to 4 so that the sample size of ponzitracts is $32 > 30$ in each group. Table III shows the experimental results of *4*-fold cross-validation.

In Table III, R1, R2, R3 and R4 are the four groups of results for *4*-fold cross-validation, respectively. We can get the following observations: (1) *PonziDetector* still achieved the best performance at $\delta = 0.55$; (2) When $\delta \in [0.55, 1.00]$, the value of all three evaluation metrics (i.e., precision, recall, F_1-measure) are larger than 0.9 (the TVI discussed on the above). Thus, based on these observations, we can conclude that *PonziDetector* is superior to the state-of-the-art [7]. In addition, the value (including TVI) of threshold our recommended is reasonable and stable so that we can envision *PonziDetector* to achieve better generalization ability in the future.

2) PonziDetector's Efficiency: During the evaluation of *PonziDetector*, most of the time was spent on building a $Ponzi\mathcal{BF}$ database. This is a fundamental and essential work for our approach, but it is a preparation work that does not take up the new smart contract detection time. When detecting a new smart contract X, the time-consuming processes include (1) interaction with X, and (2) searching for the candidate behavior forest set ($C\mathcal{BF}$) in $Ponzi\mathcal{BF}$.

In the process (1), in order to improve the interaction efficiency, we set the number of tests to 10 instead of more, to limit the times of interactions with the smart contract deployed on Ethereum Testnet.

In the process (2), *PonziDetector* tries to search for the candidate ponzitract set $C\mathcal{BF}$ that used to assist in judging whether X is a ponzitract. To improve search efficiency, we try to filter some behavior trees from $Ponzi\mathcal{BF}$ that are irrelevant to judge whether X is a ponzitract.

Table IV shows the comparison of experimental results before and after filtering $Ponzi\mathcal{BF}$. Although we only filtered 40 irrelevant behavior trees from $Ponzi\mathcal{BF}$, the efficiency of search for $C\mathcal{BF}$ is still significantly improved (increased by nearly 25%). By optimizing the above two processes, *PonziDetector* takes an average of 4.8 seconds to detect X.

D. Textual and Behavioral Similarity

This part of the experimental study is mainly used to answer **RQ3**. We have further studied the textual similarity of smart contracts to prove that it is more accurate and reliable for *PonziDetector* to detect ponzitracts from the perspective of behavioral similarity.

We use Difflib [24], a baseline tool for textual similarity measure, to compare smart contracts, and calculate the textual similarity with the following equations:

$$sim_D(s_1, s_2) = 1 - \frac{min(|s_2|^{LOC}, |D(s_1, s_2)|^{LOC})}{|s_2|^{LOC}} \quad (5)$$

$$sim(s_1, s_2) = max\{sim_D(s_1, s_2), sim_D(s_2, s_1)\} \quad (6)$$

Difflib relies on the class “difflib.Differ” to compare sequences of lines of text and produce human-readable differences or deltas [24]. It has been used in code similarity analysis [27], [28]. Referring to these papers, given two smart contracts s_1 and s_2, their textual similarity is calculated by Equation (5), where $|s_1|^{LOC}$ and $|s_2|^{LOC}$ correspond to the number of lines of code (LOC) in s_1 and s_2 respectively. $D(s_1, s_2)$ represents the output of Difflib. Note

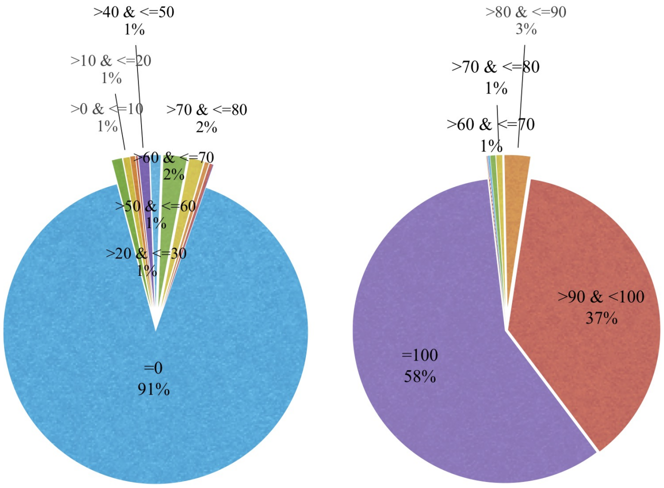
TABLE III: The experimental results of *4*-fold cross-validation

