Future Generation Computer Systems 102 (2020) 259–277

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact[☆]

Massimo Bartoletti [*], Salvatore Carta, Tiziana Cimoli, Roberto Saia

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

ARTICLE INFO

Article history:
Received 21 January 2018
Received in revised form 2 July 2019
Accepted 8 August 2019
Available online 14 August 2019

Keywords:
Smart contracts
Cryptocurrencies
Ponzi schemes
Electronic frauds

ABSTRACT

Ponzi schemes are financial frauds which lure users under the promise of high profits. Actually, users are repaid only with the investments of new users joining the scheme: consequently, a Ponzi scheme implodes soon after users stop joining it. Originated in the offline world 150 years ago, Ponzi schemes have since then migrated to the digital world, approaching first the Web, and more recently hanging over cryptocurrencies like Bitcoin. Smart contract platforms like Ethereum have provided a new opportunity for scammers, who have now the possibility of creating "trustworthy" frauds that still make users lose money, but at least are guaranteed to execute "correctly". We present a comprehensive survey of Ponzi schemes on Ethereum, analysing their behaviour and their impact from various viewpoints.

1. Introduction

The advent of Bitcoin [1,2] has given birth to a new way to exchange currency, allowing secure and (almost) anonymous transfers of money without the intermediation of trusted authorities. This has been possible by suitably combining several techniques, among which digital signature schemes, moderately hard "proof-of-work" puzzles, and the idea of *blockchain*, an immutable public ledger which records all the money transactions, and is maintained by a peer-to-peer network through a distributed consensus protocol.

Soon after Bitcoin has become widespread, it has started arousing the interest of criminals, eager to find new ways to transfer currency without being tracked by investigators and surveillance authorities [3].

Recently, *Ponzi schemes* [4] – a classic fraud originated in the offline world at least 150 years ago – have approached the digital world, first on the Web [5], and more recently also on Bitcoin [6]. Ponzi schemes are often disguised as "high-yield" investment programs. Users enter the scheme by investing some money. The actual conditions which allow to gain money depend on the specific rules of the scheme, but all Ponzi schemes have in common that, to redeem their investment, one has to make new users enter the scheme. A more authoritative definition of Ponzi schemes comes from the U.S. Securities and Exchange Commission (SEC)[1]:

> "*A Ponzi scheme is an investment fraud that involves the payment of purported returns to existing investors from funds contributed by new investors. Ponzi scheme organizers often solicit new investors by promising to invest funds in opportunities claimed to generate high returns with little or no risk. With little or no legitimate earnings, Ponzi schemes require a constant flow of money from new investors to continue. Ponzi schemes inevitably collapse, most often when it becomes difficult to recruit new investors or when a large number of investors ask for their funds to be returned.*"

Often, the investment mechanism of Ponzi schemes creates a pyramid-shape topology of users, having at the top level the initiator of the scheme, and at level $\ell + 1$ the users who compensate the investment of those at level ℓ. The scheme will eventually collapse because at some point it will no longer be possible to find new investors, as their number grows exponentially in the number of levels of this pyramid. Therefore, users at the top levels of the pyramid will gain money, while those at the bottom levels will just lose their investment.

Despite many investors are perfectly conscious of the fraudulent nature of these schemes, and of the fact that they are illegal in many countries, Ponzi schemes continue to attract remarkable amounts of money. A recent study [6] estimates that Ponzi schemes operated through Bitcoin have gathered more than 7

☆ Work partially supported by Aut. Region of Sardinia under grant P.I.A. 2013 "NOMAD".

* Correspondence to: Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, via Ospedale 72, 09124 Cagliari, Italy.
E-mail address: bart@unica.it (M. Bartoletti).

1 Source: `www.sec.gov/spotlight/enf-actions-ponzi`.

https://doi.org/10.1016/j.future.2019.08.014

millions *USD* in the period from September 2013 to September 2014.[2]

"Smart" Ponzi schemes. The spread of *smart contracts*, i.e., computer programs whose correct execution is automatically enforced without relying on a trusted authority [7], creates new opportunities for fraudsters. Indeed, implementing Ponzi schemes as smart contracts would have several attractive features:

1. The initiator of a Ponzi scheme could stay anonymous, since creating the contract and withdrawing money from it do not require to reveal her identity;
2. Since smart contracts are "unmodifiable" and "unstoppable", no central authority (in particular, no court of law) would be able to terminate the execution of the scheme, or revert its effects in order to refund the victims. This is particularly true for smart contracts running on *permissionless* blockchains, which are controlled by a peer-to-peer network of nodes.
3. Investors may gain a false sense of trustworthiness from the fact that the code of smart contracts is public and immutable, and their execution is automatically enforced. This may lead investors to believe that the owner cannot take advantage of their money, that the scheme would run forever, and that they have a fair probability of gaining the declared interests.

All these features are made possible by a combination of factors, among which the growth of platforms for smart contracts [8], which advertise anonymity and contract persistence as main selling points, and the fact that these technologies are very recent, and still live in a grey area of legal systems [9,10].

Understanding the behaviour of "smart" Ponzi schemes would be crucial to devise suitable intervention policies. To this purpose, one has to analyse various aspects of the fraud, answering several questions: how many victims are involved? How much money is invested? What are the temporal evolution and the lifetime of a fraud? What kind of users fall in these frauds? Can we recognize fingerprints of Ponzi schemes during their execution, or possibly even before they are started? Investigating on these issues would help to disrupt this kind of frauds.

Contributions. This paper is the first comprehensive survey on Ponzi schemes in Ethereum [11], the most prominent platform for smart contracts so far. We construct a dataset of Ponzi schemes, and we analyse them from various perspectives. More specifically, our contributions can be summarized as follows:

- a set of criteria for determining when a smart contract implements a Ponzi scheme. Our criteria take into account only the logic implemented by the contract to gather and distribute money, while neglecting external factors, like e.g., the way the scheme is advertised or gamified.
- a public dataset of Ponzi schemes deployed on Ethereum (`goo.gl/CvdxBp`), Coherently with our classification criteria, the dataset is constructed by examining the source code of contracts. We start from the contracts whose source code is available on blockchain explorers, finding among them 138 Ponzi schemes. We expand this collection to 184 schemes, by searching the blockchain for contracts whose bytecode is highly similar to a contract already classified as Ponzi. False negatives are excluded by manually inspecting their decompiled code.
- an open-source tool (github.com/blockchain-unica/ethereum-ponzi) which extracts from the Ethereum blockchain all the transactions of the Ponzi schemes in our dataset, records all the incoming and outgoing movements of money, and computes the analyses presented in this paper.
- an analysis of the source code of the contracts in our collection (Section 4). We discover that most contracts share a few common patterns, and that many of them are obtained by minor variations of already existing ones. We devise a rough taxonomy of Ponzi schemes, which classifies them according to the pattern used to redistribute payouts. We show that the schemes in each category fail to achieve a fair distribution of money. Further, we spot several security vulnerabilities in the analysed contracts, which could be exploited by adversaries to steal money.
- a measure of the economic impact of Ponzi schemes, quantifying the overall value exchanged through them (Section 5).
- a measure of the gains and losses of the users of Ponzi schemes (Section 6). We focus on the top 10 schemes (those with the highest number of transactions), with most interesting features (number of users, transactions, or ether exchanged). In most cases we observe the typical pattern of Ponzi schemes: a few users gain a lot, while the majority of users simply lose their money.
- an analysis of the temporal behaviour of Ponzi schemes under various viewpoints (Section 7). First, we investigate the lifetime of Ponzi schemes, an important indicator to predict when a scheme is going to collapse. Then, we analyse the correlation between inflow and outflow of contracts over time. Finally, we measure the monthly volume of transactions.
- a measure of the inequality of payments to and from the schemes (Section 8). This indicator may reveal how scammers select their victims: a fair distribution of payments means that the scheme is fed by a large number of victims who pay small amounts of money; instead, an unequal distribution often means that the scheme profits from a small number of "big fishes" who invest a lot of money.
- a set of guidelines that users could follow to protect themselves against Ponzi schemes (Section 9).

2. Ethereum in a nutshell

Ethereum [11] is a decentralized virtual machine, which can execute programs – called *contracts* – written in a Turing-complete bytecode language, called EVM [12]. Every contract has a permanent storage where to keep data, and a set of functions which can be invoked either by users or by other contracts. Users and contracts can own a crypto-currency (called *ether*, or *ETH* in short), and send/receive ether to/from users or other contracts.

Users can send *transactions* to the Ethereum network in order to: (i) create new contracts; (ii) invoke a function of a contract; (iii) transfer ether to contracts or to other users. All the transactions sent by users, called *external* transactions, are recorded on a public, append-only data structure — the *blockchain*. Upon receiving an external transaction, a contract can fire some *internal* transactions, which are not explicitly recorded on the blockchain, but still have effects on the balance of users and of other contracts.

Since transactions can move money, it is crucial to guarantee that their execution is performed correctly. To this purpose, Ethereum does not rely on a trusted central authority: rather, each transaction is processed by a decentralized network of nodes. There is a *consensus* protocol to address mismatches (due e.g., to failures or to attacks), which is currently based on a "proof-of-work" puzzle. The security of the consensus protocol

