2018 IEEE
1st International Workshop on Blockchain Oriented Software Engineering (IWBOSE)

Proceedings

Roberto Tonelli, Stéphane Ducasse, Gianni Fenu, and Andrea Bracciali

March 20, 2018
Campobasso, Italy

2018 IEEE
1st International Workshop on Blockchain Oriented Software Engineering (IWBOSE)

IEEE Catalog Number: CFP18P97-ART
ISBN: 978-1-5386-5986-1

Production: Conference Publishing Consulting, D-94034 Passau, Germany
info@conference-publishing.com

Message from the Chairs

Welcome to the 1st **International Workshop on Blockchain Oriented Software Engineering** (**IWBOSE2018**). The workshop is co-lacated with SANER 2018 and will be held on the 20th of March in Campobasso, Italy.
The workshop aims at gathering together researchers from the academia and from the industry to focus on the new challenges posed by the new software technology supporting the various Blockchains infrastructure. The Workshop's goal is to gather together practitioners and researchers to discuss on progresses on the research and on the practical usage of Blockchain technologies and smart contracts, focusing on the application and definition of software engineering principles and practices specific for such software technology, and for the technologies relying on it. Motivations for this workshop are the ever-increasing interest both in the research community and in the industry on Blockchain and smart contracts principles and applications, being the management of cryptocurrencies the most popular topic. These novelties call for specific tools, paradigms, principles, approaches and research to deal with it and for a specific Blockchain Oriented Software Engineering (BOSE) [1].
The Workshop features six accepted papers, three focusing on Smart Contracts and three focusing on Bockchain and ICO (Initial Coin Offers), where security patterns in Solidity, the inspection of Smart Contracts, their velnerabilities are analized altogether with Property-Based Testing for Blockchain and extended analysis of the success factors of ICOs.

Workshop Keynote invited speaker is Prof. Michele Marchesi, from Cagliari University, Italy, with a talk titled: Why Blockchain Is Important for Software Developers, and Why Software Engineering Is Important for Blockchain Software.

We would like to thank all participants and submitters for contributing to the workshop's success.

[1] S. Porru, A. Pinna, M. Marchesi, R. Tonelli, "Blockchain-oriented software engineering: challenges and new directions" Proceedings of the 39th International Conference on Software Engineering Companion, p. 169- 171, May 2017, Buenos Aires, Argentina.

A. Bracciali, S. Ducasse,
G. Fenu, R. Tonelli,

Campobasso
March 2018

IWBOSE 2018 Chairs

IWBOSE 2018 Organization

Organizing Committee

General Chairs

- **Roberto Tonelli**, University of Cagliari, Italy (roberto *dot* tonelli *at* dsf *dot* unica *dot* it)
- **Stephane Ducasse** , INRIA-LILLE, France (stephane dot ducasse at inria dot fr)

Program Chairs

- **Gianni Fenu**, University of Cagliari, Italy (*fenu at* unica *dot* it**)**
- **Andrea Bracciali** University of Stirling, UK (*abb at* cs *dot* stir *dot* ac *dot* uk)

Program Committee

Vasco Amaral, Nova university of Lisbon, Portugal
Francesca Arcelli Fontana, University of Milano Bicocca, Italy
Massimo Bartoletti, University of Cagliari Italy
Walter Bartosz, Poznan University of Technology
Stefano Bistarelli, University of Perugia, Italy
Luigi Buglione, Engineering.IT / ETS, Canada
Damien Challet, Fribourg Univ., Germany
Steve Counsell, Brunel University, UK
Giuseppe Destefanis, University of Hertfordshire UK
Boris Duedder, University of Copenhagen, Denmark
Shayan Eskandari, Concordia University / Bitaccess
Yossi Gil, Technion, Israel
Reiko Heckel, University of Leicester, UK
Robert Hierons, Brunel University, UK
Raimundas Matulevicius, University of Tartu, Estonia
Sandro Morasca, University of Insubria, Italy
Matteo Orru', Technion, Israel
Laura Ricci, University of Pisa, Italy
Paramvir Singh, National Institute of Technology - Jalandhar, India
Stephen Swift, Brunel University, UK
Maurice Ter Beek, ISTI CNR, IT
Andrea Vitaletti, University La Sapienza, Rome, Italy
Hongyu Zhang, The University of Newcastle, Australia
Roberto Zunino, University of Trento, Italy.

Contents

Why Blockchain Is Important for Software Developers, and Why Software Engineering Is Important for Blockchain Software (Keynote)

Michele Marchesi
University of Cagliari, Italy
marchesi@unica.it

Abstract*—In the past few years, cryptocurrencies and blockchain applications has been one of the most rapidly emerging fields of computer science, leading to a strong demand of software applications. Several new projects have been emerging almost daily, with an impetus that was not seen since the days of the dawn of the Internet. However, the need of being timely on the market and the lack of experience in a brand new field led to epic disasters, such as those of DAO in 2016 and of Parity Ethereum wallet in 2017. Also, there have been several hacks successfully performed on cryptocurrency exchanges, the biggest being those of MtGox in 2014 (350 million US$), Bitfinex in 2016 (72 million US$), and Coincheck in 2017 (400 million US$). The application of sound SE practices to Blockchain software development, both for Smart Contract and generic Blockchain software, might be crucial to the success of this new field. Here the issues are the need for specific analysis and design methods, quality control through testing and metrics, security assessment and overall development process. At the same time, Blockchain development offers new opportunities, such as the certification of empirical data used for experiment; the ability to design processes where developers are paid upon completion of their tasks through Blockchain tokens, after acceptance tests performed using Smart Contracts; and more sound techniques enabling pay-per-use software, again using tokens.

***Index Terms*—Blockchain; Smart Contracts; Distributed Ledgers**

BIOGRAPHY

Michele Marchesi graduated in electronic engineering and mathematics from the University of Genoa. He is professor of Software Engineering at the Department of Mathematics and Computer Science of the University of Cagliari.

His research interests include agile methods of software application development, software metrics, social networks, modeling and simulation of the software process, and of economic and financial systems, Blockchain technology and Smart Contracts. He is one of the proposer of the new field of Blockchain-oriented software engineering.

He is author of over 250 international publications on these topics, which have more than 8000 citations (source: Google Scholar). He collaborates with many universities, including Genoa, Brunel University of London, University of Hertfordshire, UK, Madrid Polytechnic University, Auckland University. He was and is the coordinator of several international and domestic research projects.

He is a founding member of two spinoff firms working in software production and blockchain applications.

Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity

Maximilian Wöhrer and Uwe Zdun
University of Vienna
Faculty of Computer Science
Währingerstraße 29, 1090 Vienna, Austria
Email: {maximilian.woehrer,uwe.zdun}@univie.ac.at

***Abstract*—Smart contracts that build up on blockchain technologies are receiving great attention in new business applications and the scientific community, because they allow untrusted parties to manifest contract terms in program code and thus eliminate the need for a trusted third party. The creation process of writing well performing and secure contracts in Ethereum, which is today's most prominent smart contract platform, is a difficult task. Research on this topic has only recently started in industry and science. Based on an analysis of collected data with Grounded Theory techniques, we have elaborated several common security patterns, which we describe in detail on the basis of Solidity, the dominating programming language for Ethereum. The presented patterns describe solutions to typical security issues and can be applied by Solidity developers to mitigate typical attack scenarios.**

I. Introduction

Ethereum is a major blockchain-based ecosystem that provides an environment to code and run smart contracts. Writing smart contracts in Solidity is so far a challenging undertaking. It involves the application of unconventional programming paradigms, due to the inherent characteristics of blockchain based program execution. Furthermore, bugs in deployed contracts can have serious consequences, because of the immediate coupling of contract code and financial values. Therefore, it is beneficial to have a solid foundation of established and proven design and code patterns that ease the process of writing functional and error free code.

With this paper we want to make the first steps in order to create an extensive pattern language. Our research aims to answer which code and design patterns commonly appear in Solidity coded smart contracts and the problems they intent to solve. In order to answer these questions we gathered data from different sources and applied Grounded Theory techniques to extract and identify the patterns.

This paper is structured as follows: First, we provide a short background to blockchain technology in Section II and the Ethereum platform in Section III. Then, we discuss some platform related security aspects in Section IV-C, before we present elaborated security patterns in Section V in detail. Finally, we discuss related work in Section VI, and draw a conclusion at the end in Section VII.

II. Background

A. Blockchains, Cryptocurrencies, and Smart Contracts

Blockchains are a digital technology that build on a combination of cryptography, networking, and incentive mechanisms to support the verification, execution and recording of transactions between different parties. In simple terms, blockchain systems can be seen as decentralized databases that offer very appealing properties. These include the immutability of stored transactions and the creation of trust between participants without a third party. That makes blockchains suitable as an open distributed ledger that can store transactions between parties in a verifiable and permanent way. One prominent application is the exchange of digital assets, so-called cryptocurrencies. Widely known cryptocurrencies are Bitcoin, Ethereum and Litecoin. They offer, beyond the transfer of digital assets, the execution of smart contracts. Smart contracts are computer programs that facilitate, verify, and enforce the negotiation and execution of legal contracts. They are executed through blockchain transactions, interact with crypto currencies, and have interfaces to handle input from contract participants. When run on the blockchain, a smart contract becomes an autonomous entity that automatically executes specific actions when certain conditions are met. Because smart contracts run on the blockchain, they run exactly as programmed, without any possibility of censorship, downtime, fraud or third party interference [1]. Today, the most-used smart contract platform in this regard is Ethereum.

III. Ethereum Platform

Ethereum is a public blockchain based distributed computing platform, that offers smart contract functionality. It provides a decentralised virtual machine as runtime environment to execute smart contracts, known as Ethereum Virtual Machine (EVM).

A. Ethereum Virtual Machine (EVM)

The EVM handles the computation and state of contracts and is build on a stack-based language with a predefined set of instructions (opcodes) and corresponding arguments [2]. So, in essence, a contract is simply a series of opcode statements, which are sequentially executed by the EVM. The EVM can be thought of as a global decentralized computer on which all smart contracts run. Although it behaves like one giant computer, it is rather a network of smaller discrete machines in constant communication. All transactions, handling the execution of smart contracts, are local on each node of the network and processed in relative synchrony. Each node validates and groups the transactions sent from users into blocks, and tries to append them to the blockchain in order to collect an associated reward. This process is called mining and the participating nodes are called miners. To ensure a proper resource handling of the EVM, every instruction the EVM executes has a cost associated with it, measured in units of gas. Operations that require more computational resources cost more gas, than operations that require fewer computational resources. This ensures that the system is not jammed up by denial-of-service attacks, where users try to overwhelm the network with time-consuming computations. Therefore, the purpose of gas is twofold. It encourages developers to write quality applications by avoiding wasteful code, and ensures at the same time that miners, executing the requested operations, are compensated for their contributed resources. When it comes to paying for gas, a transaction fee is charged in small amounts of Ether, the built-in digital currency of the Ethereum network, and the token with which miners are rewarded for executing transactions and producing blocks. Ultimately, Ether is the fuel for operating the Ethereum platform.

B. *Ethereum Smart Contracts*

Smart contracts are applications which are deployed on the blockchain ledger and execute autonomously as part of transaction validation. To deploy a smart contract in Ethereum, a special creation transaction is executed, which introduces a contract to the blockchain. During this procedure the contract is assigned an unique address, in form of a 160-bit identifier, and its code is uploaded to the blockchain. Once successfully created, a smart contract consists of a contract address, a contract balance, predefined executable code, and a state. Different parties can then interact with a specific contract by sending contract-invoking transactions to a known contract address. These may trigger any number of actions as a result, such as reading and updating the contract state, interacting and executing other contracts, or transferring value to others. A contract-invoking transaction must include the execution fee and may also include a transfer of Ether from the caller to the contract. Additionally, it may also define input data for the invocation of a function. Once a transaction is accepted, all network participants execute the contract code, taking into account the current state of the blockchain and the transaction data as input. The network then agrees on the output and the next state of the contract by participating in the consensus protocol. Thus, on a conceptual level, Ethereum can be viewed as a transaction-based state machine, where its state is updated after every transaction.