	R1			**R2**			**R3**			**R4**			**Average**		
δ	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
0.50	0.919	0.938	0.928	0.923	0.875	0.898	0.893	0.938	0.941	0.921	0.969	0.944	0.914	0.930	0.921
0.55	0.958	0.938	0.947	0.945	0.875	0.908	0.931	0.938	0.933	0.949	0.969	0.959	**0.946**	**0.930**	**0.937**
0.60	0.967	0.906	0.935	0.955	0.875	0.913	0.947	0.906	0.926	0.958	0.938	0.947	0.956	0.906	0.930

TABLE IV: Comparison of *PonziDetector*'s efficiency before and after filtering the database $Ponzi\mathcal{BF}$

	Count_$Ponzi\mathcal{BF}$	Search_Time_G2 (s)	Search_Time_G3 (s)	Search_Time_G4 (s)	Search_Time_Average (s)
Before filtering	318	506.30	403.26	313.01	407.52
After filtering	295	342.84	330.15	244.42	305.80
Reduce	23	163.46	73.11	68.59	101.72
Improvement	—	32%	18%	22%	25%

that $sim_D(s_1, s_2)$ is sensitive to parameter order, and thus $sim_D(s_1, s_2) \neq sim_D(s_2, s_1)$ in most cases. As Equation (6) shows, we regard the maximum in $sim_D(s_1, s_2)$ and $sim_D(s_2, s_1)$ as the similarity between two smart contracts.



(a) Textual Similarity Distribution

(b) Behavioral Similarity Distribution

Fig. 7: Comparison of similarity distribution

Fig. 7 shows the distribution of the textual and behavioral similarity on different similarity intervals. Among 128 ponzitracts, we found that 95% (i.e., 37% + 58% in Fig. 7-(b)) of smart contract pairs have a behavioral similarity higher than 90%, while the textual similarity of the 91% smart contract pairs is 0 (as shown in Fig. 7-(a)). Undoubtedly, it would be more accurate for us to measure the similarity of smart contracts by behavior rather than text.

We counted LOC for 128 ponzitracts and found that LOC is mainly between 20 and 80, accounting for about 70%. After a statistic of LOC distribution, we further tried to figure out how the textual and behavioral similarity between smart contracts would change as the LOC gap increases, and experimental results shown in Table V and Fig. 8. In Table V, AVE_{TS} and AVE_{BS} represent the average value of textual and behavioral similarity values respectively; SCP represents the number of smart contract pairs; AVE^U_{BS} and SCP^U are the unified processing of the value intervals of AVE_{BS} (calculation formula: $AVE^U_{BS} = AVE_{BS}/10$) and SCP respectively for presentation and comparison in Fig. 8. The formulas for calculating and SCP^U are as follows:

$$SCP^U = \begin{cases} SCP\ /\ 100 & \{LOC\ Gap < 21\} \\ \dfrac{SCP\ /\ 100}{IS\ /\ 5} & \{LOC\ Gap > 20\} \end{cases} \qquad (7)$$

where IS refers to the interval size of the LOC gap, and the value of IS is 5 or 20 when the LOC gap is less than 20 or more than 20.

TABLE V: The effect of LOC gap on similarity

LOC Gap	$AVE_{TS}\%$	$AVE_{BS}\%$	AVE^U_{BS}	SCP	SCP^U
0-5	19.9	99.6	9.96	1201	12.01
6-10	10.40	99.3	9.93	947	9.47
11-15	6.00	99.1	9.91	581	5.81
16-20	2.10	98.6	9.86	416	4.16
21-40	0.10	98.6	9.86	1156	**2.89**
41-60	0.10	95.3	9.53	651	**1.63**
61-80	0.00	97.3	9.73	829	**2.07**
81-100	0.00	96.7	9.67	475	**1.19**
101-120	0.00	98.3	9.83	590	**1.48**
121-140	0.00	97.3	9.73	465	**1.16**
141-160	0.00	97.7	9.77	245	**0.61**
161-180	0.00	98.3	9.83	50	**0.13**
181-200	0.00	98.5	9.85	17	**0.04**
> 200	0.00	95.9	9.59	505	**0.06**
Total & AVE	**2.76**	**97.9**	9.79	8128	—