[2] This estimate considers both traditional Ponzi schemes which also accept payments in bitcoins, and schemes that only handle bitcoins.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4      mapping (address => uint) public inflow;
5
6      function AWallet(){ owner = msg.sender; }
7
8      function pay(uint amount, address recipient) returns (bool){
9          if (msg.sender != owner ||  msg.value != 0) throw;
10         if (amount > this.balance) return false;
11         outflow[recipient] += amount;
12         if (!recipient.send(amount)) throw;
13         return true;
14     }
15
16     function(){ inflow[msg.sender] += msg.value; }
17 }
```

Fig. 1. A simple wallet contract.

relies on the fact that following the protocol is more convenient than trying to attack it. Indeed, nodes receive economic incentives for correctly performing all the computations required by the protocol. The execution of contracts is guaranteed to be correct, as long as the adversary does not control a very large portion of the computational power of the network [13].

Contracts. Abstractly, contracts can be seen as objects in object-oriented languages, which are composed of fields and functions. A user can invoke a function by sending a suitable transaction to the Ethereum nodes. The transaction must include the execution fee (for the miners), and may include a transfer of ether from the caller to the contract.

We illustrate contracts through a small example (`AWallet`, in Fig. 1), which implements a personal wallet associated to an owner. Rather than programming it directly as EVM bytecode, we use *Solidity*, a Javascript-like programming language which compiles into EVM bytecode [14]. The contract can receive ether from other users, and its owner can send (part of) that ether to other users via the function `pay`. The hashtable `outflow` records all the *addresses*[3] to which it sends money, and associates to each of them the total transferred amount. The hashtable `inflow` records all the addresses from which it has received money. All the ether received is held by the contract. Its amount is automatically recorded in `balance`: this is a special variable, which cannot be altered by the programmer. When a contract receives ether, it also executes a special function with no name, called *fallback* function.

The function `AWallet` at line 6 is a constructor, run only once when the contract is created. The function `pay` sends `amount` *wei* ($1\,wei = 10^{-18}ETH$) from the contract to `recipient`. At line 9 the contract throws an exception if the caller (`msg.sender`) is not the owner, or if some ether (`msg.value`) is attached to the invocation and transferred to the contract. Since exceptions revert side effects, this ether is returned to the caller (who however loses the fee). At line 10, the call terminates if the required amount of ether is unavailable; in this case, there is no need to revert the state with an exception. At line 11, the contract updates the `outflow` registry, before transferring the ether to the recipient. The function `send` used at line 12 to this purpose presents some quirks, e.g. it may fail if the recipient is a contract. The fallback function at line 16 is triggered upon receiving ether, when no other function is invoked. In this case, the fallback function just updates the `inflow` registry. In both cases, when receiving ether and when sending, the total amount of ether of the contract, stored in variable `this.balance`, is automatically updated.

[3] Addresses are sequences of 160 bits which uniquely identify contracts and users.

R1 the contract distributes money among investors, according to some logic.

R2 the contract receives money *only* from investors.

R3 each investor makes a profit *if* enough investors invest enough money in the contract afterwards.

R4 the later an investor joins the contract, the greater the risk of losing his investment.

Fig. 2. Criteria for classifying a contract as a Ponzi scheme.

3. Collection of Ponzi schemes

In this section we establish a set of criteria for classifying contracts as Ponzi schemes. We then describe our methodology for constructing a collection of Ponzi schemes, and for extracting the related transactions.

3.1. What is a "smart" Ponzi scheme?

We start by clarifying what is considered a Ponzi scheme in this paper. The first key choice that we make is to restrict to the schemes which are implemented as smart contracts — or *"smart" Ponzis* [15]. This choice rules out scams which use Ethereum only as a mean of payment (or just for advertisement). These scams include some well-known "high-yield" investment programs, many of which are reported on the blacklist maintained by *BadBitcoin*[4] We chose to exclude this kind of scams from our analysis, since it is seldom possible to retrieve any information about the Ethereum addresses they use (if any).

In Fig. 2 we propose four requirements to determine if a contract is a Ponzi scheme, based exclusively on the logic implemented within the contract. When a contract satisfies *all* four requirements, we classify it as a Ponzi scheme.

- **R1** asks that the contract distributes money to *investors*, i.e. users who join the contract by sending some money to it. This requirement does not put any constraints on the logic used to distribute the money, so **R1** alone is not enough to classify a contract as Ponzi: for instance, gambling games, lotteries, insurances and bonds, satisfy **R1**. However, **R1** rules out the contracts which provide users with some kind of assets, but do not implement the logic to distribute them: rather, these assets are exchanged through external marketplaces, like cryptocurrency exchanges. This is the case, e.g., of most implementations of ERC-20 tokens [16], among which Initial Coin Offerings [17].

[4] https://badbitcoin.org/thebadlist/index.php.

- **R2** asks that the money gathered by the contract comes from investors, *only*. This rules out the cases where the money distributed to investors comes from external sources, like e.g. a bank which pays the interests of a "smart" bond, or a bookmaker who pays off bets using his own funds.
- **R3** asks that each investor makes a profit, *provided that* new investors continue to send money to the contract. Together with the first two requirements, this implies that users make profits *only* through the investments of other users. Note that gambling games, betting, and lotteries violate **R3**: there, even if there is a constant flow of investments, an unlucky user is not guaranteed to make any profit (e.g., he can always lose the lottery).
- **R4** asks that the risk of losing one's investment grows with the time one joins the scheme. This is a landmark feature Ponzi schemes also in the real world: at a certain point it becomes difficult to find new investors, so no one makes profits anymore, and the scheme collapses.

Compared to the SEC definition of Ponzi scheme quoted in Section 1, the requirements **R1** and **R2** together capture that fact that a Ponzi scheme *"involves the payment of purported returns to existing investors from funds contributed by new investors"*; **R3** corresponds to the fact that they *"require a constant flow of money from new investors to continue"*; **R4** implies that they *"inevitably collapse, most often when it becomes difficult to recruit new investors"*. Note that our requirements do not capture the fact that *"Ponzi scheme organizers often solicit new investors by promising to invest funds in opportunities claimed to generate high returns with little or no risk"*. This is because, by design, our requirements are based exclusively on the logic implemented within the contract, while advertising is done outside the contract code.

Ponzi vs. pseudo-Ponzi schemes. Note that requirement **R4** rules out some contracts which are sometimes blamed to be Ponzi schemes, even if the Ponzi mechanism to distribute investments is not *hard-coded* in the contract. This is the case e.g. of contracts which implement crypto-collectible markets — the most notable instance being *CryptoKitties*, a game where players can breed and trade virtual cats, implemented as ERC-721 tokens. Indeed, **R4** is violated, because a lucky user, regardless of the moment when he joins the contract, may breed a rare cat, and make a profit by its sale.

For similar reasons, **R4** rules out *Fomo3D*, a sort of game which is sometimes pointed out as a Ponzi scheme. *Fomo3D* works as a lottery game where, at each round, players can purchase some "keys", and the last buyer in the round wins a jackpot. Whenever a key is purchased, the deadline to the end of the round is extended, and the earning from key selling is split in two parts: a part is added to the jackpot, while the other is shared among the participants in the round. The lottery mechanism decouples the time when a user joins the scheme from its risk of losing her investment, violating **R4**.

Requirement **R4** rules out also *PoWH3D*, another alleged Ponzi scheme. *PoWH3D* implements a token and its exchange: the contract mechanism ensures that the value of the token grows when people buy, and decreases when they sell; further every token trade has a 10% fee. Investors can earn in three ways: by selling a token for more than it was paid; by inviting a new investor to buy tokens (in this case, they get the fees of the invitee); and by receiving the fees paid by a (not invited) investor (these fees are distributed among all the token holders). Requirement **R4** is violated because investing late, e.g. in a period of stagnation, does not necessarily imply a greater risks of losing one's investment, since the mechanism ensures that the value of tokens is low.

We remark that, even if a contract does not explicitly implement a Ponzi mechanism (so, violating some of our requirements), it may potentially induce a behaviour which closely resembles that of a Ponzi scheme. For instance, *CryptoKitties* and its followers gave rise to a market of crypto-collectibles which is often compared to the "tulip mania", a large speculative bubble in the 1600s. The extreme popularity of *CryptoKitties* almost caused the congestion of the Ethereum network in 2017; some virtual cats were sold for more than 170*KUSD*, and the market has processed more than $12 million in sales of virtual cats [18].

3.2. Collection of Ponzi schemes

To construct a dataset of Ponzi schemes, we start by retrieving the Solidity code of contracts published on the Ethereum blockchain. Since the blockchain only stores the EVM bytecode, to this purpose we rely on the blockchain explorer *Etherscan*, which allows developers to upload the Solidity code of their contracts, and verifies that their compilation matches the EVM code on the blockchain.[5]

By manually inspecting the Solidity code of these contracts, we detect 138 contracts which satisfy *all* the requirements in Fig. 2, and therefore can be classified as Ponzi schemes. Since all the contracts in this sample are relatively small ($<$ 120 LOC, including comments), manual inspection was accurate enough to check the requirements. As a further check, for all these contracts we study the pattern they use to redistribute money, which is the basis for our taxonomy in Section 4. To stay on the safe side (i.e., to avoid false positives), we have not included in this collection those contracts which are too complex to establish with certainty whether they satisfy the requirements or not.

We perform a second search phase to enlarge our collection. More specifically, we search the Ethereum blockchain for contracts whose bytecode is *similar* to that of some Ponzi scheme identified in our initial collection. This is done through the following steps:

1. We use a Monte Carlo algorithm to estimate the *normalized Levenshtein distance* [20] (NLD) between two *arbitrary* EVM contracts on the Ethereum blockchain. The NLD is a standard measure of similarity between two strings. The *non-normalized* Levenshtein distance between two strings measures the number of character which one has to change to transform the first string in the second one (e.g., the distance between "Ponzi" and "Banzai" is 3). The *normalized* version is a metric, and its value is a real number ranging between 0 (perfect equality) and 1 (perfect inequality). After these calculations, we estimate as 0.79 the NLD between two arbitrary EVM contracts downloaded from the blockchain.
2. We compute the NLD between the contracts in our initial sample, and all the contracts on the Ethereum blockchain. We consider as a *potential* Ponzi scheme any contract with a NLD less than 0.35 from some contract in our sample. The two values 0.35 and 0.79 are sufficiently far apart to ensure a low incidence of *false positives*, i.e. contracts whose NLD from the initial sample is below 0.35, but they are not Ponzi schemes. This search resulted in 0 *potential* new Ponzi schemes, not included in our original collection of 138 contracts.

[5] When we first created our collection in 2017, it was still possible to list all contracts with verified source code through the URL https://etherscan.io/contractsVerified. Currently, only the last 500 contracts with verified source are listed. To overcome this limitation, one can use a blockchain parser, like e.g. BlockAPI [19], to scan all the transactions on the blockchain, and fetch their Solidity code from *Etherscan*. For a contract address xyz, the URL https://etherscan.io/address/xyz#code contains the contract Solidity code, if verified by *Etherscan*.

Table 1
Top-10 Ponzi schemes by amount of invested ether.

Contract name	#Trans.		ETH		USD		Users		Transactions	
	In	Out	In	Out	In	Out	Paying	Paid	First	Last
DynamicPyramid	444	143	7 474	7 437	84 187	83 541	175	51	2016-02-23	2018-10-01
DianaEthereum-x1.8	288	168	5 307	5 303	61 166	61 266	129	84	2016-03-08	2018-05-17
Doubler2	395	161	4 858	4 825	26 376	26 220	211	68	2016-02-16	2018-11-22
ZeroPonzi	627	499	4 490	4 489	49 816	49 770	47	28	2016-04-04	2017-10-27
Doubler	156	57	3 073	3 073	31 292	35 927	92	17	2016-02-19	2018-06-26
Government	723	846	2 939	2 939	35 738	40 066	40	27	2016-03-08	2017-03-20
Rubixi	686	66	1 367	1 363	16 986	16 775	104	28	2016-03-14	2019-01-24
ProtectTheCastle2	890	1257	1 332	1 332	186 040	190 802	101	68	2016-03-20	2018-02-22
EthereumPyramid	978	339	986	917	5 044	5 290	327	125	2015-09-07	2018-04-11
Total (184 schemes)	18 925	9100	43 881	43 332	630 662	702 878	2378	1232	–	–

3. We apply the Online Solidity Decompiler[6] to the EVM bytecode of the 0 contracts found in the second phase, and we manually compare the obtained Solidity code with that of the corresponding Ponzi scheme found in the first phase. In 46 cases we find a substantial match between the contract codes, so we add these contracts to our collection.

In conclusion, we end up with a dataset of 184 Ponzi schemes, which we make available at `goo.gl/CvdxBp` (an excerpt is in Table 1 in Section 5). We stress that our collection does *not* include all the Ponzi schemes which have been published on Ethereum over the years. For instance, the contract `PonziUnlimited`[7] is blatantly a Ponzi scheme, but it is not immediate to detect if its logic satisfies the requirements in Fig. 2, so we do not include it in our collection.[8]

3.3. Extraction of transactions

For each Ponzi scheme in our dataset, we gather all its transactions (both external and internal) from the Ethereum blockchain. More specifically, for each transaction we record the following data: (i) the number of the enclosing block; (ii) the date when it was published on the blockchain; (iii) the address of the sender; (iv) the address of the receiver; (v) the amount of ether transferred by the transaction; (vi) a boolean value which records whether the transaction execution resulted in an error; (vii) a boolean value which indicates whether the transaction is external or internal. The scripts that we have developed to this purpose exploit the Etherscan Ethereum Developer APIs,[9] and they are available at `github.com/blockchain-unica/ethereum-ponzi`.

4. Anatomy of Ponzi schemes

In this section we analyse the source code of Ponzi schemes, to understand their behaviour, and find analogies between different schemes. We then discuss some security issues found in the analysed contracts.

4.1. Taxonomy of Ponzi schemes

Based on the analysis of the contracts source code carried out in Section 3.2, we devise a rough taxonomy of Ponzi schemes, which classifies them according to the pattern used to redistribute money. Our taxonomy consists of four categories, whose archetypal representatives are displayed in Figs. 3–6.[10] We discuss below the categories of our taxonomy.

Tree-shaped schemes use a tree data structure to induce an ordering among users. Whenever a user joins the scheme, she must indicate another user as inviter, who becomes her *parent node*. If no inviter is indicated, the parent will be the root node, i.e. the owner of the scheme. In most schemes, the amount of money to be invested is chosen by the user, and there is a lower bound to that amount. The money of the new user is split among her ancestors with the logic that the nearest an ancestor is, the greater her share. Since there is no limit to the number of children of a node, the more children (and descendants) a node has, the more money it will make.

We show in Fig. 3 an archetypal scheme of this kind. To join the scheme, a user must send some money, and must indicate an `inviter` that will be her parent node. If the amount is too low (line 15), or if the user is already present (line 16), or if the inviter does not exist (line 17), the user is rejected; otherwise she is inserted in the tree (line 19). Once the user has joined, her investment is shared among her ancestors (lines 25–29), halving the amount at each level.

In this scheme, a user cannot foresee how much she will gain: this depends on how many users she is able to invite, and on how much they will invest. The only one who is guaranteed to have profit is the owner, i.e. the root node of the tree. Examples for this kind of scheme are `Etheramid` and `DynamicPyramid`.

Chain-shaped schemes are a special case of tree-shaped schemes, where each node of the tree has exactly one child (so, the ordering induced among users is linear). The schemes in this category usually multiply the investment by a predefined constant factor, which is equal for all users. The scheme starts paying back users, one at a time, in order of arrival, and in full: all new investments are gathered until the due amount is obtained. At that moment, the contract sends the payout back in a single shot, and moves on to the next user in the chain. The amount to be invested can be fixed, or free, or have a lower bound. Usually, the contract owner retains a fee from each investment.

We show in Fig. 4 an archetypal chain-shaped scheme, which doubles the investment of each user. To join the scheme, a user sends `msg.amount` *ETH* to the contract, hence triggering the fallback function (line 14). The contract requires a minimum fee of 1*ETH*: if `msg.amount` is below this minimum, the user is rejected (line 15); otherwise, her address is added to the array (line 17),

6 https://ethervm.io/decompile.

7 https://etherscan.io/address/0x582b2489710A4189AD558B6958641789587fCc27.

8 A relevant question, without an easy answer, would be that of estimating the total number of smart Ponzi schemes on Ethereum. The analysis in [21] conjectures that there could be 507 smart Ponzi schemes created on Ethereum before September 2017.

9 https://etherscan.io/apis.

10 The code snippets presented there assume version v0.2.2 of the Solidity compiler, which is the version used by most of the contracts in our collection. Although more recent versions of Solidity change the way to declare functions and to manage arrays, these changes do not really affect the spirit of our examples.