C. *Ethereum Programming Languages*

Smart contracts in Ethereum are usually written in higher level languages and are then compiled to EVM bytecode. Such higher level languages are LLL (Low-level Lisp-like Language) [3], Serpent (a Python-like language) [4], Viper (a Python-like language) [5], and Solidity (a Javascript-like language) [6]. LLL and Serpent were developed in the early stages of the platform, while Viper is currently under development, and is intended to replace Serpent. The most prominent and widely adopted language is Solidity.

D. *Solidity*

Solidity is a high-level Turing-complete programming language with a JavaScript similar syntax, being statically typed, supporting inheritance and polymorphism, as well as libraries and complex user-defined types.

When using Solidity for contract development, contracts are structured similar to classes in object oriented programming languages. Contract code consists of variables and functions which read and modify these, like in traditional imperative programming.

```
pragma solidity ^0.4.17;
contract SimpleDeposit {
  mapping (address => uint) balances;

  event LogDepositMade(address from, uint amount);

  modifier minAmount(uint amount) {
    require(msg.value >= amount);
    _;
  }

  function SimpleDeposit() public payable {
    balances[msg.sender] = msg.value;
  }

  function deposit() public payable minAmount(1 ether)
       {
    balances[msg.sender] += msg.value;
    LogDepositMade(msg.sender, msg.value);
  }

  function getBalance() public view returns (uint
       balance) {
    return balances[msg.sender];
  }

  function withdraw(uint amount) public {
    if (balances[msg.sender] >= amount) {
      balances[msg.sender] -= amount;
      msg.sender.transfer(amount);
    }
  }
}
```

Listing 1. A simple contract where users can deposit some value and check their balance.

Listing 1 shows a simple contract written in Solidity in which users can deposit some value and check their balance. Before describing the code in more detail, it is helpful to give some insights about Solidity features like global variables, modifiers, and events.

Solidity defines special variables (`msg`, `block`, `tx`) that always exist in the global namespace and contain properties to access information about an invocation-transaction and the blockchain. For example, these variables allow the retrieval of the origin address, the amount of Ether, and the data sent alongside an invocation-transaction.

Another particular convenient feature in Solidity are so-called modifiers. Modifiers can be described as enclosed code units that enrich functions in order to modify their flow of code execution. This approach follows a condition-orientated programming (COP) paradigm, with the main goal to remove conditional paths in function bodies. Modifiers can be used to easily change the behaviour of functions and are applied by specifying them in a whitespace-separated list after the function name. The new function body is the modifiers body where '_' is replaced by the original function body. A typical use case for modifiers is to check certain conditions prior to executing the function.

An additionally important and neat feature of Solidity are events. Events are dispatched signals that smart contracts can fire. User interfaces and applications can listen for those events on the blockchain without much cost and act accordingly. Other than that, events may also serve logging purposes. When called, they store their arguments in a transaction's log, a special data structure in the blockchain that maps all the way up to the block level. These logs are associated with the address of the contract and can be efficiently accessed from outside the blockchain.

Given this short feature description, we can now return and analyse the code example. First, the compiler version is defined (line 1), then the contract is defined in which a state variable is declared (line 3), followed by an event definition (line 5), a modifier definition (line 7), the constructor (line 12), and the actual contract functions (line 16 onwards). The state of the contract is stored in a mapping called `balances` (which stores an association between a users address and a balance). The special function `SimpleDeposit` is the constructor, which is run during the creation of the contract and cannot be called afterwards. It sets the balance of the individual creating the contract (`msg.sender`) to the amount of Ether sent along the contract creation transaction (`msg.value`). The remaining functions actually serve for interaction and are called by users and contracts alike. The `deposit()` function (line 16) manipulates the balances mapping by adding the amount sent along the transaction-invocation to the senders balance, while utilizing a modifier to preliminary ensure that at least 1 Ether is sent. The `withdraw()` function (line 25) manipulates the balances mapping by subtracting the requested amount to be withdrawn from the senders balance and the `getBalance()` function (line 21) returns the actual balance of the sender by querying the balances mapping.

In summary, this simple example shows the basic concepts of a smart contract coded in Solidity. Moreover, it illustrates the most powerful feature of smart contracts, which is the ability to manipulate a globally verifiable and universally consistent contract state (the balances mapping).

IV. Development Aspects

A. *Limits of Blockchain Technology*

First, it is important to state that not every application is predestined to be run on a blockchain. There are many applications that do not need a decentralized, immutable, append-only data log with transaction validation. Due to the inherent characteristics of blockchains, distributed ledger systems are not suitable for a variety of use cases. For example, computation-heavy applications are impractical to run on blockchains, because of the accumulated computation fees and the fact that many types of computations are impractical to execute on a stack-based virtual machine. Another limitation of blockchains is that they are not suitable for storing large amounts of data. This implicit limitation results in the extensive redundancy from the large number of network nodes, holding a full copy of the distributed ledger. Nonetheless, this can be overcome by not storing large data directly on the blockchain, but only a hash or other meta-data on the chain. In the context of data storage it is also important to realize that the data on the blockchain is visible to all network participants. This implies that keeping sensitive data confidential, requires the obfuscation of plaintext data by some means. A further limitation of blockchains is their performance. They are currently not suitable for applications which demand a high-frequency or low latency execution of transactions, because of the additional work owed to the cryptography, consensus, and redundancy apparatus of blockchain systems. Within these limits smart contracts should be used for applications that have something to gain from being distributed and publicly verifiable and enforceable. In general, most applications that handle the transfer or registration of resources in a traceable way are suitable, e.g. land register, provenance documentation, or electronic voting.

B. *Coding Smart Contracts in Ethereum*

Contract development on the Ethereum blockchain requires a different engineering approach than most web and mobile developers are familiar with. Unlike modern programming languages, which support a broad range of convenient data types for storage and manipulation, the developer is responsible for the internal organization and manipulation of data at a deeper level. This implies that the developer has to address details he may not be used to deal with. For example, a developer would have to implement a method to concat or lowercase strings, which are tasks developers usually do not have to think about in other languages. Furthermore, the Ethereum platform and Solidity are constantly evolving in a fast pace and the developer is confronted with an ongoing transformation of platform features and the security landscape, as new instructions are added, and bugs and security risks are discovered. Developers have to consider that code that is written today, will probably not compile in a few months, or will at least have to be refactored.

C. *Smart Contract Security*

An analysis of existing smart contracts by Bartoletti and Pompianu [7] shows that the Bitcoin and Ethereum platform mainly focus on financial contracts. In other words, most smart contract program code defines how assets (money) move. Therefore, it is crucial that contract execution is performed correctly. The direct handling of assets means that flaws are more likely to be security relevant and have greater direct financial consequences than bugs in typical applications. Incidents, like the value overflow incident in Bitcoin [8], or the DAO hack [9] in Ethereum, caused a hard fork of the blockchain to nullify the malicious transactions. These incidents show that security issues have been used for fraudulent purposes ruthlessly in the past. A survey of possible attacks on Ethereum contracts was published by Atzei et al. [10] and lists 12 vulnerabilities that are assigned by context to Solidity, the EVM, and blockchain peculiarities itself. Many of these vulnerabilities can be addressed by following best practices for writing secure smart contracts, which are scattered throughout the Ethereum community [11, 12] and different Ethereum blogs. Most best practices mainly contain information about typical pitfalls to avoid and the description of favourable design and problem approaches. The latter being the focus of this paper in order to collate smart contract security design patterns.

V. Smart Contract Design Patterns

A software pattern describes an abstraction or conceptualization of a concrete, complex, and reoccurring problem that software designers have faced in the context of real software development projects and a successful solution they have implemented multiple times to resolve this problem [13].

So far, only few efforts have been made to collect and categorize patterns in a structured manner [14, 15]. This section gives an overview of typical security design patterns that are inherently frequent or practical in the context of smart contract coding. The presented patterns address typical problems and vulnerabilities related to smart contract execution. The patterns are based on multiple sources, such as review of Solidity development documentation, on studying Internet blogs and discussion forums about Ethereum, and the the examination of existing smart contracts. The source code of the presented patterns is available on github [16]. To illustrate the patterns in practice, Table I at the end of this section lists for each pattern an example contract with published source code deployed on the Ethereum mainnet.

A. Security Patterns

Security is a group of patterns that introduce safety measures to mitigate damage and assure a reliable contract execution.

1) Checks-Effects-Interaction:

CHECKS-EFFECTS-INTERACTION PATTERN

Problem When a contract calls another contract, it hands over control to that other contract. The called contract can then, in turn, re-enter the contract by which it was called and try to manipulate its state or hijack the control flow through malicious code.

Solution Follow a recommended functional code order, in which calls to external contracts are always the last step, to reduce the attack surface of a contract being manipulated by its own externally called contracts.

The Checks-Effects-Interaction pattern is fundamental for coding functions and describes how function code should be structured to avoid side effects and unwanted execution behaviour. It defines a certain order of actions: First, check all the preconditions, then make changes to the contract's state, and finally interact with other contracts. Hence its name is "Checks-Effects-Interactions Pattern". According to this principle, interactions with other contracts should be, whenever possible, the very last step in any function, as seen in Listing 2. The reason being, that as soon as a contract interacts with another contract, including a transfer of Ether, it hands over the control to that other contract. This allows the called contract to execute potentially harmful actions. For example, a so-called re-entrancy attack, where the called contract calls back the current contract, before the first invocation of the function containing the call, was finished. This can lead to an unwanted execution behaviour of functions, modifying the state variables to unexpected values or causing operations (e.g. sending of funds) to be performed multiple times. An example for a contract function, prone to the described attack scenario, is shown in Listing 3. The re-entrancy attack is especially harmful when using low level `address.call`, which forwards all remaining gas by default, giving the called contract more room for potentially malicious actions. Therefore, the use of low level `address.call` should be avoided whenever possible. For sending funds `address.send()` and `address.transfer()` should be preferred, these functions minimize the risk of re-entrancy through limited gas forwarding. While these methods still trigger code execution, the called contract is only given a stipend of 2,300 gas, which is currently only enough to log an event.

```
function auctionEnd() public {
  // 1. Checks
  require(now >= auctionEnd);
  require(!ended);
  // 2. Effects
  ended = true;
  // 3. Interaction
  beneficiary.transfer(highestBid);
}
```

Listing 2. Applying the Checks-Effects-Interaction pattern within a function.

```
mapping (address => uint) balances;

function withdrawBalance() public {
  uint amount = balances[msg.sender];
  require(msg.sender.call.value(amount)()); // caller's
      code is executed and can re-enter withdrawBalance
      again
  balances[msg.sender] = 0; // INSECURE - user's balance
      must be reset before the external call
}
```

Listing 3. An example of an insecure withdrawal function prone to a re-entrancy attack.

2) Emergency Stop (Circuit Breaker):

EMERGENCY STOP (CIRCUIT BREAKER) PATTERN

Problem Since a deployed contract is executed autonomously on the Ethereum network, there is no option to halt its execution in case of a major bug or security issue.

Solution Incorporate an emergency stop functionality into the contract that can be triggered by an authenticated party to disable sensitive functions.

Reliably working contracts may contain bugs that are yet unknown, until revealed by an adversary attack. One countermeasure and a quick response to such attacks are emergency stops or circuit breakers. They stop the execution of a contract or its parts when certain conditions are met. A recommended scenario would be, that once a bug is detected, all critical functions would be halted, leaving only the possibility to withdraw funds. A contract implementing the described strategy is shown in Listing 4. The ability to fire an emergency stop could be either given to a certain party, or handled through the implementation of a rule set.

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract EmergencyStop is Owned {
  bool public contractStopped = false;

  modifier haltInEmergency {
    if (!contractStopped) _;
  }

  modifier enableInEmergency {
    if (contractStopped) _;
  }

  function toggleContractStopped() public onlyOwner {
    contractStopped = !contractStopped;
  }

  function deposit() public payable haltInEmergency {
    // some code
  }

  function withdraw() public view enableInEmergency {
    // some code
  }
}
```

Listing 4. An emergency stop allows to disable or enable specific functions inside a contract in case of an emergency.

3) *Speed Bump:*

SPEED BUMP PATTERN
Problem The simultaneous execution of sensitive tasks by a huge number of parties can bring about the downfall of a contract.
Solution Prolong the completion of sensitive tasks to take steps against fraudulent activities.

Contract sensitive tasks are slowed down on purpose, so when malicious actions occur, the damage is restricted and more time to counteract is available. An analogous real world example would be a bank run, where a large number of customers withdraw their deposits simultaneously due to concerns about the bank's solvency. Banks typically counteract by delaying, stopping, or limiting the amount of withdrawals. An example contract implementing a withdrawal delay is shown in Listing 5.