From Fig. 8, we can find, obviously, with the increasing of the LOC gap, in addition to a significant decrease in the number of smart contract pairs (SCP, corresponding to rectangles), the average textual similarity of smart contracts drops rapidly (AVE_{TS}, corresponding to a curve connected by circles), while change of the average behavioral similarity of them is slight (AVE^U_{TS}, corresponding to a curve connected by triangles). Therefore, based on the above experiments, *we can conclude that it is more accurate and independent of smart contract size (i.e., the LOC) to use the behavioral similarity*

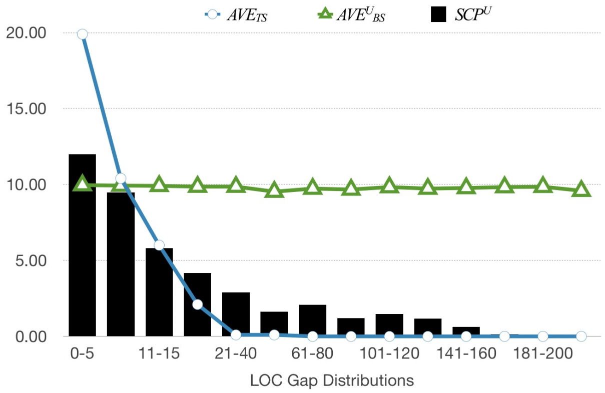


Fig. 8: The changing trend of the number of smart contract pairs and similarity as the LOC gap increases

of smart contracts rather than textual similarity to represent the similarity between them.

V. Threats to Validation

A. Internal validity

The behaviors extracted by *PonziDetector* may be incomplete. For example, some branches may require special inputs to achieve. Since each interaction with the smart contract takes a long time, the more interactions, the more complete the mining behavior will be, and of course the longer it takes. So we took a compromise and generated 10 tests for each smart contract to dig out the full smart contract behavior as much as possible. We will further improve and optimize the extraction of behaviors by combining mature test technologies [29], [30], in the future.

We use the similarity threshold to assist in judging whether a smart contract under detection is a ponzitract. Thus, the choice of threshold affects *PonziDetector*'s performance. In order to reduce the bias of threshold δ, referencing the paper [28], we tried different δ (δ increased from 0 to 1 by 0.05 each time) for many experiments. And we also employ k-fold cross-validation to ensure that we got a reasonable threshold.

B. External validity

Our experiments were conducted on a public dataset that contained smart contracts that were significantly different from the smart contracts we recently observed on Ethereum. Most of the smart contracts observed today are based on specific Ethereum standards (e.g, ERC-20 [31] and ERC-721 [32]), and although the structure is clear, the logic is more complex, which may affect the generalization of our approach. In the future, we are going to experiment on more new datasets.

VI. Related Work

Smart Ponzi Schemes Detection are mainly focused on the two widely used blockchain platforms, i.e., Bitcoin [33], [34] and Ethereum [7], [9], [35]. Massimo et al. [34] and Weili et al. [7] employed a specific method, i.e., data mining and machine learning techniques, to detect Ponzi schemes on Bitcoin and Ethereum respectively. The research most relevant to us is the study by Weili et al. [7]. Both of us think opcodes as one of the critical features for detecting ponzitracts that appear on Ethereum. The opcodes analyzed by [7] are extracted from the compilation results of smart contracts while we collected that by interacting with smart contracts. The dynamically collected opcodes can more accurately describe the complete running-time behaviors of smart contracts. Besides, they only statistic the frequency of opcodes appearing in compilation result but we utilize them to full characterize the semantic information of smart contracts. Because of these, we succeeded in getting rid of the control of trading data.

Behavioral Similarity usually refers to semantic similarity in existing researches [18], [36], [37]. Sihan et al. [18] regard the proportion of inputs producing the same output on both programs over the specific input domain as the behavioral similarity between two programs. They have applied code behavior similarity measure to promote software engineering education. Coen De et al. [37] aim at using a single abstract pattern description to detect multiple concrete pattern instances, in order to achieve this, they take information about a program's run-time behavior into account. By incorporating a similarity-based unification algorithm like the one in LikeLog [38], they can overcome failures in the refutation process caused by a syntactic difference between parse tree nodes which might actually evaluate to overlapping sets of objects at run-time and are also able to evaluate the confidence our platform has in the discovered software pattern instances. In [36], it divides the types of program similarity into syntactic (also called representational) and semantic or behavioral similarity two categories and details several potential ideas for measuring semantic similarity, such as Input-Output Relation Similarity idea for behavioral similarity measure. After learning these papers, we find, obviously, that utilize behavioral similarity to represent program similarity is more accurate, which also leads us to believe that employ behavioral similarity to detect ponzitracts is feasible.