```
contract TreePonzi {

  struct User {
    address inviter;
    address itself;
  }
  mapping (address=>User) tree;
  address top;

  function TreePonzi() {
    tree[msg.sender] =
      User({itself: msg.sender,
           inviter:msg.sender});
    top = msg.sender;
  }
  function enter(address inviter) public {
    if ((msg.value < 1 ether) ||
      (tree[msg.sender].inviter != 0x0) ||
      (tree[inviter].inviter == 0x0)) throw;

    tree[msg.sender] = User({itself: msg.sender,
                            inviter: inviter});
    address current = inviter;
    uint amount = msg.value;
    while (next != top) {
      amount = amount/2;
      current.send(amount);
      current = tree[current].inviter;
    }
    current.send(amount);
  }}
```

Fig. 3. A tree-shaped scheme.

```
1  contract ChainPonzi {
2
3    struct User {
4      address addr;
5      uint amount;
6    }
7    User[] public users;
8    uint public paying = 0;
9    address public owner;
10   uint public totalUsers=0;
11
12   function ChainPonzi() {
13     owner = msg.sender;
14   }
15
16   function() {
17     if (msg.value < 1 ether) throw;
18
19     users[users.length] = User({addr: msg.sender,
20                                 amount: msg.value});
21     totalUsers += 1;
22     owner.send(msg.value/10);
23
24     while (this.balance > users[paying].amount * 2) {
25       users[paying].addr.send(users[paying].amount * 2);
26       paying += 1;
27     }
28   }}
```

Fig. 4. A chain-shaped scheme.

and the array length is incremented.[11] The contract owner retains 10% of the investment (line 22). With the remaining funds, the contract tries to pay back the previous users. If the `balance` is enough to pay the user at index `paying`, the contract pays the user her investment multiplied by 2 (line 25). After that, the contract tries to pay the next user, and so on until the balance is enough.

In this scheme, a user can foresee exactly how much she will gain, provided that the scheme keeps running; the amount is proportional to what she has invested. Examples of this kind are `Doubler`, `DianaEthereum`, and `ZeroPonzi`.

Waterfall schemes are similar to chain shaped-schemes for the user ordering, yet different for the logic of money distribution. Each new investment is poured along the chain of investors, so that each can take their share. Since the logic is first-come first-served, and the distribution starts always from the beginning of the chain, the users later in the chain are likely to never get any money.

We show in Fig. 5 an archetypal scheme of this kind, with an entry toll of 1*ETH* (line 19), 10% fees for the owner (line 24), and a payout of 6% of user investments at each turn. The payout logic starts at line 27. If the contract `balance` is enough to pay the first user in the array (at position `pos = 0`), then the contract sends to that user 6% of her original investment (lines 29–30). After that, the contract tries to pay the next user in the array, and so on until the balance is exhausted. On the subsequent investment, the array is iterated again, still starting from the first user.

To ensure that all users receive payouts (coherently with requirement **R3**), the investments of new users must grow proportionally to the number of users. Examples for this kind of scheme are `TreasureChest` and `PiggyBank`.

Handover schemes are an instance of chain-shaped scheme, where the entry toll is determined by the contract, and it increased each time a new investor joins the scheme. The toll of a new investor is given *in full* to the previous one: since the entry toll is increasing, the previous investor makes an instant profit. At each moment, there is only one investor who is receiving money, and as soon as she is paid, she hands that privilege over the next user.

An archetypal example is shown in Fig. 6. To join the scheme a user must send at least `price` *ETH* to the contract, hence triggering the fallback function of line 11. The contract forwards that sum to the former `user`, minus a fee which is kept within the contract (line 13). Then, the address of the new user is recorded (line 14), and the `price` is doubled (line 15). The contract owner can withdraw her share by calling `sweepCommission`.

In handover schemes, at the time of the investment users know exactly how much they will gain. However, since the toll increases as the scheme goes on, later users are more likely to lose their money (coherently with requirement **R4**). A paradigmatic representative of handover schemes is `KingOfTheEtherThrone`.

4.2. Analysis of money redistribution

Ponzi schemes have the peculiarity that each investor can make a profit, provided that *enough* investors invest *enough* money in the contract, after him (requirement **R3**). Focusing on the kinds of schemes identified in the previous section, we now study how many users an investor must wait for, and how much they must invest to make her (say) double her money.

Chain-shaped schemes. Consider a chain-shaped scheme which *doubles* the received money, accepts entry tolls of exactly 1*ETH*, and has no owner fees except the first 1*ETH* sent to the contract. Let us assume that the first user U_1 sends 1*ETH*. Her money is given to the owner, and so it is removed from the contract, whose balance is 0. For U_1 to see back her 1*ETH* plus the other one promised (since the contract doubles the investment), he must wait for two others users U_2 and U_3 to join the scheme, by sending 1*ETH* each.

In Fig. 7, each node represents one user, and its children are the users needed to redeem her share. So, U_2 must wait for U_1 to

[11] In Solidity, dynamicarrays can be resized by changing the `length` member.

```
contract WaterfallPonzi {

  struct User {
      address addr;
      uint  amount;
  }

  User[] public users;

  uint pos = 0;
  uint public totalUsers=0;
  address public owner;
  uint public fees = 0;

  function WaterfallPonzi() {
    owner = msg.sender;
  }

  function() {
    if (msg.value < 1 ether) throw;

    users[totalUsers] = User({addr: msg.sender,
                              amount:msg.value});
    totalUsers += 1;
    fees = mgs.value / 10;
    owner.send(fees);

    pos=0;
    while (this.balance >= users[pos].amount
        *6/100 && pos<totalUsers){
      users[pos].etherAddress.send
        (users[pos].amount * 6/100);
      pos += 1;
}}}
```

Fig. 5. A waterfall scheme.

```
contract HandoverPonzi {
  address owner;
  address public user;
  uint public
    price = 100 finney;

  function HandoverPonzi() {
    owner = msg.sender;
    user = msg.sender;
  }

  function() {
    if (msg.value < price) throw;
    user.send(msg.value * 9 / 10);
    user = msg.address;
    price = price * 2;
  }

  function sweepCommission(uint amount)  {
    if (msg.sender == owner) owner.send(amount);
}}
```

Fig. 6. An handover scheme.

redeem her share, and then he must wait for U_3 and U_4 to send money (hence he has to wait a total of 3 users). User U_3, who is the last one on her level, must wait that the subsequent level is full, which gives a total of 4 users to wait. In general, a user U_k at level i must wait that all those users on the previous level have redeemed their share, and then he must wait for all the ones on her level that have arrived before him. If U_k is the first node at level i, he must wait for all the other users at level i to join, plus the two ones needed to redeem her share. This needs $2^i - 1 + 2$ users. Since the amount of nodes up to level $i - 1$ is $2^i - 1$ and since U_k is the first at level i, we have that $k = 2^i$ and hence, in the best case, U_k must wait $k + 1$ users. Instead, if U_k is the last user at level i, he must wait for all the other users at level $i + 1$. This needs 2^{i+1} users. Since, $k = 2^{i+1} - 1$, in this case U_k must wait for $k + 1$ users. For instance, a user joining the scheme at position 3 must wait 4 new users, and to invest 4*ETH*, in order to double her investment; instead, a user at position 50 must wait for 51 new users.

Although this simple example considers a scheme with no fees and a fixed investment for each user, the general considerations about the chances of redeeming one's investment remain true for all the contracts in our collection. In a contract which poses no limits on how much one can invest, an unusually high investment could make the contract stop sending payouts for a lot of time, while accumulating the payout, thus discouraging new users to join. Also, higher owner fees and higher multiplying factors will slow the flow down so that our results constitutes a lower bound to the number of users to be waited. For instance, the chain-shaped scheme `Doubler2` doubles the invested amount, asks a minimum investment of 1*ETH*, and 10% fee. The contract has paid out only up to the 68th user out of 210. Looking at its transactions, we see that the most common toll is between 1 to 5*ETH*, but here and there, there are some higher ones (up to 50*ETH*) which make the system very slow to fill up a level.

Tree-shaped schemes. The considerations above hold as a lower bound also for tree-shaped schemes, since they are slowed up by the fact that new users could not all be descendant of a given node.

Waterfall schemes. Assume a waterfall scheme with a fixed toll of 1*ETH*, no fees, and which gives each user 10% of the amount invested. For each new investor, the old ones are entitled of 0.1*ETH*: hence, 10 new users are needed to repay the investment of the first user, and further 10 users to let her double her investment. Note that for the first 10 users, the amount they are giving is not entirely distributed: a part is left within the contract. However, after the 10th investor joins the chain, the money she is giving is not enough to be shared among *all* the users: from that moment on, the contract must use its own funds to fill up the gap. Eventually, also this amount will end: as the scheme goes on, no matter how many other investors will join, only the first 10 users are guaranteed to receive shares.

So, to ensure that *each* user can double her investment, we must make sure that investments are spread over all the users. Let us now assume that, to join the scheme, a user must give 0.1*ETH* times number of users already in. So the first user will invest 0, and the 11*th* user will invest 1*ETH*. With such a rule, if the scheme contains n users, the kth user has given $0.1(k-1)ETH$ while receiving $0.1(n-k)ETH$. As the number of users grows, also the received money grows. For instance, the 3rd user joining the scheme must invest 0.2*ETH*, and will receive 0.4*ETH* as soon as 4 other users join in. Instead, the 50th user must give a toll of 4.9*ETH*, and must wait 98 new users to double the investment. However, since the toll increases for each new investor, entering the scheme is less appealing as the scheme goes on.

Handover schemes. In handover schemes, for an investor to receive a payout it is enough to wait one other user to join. However, since the toll keeps increasing, entering the scheme is less appealing as the scheme goes on.

Overall, we have shown that the requirements **R3** and **R4** hold for all the kinds of schemes we have identified: namely, each investor is guaranteed to earn money if enough money are invested afterwards, but late investors have a greater the risk of losing their investment.

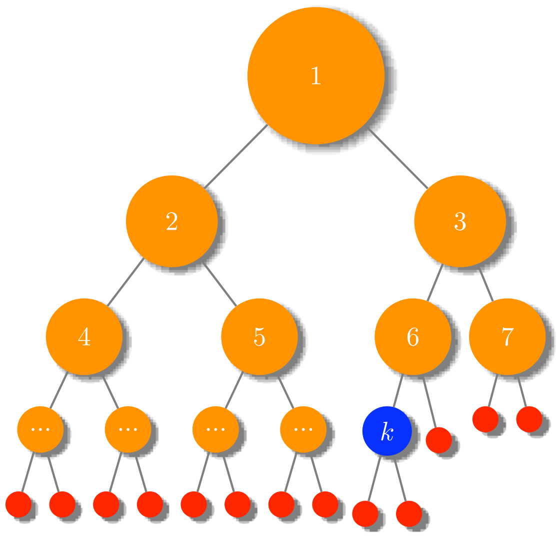

Fig. 7. Payout tree for a scheme which doubles the invested money and accepts exactly 1*ETH* from each user. The first ether is given to the owner.

4.3. Security issues

In this section we highlight several vulnerabilities we have encountered in the implementations of Ponzi schemes, which undermine their advertised behaviour. We will organize vulnerabilities according to their effects: those harming investors, and those harming the scheme itself.

Harming investors. Some vulnerabilities are due to bugs of the code, which is some cases seem plain intentional: they harm users while being profitable for the owner of the scheme. The most common vulnerability is caused by an improper use of the `send` primitive, whose security issues have been already pointed out in [22,23]. If a `send` fails, it returns an error code: if a contract does not check this error, it cannot acknowledge that there has been a problem. So, in case of errors during the `send`, the money remains within the contract, while the user does not receive anything. Notably, the large majority of the contracts we have analysed do not check that the ether transfer succeeds. Their code is similar to that in Fig. 3 (line 27), Fig. 4 (line 22), Fig. 5 (line 29) and Fig. 6 (line 13). This vulnerability is known, at least, since February 11st 2016, when the owner of `KingOfTheEtherThrone` realized that there was too much ether left on his contract[12]

Another issue affects many of the contracts which require a lower bound on the entry toll. If the toll is not met, the user is not allowed to join the scheme, and the sent amount should be returned. However, some contracts (e.g., `DynamicPyramid`, `GreedPit`, `NanoPyramid`, `Tomeka`), forget to return it to user, and keep the amount by themselves (see e.g. Fig. 8 left). This is a questionable, especially when the minimum amount is quite relevant (e.g., in `Tomeka` the minimum is 1 *ETH*).

Another bug that benefits the owner is in `PiggyBank`.[13] According to its advertisement,[14] this is a waterfall scheme, where the owner keeps 3% fees, and each user receives 3% of their investment every time a new user joins the scheme. Hence, the command to compute the owner fees should be as follows:

```
fees = amount/33
```

[12] Source: www.reddit.com/r/ethereum/comments/44h1m1/.

[13] Source: www.reddit/piggybank_earn_eth_forever.

[14] Source: bitcointalk.org/topic=1410587.0.

However, the actual command used in `PiggyBank` is just a little different:

```
fees += amount/33
```

This difference is subtle to spot, but relevant: indeed, the second command makes the fees grow at each deposit, and consequently the owner share subtracted to each investment steadily increases. In practice, the fees calculated for the 7th deposit have exceeded the deposit itself. Beside this, the global variable used to scan the array (`pos` in Fig. 5) is not reset, unlike in line 25. Hence, at each deposit, the iteration does not go from the *first* user to the last one, but from the *last* to the last itself. Hence, only one user at each deposit is paid, and only once. Notably, the conjunction of these two bugs results in giving (almost) all the money invested to the owner. Were only the second bug present, the contract would have kept accumulating a lot of unredeemable ether.

Besides bugs hidden in the code, other dangers for users come from functions which allow the owner to perform special operations, which can make the contract stray from its expected behaviour. One example is in `DynamicPyramid`, where the owner can change the interest rate, and also his fee shares (see Fig. 8, right). Other cases are in `Doubler3` and `TheGame`, where the owner can withdraw all the money in the contract (not only his share, see Fig. 9, left), draining the amount to be given back to investors. Further, some schemes have a `selfdestruct` function that can be called only by the owner, and terminates the contract (see Fig. 9, right). When this happens, investors lose their money.

Harming the scheme. Even when `send` commands are checked, an improper handling of their return value can backfire, and can expose the scheme to Denial-of-Service attacks or blackmailing. An example is `HYIP` (see Fig. 10), a waterfall scheme where investors are recorded in an array, and they are all paid at the end of every day. The scheme checks that each `send` is successful: in case of errors, it throws an exception. However, any error in one of the `send` (lines 25 and 31) will revert *all* the ether transfers. Errors may happen, e.g., for the following reasons: (i) the array of investors grows so long that scanning it causes an out-of-gas exception; (ii) the balance of the contract goes to zero somehow in the middle of the `for` command (line 28), having not paid all the investors; (iii) one of the investor is a contract, whose fallback raises an exception. By exploiting the last issue, an attacker could create a contract with a fallback which always `throws` (see e.g., `Mallory` in Fig. 10). The attacker contract sends a fraction of ether to `HYIP` to enter in the array of investors; when `HYIP` tries to send her the payout, the invoked fallback throws an exception. Note that there is no way to cancel `Mallory` from the array, hence `HYIP` is stuck, and its balance is frozen forever. At this point, the attacker could blackmail `HYIP`, asking for money to stop the attack (via `stopAttack`, line 21).

Although the unchecked `send` is the most widespread issue, there are other bugs which affect contracts. For instance, `Government`,[15] has a notorious bug, which has been found, so far, only in that contract. `Government` is a chain-shaped Ponzi scheme with a quirk: in addition to the usual way to get back money if enough users keep investing, someone can win a jackpot if no one invests after him for 12 h. The list of users is kept in an array, and when the 12 h have expired, the array is cleared. However, the command used to clear the array had to scan *each* of its elements. At a certain point, the array grew so long that clearing every element required too much gas — more than the maximum allowed per single transaction. Hence, the contract got stuck, with the legit jackpot winner unable to claim her price.

[15] `Government` is often called "GovernMental" or "PonziGovernMental" on web forums.

```
function init() private{
  //Ensures only tx with 1 ether
  if (msg.value < 1 ether) {
    collectedFees += msg.value;
    return;
  }...
```

```
function changeMultiplier(uint _mult){
  if (msg.sender != owner) throw;
  if (_mult > 300 || _mult < 120) throw;
    pyramidMultiplier = _mult;
}

function changeFeePercentage(uint _fee){
  if (msg.sender != owner) throw;
  if (_fee > 10) throw;
  feePercent = _fee;
}
```

Fig. 8. On the left, rejecting enrolment without returning the fee in Tomeka. On the right, the function used by the owner of TheGame to set multipliers and fees.

```
function Emergency() {
  if (owner!=msg.sender) throw;
  if (balance!=0){
    owner.send(balance);
    balance=0;
  }
}
```

```
function restart() {
  if (msg.sender==mainPlayer) {
    mainPlayer.send(address(this).balance);
    selfdestruct(mainPlayer);
  }
}
```

Fig. 9. On the left, withdrawing all the balance in EthVentures1. On the right, a termination function in TheGame.

```
contract HYIP {
uint constant INTERVAL = 1 days;

struct Investor {
  address addr;
  uint amount;
}
Investor[] private investors;
address private owner;
uint private paidTime;

function HYIP() {
  owner = msg.sender;
  paidTime = now;
}

function() payable {
  investors.push(Investor(msg.sender, msg.
      value) );
}

function performPayouts() {
  if(paidTime + INTERVAL > now)  throw;

  uint fees = (this.balance * 37)/1000;
  if (!owner.send(fees)) throw;

  uint idx;
  for (idx = investors.length; idx-- > 0; ) {
    uint payout =
      (investors[idx].amount * 33) / 1000;
    if(!investors[idx].addr.send(payout))
      throw;
  }
  paidTime += INTERVAL;
}}
```

```
contract Mallory {

  address victim = 0x23...;
  address private owner;
  bool private attack = true;

  //to be created with
  //1wei of balance
  function Mallory() {
    owner = msg.sender;
  }

  function() payable {
    if (attack) throw;
  }

  function invest() {
    victim.send(1 wei);
  }

  function stopAttack(){
    if (msg.sender == owner)
      attack = false;
  }
}
```

Fig. 10. On the left, a snippet of the code of HYIP, a scheme vulnerable to Denial-of-Service attacks. On the right, the corresponding attack.

Another bug concerns the constructor function, which is executed just once at creation time (usually, to initialize the owner of the contract with the address `msg.sender` of the sender of the first transaction). The constructor must have the same name of the contract, but we found four contracts where it has a wrong name: GoodFellas, Rubixi, FirePonzi, and StackyGame. Fig. 11 shows an extract from the first two. On the left, Goodfellas has a function called LittleCactus (line 5) which sets the owner, and then the owner is sent the fees collected so far (line 11). On the right, Rubixi has a function called DynamicPyramid (line 5) which sets the owner (called creator), and then there is a function collectAllFees which can be invoked to send the fees to the owner (line 11). Giving a wrong name to the function meant to be a constructor is harmful: the function does not qualify to be a constructor at all, and it can be invoked by anyone at anytime, hence changing the owner address. When users discovered the bug, they started to invoke these functions to obtain the ownership and redeem the fees.

To conclude this list of issues, we illustrate a simple trick that can be performed to shut down a chain-shaped scheme. To illustrate it, we consider Doubler, which sends back the amount multiplied by two. To perform the attack, Oscar needs to invest a large amount of ether (say, 100*ETH*). Oscar first sends 100*ETH* to the contract, and then additional 100*ETH* (plus some

```
contract Goodfellas {

  address public owner;

  function LittleCactus() {
    owner = msg.sender;
  }

  function enter() {
    ...
    owner.send(collectedFees);
    ...
  }
```

```
contract Rubixi {
  address private creator;