```
pragma solidity ^0.4.17;
contract SpeedBump {
  struct Withdrawal {
    uint amount;
    uint requestedAt;
  }
  mapping (address => uint) private balances;
  mapping (address => Withdrawal) private withdrawals;
  uint constant WAIT_PERIOD = 7 days;

  function deposit() public payable {
    if(!(withdrawals[msg.sender].amount > 0))
      balances[msg.sender] += msg.value;
  }

  function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
      uint amountToWithdraw = balances[msg.sender];
      balances[msg.sender] = 0;
      withdrawals[msg.sender] = Withdrawal({
        amount: amountToWithdraw,
        requestedAt: now
      });
    }
  }

  function withdraw() public {
    if(withdrawals[msg.sender].amount > 0 && now >
        withdrawals[msg.sender].requestedAt + WAIT_PERIOD)
         {
      uint amount = withdrawals[msg.sender].amount;
      withdrawals[msg.sender].amount = 0;
      msg.sender.transfer(amount);
    }
  }
}
```

Listing 5. A contract that delays the withdrawal of funds deliberately.

4) *Rate Limit:*

RATE LIMIT PATTERN
Problem A request rush on a certain task is not desired and can hinder the correct operational performance of a contract.
Solution Regulate how often a task can be executed within a period of time.

A rate limit regulates how often a function can be called consecutively within a specified time interval. This approach may be used for different reasons. A usage scenario for smart contracts may be founded on operative considerations, in order to control the impact of (collective) user behaviour. As an example one might limit the withdrawal execution rate of a contract to prevent a rapid drainage of funds. Listing 6 exemplifies the application of this pattern.

```
pragma solidity ^0.4.17;
contract RateLimit {
  uint enabledAt = now;

  modifier enabledEvery(uint t) {
    if (now >= enabledAt) {
      enabledAt = now + t;
      _;
    }
  }

  function f() public enabledEvery(1 minutes) {
    // some code
  }
}
```

Listing 6. An example of a rate limit that avoids excessively repetitive function execution.

5) *Mutex:*

MUTEX PATTERN
Problem Re-entrancy attacks can manipulate the state of a contract and hijack the control flow.
Solution Utilize a mutex to hinder an external call from re-entering its caller function again.

A mutex (from mutual exclusion) is known as a synchronization mechanism in computer science to restrict concurrent access to a resource. After re-entrancy attack scenarios emerged, this pattern found its application in smart contracts to protect against recursive function calls from external contracts. An example contract is depicted below in Listing 7.

```
pragma solidity ^0.4.17;
contract Mutex {
  bool locked;

  modifier noReentrancy() {
    require(!locked);
    locked = true;
    _;
    locked = false;
  }
  // f is protected by a mutex, thus reentrant calls
  // from within msg.sender.call cannot call f again
  function f() noReentrancy public returns (uint) {
    require(msg.sender.call());
    return 1;
  }
}
```

Listing 7. The application of a mutex pattern to avoid re-entrancy.

6) *Balance Limit:*

BALANCE LIMIT PATTERN
Problem There is always a risk that a contract gets compromised due to bugs in the code or yet unknown security issues within the contract platform.
Solution Limit the maximum amount of funds at risk held within a contract.

It is generally a good idea to manage the amount of money at risk when coding smart contracts. This can be achieved by limiting the total balance held within a contract. The pattern monitors the contract balance and rejects payments sent along a function invocation after exceeding a predefined quota, as seen in Listing 8. It should be noted that this approach cannot prevent the admission of forcibly sent Ether, e.g. as beneficiary of a `selfdestruct(address)` call, or as recipient of a mining reward.

```
pragma solidity ^0.4.17;
contract LimitBalance {
  uint256 public limit;

  function LimitBalance(uint256 value) public {
    limit = value;
  }

  modifier limitedPayable() {
    require(this.balance <= limit);
    _;
  }

  function deposit() public payable limitedPayable {
    // some code
  }
}
```

Listing 8. A contract limiting the total balance acquirable through payable function invocation.

TABLE I
PATTERN USAGE EXAMPLES IN PUBLISHED SOURCE CODE CONTRACTS ON THE ETHEREUM MAINNET.

Category	Pattern	Example Contract
Security	Checks-Effects-Interaction	CryptoKitties
	Emergency Stop	Augur/REP
	Speed Bump	TheDAO
	Rate Limit	etherep
	Mutex	Ventana Token
	Balance Limit	CATToken

VI. Related Work

According to Alharby and van Moorsel [17] current research on smart contracts is mainly focused on identifying and tackling smart contract issues and can be divided into four categories, namely coding, security, privacy and performance issues. The technology behind writing smart contracts in Ethereum is still in its infancy stage, therefore coding and security are among the most discussed issues. Unfortunately, a lot of research and practical knowledge is scattered throughout blog articles and grey literature, therefore information is often not very structured. Only relatively few papers focus on software patterns in blockchain technology respectively on design patterns in the Solidity language for the Ethereum ecosystem. Bartoletti and Pompianu [7] conducted an empirical analysis of Solidity smart contracts and identified a list of nine common design patterns that are shared by studied contracts. These patterns broadly summarize the most frequent solutions to handle common usage scenarios. A paper by Zhang et al. [18] describes how the application of familiar software patterns can help to resolve design specific challenges. In particular, commonly known design patterns such as the Abstract Factory, Flyweight, Proxy, and Publisher-Subscriber pattern are applied to implement a blockchain-based healthcare application. The above mentioned papers do not contain security related design patterns, but show that design patterns are an interesting topic in smart contract coding.

VII. Conclusion

We have given a brief introduction to Ethereum and Solidity and outlined six design patterns that address security issues when coding smart contracts in Solidity. In general, the main problem that these patterns solve is the lack of execution control once a contract has been deployed, resulting form the distributed execution environment provided by Ethereum. This one-of-a-kind characteristic of Ethereum allows programs on the blockchain to be executed autonomously, but also has drawbacks. These drawbacks appear in different forms, either as harmful callbacks, adverse circumstances on how and when functions are executed, or uncontrollably high financial risks at stake. By applying the presented patterns, developers can address these security problems and mitigate typical attack scenarios.

In future work, we plan to extend the already collated patterns to create a structured and informative design pattern language for Solidity, that can be used as guidance for developers or find its application in automatic code generating frameworks.

References

[1] Ethereum project. [Online]. Available: https://www.ethereum.org/

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[3] Lll poc 6. [Online]. Available: https://github.com/ethereum/cpp-ethereum/wiki/LLL-PoC-6/7a575cf91c4572734a83f95e970e9e7ed64849ce

[4] Serpent. [Online]. Available: https://github.com/ethereum/wiki/wiki/Serpent

[5] ethereum/viper: New experimental programming language. [Online]. Available: https://github.com/ethereum/viper

[6] Solidity — solidity 0.4.18 documentation. [Online]. Available: https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[7] M. Bartoletti and L. Pompianu, “An empirical analysis of smart contracts: platforms, applications, and design patterns,” *arXiv preprint arXiv:1703.06322*, 2017.

[8] Value overflow incident - bitcoin wiki. [Online]. Available: https://en.bitcoin.it/wiki/Value_overflow_incident

[9] P. Daian, “Analysis of the dao exploit,” 2016, [Online; accessed 6-September-2017]. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[10] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[11] Security considerations — solidity 0.4.18 documentation. [Online]. Available: http://solidity.readthedocs.io/en/develop/security-considerations.html

[12] “Ethereum contract security techniques and tips,” 2017, [Online; accessed 6-September-2017]. [Online]. Available: https://github.com/ConsenSys/smart-contract-best-practices

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: elements of,” 1994.

[14] J. Bontje. (2015) Dapp design patterns. [Online]. Available: https://www.slideshare.net/mids106/dapp-design-patterns

[15] cjgdev. (2016) Smart-contract patterns written in solidity, collated for community good. [Online]. Available: https://github.com/cjgdev/smart-contract-patterns

[16] “maxwoe/solidity_patterns.” [Online]. Available: https://github.com/maxwoe/solidity_patterns

[17] M. Alharby and A. van Moorsel, “Blockchain-based smart contracts: A systematic mapping study,” *arXiv preprint arXiv:1710.06372*, 2017.

[18] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, “Applying software patterns to address interoperability in blockchain-based healthcare apps,” *arXiv preprint arXiv:1706.03700*, 2017.

SmartInspect: Solidity Smart Contract Inspector

Santiago Bragagnolo*, Henrique Rocha*, Marcus Denker*, and Stéphane Ducasse*
* Inria Lille - Nord Europe
Villeneuve D'ascq, France
{santiago.bragagnolo, henrique.rocha, marcus.denker, stephane.ducasse}inria.fr

***Abstract*—Solidity is a language used for smart contracts on the Ethereum blockchain. Smart contracts are embedded procedures stored with the data they act upon. Debugging smart contracts is a really difficult task since once deployed, the code cannot be re-executed and inspecting a simple attribute is not easily possible because data is encoded. In this paper, we address the lack of inspectability of a deployed contract by analyzing contract state using decompilation techniques driven by the contract structure definition. Our solution, SmartInspect, also uses a mirror-based architecture to represent locally object responsible for the interpretation of the contract state. SmartInspect allows contract developers to better visualize and understand the contract stored state without needing to redeploy, nor develop any ad-hoc code.**

***Index Terms*—Blockchain, Inspecting, Solidity, Smart Contracts**

I. Introduction

Blockchain technology has attracted a lot of attention recently [1]. A blockchain is a distributed database, managed by a peer-to-peer network that stores a list of blocks or records. Ethereum [2], and BitCoin [3] are examples of blockchain technologies. Blockchains can be used for many applications such as cryptocurrency, digital wallets, ad-hoc networks, remote transactions, among other uses [1]–[7]. One notable application of blockchain is the execution of smart contracts [8].

Smart contracts are what embedded procedures are for databases: programs executed in the blockchain to manage and transfer digital assets. When used in platforms like Ethereum, the contract language is Turing-complete [9]. Therefore, smart contracts can be used in many different scenarios. For example, there are smart contracts employed to subcurrency [10], and outsourced computation [1]. Solidity [10] is the predominant programming language used to specify smart contracts on the Ethereum blockchain platform.

Smart contracts define a data structure as well as the operations used to interact with these data [10]. Far from a typical database, where the primary representation is data, and the available operations are about the structure and the content, the principal element of an Ethereum database is not just the data. In addition, the database stores the behavior provided to interact with these data, and to trigger other behaviors, by sending messages to other contracts. Ethereum is a database that works as a stored environment of contract instance (objects). Compiled versions of the contract instances are then published as part of transactions to the blockchain.

One of the challenges faced by developers of smart contracts is finding and fixing bugs. Indeed, contracts are opaque in the sense that once deployed in the blockchain it is difficult to access the value of a given contract attribute. In this paper, we focus on inspecting a smart contract state as a first step to support contract debugging.

The difficulty to inspect contract data is not a widely known problem. Although there are many tools for traditional databases to access its stored data, Ethereum and Solidity provide no such tool to inspect contract information. On the other hand, there are two practices that we can use to access contract data: (i) introducing getter methods, which requires the redeployment of the contract (if it is already running in the blockchain) and a possible data conversion (if the type is not supported as return); and (ii) using the API to acquire raw data and applying an ad-hoc decoding of the content.

Both described practices are tedious time-consuming tasks for a developer. By contrast, nowadays any programming language offers simpler ways to inspect data. Developers use such inspection to access run-time data during development or maintenance activities. Similarly, developers could benefit from smart contract run-time inspection to verify the currently stored data. Moreover, from a business perspective, companies could use contract inspection to help clients better understand the information that is actually stored in the contract. In fact, the UTOCAT[1] company deemed this interaction with clients as an important scenario, regarding the complexity to explain and understand Ethereum technology possibilities.

As a solution, we propose SmartInspect, an inspector based on pluggable property reflection. The main idea is that the binary structure of the contract is decompiled using a memory layout reification. The memory layout reification is built from the Solidity source code. Our SmartInspect architecture is based on decompilation capabilities encapsulated in mirrors [11]. Such mirrors are automatically generated from an analysis of Solidity source code. This approach allows us to access unstructured information from a deployed contract in a structured way. Therefore, our SmartInspect approach can introspect the current state of a smart contract instance without needing to redeploy it nor develop additional code for decoding.

The remainder of this paper is organized as follows. Section II starts with an example of a smart contract. In Section III, we detail the main problem our proposed approach aims to address. Section IV describes our proposed solution, SmartInpect. Section V shows a preliminary evaluation of the SmartInspect approach by comparing to other practices.

[1]https://www.utocat.com/en/, verified 2018-02-22.

Section VI provides a brief discussion on contract inspection. In Section VII, we describe the related work. Finally, Section VIII presents our conclusions and outlines possible future work ideas.

II. Smart Contract by Example

In this section, we present an example of a smart contract written in Solidity (Section II-A), and we also describe a client application in Pharo Smalltalk to interact with it (Section II-B). We use this example throughout the paper to explain the opacity problem.

A. *Poll Smart Contract*

In this example, the contract manages a poll where users are allowed to vote a single time. Only the contract owner is allowed to modify the list of voters. The poll is managed with a contract because it is used for management decisions that rely on the veracity of the information.

The following listing contains the code of this contract:

Listing 1. Solidity Poll Contract Example