Malware Detection is a subject of extensive research [39], [40]. Malware authors use obfuscation techniques [41] like dead code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition, and code integration to evade detection by traditional defenses like firewalls, antivirus and gateways which typically use signature based techniques and are unable to detect the previously unseen malicious executables [39]. The initiator of the ponzitract conceal malicious scam behavior by implementing Ponzi schemes as smart contracts. However, from the perspective of software program property, the ponzitract is also a type of malware. In [17], the authors construct the opcode running tree to simulate the dynamic execution of a program and use the n-gram method [42] to extract executables features to train a classifier to detect malware. The approach of this study is the closest to ours, and we all think the opcode can represent the behavior of the program or smart contract. But there are two clear distinctions: (1) The opcodes our approach used are dynamically collected during interacting with smart contracts while that in [17] are extracted from executables; (2) Our approach uses complete opcode behaviors to support detection while that used in [17] is partial (i.e, extracting partial features from the opcode running tree). Therefore, even if we can't directly compare with their methods, we believe that, to a certain extent, our approach should be more accurate.

VII. Conclusion

Smart contracts running on Ethereum have become widely used for a broad spectrum of financial applications. Unfortunately, the advantages of blockchain-managed financial assets have also been exploited by criminals, and fraudulent means such as ponzitracts have appeared. To solve this problem in the early days of ponzitract deployment, We developed *PonziDetector* to supporting detect ponzitracts. More importantly, *PonziDetector* does not rely on a large number of practical transaction data which indicated that many investors had been deceived. Thus, PonziDetector can detect ponzitracts early. Our experimental results have proven the effectiveness and efficiency of *PonziDetector*. We are setting up a smart Ponzi schemes detection platform for users to easily detect deployed smart contracts early and prevent to be deceived.

We observed the latest smart contracts on the Etherscan platform [43] that are significantly different from that released by [7]. Most of today's smart contracts are based on new Ethereum standards (e.g, ERC-20 [31] and ERC-721 [32]). In the future, we are going to further validate and improve *PonziDetector* against the latest smart contracts. Also, more applicable testing techniques, such as symbolic execution [29], will be explored to support *PonziDetector* mining more complete behaviors to further improving the performance of ponzitract detection.

Acknowledgment

This work is supported partially by National Key R&D Program of China (2018YFB1403400), National Natural Science Foundation of China(61772014), and Fundamental Research Funds for the Central Universities(14380021).

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," pp. 1–9, 2008.

[2] M. Swan, *Blockchain: Blueprint for a new economy*. "O'Reilly Media, Inc.", 2015.

[3] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[4] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[5] T. Moore, "The promise and perils of digital currencies," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 3-4, pp. 147–149, 2013.

[6] A. Beregszaszi and Chris. Solidity. [Online]. Available: https://solidity.readthedocs.io

[7] W. Chen, Z. Zheng, J. Cui, E. C. H. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology," in *Proceedings of the 27th World Wide Web Conference on World Wide Web*. Lyon, France: ACM, 2018, pp. 1409–1418.

[8] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Proceedings of the 30th International Conference on Computer Aided Verification*. Oxford, UK: Springer, 2018, pp. 51–78.

[9] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact," *The Computing Research Repository*, vol. 1, pp. 1–32, 2017.

[10] T. U. Securities and E. Commission. What is a Ponzi scheme? [Online]. Available: https://www.sec.gov/spotlight/enf-actions-ponzi.shtml

[11] M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, 2016.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, LA, USA: IEEE Computer Society, 2003, pp. 220–231.

[13] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*. Dubrovnik, Croatia: ACM, 2007, pp. 5–14.

[14] H. Liu and H. B. K. Tan, "Covering code behavior on input validation in functional testing," *Information & Software Technology*, vol. 51, no. 2, pp. 546–553, 2009.