  //Sets creator
  function DynamicPyramid() {
    creator = msg.sender;
  }
  //Fee functions for creator
  function collectAllFees()  {
    if (collectedFees == 0) throw;
    creator.send(collectedFees);
    collectedFees = 0;
  }
```

Fig. 11. Constructor bug in `Goodfellas` and `Rubixi`.

fees).[16] Upon receiving the second slot, the scheme will pay all the 200*ETH* back to Oscar, so he does not lose anything. From that moment on, all the subsequent investments will be gathered to pay back the second 100*ETH* of Oscar. If the average invested amount is smaller than 100*ETH*, a large number of investors (and a lot of time) are needed to pay back Oscar: hence, the scheme will not be able to pay out other investors for a while. Since the success of these schemes is based on the fact that they are fast to pay out, it is likely that with this attack, the scheme will be abandoned. This attack can be performed at any time to disincentivize users to join a chain-shaped scheme.[17] If performed at an early stage of the lifecycle of the scheme, the attack succeeds with a negligible loss of money.

5. Impact of Ponzi schemes

In Table 1 we draw some general statistics about all the 184 contracts in our collection, and we give details about the first 10 contracts in our list, ordered by total amount of invested ether. Full data about the collected Ponzi schemes, including their unique addresses, are reported online at `goo.gl/CvdxBp`.

The columns in Table 1 contain the number of incoming and outcoming transactions, and the overall transferred value, both in *ETH* and in *USD* (rounded to an integer). To convert the amount of each transaction to *USD*, we use the average exchange rate on the day of the transaction, obtained from `etherscan.io`.[18] The value transferred through a transaction has a different meaning, according to whether the transaction is external or internal:

- *external* transactions are created by users to invoke contract functions. These transactions can transfer some ether from a user to the called contract. Hence, this amount of ether is part of the *inflow* of the contract (i.e., its incoming ether).
- *internal* transactions are of two kinds: the ones sent *to* the contract under observation, and the ones sent *from* it. The first case happens when, instead of sending her money directly to a Ponzi scheme, a user goes through another contract (typically, a wallet contract): hence the amount linked to the transaction is part of the *inflow* of the contract. The second case happens when the contract sends a payout to some user: in this case, the transaction amount is part of the *outflow* of the contract.

Note that, similarly to [24] for Ponzi schemes on Bitcoin, also in Ethereum we cannot precisely quantify the profit of scammers. Indeed, it is not clear how to define *who* is the scammer: of course the contract owner can be considered the originator of the scam, but he may have more than one addresses through which redeeming money. Hence, we do not know how to separate the money sent to legit users from the money sent to scammers. A rough over-approximation of the profit of scammers is the total inflow of the scheme.

The columns "Paying users" and "Paid users" in Table 1 indicate, respectively, the number of users who entered the scheme (i.e., the distinct addresses that send money to the contract), and the number of users that have received a payment from the contract.

The columns "*USD*" and "*ETH*" in Table 1 give a first measure of the economic impact of Ponzi schemes on Ethereum. Notice that *ETH* alone is not significant as a unit of measure: actually, the exchange rate from *ETH* to *USD* has been highly volatile, as shown by the diagram in Fig. 12. Overall, we observe that the Ponzi schemes in our list collected 630 662*USD* from 2378 distinct users. While the difference between incoming and outgoing *ETH* is always non-negative (as contracts cannot send more *ETH* than what they receive), the difference between incoming and outgoing *USD* can be negative. This is not a contradiction: it can be explained by the fact that the exchange rate between *ETH* and *USD* has varied over time, as depicted in Fig. 12.

5.1. Statistics by schemes kind

In Table 2 we display the statistics about the impact of Ponzi schemes grouped by scheme kind, according to the taxonomy in Section 4. The columns show: the number of schemes for each kind; the amount of incoming and outgoing *ETH* to the contract, and the corresponding values in USD; the number of users that invested in the scheme and that have received some payout; the *ratio* between these two values.

We see that, out of all 184 contracts, almost the totality are chain-shaped schemes (151). There are 12 schemes between waterfall, tree-based and handover kinds, while a portion of 21 schemes has not found place in any of the analysed categories. Mostly they are experiments, or variants of existing schemes, or just singularities in their own way.

The source code of Ponzi schemes falling in the same category features only slight differences between distinct instance: most contracts only differ in the multiplication factor, in the applied fees, or in the presence of auxiliary functions, like e.g. fields getter/setter, or other utility functions for the owner. The low variance in the code of Ponzi schemes is also witnessed by the average normalized Levenshtein distance among their bytecode, which is 0.54, far less than 0.79, the average distance between the bytecode of two *arbitrary* contracts. This may suggest that, after the first Ponzi schemes have been created, the subsequent ones have been obtained by adapting the existing instances.

[16] To guarantee the atomicity of the sends, Oscar sends the money through a contract.

[17] As far as we know, this attack has been performed only on contract `Quadrupler`. See etherscan and bitcointalk for details.

[18] Source: https://etherscan.io/chart/etherprice.

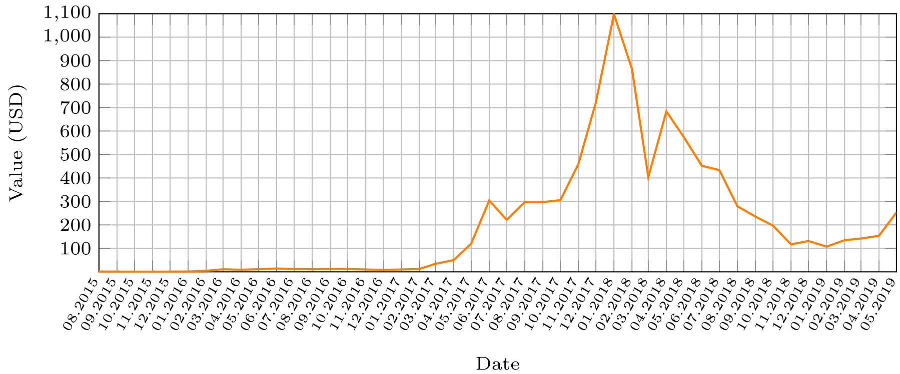

Fig. 12. Ether/USD exchange rate (monthly average).

Table 2
Statistics by kind of scheme.

Kind	#Num	ETH		USD		Users		
		In	Out	In	Out	Paying	Paid	%
Tree-shaped	4	410	400	1 429	1 416	161	83	51%
Chain-shaped	151	41 514	40 170	587 086	599 347	1967	968	48%
Waterfall	4	452	444	8 836	11 261	111	82	73%
Handover	4	486	483	2 618	3 124	97	63	64%
Other	21	1 017	933	30 693	87 728	42	36	85%
Total	184	43 881	43 332	630 662	702 878	2378	1232	51%

6. Measuring gains and losses

We now study the distribution of gains and losses among users. We expect to observe the common pyramidal pattern of Ponzi schemes, where only a few users make profit from their investment, while the vast majority loses money.

We start our analysis by considering two contracts: `Doubler2` (a chain-shaped scheme) and `Etheramid1` (a tree-shaped scheme).

Fig. 13 shows the gain in ether for each user who entered the scheme `Doubler2`. A negative amount indicates a loss. The graph shows that the vast majority of users has a balance close to zero, while just a few of them have a substantial gain, with a peak of 486*ETH*. The integral of the graph is close to zero, i.e. the contract has redistributed almost all the money it has received.

We now analyse the *gain ratio*, i.e. the ratio between received ether versus invested ether. According to advertisement, which promises to double the investment, each user should have a ratio of exactly 2. Instead, Fig. 14 shows that, out of a total of 210 users, 142 never received any money back (the ones with label 0); 23 have a gain ratio between 0 and 1 (meaning that they barely were able to regain what they had invested); 44 have a ratio between 1 and 2, and only one has a very high ratio (486).

Fig. 15 (left) shows the users with the highest gains. The user who earned 486*ETH* has invested only 1*ETH*. Subsequent users have a gain ratio strictly less than 2: this means that to have such an high gain, they had to invest a lot of ether. Since `Doubler2` is a chain-shaped scheme which doubles the investment, from Fig. 15 we infer that the gaining users have invested more than once, and sometimes the contract has not given the promised ether back.

Fig. 13 shows the gain of each user in `Etheramid1`. As before, the majority of users lost their money. Only a few users gained a little (up to 5*ETH*), and exactly one had an high income of 30*ETH*. Fig. 14 shows a behaviour similar to that of `Doubler2`. From Fig. 15 we see that two users have a peculiar gain ratio: one received 7*ETH* upfront an investment of 1*ETH*, and another one (probably, the initiator of the scheme) received 30.6*ETH* without investing money.

We now consider the other contracts in our collection. In Appendix A we show the diagrams of user gain, gain ratio, and inflow–outflow, on a selection of 23 contracts, chosen among those with the most interesting features, e.g. high volume of payments, number of users, or number of transactions. The users-gains diagrams show a similar pattern to `Doubler2`, notwithstanding the differences in number of users, scheme type and volume of ether exchanged of the contracts we have selected. In general, the curve is quite shallow in the centre with a narrow positive peak on the far right, and it stays mainly below the *x*-axis. This means that a lot of users lose money; a few gain something, and even less have extremely high incomes. Also, the difference between inflow and outflow is almost zero — if we include the owner among the users.

From the gain ratio diagrams in Appendix A, we observe that the most numerous classes are those of users who never received any money back, or have a ratio between 0 and 1. This means that the majority of users could not gain anything. The percentage of users not gaining anything is on average around 70% among the contracts in our selection. In particular, Figure A.34 shows that the most unfair contract is `Doubler` (where 88% of users do not gain anything), followed by `ShinySquirrels` (87%) and `GreedPit` (85%).

From the joint analysis of the gain ratio diagrams and of the users with highest gains (Figure A.33), we see that just one or two users per contract have exceptionally high revenues, and that, in some cases, they have not invested money. Generally, the exceptional high gains belongs to the owner, and are due to the contract fees. In some cases, there is more than one owner: e.g., `EthereumPyramid` has two owners, and `Rubixi` has six. However, the case of `Rubixi` is singular: a bug in the code allowed users to steal the role of the owner and hence to receive the fees. In other cases, the scheme has no owner fees, and hence the ratio graph is more levelled (e.g., in `ZeroPonzi`, 59% of users do not gain anything), as the absence of fees allows for refunding more users.

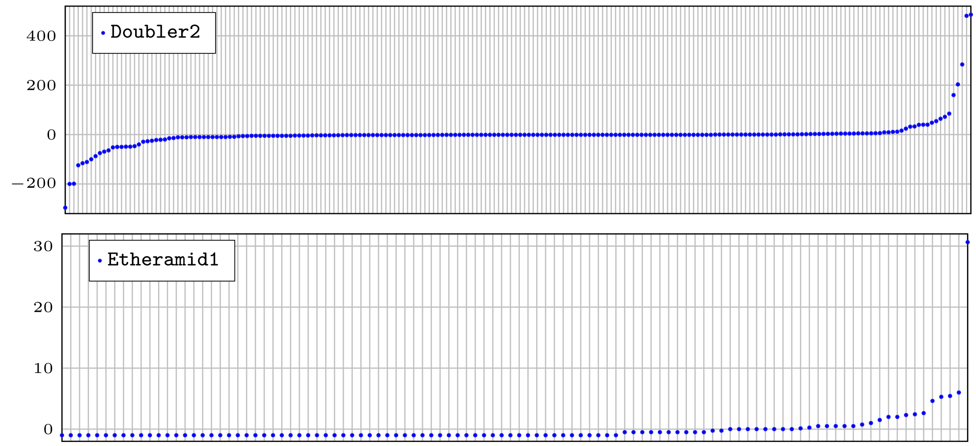

Fig. 13. Gain in *ETH* (y-axis) per user (x-axis).