```
1  pragma solidity ^0.4.16;
2
3  contract Public3StatesPoll {
4    /* Type Definition */
5     enum Choice { POSITIVE, NEGATIVE, NEUTRAL }
6     struct PollEntry { address user; Choice
          choice; bool hasVoted; }
7
8    /* Properties */
9     PollEntry[] pollTable;
10    address owner;
```

This contract defines two user types Choice (line 5) and PollEntry (line 6). A Choice models the answers to the poll (whether the vote was *positive*, *negative* or *neutral*). A PollEntry is a record representing a vote, *i.e.,* the voting user, the selected option, and if he/she has voted or not. Note that to refer to the user we need an account address (using the primitive type address) that refers to an external account.

The contract stores internally a poll table (an array of PollEntry) (line 9) and an address to the contract's owner account (line 10). The poll table is an empty array where the contract owner will eventually store the poll information (i.e., the array will have an entry for each user that is allowed to vote). The contract owner's address is used for security checks.

```
11     /* Constructor */
12    function Public3StatesPoll () {
13       owner = msg.sender;
14    }
```

Lines 11-14 define the contract constructor. This constructor is executed when the contract is deployed in the blockchain. It keeps track of the user who owns the smart contract for future reference.

```
15    function isRegistered (address voterAccount)
          returns (bool) {
16       return (voterIndex (voterAccount) > -1);
17    }
18
19    function voterIndex (address voterAccount)
          returns (int) {
20       for (uint x = 0; x < pollTable.length;
             x++) {
21          if (pollTable[x].user == voterAccount)
                {
22             return int(x);
23          }
24       }
25       return -1;
26    }
```

We define the helper function voterIndex (lines 19-26), which returns the index of the voter in the poll table. We also created the function isRegistered (lines 15-17) to determine whether the user was registered to vote by using the voterIndex function. Since array indexes in Solidity are unsigned integers (uint), we need to explicitly convert it to a regular integer (line 22).

```
27    function addVoter(address voterAccount)
          returns (uint) {
28       assert( owner == msg.sender );
29       assert( !isRegistered(voterAccount) );
30       pollTable.push(PollEntry(voterAccount,
             Choice.NEUTRAL, false));
31       return pollTable.length -1;
32    }
33
34    function vote (Choice choice) {
35       assert( isRegistered(msg.sender) );
36       uint index = uint(voterIndex(msg.sender));
37       assert( !pollTable[index].hasVoted );
38       pollTable[index].choice = choice;
39       pollTable[index].hasVoted = true;
40    }
41
42    function votesFor(Choice choice) returns
           (uint) {
43       uint votes = 0;
44       for (uint x = 0; x < pollTable.length;
             x++) {
45          if (pollTable[x].hasVoted &&
                pollTable[x].choice == choice)
46             votes = votes +1;
47       }
48       return votes;
49    }
50
51    function allParticipantsHaveVoted () returns
          (bool) {
52       for(uint x = 0; x < pollTable.length;
             x++) {
53          if (!pollTable[x].hasVoted) return
                false;
54       }
55       return true;
56    }
57
58 } //end of contract
```

The rest of the contract defines the following functions:

- *addVoter* (lines 27-32). This function registers a voter into the poll table. It tries to assert[2] that the caller is the contract owner and the voter is not already registered.

[2]The *assert* function checks for a condition and throws an exception if such condition is not met. In Solidity, exceptions undo all changes made in the invoked method.

- *vote* (lines 34-40). This function assigns the given choice to the entry related to the calling user. The user must be registered and not voted yet.
- *votesFor* (lines 42-49). It returns the number of users that voted for the given choice.
- *allParticipantsHaveVoted* (lines 51-56). It returns true if all the registered users have voted.

B. Client Side

Once our poll contract is deployed in the blockchain, we need a client application to interact with it. For example, we can implement a web application providing a user interface or a web service as a means to invoke the functions in the contract and vote or get the poll results. The following listing illustrates the code of a Poll class implemented in Pharo for our client application that will act as a façade to our contract:

Listing 2. Client side for the Voter Smart Contract

```
1 Object subclass: #Poll
2  instanceVariableNames: 'deployedContract'
3  package: #PollContract.
```

Lines 1-3 declare a Poll class with a deployedContract instance variable. This instance variable refers to a proxy to the deployed contract. This is a common implementation used for Ethereum clients coded in other languages.

```
4  Poll class>> config
5   ^{#fromAccount →self systemAccount.
6     #gas →30000. #etc}.
7
8  Poll class>> deployNewContract: src
9        accounts: accounts
10       connection: conn
11
12  deployedContract := conn deploy: source
13                     configuration: self config.
14  accounts do: [ :account |
15    deployedContract addVoter: account
          configuration: self config ].
```

The class method[3] deployNewContract:accounts:connection: (lines 8-15) receives the contract source code, a list of user accounts that are allowed to vote and a connection to the blockchain. It first deploys the contract in the blockchain using the contract source code, and then calls the addVoter:configuration: function of the new contract for each of the given user accounts. When we call functions from a deployed contract, we need to provide configuration information as well.

```
16 Poll>> config: usr
17  ^ {#fromAccount →usr account.
18     #gas →30000. #etc}.
19
20 Poll>> user: usr votes: aValue
21  deployedContract vote: aValue configuration:
        (self config: usr).
22
23 Poll>> isFinished
24   ^ deployedContract allParticipantsHaveVoted
        configuration: (self class config).
25
26 Poll>> results
27   ^ { #POSITIVE . #NEGATIVE . #NEUTRAL }
        collect: [ :value | value →
        deployedContract votesFor: value
        configuration: (self class config) ].
```

The method config: (lines 16-18) provides configuration data with the user's account. The method user:votes: (lines 20-22) invokes the function vote:configuration: from the contract using the user's configuration. Likewise, the method isFinished (lines 23-25) invokes the function allParticipantsHaveVoted() of the contract. The method results (lines 26-28) invokes repeatedly votesFor() for each of the contract choices and returns a map relating each choice to the number of people that voted for it.

It is noteworthy that the Poll class works as a thin layer over the remote contract performing remote calls to it. In the scenario of a real business application, this layer may define the complete process of a large business and its logic.

III. The Problem: Contract opaqueness

Contrary to traditional SQL databases such as Oracle or PostgreSQL which have a multitude of tools (e.g., DBeaver, Navicat, SQL Maestro, Toad, PgAdmin, etc.) to access the database schema and the *actual* data stored in a given column or row, Ethereum/Solidity does not provide any tool to inspect contract state in the referential model of the application. Since the contract is an arbitrary data type, the offered API to interact and inspect is both restricted and at a low-level of abstraction.

Contract state is *read-only* in the sense that unauthorized clients cannot interact with it. Finally, contract state is *opaque*: since it is encoded there is no simple way for a software developer to know the actual value of a contract specific attribute effectively stored in the blockchain. The remote architecture of deployed solutions should also be taken into account.

A. Contract Remote Structure

A contract is stored in the blockchain database and its object representation can be accessed in the application client layer. The Ethereum platform employs proxies based on the contract ABI[4] for covering the gap introduced by the physical location of the object (which is stored remotely in the blockchain). For example, when we invoke the method vote: configuration: in a Pharo client, the method call will be sent to a proxy that will connect through RPC (Remote Procedure Call) to the remote ABI object. The result of this method is a transaction receipt hash and, if applicable, the client will also receive any returned values of the method call. It is noteworthy that method calls to blockchain objects may have a transaction cost related with them.

[3]A class method is comparable to a static method in the Java jargon. In Pharo method call with multiple parameters place arguments in between the method name. Hence this.foobar(arg1, arg2) is expressed as this foo: arg1 bar: arg2.

[4]Application Binary Interface (ABI) is the Ethereum standard to interact with contracts. This standard encodes contract data according to its specification.

By proxying the remote contract, a client application can use the contract methods just as any other object. Moreover, the client can activate methods that will be executed elsewhere in the blockchain (by paying the transaction cost if applicable) and it will return values that we can use (as any other values for other methods and objects). This simple way to interact with the contract is unsatisfactory, since the objects cannot be inspected. Since there is no simple way to access contract properties, it makes debugging session tedious or even impossible.

B. Opaqueness problem example

We use the poll contract example (Section II) to illustrate the opaqueness problem. Let's suppose that a new user arrives a couple of days before the poll expiration date. When he tries to vote, the client system executes a routine that calls the contract's vote function and reports whether the call was successful (Listing 3). More specifically, the routine calls our method `user: votes:` defined in the Poll class (Listing 2, lines 16-17) that uses a proxy to remote call the vote function in the deployed contract.

Listing 3. Client voting routine

```
UserSession >> vote: aValue
 transactionReceipt := poll user: user votes:
     aValue.
 transactionReceipt
   onSuccess: [ :t | self informToUser ];
   onError: [ :e | self informError: e ].
```

The call will fail because the user was not a registered voter. We can see in the client code that the only point where there is a setup of users in the contract is during its deployment (Listing 2, lines 8-15). Therefore, the contract will not encounter the user and it will throw an exception. In the client, the details that caused the exception will be hidden, it will only know that the invoked method failed. Moreover, since the error is being thrown by the remote object, inspecting the contract code could identify the problem. However, a regular user does not have access to the contract code, only the people in charge of the contract have such access. For this reason, the user's only option is to submit a bug report stating that he cannot vote.

Listing 4. Contract vote function highlithing the asserts