[15] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*. Berlin, Germany: IEEE Computer Society, 2011, pp. 1–10.

[16] Y. Feng and Z. Chen, "Multi-label software behavior learning," in *Proceedings of the 34th International Conference on Software Engineering - New Ideas and Emerging Results*. Zurich, Switzerland: IEEE Computer Society, 2012, pp. 1305–1308.

[17] Y. Ding, W. Dai, Y. Zhang, and C. Xue, "Malicious code detection using opcode running tree representation," in *Proceedings of the 9th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Guangdong, China: IEEE Computer Society, 2014, pp. 616–621.

[18] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *Proceedings of the 38th International Conference on Software Engineering, Companion Volume*. Austin, TX, USA: ACM, 2016, pp. 501–510.

[19] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19.

[20] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification & Reliability*, vol. 26, no. 5, pp. 366–401, 2016.

[21] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 59–68.

[22] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd International Conference on Automated Software Engineering*. Montpellier, France: ACM, 2018, pp. 259–269.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 96–105.

[24] T. P. S. Library. difflib-helpers for computing deltas. [Online]. Available: https://docs.python.org/3.7/library/difflib.html

[25] Wikipedia. Cross-validation (statistics). [Online]. Available: https://en.wikipedia.org/wiki/Cross-validation_(statistics)

[26] R. Hill, "What sample size is "enough" in internet survey research," *Interpersonal Computing and Technology: An electronic journal for the 21st century*, vol. 6, no. 3-4, pp. 1–12, 1998.

[27] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[28] W. Sun, X. Wang, H. Wu, D. Duan, Z. Sun, and Z. Chen, "MAF: method-anchored test fragmentation for test code plagiarism detection," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training*. Montreal, QC, Canada: IEEE / ACM, 2019, pp. 110–120.

[29] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[30] X. Wang, H. Wu, W. Sun, and Y. Zhao, "Towards generating cost-effective test-suite for ethereum smart contract," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. Hangzhou, China: IEEE, 2019, pp. 549–553.

[31] F. Vogelsteller. (2015) Eip 20: Erc-20 token standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[32] D. Shirley. (2018) Eip 721: Erc-721 non-fungible token standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[33] M. Vasek and T. Moore, "Analyzing the bitcoin ponzi scheme ecosystem," in *Proceedings of the 22th Financial Cryptography and Data Security - Workshops, BITCOIN, VOTING, and WTSC*. Nieuwpoort, Curaçao: Springer, 2018, pp. 101–112.

[34] M. Bartoletti, B. Pes, and S. Serusi, "Data mining for detecting Bitcoin Ponzi schemes," in *Proceedings of the 1th Crypto Valley Conference on Blockchain Technology*. Zug, Switzerland: IEEE, 2018, pp. 75–84.

[35] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," *IACR Cryptology ePrint Archive*, vol. 2016, pp. 164–186, 2016.

[36] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. von Gudenberg, "Similarity in programs," in *Duplication, Redundancy, and Similarity in Software*. Internationales Begegnungs- und Forschungszentrum fuer Informatik(IBFI), Schloss Dagstuhl, Germany, 2006, pp. 1–8.

[37] C. D. Roover, J. Brichau, and T. D'Hondt, "Combining fuzzy logic and behavioral similarity for non-strict program validation," in *Proceedings of the 8th International Conference on Principles and Practice of Declarative Programming*. Venice, Italy: ACM, 2006, pp. 15–26.

[38] F. A. Fontana and F. Formato, "Likelog: A logic programming language for flexible data retrieval," in *Proceedings of the 14th Symposium on Applied Computing*. San Antonio, Texas, USA: ACM, 1999, pp. 260–267.

[39] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 5, no. 02, p. 56, 2014.

[40] Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys*, vol. 50, no. 3, pp. 41:1–41:40, 2017.

[41] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proceedings of the 5th International Conference on Broadband and Wireless Computing, Communication and Applications*. Fukuoka Institute of Technology, Fukuoka, Japan: IEEE Computer Society, 2010, pp. 297–300.

[42] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. Cambridge, Massachusetts, London, England: MIT Press, 2001.

[43] ETHERSCANERS. Etherscan platform. [Online]. Available: https://etherscan.io