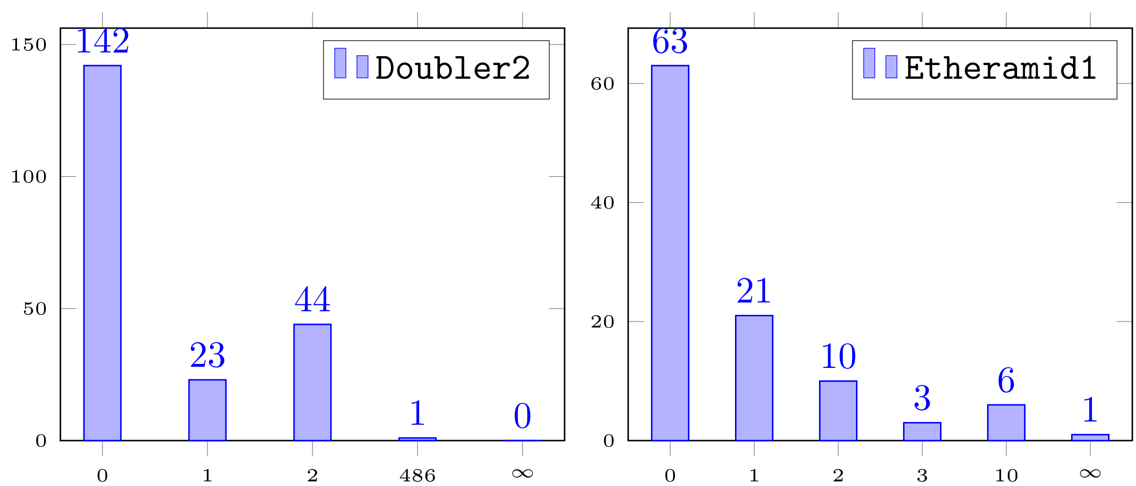

Fig. 14. Number of users grouped by gain ratio for Doubler2 and Etheramid1. Label *0* means no money has been received; for $n > 0$, label n indicates a ratio between $n-1$ (strict) and n (included). Label ∞ indicates users who have never sent anything but have received something.

Doubler2				Etheramid1			
Invested	Received	Net gain	Ratio	Invested	Received	Net gain	Ratio
1.0	486.8	485.8	486	0	30.6	30.6	∞
601	1081.8	480.8	1.2	1	7	6	7
370.6	654.5	283.9	1.7	1	6.4	5.4	6.4
254	457.2	203.2	1.7	1	6.2	5.2	6.2
200	360	159.9	1.8	2	6.6	4.6	3.3

Fig. 15. Details of the 5 users who have gained most for Doubler2 and Etheramid1. Each row represents one user. Values are in *ETH*, cut to the first decimal non-zero digit. Columns show: (1) how much the user has sent to the contract; (2) how much she has received, (3) difference between columns 2 and 1; (4) gain ratio. The ratio of users who have sent nothing is denoted by ∞.

7. Evolution over time

In this section we study how Ponzi schemes behave over time. In particular, in Section 7.1 we analyse the timing correlation between inflow and outflow; in Section 7.2 we analyse the lifespan of Ponzi schemes; in Section 7.3 we observe how the volume of payments evolves over time.

7.1. Inflow and outflow correlation

We now study the behaviour of inflow and outflow transactions over time. Similarly to the previous section, we first analyse the contracts Doubler2 and Etheramid1, before discussing the general pattern.

Fig. 16 shows that Doubler2 was active for 6 months, with most of the activity concentrated in the first month. We see a correlation between the inflow and the outflow: each outflow (red dot) is preceded by a sequence of inflows (blue dots) of smaller amounts. This is because chain-shaped schemes gather funds, and then pay back a single user in a single shot. Hence, the red dots are higher in value, but less in number. The red dots from 06–05 correspond to the contract owner withdrawing the fees. Etheramid was active only for 1 month, with 5 days of intense activity. Also in this case there is a strong correlation between inflow and outflow transactions: however, the pattern is different from Doubler2, since new investments (blue spots) are immediately distributed among (some of) the users. This results

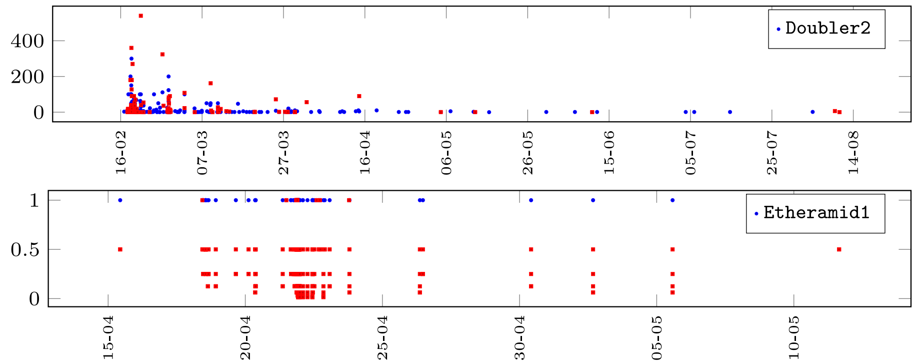

Fig. 16. Inflow (blue) and outflow (red) timing: on the *x* axis, the time of transactions (day-month); on the *y* axis, the amount of ether received/sent. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

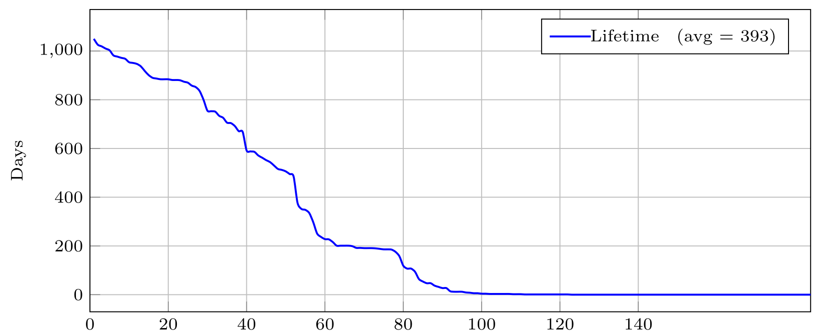

Fig. 17. Lifetime of Ponzi schemes. On the *x*-axis, the number of contracts; on the *y*-axis, their lifetime measured in days.

in a set of red dots in decreasing value whenever there is a blue dot.

The diagrams of the other contracts in our selection are in Appendix A, from which we still see the strong correlation between inflow and outflow. The actual correlation depends on the category of the Ponzi scheme: a tree-shaped one spreads each investment as soon as it receives them (e.g., `Etheramid1`), while a chain-shaped scheme waits to have enough balance. Other minor differences in the diagrams depend on contract peculiarities. For instance, some contracts also implement a lottery, so that there is a jackpot winner which results in an unexpected red dot (e.g., `Government`). Other contracts are designed so that the owners can withdraw fees at their wish, and again in this case, the result is red dots in a zone with no blue ones. In general, most contracts have a very short lifespan, with a peak of intense activity followed by almost no activity at all.

7.2. Lifetime

We now study the lifetime of the Ponzi schemes in our collection. Fig. 17 displays, in blue, the lifetime measured as the number of days from the first to the last inflow or outflow transaction of the contract. We see that $\sim$ 60% of Ponzi schemes have a lifetime close to 0 days. Basically, this means that they were deployed on the Ethereum blockchain, and in many cases advertised in forums or dedicated web sites, but they did not manage to attract any users.

Note that using the last transaction of a contract to measure its actual period of activity may be too coarse. Indeed, our overall diagrams show that Ponzi schemes are characterized by an high number of transactions operated in a short time frame, followed by a period of isolated transactions, and inactivity.

Fig. 18 shows how many Ponzi schemes have been created over time. We see a peak in April 2016, with 91 new public Ponzi schemes. After this first wave of creations, the situation has settled, with an average of $\sim$ 3 new public schemes per month. In Section 9 we discuss possible explanations for this fall in the creation of Ponzi schemes, and in particular we conjecture that, rather than disappearing, they are evolving into something more difficult to classify.

7.3. Volume of payments

In this section we study how Ponzi schemes perform over time. Fig. 19 shows the daily volume of payments (measured in *USD*) of all the 184 Ponzi schemes in our collection. The *x*-axis represents time, and the *y*-axis gives the volume of money transferred (measured in *USD*). The red dashed line represents money sent by users to the schemes, while the blue solid line represents money sent by the schemes to users. The diagram clearly reports an equilibrium between outcoming and incoming flows, meaning that most of the money invested in the schemes are redistributed to users. However, the distribution of money follows the pattern of inequality that characterizes Ponzi schemes, as highlighted in Section 6, and further discussed later on in Section 8.

From Fig. 19 we observe that most value was exchanged in the period from February to May 2016, with three peaks between March and April 2016. It is plausible that the fall of activity after April 2016 is a consequence of the analogous drop in the creation of new Ponzi schemes, witnessed in Fig. 18.

We now measure the volume of transactions pointwise, on a sample of the most representative schemes. Each diagram in Fig. 20 shows the money flow (in and out) of a single contract: the red dashed lines represent money sent to the scheme (measured in *USD*), while the blue solid lines represent payouts sent by the scheme to users. The *x*-axis represents time: we consider the total incoming/outgoing money per day.

In the diagram for `DynamicPyramid`, we see that the most of the incoming flow happened on the 11st of March, the total

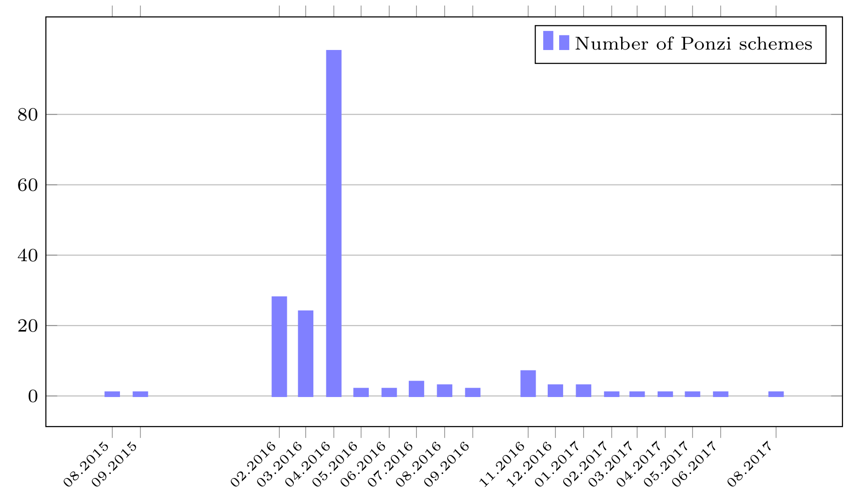

Fig. 18. Number of Ponzi schemes created by month.

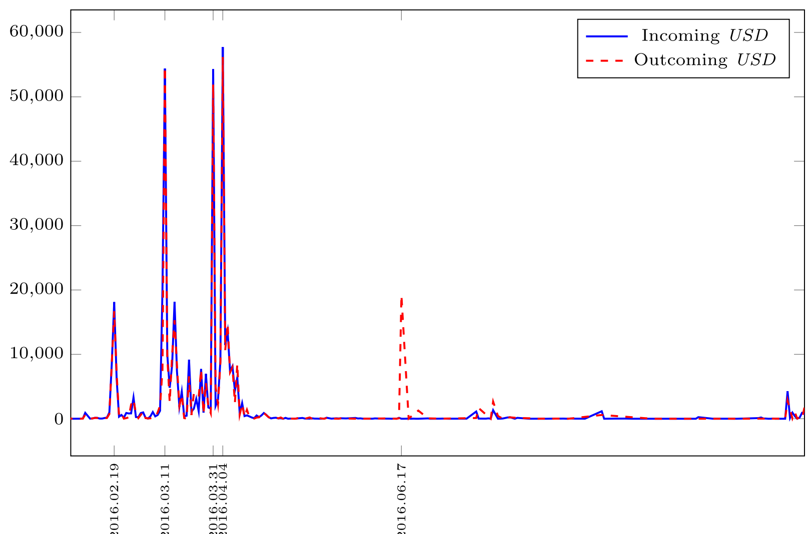

Fig. 19. Daily volume of transactions (complete set of 184 Ponzi schemes).

investments in that day alone amount to almost 60 000 *USD*.[19] We see that the blue and red flows in that day almost overlap, meaning that with such a great balance the contract was able to pay out many users. However, after that single peculiar day, users almost stopped sending ether, and so did the contract.

The diagram of `Government` is peculiar, due to a bug which affected it (already discussed in Section 4.3). This contract needs to periodically clear the array which records the list of users. However, from a certain point performing this operation would have required more gas than the maximum allowed for a single transaction. Several attempts to clear the array and to redeem the funds stored in the contract have failed with an "out-of-gas" exception. Exactly in the date of the first hard-fork (on the 17th of June, 2016), which also raised the gas limit,[20] we observe an internal transaction of 22 699 *USD*, used to withdraw the funds and correctly clear the array.

The diagram for `EthereumPyramid` shows that the investments were made basically in two slots of time: one around the last days of February 2016, and another on a single day, the 1st of April. All those later investments were strangely made by the owner: they were almost 50 in a single day. We see that the inflow and outflow almost overlap. `EthereumPyramid` asks to all users exactly 1*ETH* and triples the investment. With such a fixed toll, one user every three is paid out, and the outflow is smooth. However, we see that there is a peak in the outflow around the 26th of June, and in that day we observe a single payment of 90*ETH*. After inspecting the code and the set of transactions, we are inclined to say that it is the owner withdrawing her fees.

From the diagram of `Etheramid` we see a perfect overlap of inflow and outflow: indeed, this is a tree-shaped scheme, so everything which goes in is immediately sent to the users' ancestors. There is no need to delay payments waiting for the payout to reach its quote, like in chain-shaped schemes (e.g., see the diagram of `Doubler2`).

8. Measuring payment inequality

Our last analysis measures the inequality in the distribution of investments and revenues for the schemes in our sample. To this purpose we use *Lorenz curves* (Figs. 21 and 22) and *Gini coefficients* (Fig. 23), two standard graphical representations of the distribution of income or wealth.

[19] Source: etherscan.io/address/0xa9e4e3b1da2462752aea980698c335e70e9ab26c.

[20] Source: `blog.ethereum.org/2016/07/20/hard-fork-completed/`.

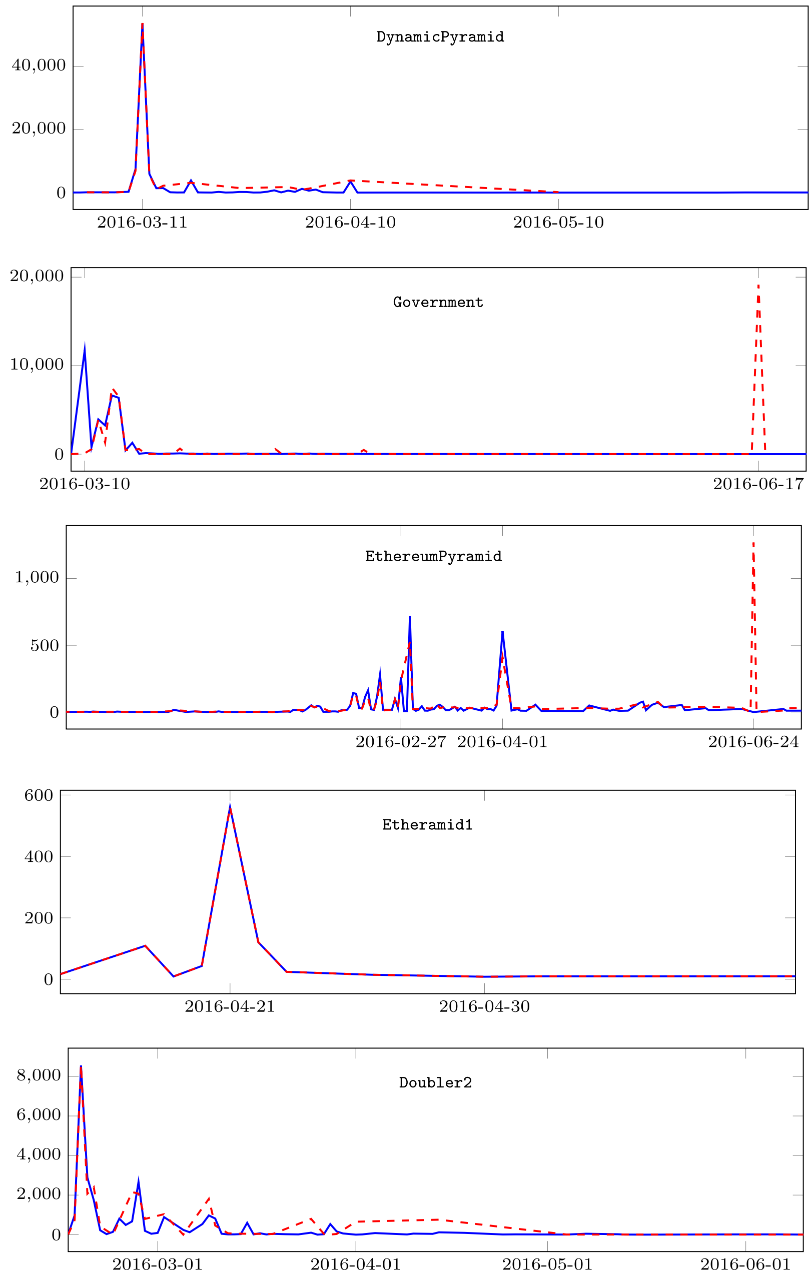

Fig. 20. Volume of payments into and out Ponzi schemes, by time. On the x-axis, the dates of transactions; on the y-axis, the *USD* sent to (blue solid line) and from (red dashed line) the contract.

The Lorenz curves represent users on the x-axis (in percentage), and on the y-axis the percentage of payments *into* (Fig. 21) and *from* (Fig. 22) the Ponzi scheme. A diagonal line at 45 degrees from the two extremes of the diagram (leftmost-bottommost to rightmost-topmost) represents the perfect equality: i.e., for all $x \in [0, 100]$, the x% of the whole population of users has invested/received the x% of the total income of the scheme. Instead, the perfect disequality is represented by the (discontinuous) function that has value 0 for all $x < 100$, and value 100 for $x = 100$: this means that a single user has invested/received the total sum in the scheme.

We can observe in Fig. 21 that `Etheramid1` is quite close to perfect equality, while the most unbalanced schemes in our sample are `Government` and `ProtectTheCastle`, where 10% of victims have invested more than 90% of the money. The Lorenz curves of these two schemes are quite close to the overall curve of Bitcoin-only Ponzi schemes in [6]. Overall, the closer is a curve to the one which represents perfect inequality, the more a Ponzi scheme benefits from "big fishes" who invest large amounts of money in the scheme; dually, if the curve is close to the one which represents perfect equality, the scheme benefits from a large population of victims who invest a small amount of money.