```
  function vote (Choice choice) {
     assert( isRegistered(msg.sender) );
     uint index = uint(voterIndex(msg.sender));
     assert( !pollTable[index].hasVoted );
     pollTable[index].choice = choice;
     pollTable[index].hasVoted = true;
  }
```

The person in charge (let's call him Bob) of solving this issue, will go and check the contract code (Listing 4), and deduce that there are two possible reasons for the code to fail: (i) the user is not authorized, i.e., he/she was not registered into the contract; or (ii) the user already voted.

However, Bob cannot know for certain what caused the issue without analyzing the contract data. In this specific case, since we know the problem was caused by the unregistered voter, the easiest solution would be to change the contract state manually by calling the function `addVoter` to add the new user.

Bob will face many difficulties to find the issue. As we can see, we did not define the contract properties as public[5] (Listing 1, lines 9-10). Therefore, if Bob wants to find the nature of the error, he will need the contract instance's current state to inspect it. Bob has two possibilities then: (i) re-instantiate the contract to add a new function to return the data (i.e., create a getter method); or (ii) develop a costly, ad-hoc decoder for reading the binary content of the contract.

If Bob takes the first possibility, he will add a new function to analyze the content of the contract. Since Solidity functions cannot return arrays or structs, Bob will need to adapt its function accordingly to acquire the poll data. Moreover, the contract will have to be redeployed, creating a new instance of it. Therefore, Bob will be able to analyze the new instance data with his function, but not the previous one (which was the one that presented the issue). Besides, there are the transactional costs to redeploy the contract to be considered, as well as the inconvenience to ask the users to vote again (since it is a new instance). In our example, where there is only a few days left to close the poll, it would not be feasible for Bob to ask all users to vote again.

The second possibility is for Bob to spend time into creating an add-hoc decoder. The main advantage on this possibility is that Bob does not need to redeploy the contract. The decoder could access the complex binary slots of the contract's related storage and converted them into the desired content that Bob is trying to analyze. Since the Solidity documentation for its binary encoding is incomplete, Bob will have a difficult time to create the decoder. Moreover, this decoder is a one time solution, as it is designed for a specific data in a particular contract, i.e., Bob will not be able to easily reuse this solution to another contract. In our example, Bob might not have the time required to create such decoder before the poll expiration date.

This scenario illustrates a simple aspect of the impact of the opacity of contracts. The general concern is that the developer should be able to understand the value of a given contract attribute.

C. Challenges

There are many challenges to pierce through the opaqueness problem and reveal contract information.

- **Binary and incomplete specification.** From the technical aspects we only have the Ethereum API to access a binary representation of the contract. The first challenge we faced is an *incomplete* specification of the contract encoding performed by the Solidity compiler [10].

[5]In Solidity, when a contract attribute (state variable) does not specify its visibility, it assumes the default "internal". Internal members can only be accessed from within the current contract or contracts deriving from it. It is noteworthy that everything inside a contract can be visualized by external users. Restricted visibilities (e.g., internal, private) only prevents other contracts from modifying the information [10].

- **Inconsistent specification of hash computation.** Another challenge related to the specification is the hash computation for dynamic types. Static types use text as input for the hash calculation. However, dynamic types follow a different standard that it is not clearly specified in the documentation. For dynamic types, it is necessary to use binary data packed specifically for each type. For example, to access an array it is necessary to pack the index number and the array position (offset) into a binary representation to obtain the correct hash. Other dynamic types would require a different input to get its hash.
- **Packed and ordered data.** We also highlight the challenges on decoding types, as the compiler packs as much data as possible into contiguous memory. Therefore, we need to know the specific types in the correct order to acquire the contract data, and that is not an easy task when we have an incomplete specification.

We acknowledge as a problem the challenges and difficulties of analyzing our own objects which are deployed in the Ethereum blockchain database. To solve this problem we propose an inspector that allows the user to perceive a clean representation of the object he/she is dealing with.

IV. SmartInspect: Contract Inspector

SmartInspect is a local pluggable mirror-based reflection system for remotely deployed objects on a reflection-less system (the Ethereum platform). The goal of SmartInspect is to allow the inspection of known contracts based on its source code focusing on the debugging properties of interactiveness and distribution [12]. This reflective approach allows a user to see the contents of any contract instance of the given source code, without needing to redeploy, nor develop any ad-hoc code.

SmartInspect is implemeted in Pharo and it is publicly available as part of the SmartShackle tool suite.[6]

A. The Basics

The general idea of the Smart Inspector is to decompile the storage layout encoded by the Ethereum API (Figure 1). The decompilation employs a local pluggable mirror-based reflection architecture for remotely deployed objects on an Ethereum network.

Figure 2 shows the process to inspect contract, the pluggable reflective architecture generates a mirror for a given contract's source code by using its AST (Abstract Syntax Tree). Then, we use this mirror to extract information from a remote contract instance deployed in the blockchain (which is encoded as a binary memory layout). The contract data we gathered is exposed in four different formats: (i) data proxy object (REST), (ii) Pharo widget user interface, (iii) JSON, and (iv) HTML.

This approach allows us to access remote structureless information in a structured way. Our solution meets most of the desirable properties that are important for remote debugging namely: interactiveness and distribution [12].

[6]https://github.com/RMODINRIA-Blockchain/SmartShackle, http://smart-shackle.io/, verified 2018-02-22.

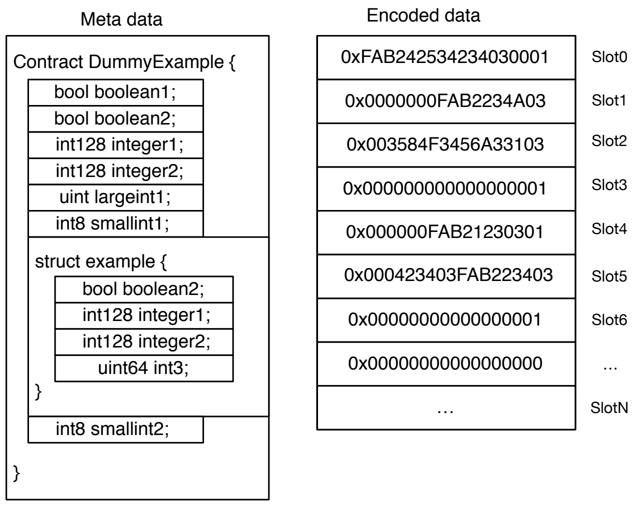


Fig. 1. Static storage with the related code definition.

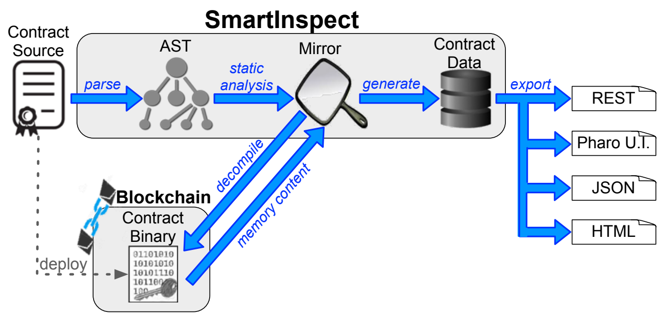


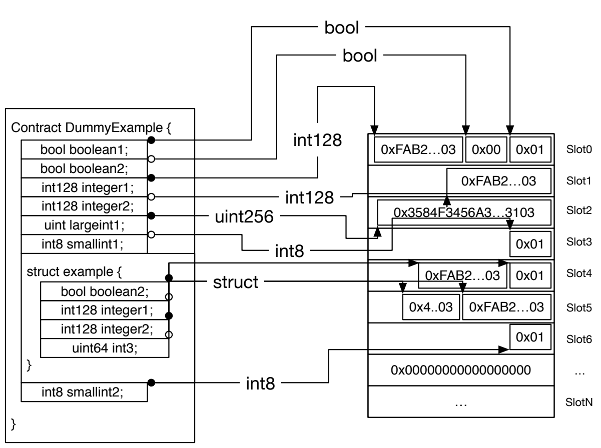
Fig. 2. SmartInspect building process.

B. Discovering the Memory Layout

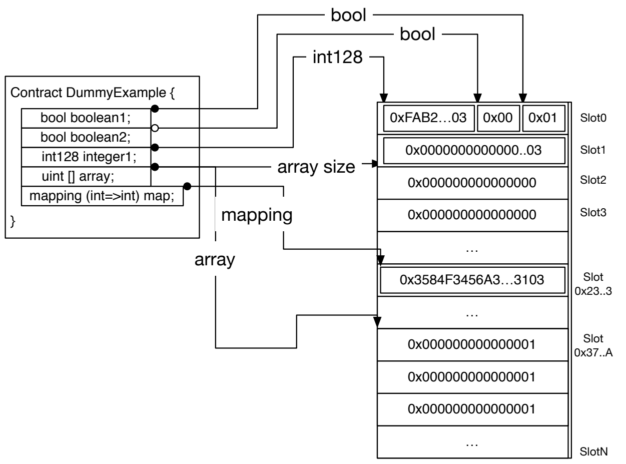
First we needed to decompile the binary representation of a deployed contract (i.e., a contract ABI) to discover the information inside the contract instance. That would resolve the opaqueness problem we described earlier, since we would be able to understand contract attributes.

The Ethereum API provides only one way to access memory layout of a contract: getStorageAt calls. This call gives access to a tree where information is encoded into slots accessible through contiguous indexes, for statically allocated memory (static types), and accessible by Keccak hash for dynamically allocated memory (variable sized arrays and mappings) [10]. It was a big challenge to decompile the memory layout because the Solidity documentation is incomplete. We had to reverse engineer some of the encoding performed by the compiler by ourselves.

There are two key restrictions in memory access: types and order. Each memory slot stores up to 32 bytes. As general policy, the compiler tries to pack as much data as possible for basic types. For example, two booleans and one *int128* occupy 18 bytes from a slot, one byte for each boolean plus 16 bytes



(a)



(b)

Fig. 3. Memory Layout Representation: (a) Static, (b) Dynamic.

for the *int128*. If we add another type that can not fit into the current slot, then the compiler places it in a new memory slot.

In the case of structs, they always start in a new slot and their data may take as many slots as needed. Any data after the struct will be encoded to start in a new slot, no matter if it could fit into the remaining struct slot. We show some of these encoding details for static types in Figure 3a.[7] Then, transversally, all the static representation of the different variables is available contiguously in the first slots of the binary representation, from 0 to *N*.

Dynamically allocated data types (e.g., arrays, mappings) are encoded in other hashed addresses, as shown in Figure 3b.[8] Therefore, the decompilation has to dive into a sometimes contiguous, sometimes indexed slots, with arbitrary allocations of space that may depend on the size of the type or in the next and/or previous type, to be able to read the stored content.

Therefore, we were successful in addressing our concern to introspect a deployed contract data, as we resolved the contract opaqueness problem by decompiling the binary representation and decoding the memory layout.

C. Building the Mirror

After we decoded the memory layout, we still needed to apply our solution to any contract for a general reusable solution. We employ a mirror-based architecture [11] that mimics the structure of any contract for us to access the memory layout that we can decode. A mirror works like an independent meta-programming layer which splits the concern of reflection capabilities into a mirror object.

First, we require the contract source code as input to start building the mirror. Then, we parse the source to create an AST (Abstract Syntax Tree). By interpreting the AST, we are able to know every type declared in the contract in the correct order. As described earlier (Section IV-B), we need know the types in order to decoded the memory layout and access the contract data.

Aiming at a general solution, we model configurable mirror objects that allow us to interact with deployed contract instances of the same configuration (usually meaning the same contract deployed in the blockchain). Our approach builds a composite mirror object, called ContractMirror, whose each component knows how to decode, in order, the contract state (Figure 4, the left side of the diagram). For each variable or struct a corresponding elementary mirror is added to the composite.

At this point we have a mirror with the structural representation of the AST that knows, in order, each contract property with its related type. Now our approach builds a representation of the memory layout to access the stored data. By using static code analysis provided by the AST, we can find the exact place of storage of every contract property. Moreover, we map the contract properties to its corresponding memory slot. Therefore, the mirror uses this mapping to gather the contract data for inspection. Figure 4 (on the right part) shows the mirror's memory mapping as a class diagram.

D. Inspection Example

Getting back at our problem example (Section III-B), where the person in charge (Bob) needed to verify the data in a deployed poll contract instance. Back then, Bob had only two possibilities to address an user's issue, and both were

[7]The figure shows contract attributes definitions with the arrows pointing to the memory representation. In this particular case, we first defined two booleans and one int128, which occupy 18 bytes from the first memory slot. Next, we define another int128, one uint (256 bits), and an int8 that will occupy the second, third, and fourth memory slots respectively. The struct encoding places it in the fifth and sixth slot. Finally, even though the last variable (int8) could fit in the struct slot, the encoding places it in the next available slot (the seventh).

[8]We decided to represent the slot as contiguous data to facilitate its representation. In this case, we first defined three static variables (two booleans and one int128) to show they are placed in the first available slots. Then, we defined two dynamic types: one variable size array, and a mapping (which is similar to a hash table). Unlike static types, they are not encoded into the first memory slots but placed elsewhere in the memory.

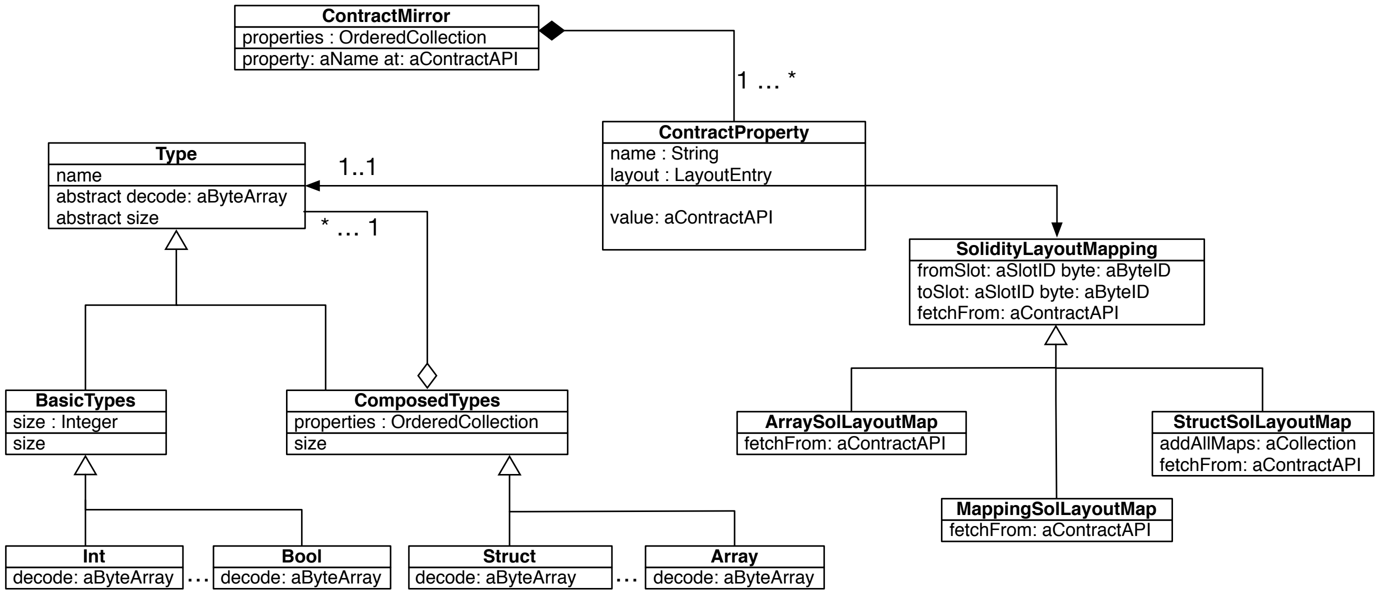
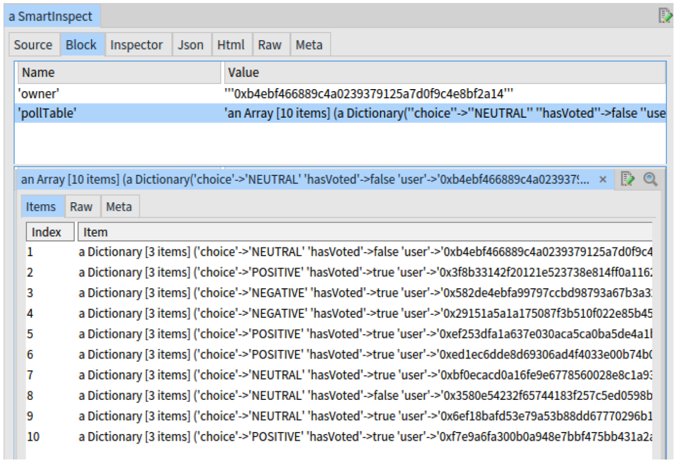


Fig. 4. Contract Mirror UML Diagram

not feasible. Now, let's suppose that Bob just learned about SmartInspect, which gives him a third possibility, inspect the contract data using our tool. Since Bob is the person in charge he has access to both the contract source code and its deployed binary representation. Bob executes SmartInspect on its poll contract and he is presented with the data from the pollTable property (Figure 5). Finally, Bob can see that the user was not registered, and he can now easily fix the problem by executing the *addVoter* function on his client application. This simple example illustrates the importance of contract inspection.



Name	Value
'owner'	'''0xb4ebf466889c4a0239379125a7d0f9c4e8bf2a14'''
'pollTable'	'an Array [10 items] (a Dictionary(''choice''->''NEUTRAL'' ''hasVoted''->false ''use

an Array [10 items] (a Dictionary('choice'->'NEUTRAL' 'hasVoted'->false 'user'->'0xb4ebf466889c4a023937...

Index	Item
1	a Dictionary [3 items] ('choice'->'NEUTRAL' 'hasVoted'->false 'user'->'0xb4ebf466889c4a0239379125a7d0f9c4
2	a Dictionary [3 items] ('choice'->'POSITIVE' 'hasVoted'->true 'user'->'0x3f8b33142f20121e523738e814ff0a116
3	a Dictionary [3 items] ('choice'->'NEGATIVE' 'hasVoted'->true 'user'->'0x582de4ebfa99797ccbd98793a67b3a3
4	a Dictionary [3 items] ('choice'->'NEGATIVE' 'hasVoted'->true 'user'->'0x29151a5a1a175087f3b510f022e85b45
5	a Dictionary [3 items] ('choice'->'POSITIVE' 'hasVoted'->true 'user'->'0xef253dfa1a637e030aca5ca0ba5de4a1
6	a Dictionary [3 items] ('choice'->'POSITIVE' 'hasVoted'->true 'user'->'0xed1ec6dde8d69306ad4f4033e00b74b
7	a Dictionary [3 items] ('choice'->'NEUTRAL' 'hasVoted'->true 'user'->'0xbf0ecacd0a16fe9e6778560028e8c1a9
8	a Dictionary [3 items] ('choice'->'NEUTRAL' 'hasVoted'->false 'user'->'0x3580e54232f65744183f257c5ed0598b
9	a Dictionary [3 items] ('choice'->'NEUTRAL' 'hasVoted'->true 'user'->'0x6ef18bafd53e79a53b88dd67770296b1
10	a Dictionary [3 items] ('choice'->'POSITIVE' 'hasVoted'->true 'user'->'0xf7e9a6fa300b0a948e7bbf475bb431a2

Fig. 5. SmartInspect Pharo User Interface Screenshot

V. Preliminary Evaluation

In this section, we present a preliminary evaluation of the SmarInspect approach. The goal of this evaluation was to investigate whether SmartInspect implements the necessary and desirable features for an inspector. We used the following four characteristics used for remote debugging by Papoulias [12]: Interactiveness, Distribution, Security, and Instrumentation. We also analyzed other five characteristics that are important for a blockchain remote inspector: Privacy, Pluggability, Consistency, Reusability, and Unrestricted types. We detail the nine characteristics as follows:

- *Interactiveness:* the inspector shows the object's state in real time. A lack of interactiveness could be a problem in blockchain platforms because of the contract's state may change during inspection and the user would be presented with outdated information.
- *Distribution:* the inspector can be extended for other technologies. Ideally, a debugger or inspector should rely on a middleware that is extensible. For smart contracts, the inspector should be extensible over different smart contract languages and blockchain technologies.
- *Security:* since remote debugging access a target through a network, it is important to ensure security from both ends. On the target side, the inspector should not have unrestricted access to its device; this is already ensured by the blockchain platform.
- *Instrumentation:* the inspector can alter the semantics of a process to assist in debugging. Basically, this is the mechanism to halt the process and inspect it at that point (e.g., breakpoints and watchpoints). This characteristic is not possible in blockchain platforms, as we cannot modify the deployed contract code to halt a function in the middle of its execution in the blockchain.
- *Privacy:* inspection should not breach or compromise data privacy by exposing data to unauthorized users. When considering smart contracts, a lack of privacy is dangerous

as it could be exploited by malicious users to acquire illicit advantages and resources.

- *Pluggability:* the inspector can be used on existing objects without the need to re-instantiate the objects or the system. For contracts, this means we can inspect existing deployed contracts without any dependency on the contract side, or the need to redeploy the contract. An unpluggable approach has the disadvantage of requiring the redeployment of a contract, which has non-trivial transactional costs.
- *Consistency:* the representation used by the inspector must reveal the information in a consistent manner, i.e., the inspection must reflect the current state of the deployed contract.
- *Reusability:* the inspector can be reused for different contracts. Lack of reusability would require a developer to spend time redefining the inspection for each individual contract.
- *Unrestricted Types*: the inspection can handle all types of objects. In contrast, a type-restricted inspector supports only a subset of data types (e.g., primitive types, static types).

We analyzed SmartInspect according to the characteristics of inspection tools just presented. Even though we wanted to compare SmartInspect against related approaches, as far as we know, there are no other inspectors available for Solidity smart contract. Therefore, we compared our approach against two other practices to access contract data: Getter methods, and ad-hoc Decoder (Table I).

TABLE I
RELATED TECHNIQUES COMPARISON

Characteristic	**SmartInspect**	**Getter**	**Decoder**
Interactiveness	Yes	Partial	Partial
Distribution	Yes	No	No
Security	Yes	Yes	Yes
Instrumentation	No	No	No
Privacy	Yes	No	Yes
Pluggability	Yes	No	Yes
Consistency	Yes	Yes	Yes
Reusability	Yes	No	No
Unrestricted Types	Yes	No	Yes

As we can see from Table I, SmartInspect's only characteristic flaw is related to *instrumentation* for remote debugging. However, this is a limitation imposed by the Ethereum blockchain technology rather than a design flaw in our approach.

Getter methods are a simple solution, since they are cheap to implement and easy to test. The developer does not need to know the memory layout of a contract to create getter methods. However, if the developer forgets to make a getter for a given attribute, he/she will need to re-deploy the contract and, most often, lose the data from the previous instance. Solidity does not support the return of many complex types (e.g, structs, mappings) on its functions. Therefore, a developer might need to adapt his/hers data or function to provide access to a complex type. Moreover, the easy access to the data may cause a loss of privacy, since getter methods are a public part of the contract binary encoding.

Another practice is the ad-hoc Decoder that uses the Ethereum API on the memory slots. This is a complex task since it demands a deep understanding of the memory layout of each contract a developer plans to inspect. It also requires a developer to know the type of each attribute and code the ad-hoc decoder accordingly. Its advantages are that it allows access to data without loss of privacy and without the need to redeploy the contract. In fact, SmartInspect uses this concept of decoding memory layouts as a part of its inspection process.

VI. Discussion

In this section, we discuss our evaluation (Section VI-A) and the possible benefits for inspecting smart contracts (Section VI-B).

A. Evaluating the Inspector

In our preliminary evaluation, we compared SmartInspect against two practices that can be used to access contract data (Section V). We acknowledge that we need other inspectors for a better comparison, since practices are a less polished solution than a fully designed approach for inspection. However, as far as we know, there is no other inspector tool available for Ethereum blockchain.

In the future, we plan to improve the evaluation by comparing with other inspectors, and performing an user driven evaluation.

B. Benefits from Inspecting a Contract

There are several benefits for inspecting a smart contract on a blockchain platform. We highlight the following:

- **Easier to Understand.** Blockchain provides a mechanism that can be used to build trust between entities without a middleman [2], [3]. In the blockchain environment, smart contracts became a popular way to transfer digital assets among such entities [8], [9]. From a beginners perspective, it is better to work with concrete examples to understand a new concept. Thus, an inspector provides a simple way to access the contract state, which facilitates its understanding.
- **Find Bugs.** Contract inspection can help developers to find bugs more easily (as we illustrated in Section III-B).
- **Transparency.** Supporting inspection of contracts can increase transparency and improve overall trust among entities dealing with blockchain. For instance, it is possible for two entities to show the current state of their contracts to each other promoting transparency in their interactions.