From Fig. 22 we observe that the distribution of payouts is in general more iniquitous than that of investments, as the Lorenz curves are more squeezed to the right, compared to those in Fig. 21. Interestingly enough, although `Etheramid1` is almost perfectly balanced for investments, the distribution of payouts is quite unbalanced.

The Gini coefficients in Fig. 23 relate the inequality of investments/payouts to the "success" of the scheme, defined as total amount of money invested/received by users. The x-axis represents the degree of inequality (0 indicates perfect equality, while 100 is perfect inequality), and the y-axis measures the total investment/payout. Each scheme is represented by an arrow,

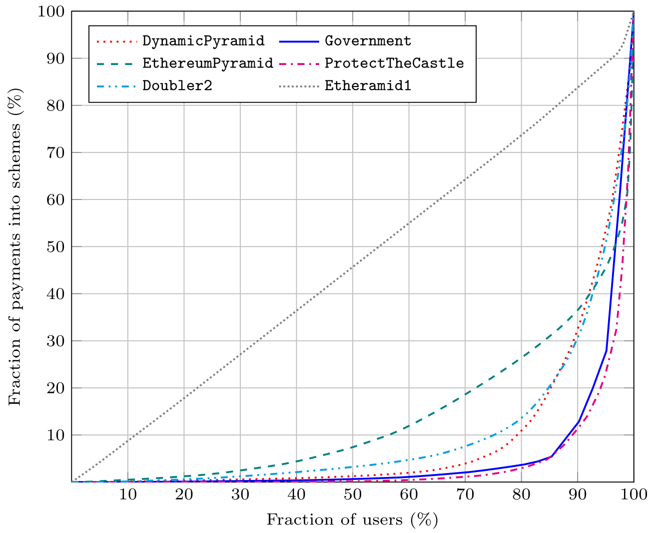

Fig. 21. Lorenz curves of a sample of Ponzi schemes (payments in).

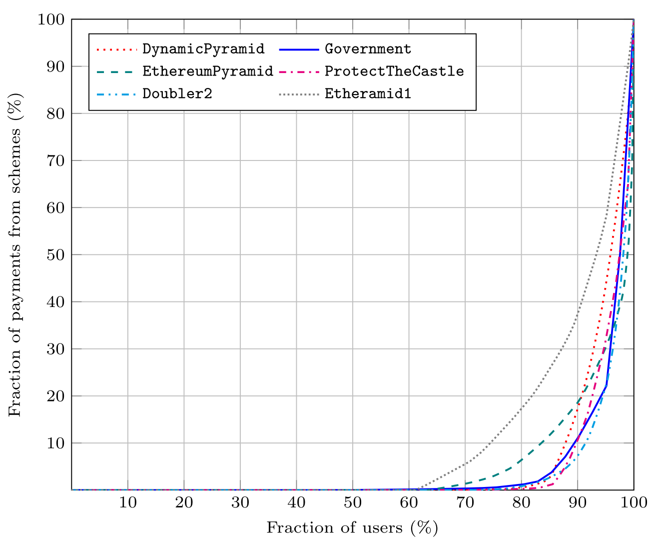

Fig. 22. Lorenz curves of a sample of Ponzi schemes (payments out).

whose tail represents investments, while the head represents payouts. For the most lucrative scheme, `DynamicPyramid`, we observe that the index of inequality is high, surpassing 80% for both investments and payouts. For `ProtectTheCastle`, we see that the head and the tail of the arrow almost overlap, meaning that the inequality distributions of investments and payouts are very close in this scheme. For the less lucrative schemes, no correlation seems to exist between the success of the scheme and the index of inequality.

9. Conclusions

Blockchains and smart contracts might really be the next "disruptive" technologies, as often reported by the media; however, they can also offer new opportunities to tax-evaders, criminals, and fraudsters, who can take advantage of their anonymity and decentralization [25,26]. In this paper we have analysed Ponzi schemes on Ethereum, the most widespread platform for smart contracts so far. Overall, we have observed that, in the first 3 years of life of Ethereum, there have been a multitude of experiments to implement Ponzi schemes as smart contracts. Although the economic impact of these experiments has been quite limited, as they involve only a small fraction of the transactions and value on the Ethereum blockchain, our analysis allows to draw some relevant conclusions, which we summarize below in the form of "recommendations" for users and surveillance authorities.

Recommendation #1: check the advertisements. During our collection activity, we have studied how Ponzi schemes are promoted on the web. In many cases, Ponzi schemes are presented as "high-yield" investment programs, promising high returns and omitting to declare any risks; in some other cases, they are promoted as mere "social games", but a constant factor is that playing involves transfers of money from the user to the contract, and the allurement of making some profits. In many cases, we have found discrepancies between the advertisement and the actual chances of obtaining a payout: the latter is presented as a plain fact, while in Section 4 we have shown that fallacies in the money distribution mechanism or in its implementation might prevent users from obtaining the expected payouts. Further, advertisements usually omit to declare that the contract owner can modify the advertised conditions, e.g. by increasing the owner fees, or destructing the contract.[21]

So, our first recommendation for potential users is to carefully study the advertisement: if the conditions appear too alluring, probably it is a scam. Websites like *BadBitcoin*, which maintains a blacklist of cryptocurrency-based scams,[22] or discussion forums like the "Gambling: Investor-based games" section of *Bitcointalk.org*[23] should be consulted before sending money to a contract. For surveillance authorities, our recommendation is to monitor the web to detect suspect advertisements, and to provide the community with official blacklists.

Recommendation #2: analyse the contract code. Despite one of the main selling points of "smart" Ponzi schemes is that their immutability and decentralized execution makes them "reliable", our analysis in Section 4 has revealed several vulnerabilities, which undermine their trustworthiness. Some of these vulnerabilities are caused by poor programming skills, while some others seem intentional: either should discourage users to join. However, to transmit a feeling of security, contract owners shelter themselves behind the motto that *the code is publicly accessible*, assuming that everyone can read it and assess its reliability. Since bugs are often missed even by their own creators, it is hard to imagine that the average user can read a contract and fully understand what it really does and what harms can be hidden behind. Differently from the notorious vulnerabilities which affected the DAO [27] and the Parity wallet [28,29], which caused money losses in the order of hundreds of millions of dollars, the vulnerabilities discussed here involve smaller contracts: indeed, the vast majority of the contracts in our collection stays in less than 100 lines of Solidity code (for comparison, the DAO was ∼1200 lines).

To counteract these vulnerabilities, researchers have started to develop tools for automatically analysing Ethereum contracts [23,30–33]. These tools manage to detect several common vulnerabilities, even though the Turing-completeness of EVM and Solidity make verification unfeasible, in general. A parallel line of research is the development of *domain-specific* languages for smart contracts (possibly, not Turing-complete), which can help

21 For instance, the advertisement of `EthStick` only warns that "the settings can be changed to adapt to the trends (but only within defined limits)". Actually, from its source code we see that the fees can only be augmented, while the payoff multiplier factor can be only decreased: everything to the sole advantage of the owner.

22 https://badbitcoin.org/thebadlist/.

23 https://bitcointalk.org/index.php?board=207.0.

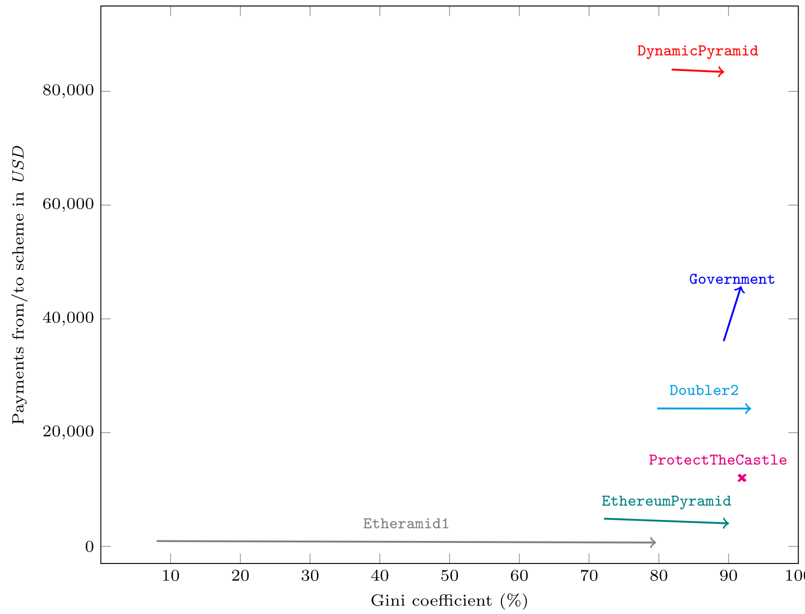

Fig. 23. Gini coefficients of a sample of Ponzi schemes.

to improve the precision of analysis techniques, by reducing the distance between contract specification and implementation. Several domain-specific languages for smart contracts have been proposed, not only targeted to the Ethereum platform. Among them, FSolidM [34] models contracts as finite automata, which can be translated into Solidity code; the works [35–37] develop languages to specify financial contracts in the style of Peyton Jones et al. [38].

Our recommendation for users is to apply these tools to verify, at least, that the contract they want to join does not suffer from vulnerabilities like those discussed in Section 4. While this alone does not guarantee that the scheme is fair, it can guarantee e.g. that the contract owner does not surreptitiously steal funds. Domain-specific languages might allow for more sophisticated analyses, which ideally could verify that the distribution of funds among users is fair.[24]

Recommendation #3: analyse the transaction logs. Our analyses in Sections 6–8 have shown that, despite the many peculiarities, the transaction logs of Ponzi schemes seem to share some general patterns: (i) only a few users have a ratio greater than 1: the most numerous classes are those of users who never received any money back, or have a ratio between 0 and 1; (ii) most Ponzi schemes have a relatively short lifespan, consisting in a peak of intense activity followed by a period of stagnation; (iii) the Gini coefficients of the *payouts* of Ponzi schemes tend to be high (more than 80% in our collection), meaning a strong inequality in the distribution of money. Even though none of these features alone seems enough to separate Ponzi schemes from other contracts, these features can be used together to train classifiers which *automatically* detect Ponzi schemes. Along these lines, a preliminary version of our dataset has been used in [21,40] to experiment with learning strategies to classify Ethereum Ponzi schemes. The classifier in [21,40] uses simple features of transaction logs (e.g., number of payments, contract balance, proportion of investors who received at least one payment, *etc.*.), as well as features of the contracts EVM code (e.g., the number of occurrences of certain opcodes). The measurements in [21,40] show that code features are more discriminating than transaction features — somehow counter-intuitively, since EVM features do not seem to carry any insight on the nature of the contract. However, this discrepancy may be due to an over-simplification in the choice of transaction features: using more sophisticated features, inspired to those discussed in Sections 6 and 7, may help improve the precision of the classification. The analysis techniques of Ponzi schemes in Bitcoin [41,42] demonstrate that the automatic classification of Ponzi schemes from the transaction history alone is feasible with a high level of accuracy.

Future works. The Ponzi schemes we have presented in this paper can be seen as the first wave of Ethereum-based scams. In a preliminary version of this paper that we put online on March 10th, 2017,[25] we had foreseen a second wave of scams:

> "*very likely they will be less recognizable as such than the ones collected in this survey. For instance, they could mix multi-level marketing, token sales, and games, to realize complex smart contracts, which would be very hard to correctly classify as Ponzi schemes or legit investments*"

We believe that this expectation may have come true with Initial Coin Offerings (ICOs), a means of crowdfunding based on the trade of crypto-tokens, through which more than 3*USD* billions have been collected in 2017,[26] as well as crypto-collectibles games like *CryptoKitties* and its followers. The absence of specific regulations in Europe and in the US, and the general difficulty of governing decentralized cryptocurrencies, have made these schemes attractive also for scammers: indeed, a few ICOs have been unmasked as Ponzi schemes by financial authorities [43,44]. A relevant research line for future works could be that of studying these kinds of "pseudo" Ponzi schemes, which share many similarities with Ponzis, although failing to meet the requirements we have specified in Section 3. Some features which apply well to "pure" Ponzi schemes, like e.g. the gain ratio, seem appropriate also to characterize these "pseudo" Ponzi.

24 For instance, BitML is an abstract language for Bitcoin contracts, which supports the verification of fairness for gambling games, like multi-player lotteries [39].

25 arxiv.org/abs/1703.03779v1.

26 Source: www.coinschedule.com/stats.html.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.future.2019.08.014.

References

[1] S. Nakamoto, Bitcoin: a Peer-To-Peer Electronic Cash System, 2008, https://bitcoin.org/bitcoin.pdf.

[2] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J.A. Kroll, E.W. Felten, Sok: Research perspectives and challenges for Bitcoin and cryptocurrencies, in: IEEE S & P, 2015, pp. 104–121, http://dx.doi.org/10.1109/SP.2015.14.

[3] T. Moore, The promise and perils of digital currencies, IJCIP 6 (3–4) (2013) 147–149, http://dx.doi.org/10.1016/j.ijcip.2013.08.002.

[4] M. Artzrouni, The mathematics of Ponzi schemes, Math. Social Sci. 58 (2) (2009) 190–201, http://dx.doi.org/10.1016/j.mathsocsci.2009.05.003.

[5] T. Moore, J. Han, R. Clayton, The postmodern Ponzi scheme: Empirical analysis of high-yield investment programs, in: Financial Cryptography and Data Security, 2012, pp. 41–56, http://dx.doi.org/10.1007/978-3-642-32946-3_4.

[6] M. Vasek, T. Moore, There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams, in: Financial Cryptography and Data Security, 2015, pp. 44–61, http://dx.doi.org/10.1007/978-3-662-47854-7_4.

[7] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9) (1997) https://firstmonday.org/ojs/index.php/fm/article/view/548/.

[8] P.L. Seijas, S. Thompson, D. McAdams, Scripting smart contracts for distributed ledger technology, Cryptology ePrint Archive, Report 2016/1156, http://eprint.iacr.org/2016/1156 (2016).

[9] E.V. Murphy, M.M. Murphy, M.V. Seitzinger, Bitcoin: Questions, Answers, and Analysis of Legal Issues, Tech. Rep., Congressional Research Service, 2015.

[10] A. Juels, A.E. Kosba, E. Shi, The ring of gyges: Investigating the future of criminal smart contracts, in: ACM CCS, 2016, pp. 283–295, http://dx.doi.org/10.1145/2976749.2978362.

[11] V. Buterin, Ethereum: A Next Generation Smart Contract and Decentralized Application Platform, 2013, https://github.com/ethereum/wiki/wiki/White-Paper.

[12] G. Wood, Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014, gavwood.com/paper.pdf.

[13] A. Gervais, G.O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, S. Capkun, On the security and performance of proof of work blockchains, in: ACM CCS, 2016, pp. 3–16, http://dx.doi.org/10.1145/2976749.2978341.

[14] Solidity documentation, https://solidity.readthedocs.io/en/v0.5.4/ (2019).

[15] I. Kaminska, It's not just a Ponzi, it's a "smart" Ponzi, Financial Times Alphaville https://ftalphaville.ft.com/2017/06/01/2189634/its-not-just-a-ponzi-its-a-smart-ponzi/ (2017).

[16] M. William, ERC-20 Tokens, Explained, Cointelegraph, https://cointelegraph.com/explained/erc-20-tokens-explained (2018).

[17] S. Holoweiko, What is an ICO? Defining a Security on the Blockchain. Available at SSRN: https://ssrn.com/abstract=3303447 (2018).

[18] J. Young, CryptoKitties Sales Hit $12 Million, Could be Ethereum's Killer App After All, https://cointelegraph.com/news/cryptokitties-sales-hit-12-million-could-be-ethereums-killer-app-after-all (2017).

[19] M. Bartoletti, S. Lande, L. Pompianu, A. Bracciali, A general framework for blockchain analytics, in: Proc. 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL@Middleware), ACM, 2017, pp. 7:1–7:6, http://dx.doi.org/10.1145/3152824.3152831.

[20] L. Yujian, L. Bo, A normalized levenshtein distance metric, IEEE Trans. Pattern Anal. Mach. Intell. 29 (6) (2007) 1091–1095, http://dx.doi.org/10.1109/TPAMI.2007.1078.

[21] W. Chen, Z. Zheng, E.C. Ngai, P. Zheng, Y. Zhou, Exploiting blockchain data to detect smart Ponzi schemes on Ethereum, IEEE Access 7 (2019) 37575–37586, http://dx.doi.org/10.1109/ACCESS.2019.2905769.

[22] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: Principles of Security and Trust (POST), in: LNCS, vol. 10204, Springer, 2017, pp. 164–186, http://dx.doi.org/10.1007/978-3-662-54455-6_8.

[23] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: ACM CCS, 2016, http://eprint.iacr.org/2016/633.

[24] M. Vasek, M. Thornton, T. Moore, Empirical analysis of denial-of-service attacks in the Bitcoin ecosystem, in: Financial Cryptography and Data Security, 2014, pp. 57–71, http://dx.doi.org/10.1007/978-3-662-44774-1_5.

[25] J. Brito, A. Castillo, Bitcoin: A Primer for Policymakers, Mercatus Center at George Mason University, 2013.

[26] T. Slattery, Taking a bit out of crime: Bitcoin and cross-border tax evasion, Brook. J. Int'l L. 39 (2014) 829.

[27] Understanding the DAO attack, http://www.coindesk.com/understanding-dao-hack-journalists/.

[28] Parity Wallet Security Alert, https://paritytech.io/blog/security-alert.html (July 2017).

[29] A Postmortem on the Parity Multi-Sig Library Self-Destruct, https://goo.gl/Kw3gXi (November 2017).

[30] I. Grishchenko, M. Maffei, C. Schneidewind, Foundations and tools for the static analysis of Ethereum smart contracts, in: CAV, in: LNCS, vol. 10981, Springer, 2018, pp. 51–78, http://dx.doi.org/10.1007/978-3-319-96145-3_4.

[31] Mythril, https://github.com/ConsenSys/mythril (2018).

[32] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, Smartcheck: Static analysis of Ethereum smart contracts, in: IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9–16.

[33] P. Tsankov, A.M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M.T. Vechev, Securify: Practical security analysis of smart contracts, in: ACM CCS, 2018, pp. 67–82, http://dx.doi.org/10.1145/3243734.3243780.

[34] A. Mavridou, A. Laszka, Designing secure ethereum smart contracts: A finite state machine based approach, in: Financial Cryptography and Data Security, 2018.

[35] A. Biryukov, D. Khovratovich, S. Tikhomirov, Findel: Secure derivative contracts for Ethereum, in: Financial Cryptography Workshops, in: LNCS, vol. 10323, Springer, 2017, pp. 453–467, http://dx.doi.org/10.1007/978-3-319-70278-0_28.

[36] B. Egelund-Müller, M. Elsman, F. Henglein, O. Ross, Automated execution of financial contracts on blockchains, Bus. Inf. Syst. Eng. 59 (6) (2017) 457–467, http://dx.doi.org/10.1007/s12599-017-0507-z.

[37] P.L. Seijas, S.J. Thompson, Marlowe: Financial contracts on blockchain, in: ISoLA, in: LNCS, vol. 11247, Springer, 2018, pp. 356–375, http://dx.doi.org/10.1007/978-3-030-03427-6_27.

[38] S.L.P. Jones, J. Eber, J. Seward, Composing contracts: an adventure in financial engineering, functional pearl, in: International Conference on Functional Programming (ICFP), 2000, pp. 280–292, http://dx.doi.org/10.1145/351240.351267.

[39] M. Bartoletti, R. Zunino, Verifying liquidity of Bitcoin contracts, in: Principles of Security and Trust, in: LNCS, vol. 11426, Springer, 2019, pp. 222–247, http://dx.doi.org/10.1007/978-3-030-17138-4_10.

[40] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, Y. Zhou, Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology, in: WWW, 2018, pp. 1409–1418, http://dx.doi.org/10.1145/3178876.3186046.

[41] M. Bartoletti, B. Pes, S. Serusi, Data mining for detecting Bitcoin Ponzi schemes, in: Crypto Valley Conference on Blockchain Technology, IEEE, 2018, pp. 75–84, http://dx.doi.org/10.1109/CVCBT.2018.00014.

[42] K. Toyoda, T. Ohtsuki, P.T. Mathiopoulos, Identification of high yielding investment programs in bitcoin via transactions pattern analysis, in: GLOBECOM, 2017, pp. 1–6, http://dx.doi.org/10.1109/GLOCOM.2017.8254420.

[43] L. Shin, $15 Million ICO Halted By SEC For Being Alleged Scam, 2017, www.forbes.com/sites/laurashin/2017/12/04/15-million-ico-halted-by-sec-for-being-alleged-scam.

[44] D.A. Zetzsche, R.P. Buckley, D.W. Arner, L. Föhr, The ICO Gold Rush: It's a scam, it's a bubble, it's a super challenge for regulators, University of Luxembourg Law Working Paper No. 11/2017; UNSW Law Research Paper No. 83; University of Hong Kong Faculty of Law Research Paper No. 2017/035. Available at SSRN: https://ssrn.com/abstract=3072298 or http://dx.doi.org/10.2139/ssrn.3072298 (2017).

Massimo Bartoletti is Associate Professor at the Department of Mathematics and Computer Science of the University of Cagliari. He got a Ph.D in Computer Science at the University of Pisa in 2005. The research activity of Massimo Bartoletti focuses on formal methods for the specification, analysis and verification of software and systems, on security, on impure functional languages, on concurrency theory, on the semantics of programming languages, on theories of contracts, and on logics for the specification and verification of properties of software and systems. He is coordinator of the node of the Cyber Security National Lab at the Department of Mathematics and Computer Science of the University of Cagliari. Massimo Bartoletti published over fifty research papers on refereed journals and international

refereed conferences and workshops. He served as chair, program committee member, and organizer for several international conferences and workshops, and as external reviewer for several journals and international conferences.

Salvatore Carta graduated received a Ph.D in Electronics and Computer Science from the University of Cagliari in 2003. In 2005 joined Department of Mathematics and Computer Science of the University of Cagliari as Assistant Professor. In 2006 and 2007 he visited the Swiss Federal Institute of Technology as invited Professor, hosted by Laboratoire des Systèmes Intégrés - LSI. Since 2014 he is Associate Professor in Computer Science at the University of Cagliari. The research activity of Salvatore Carta focuses on architectures, software and tools for embedded and portable computing, with particular emphasis on Software infrastructures, algorithms and applications for multiprocessor-system-on-chip, low power real-time scheduling algorithms, and Networks-on-Chip. Recently, he has focused on topics related to the social Web, ubiquitous computing and computational societies. In particular he works on algorithms for social search and recommendation, and on algorithms and strategies in the fields of mobile Human Computer Interaction and Mobile Health. Salvatore Carta is co-founder of the Trustworthy Computational Societies Research Group at the Mathematics and Computer Science Department of the University of Cagliari.

Tiziana Cimoli is a postdoc researcher at the Department of Mathematics and Computer Science of the University of Cagliari. Her research is focused on contract-oriented interactions and their properties. In early studies, she has worked on contracts modelled with event structures, Petri nets and session types. Recently, she has drifted her attention towards smart contracts and blockchain technologies, with particular interest in frauds and security vulnerabilities.

Roberto Saia is a postdoctoral researcher at the Department of Mathematics and Computer Science of the University of Cagliari. He got a Master Degree and a Ph.D in Computer Science at the same University, and his current research is focused on several domains, such as those of the Recommender Systems, Data Mining, Artificial Intelligence, and Security. He is the author or co-author of tens scientific journals, articles, and books.