- **Encourage the adoption of Contracts**. By allowing contract inspection, we can promote trust and transparency on blockchain platforms to companies and institutions. This will encourage more people to adopt smart contracts for their business or academic activities.

VII. Related Work

We organized the related work into three groups: (i) inspecting and debugging, (ii) reverse engineering, and (iii) blockchains and smart contracts.

A. Inspecting and Debugging

Chis et al. [13] performed an exploratory research to better understand what developers expect from object inspectors, and based on that feedback they propose a novel inspector model. The authors interviewed 16 developers for a qualitative study, and a quantitative study conducting an online survey where 62 people responded. Both studies were used to identify four requirements needed in an inspector. Then they propose the Moldable Inspector, which indicates a new model to address multiple types of inspection needs. We followed the lessons taken by the Moldable Inspector when creating SmartInspect. We deem noteworthy the multiple views aspect, as SmartInspect can present its inspected data in four different views (REST, Pharo U.I., JSON, and HTML).

Papoulias [12] gives a deep analysis on remote debugging. As discussed by the author, remote debugging is specially important for devices that cannot support local development tools. The author identifies four important characteristics for remote debugging: interactiveness, instrumentation, distribution, and security. Based on the identified properties, Papoulias proposed a remote debugging model, called Mercury. Mercury employs a mirror based approach and an adaptable middleware. We used Papoulias research as an inspiration to create SmartInspect, specially relying on mirror for the remote inspection.

Salvaneschi and Mezini [14] propose a methodology, called RP Debugging, to debug reactive programs more effectively. The authors discuss that reactive programming is more customizable and easier to understand than its alternative the observer design pattern. The authors also present the main problems and challenges to debug reactive programs, and the main design decisions when creating their methodology. Although our inspector is from a different application domain, the RP Debugging design served as inspiration to plan our own inspecting approach.

B. Reverse Engineering

Srinivasan and Reps [15] developed a reverse engineering tool to recover class hierarchical information from a binary program file. Their tool also extracts composition relationships as well. They use dynamic analysis to obtain object traces and then they identify the inheritance and composition information among classes on those traces. The authors experiments show that their recovered information is accurate according to their metrics. The author's tool contrasts with SmartInspect as, we use static analysis and they use dynamic analysis.

Caballero et al. [16] propose an approach to reverse engineer protocols by using dynamic analysis on program binaries. As stated by the authors, this approach differs from others that extract protocol information purely from network traces. The authors argue the importance to extract the protocol information, specially when there is no access to its specification, for network security applications. They used 11 programs that implemented five different protocols for their evaluation. The authors' technique also contracts with SmartInspect since they use dynamic analysis.

Fisher et al. [17] propose a multi phase algorithm to process *ad-hoc data* without human interaction. The authors describe *ad-hoc data* as semistructured information that does not have tools easily available. Basically, their algorithm reverse engineer the *ad-hoc data* into a domain specific language, which is used to generated a set of tools such as parsers, printers, query engine, and others. The authors evaluate the performance and correctness of their approach by using different benchmarks.

Lim et al. [18] designed an analysis tool called File Format Extractor for x86 (FFE/x86). They extract information from an executable file to perform several process, including static analysis. Their evaluation consists in applying their tool in three systems: gzip, png2ico, and ping. The authors work is similar to SmartInspect in a way that both approaches use static analysis in one of its steps to enhance the reverse engineering.

C. Blockchain and Smart Contracts

Dinh et al. [7] describe a benchmarking framework to analyze private blockchain platforms. The authors contrast the different among public blockchain platforms (e.g., Ethereum) and private ones. For instance, private blockchain show more focus towards secure authentication. Although, their framework was designed for private blockchains, they evaluate it using public ones as well. Their evaluation measured four aspects (throughput, latency, scalability, and fault tolerance) in two blockchains (Ethereum and Hyperledger) and the Parity technology. The benchmark framework provided an in dept analysis of blockchain platforms that we used when we designed SmartInspect.

Luu et al. [8] investigates the security problems of executing smart contracts on the Ethereum platform. They also propose solutions to make the contracts more secure. The authors presents several scenarios and the possible malicious exploits for those scenarios. Based on the presented vulnerabilities, they propose solutions to make contracts more secure. The authors also propose a tool, called Oyente, that flags potential security flaws when coding smart contracts. Similarly to SmartInspect, Oyente also uses the contract bytecode to makes its security recommendations. However, our tool uses the memory layout to access the data, and Oyente uses the bytecode for a symbolic execution and security analysis.

Bhargavan et al. [9] proposed a framework to convert contracts to F* and then improve their security. F* is a functional programming language proposed by the authors in their work. According to them, F* was designed to better verify correctness of contracts. Their approach decompiles the contract bytecode into a special F* code to verify low-level properties; similarly it also compiles a contract source into an F* version to verify high-level properties. The authors did a preliminary evaluation where 46 contracts were translated to F*. Their approach also employs decompilation of contract bytecode and parses source code, similar to our SmartInspect.

VIII. Conclusion

In this paper, we present the specific problems of inspecting Solidity smart contracts. Smart contract opaqueness added to the problems of reverse engineering compiler encoding and packing on different entity types makes the inspecting of values in smart contracts almost impossible for regular developers.

Our approach implementation, SmartInspect, is a local pluggable mirror-based reflection system for remotely deployed objects on reflection-less systems (the Ethereum platform). SmartInspect allows the inspection of known contracts based on its source code focusing on the debugging properties of interactiveness and distribution. This reflective approach allows a user to see the contents of any contract instance of the given source code, without needing to redeploy, nor develop any ad-hoc code to decode the memory representation.

We planned the following ideas for future work: (i) extend SmartInspect to inspect other smart contract languages employed in the Ethereum platform besides Solidity (e.g., Serpent, Viper, LLL); (ii) extend SmartInspect to support other blockchain platforms (e.g., Hyperledger); (iii) improve the introspection capabilities to support full debugging on smart contracts; and (iv) improve the evaluation by using SmartInspect on a big contract database, defining metrics for evaluating contract inspectors, comparing it with other inspectors, and performing an extensive user evaluation.

Acknowledgment

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. This research was also supported by UTOCAT.

References

[1] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 706–719. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813659

[2] Ethereum Foundation, "Ethereum's white paper." 2014. [Online]. Available: https://en.wikibooks.org/wiki/LaTeX/Bibliography_Management

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." 2009. [Online]. Available: bitcoin.org

[4] A. Hari and T. V. Lakshman, "The internet blockchain: A distributed, tamper-resistant transaction framework for the internet," in *15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: ACM, 2016, pp. 204–210. [Online]. Available: http://doi.acm.org/10.1145/3005745.3005771

[5] B. Leiding, P. Memarmoshrefi, and D. Hogrefe, "Self-managed and blockchain-based vehicular ad-hoc networks," in *2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, ser. UbiComp '16. New York, NY, USA: ACM, 2016, pp. 137–140. [Online]. Available: http://doi.acm.org/10.1145/2968219.2971409

[7] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1085–1100. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064033

[6] S. Dziembowski, "Introduction to cryptocurrencies," in *22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1700–1701. [Online]. Available: http://doi.acm.org/10.1145/2810103.2812704

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 16)*. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[10] Ethereum Foundation, "Solidity documentation release 0.4.18," Tech. Rep., 2017. [Online]. Available: https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[11] G. Bracha and D. Ungar, "Mirrors: design principles for meta-level facilities of object-oriented programming languages," in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*. New York, NY, USA: ACM Press, 2004, pp. 331–344. [Online]. Available: http://bracha.org/mirrors.pdf

[12] N. Papoulias, "Remote Debugging and Reflection in Resource Constrained Devices," Theses, Université des Sciences et Technologie de Lille - Lille I, Dec. 2013. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00932796

[13] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba, "The moldable inspector," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 44–60. [Online]. Available: http://doi.acm.org/10.1145/2814228.2814234

[14] G. Salvaneschi and M. Mezini, "Debugging reactive programming with reactive inspector," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 728–730. [Online]. Available: http://doi.acm.org/10.1145/2889160.2893174

[15] V. Srinivasan and T. Reps, "Recovery of class hierarchies and composition relationships from machine code," in *23rd International Conference Compiler Construction (ETAPS/CC 2014), Grenoble, France, April 5-13, 2014. Proceedings*, A. Cohen, Ed. Springer Berlin Heidelberg, 2014, pp. 61–84. [Online]. Available: https://doi.org/10.1007/978-3-642-54807-9_4

[16] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007, pp. 317–329. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315286

[17] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: Fully automatic tool generation from ad hoc data," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. New York, NY, USA: ACM, 2008, pp. 421–434. [Online]. Available: http://doi.acm.org/10.1145/1328438.1328488

[18] J. Lim, T. W. Reps, and B. Liblit, "Extracting output formats from executables," in *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, 2006, pp. 167–178. [Online]. Available: https://doi.org/10.1109/WCRE.2006.29

Smart Contracts Vulnerabilities: A Call for Blockchain Software Engineering?

Giuseppe Destefanis
School of Computer Science
University of Hertfordshire, UK
g.destefanis@herts.ac.uk

Michele Marchesi, Marco Ortu, Roberto Tonelli
University Of Cagliari, Italy
marchesi@unica.it
marco.ortu@diee.unica.it, roberto.tonelli@dsf.unica.it

Andrea Bracciali
University Of Stirling, UK
abb@cs.stir.ac.uk

Robert Hierons
Brunel University, UK
rob.hierons@brunel.ac.uk

***Abstract*—Smart Contracts have gained tremendous popularity in the past few years, to the point that billions of US Dollars are currently exchanged every day through such technology. However, since the release of the Frontier network of Ethereum in 2015, there have been many cases in which the execution of Smart Contracts managing Ether coins has led to problems or conflicts. Compared to traditional Software Engineering, a discipline of Smart Contract and Blockchain programming, with standardized best practices that can help solve the mentioned problems and conflicts, is not yet sufficiently developed. Furthermore, Smart Contracts rely on a non-standard software life-cycle, according to which, for instance, delivered applications can hardly be updated or bugs resolved by releasing a new version of the software.**

In this paper we advocate the need for a discipline of Blockchain Software Engineering, addressing the issues posed by smart contract programming and other applications running on blockchains. We analyse a case of study where a bug discovered in a Smart Contract library, and perhaps "unsafe" programming, allowed an attack on *Parity*, a wallet application, causing the freezing of about 500K Ethers (about 150M USD, in November 2017). In this study we analyze the source code of Parity and the library, and discuss how recognised best practices could mitigate, if adopted and adapted, such detrimental software misbehavior. We also reflect on the specificity of Smart Contract software development, which makes some of the existing approaches insufficient, and call for the definition of a specific Blockchain Software Engineering.

***Index Terms*—smart contracts; blockchain; software engineering;**

I. Introduction

Smart contracts are becoming more and more popular nowadays. They were first conceived in 1997 and the idea was originally described by computer scientist and cryptographer Nick Szabo as a kind of digital vending machine. He described how users could input data or value and receive a finite item from a machine (in this case a real-world snack or a soft drink).

More in general, *smart contracts* are self-enforcing agreements, i.e. contracts, as we intend them in the real world, but expressed as a computer program whose execution enforces the terms of the contract. This is a clear shift in the paradigm: untrusted parties demand the trust on their agreement to the *correct* execution of a computer program. A properly designed smart contract makes possible a crow-funding platform without the need for a trusted third party in charge of administering the system. It is worth remarking that such a third party makes the system centralized, where all the trust is demanded to a single party, entity, or organisation.

Blockchain technologies are instrumental for delivering the trust model envisaged by smart contracts.

In the example of a crowd-funding platform for supporting projects, the smart contract would hold all the received funds from a project's supporter (it is possible to pay a smart contract). If the project fully meets its funding goals, the smart contract will automatically transfer the money to the project. Otherwise, the smart contract will automatically refund the money to the supporters.

Since smart contracts are stored on a blockchain, they are immutable, public and decentralised. Immutability means that when a smart contract is created, it cannot be changed again and no one will be able to tamper with the code of a contract.

The decentralised model of immutable contracts implies that the execution and output of a contract is validated by each participant to the system and, therefore, no single party is in control of the money. No one could force the execution of the contract to release the funds, as this would be made invalid by the other participants to the system. Tampering with smart contracts becomes almost impossible.

The first Blockchain to go live was the Bitcoin's Blockchain [1] in 2009. It introduced the idea of programs used to validate agreement amongst untrusted parties: Bitcoin transactions are subject to the successful termination of a non-Turing complete program in charge of validating things like ownership and availability of the crypto money. The biggest blockchain that currently supports smart contracts is Ethereum, which was specifically created and designed with an extended execution model for smart contracts in 2014 [2]. Contracts in Ethereum can be programmed with Solidity, a programming language developed for Ethereum.

A few years down the line, several detrimental software misbehaviors, which caused considerable monetary loss and community splits, have posed the problem of the correct design, validation and execution of smart contracts.

In this paper we advocate the need for a discipline of *Blockchain Software Engineering*, addressing the issues posed by smart contract programming and other applications running on blockchains. Blockchain Software Engineering will specifically need to address the novel features introduced by decentralised programming on blockchains. These will be discussed in more detail in the rest of this paper.

We consider a case study, the recent attack to the *Parity* wallet (2017). A bug discovered in a smart contract library used by the Parity application, caused the freezing of about 500K Ethers (see [3] for a summary).

We analyze the source code of Parity and the library, and reflect on the specificity of smart contract software development, noting some shortfalls of standard approaches to software development. We then discuss how recognized best practices in traditional Software Engineering could have mitigated, if adopted and adapted, such detrimental software misbehavior.

This paper aims to contribute a first step towards the definition of Blockchain Software Engineering.

II. Background

In this section we briefly introduce the blockchain and smart contracts technology, their execution environment and model. Since our study is focused on the Ethereum platform we will use it as example but the concepts presented here are of general validity.

A. Decentralized Ledgers

A blockchain is essentially a shared ledger that stores transactions, holding pieces of information, in a decentralized peer-to-peer network. Nodes are called *miners* and each one maintains a consistent copy of the ledger. Transactions are grouped together into blocks, each hash-chained with the previous block. Such a data structure is the so called *blockchain*, shown in Figure 1.

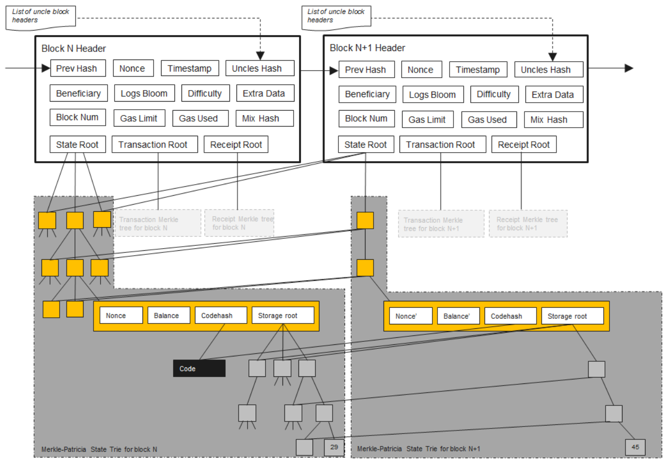


Fig. 1. Blockchain and Ethereum architecture. Each block of the chain consists of a large number of single transactions.

Miners use a consensus protocol in order to agree on the validity of each block, called *Nakamoto Consensus Protocol* [1]. At any time miners group their choice of incoming new transactions in a new block, which they intend to add to the public blockchain. Nakamoto consensus uses a probabilistic algorithm for electing the miner who will publish the next valid block in the blockchain. Such a miner is the one who solves a computationally demanding cryptographic puzzle. Such a procedure is called *proof-of-work*. All other miners verify that the new block is correctly constructed (e.g. no virtual coin is spent twice) and update their local copy of the blockchain with the new block. Bitcoin transactions essentially record the transfer of coins from one address, a wallet say, to another one. Differently, Ethereum transactions also include *contract-creation* transactions and *contract-invoking* transactions. The former ones record a smart contract on the blockchain, and the latter ones cause the execution of a contract functionality (which enforces some terms of the contract). We refer the reader to the original white papers of Bitcoin and Ethereum [1], [2] for further details.

B. Ethereum Smart Contracts

A *Smart Contract* (SC) is a *full-fledged program* stored in a blockchain by a *contract-creation* transaction. A SC is identified by a *contract address* generated upon a successful creation transaction. A blockchain state is therefore a mapping from addresses to accounts. Each SC account holds an *amount of virtual coins* (Ether in our case), and has its own private *state* and *storage*. An Ethereum SC account hence typically holds its executable code and a state consisting of:

- private storage
- the amount of virtual coins (Ether) it holds, i.e. the contract *balance*.

Users can transfer Ether coins using transactions, like in Bitcoin, and additionally can *invoke* contracts using *contract-invoking* transactions. Conceptually, Ethereum can be viewed as a huge *transaction-based state machine*, where its state is updated after every transaction and stored in the blockchain.

A Smart Contract's source code manipulates variables in the same way as traditional imperative programs. At the lowest level the code of an Ethereum SC is a stack-based bytecode language run by an Ethereum virtual machine (EVM) in each node. SC developers define contracts using high-level programming languages. One such language for Ethereum is Solidity [4] (a JavaScript-like language), which is compiled into EVM bytecode. Once a SC is created at an address *X*, it is possible to invoke it by sending a contract-invoking transaction to the address *X*. A contract-invoking transaction typically includes:

- payment (to the contract) for the execution (in Ether).
- input data for the invocation.

1) Working Example: Figure 2 shows a simple example of SC reported in [5], which rewards anyone who solves a problem and submits the solution to the SC.

A *contract-creation* transaction containing the EVM bytecode for the contract in Figure 2 is sent to miners. Eventually,

```
1  contract Puzzle {
2
3    address public owner ;
4    bool public locked ;
5    uint public reward ;
6    bytes32 public diff ;
7    bytes public solution ;
8
9    function Puzzle ()  {// constructor
10     owner = msg.sender ;
11     reward = msg.value ;
12     locked = false ;
13     diff = bytes32 (11111); // pre-defined
         difficulty
14   }
15
16   function (){ // main code , runs at every
       invocation
17     if ( msg.sender == owner ) { // update reward
18       if ( locked )
19         throw ;
20       owner.send(reward);
21       reward = msg.value ;
22     } else if ( msg.data.length > 0) {
23       // submit a solution
24       if ( locked ) throw ;
25       if ( sha256 ( msg.data ) < diff ) {
26         msg.sender.send(reward); // send reward
27         solution = msg.data ;
28         locked = true ;
29       }
30     }
31   }
32 }
```

Fig. 2. Smart Contracts example.

the transaction will be accepted in a block, and all miners will update their local copy of the blockchain: first a *unique* address for the contract is generated in the block, then each miner executes locally the *constructor* of the **Puzzle** contract, and a local storage is allocated in the blockchain. Finally the EVM bytecode of the anonymous function of **Puzzle** (Lines 16+) is added to the storage.

When a *contract-invoking* transaction is sent to the address of **Puzzle**, the function defined at Line 16 is executed by default. All information about the sender, the amount of Ether sent to the contract, and the input data of the invoking transaction are stored in a default input variable called *msg*. In this example, the *owner* (namely the user that created the contract) can update the *reward* (Line 21) by sending Ether coins stored in `msg.value` (`if` statement at Line 17), after sending back the current *reward* to the *owner* (Line 20).

In the same way, any other user can submit a solution to **Puzzle** by a *contract-invoking* transaction with a *payload* (i.e., `msg.data`) to claim the reward (Lines 22-29). When a correct solution is submitted, the contract sends the reward to the sender (Line 26).

2) Gas System: It is worth remarking that a SC is run on the blockchain by each miner deterministically replicating the execution of the SC bytecode on the local copy of the blockchain. This, for instance, implies that in order to guarantee coherence across the copies of the blockchain, code must be executed in a strictly deterministic way (and therefore, for instance, the generation of random numbers may be problematic).

Solidity, and in general high-level SC languages, are Turing complete in Ethereum. In a decentralised blockchain architecture Turing completeness may be problematic, e.g. the replicated execution of infinite loops may potentially *freeze* the whole network.

To ensure fair compensation for expended computation efforts and limit the use of resources, Ethereum pays miners some fees, proportionally to the required computation. Specifically, each instruction in the Ethereum bytecode requires a pre-specified amount of *gas* (paid in Ether coins). When users send a *contract-invoking* transaction, they must specify the amount of gas provided for the execution, called *gasLimit*, as well as the price for each gas unit called *gasPrice*. A miner who includes the transaction in his proposed block receives the transaction fee corresponding to the amount of gas that the execution has actually burned, multiplied by *gasPrice*. If some execution requires more gas than *gasLimit*, the execution terminates with an exception, and the state is rolled back to the initial state of the execution. In this case the user pays all the *gasLimit* to the miner as a counter-measure against resource-exhausting attacks [6].

III. Case Study and Methodology

Recently Ethereum suffered a supposedly involuntary hack, in which an inexperienced developer froze multiple accounts managed by the Parity Wallet application. The hack has suddenly risen to the news since the amount of Ether coins in the frozen account was estimated to be 513,774.16 Ether (equivalent to 162M USD at the time). In November 2017 the hack became (in)famously known as the *Parity Wallet hack*. This was the result of a single library code deletion.

This case replicated a similar problem exploited by another hacker on the same Parity Wallet code, a few months earlier (July 2017). In that case, a multi-signature wallet was hacked and set in control of a single owner, who acquired all the Ether coins of that single wallet.

The Parity Wallet hack represents a paradigmatic example of problems that may currently occur in the development of smart contracts and blockchain-related software in general. Such problems are associated to the lack of suitably standardised best practices for blockchain software engineering.

In the rest of this paper, we will analyse the Parity Wallet hack case by applying static code analysis to the Parity library. We aim to understand the code structure and, more importantly, the link between smart contracts and the functions defined in smart contract libraries. After analyzing the Solidity code of the Wallet, we will outline the events that ended up with more than 500k Ether frozen. Then, we will elaborate on viable solutions from the perspective of Software Engineering. These represent potential general solutions for cases analogous to the Parity Wallet hack.

IV. Structure and Functionality of Parity

Parity is an Ethereum client that is integrated directly into web browsers. It allows the user to access the basic

Ether and token wallet functions. It is also an Ethereum GUI browser that provides access to all the features of the Ethereum network, including DApps (decentralised applications). Parity also operates as an Ethereum full node, which means that the user can store and manage the blockchain on his own computer. It is a complex and critical decentralised application.

Solidity and the EVM provide three ways to call a function on a smart contract: CALL, CALL-CODE, and DELEGATE-CALL. The former is a call to a function that will be executed in the environment of the called contract. The other two calls execute the called code in the caller environment. Many library calls on Ethereum are implemented with DELEGATE-CALL, typically by deploying a contract that serves as a library: the contract has functions that anyone can call, and these may be used, for instance, to make changes in the storage of the calling contracts. Solidity has some syntactic constructs which allow *libraries* offering DELEGATE-CALLs to be defined and ”imported” by other contracts. However, at the EVM level the library construction disappears, and DELEGATE-CALLs and other calls are actually deployed as smart contract functionalities.

1) Statically linked libraries: It is possible to embed all the library code in a smart contract, e.g. the multi-signature wallet contract itself, instead of using DELEGATE-CALLs to an external contract. In a sense, this would be similar to the standard static linking of libraries. However, statically linked code increases the gas cost of contract deployment (space also has a cost).

2) Parity library contract: Parity made the choice to adopt library-driven smart contract development for their multi-signature wallets. That is, Parity initially deployed a multi-signature contract as their library, and all the other Parity multi-signature wallets referenced that single library contract for all their functionality. The library itself was actually a properly working multi-signature wallet. In hindsight, it probably shouldn’t have been.

All the Parity multi-signature wallets (except for the library one) reference the library by declaring the following constant[1]:

```
address constant _walletLibrary =
0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4
```

Since it is a constant, it is generated at compile time, meaning it’s permanently stored in ”code”, not in ”storage”. The value would be the address of the library to DELEGATE-CALL on. By running

```
eth.getCode(walletAddress)
```

on one of the affected wallets (`walletAddress`), it is still possible to see the address of the now-dead library at the line code of index 422.

Another observation is that it would probably be better practice to allow the owners of the wallets to change the linked library, instead of coding it in the bytecode.

[1]https://medium.com/crypt-bytes-tech/parity-wallet-security-alert-vulnerability-in-the-parity-wallet-service-contract-1506486c4160

V. Analysis of the Attack

In this section we report a summary of the description of the attack presented on “ethereum.stackexchange.com” at the link https://ethereum.stackexchange.com/questions/30128/explanation-of-parity-library-suicide.

Remarkably, the attack that we are discussing was announced by a post of the supposedly unaware author: *”I accidentally killed it.”*[2] The author took control of a library contract, killed it, obliterating functionality for ∼500 multi-signature wallets and effectively, irreversibly freezing ∼$150M. A hard fork would be required to restore the contract and/or return funds.

It is important to highlight that the library we are considering was a working wallet. However, it had *not been initialized* since it was a *library* contract and the variable `only_uninitialized` had not been set. The attack could have been avoided if, after Parity deployed the library contract, it would have called `initWallet-once()` to claim the contract and set the uninitialized variables, including `owner`.

Anyone could then call the function *initWallet* on the library contract. As the ”hacker” did. Such a call, amongst other things, sets the caller, i.e. the hacker, as the owner of the contract being initialised. It is worth remarking that such a call is perfectly legal and it just initializes a wallet which has not been yet initialized. At this point, the owner of the library contract (e.g., an hacker), can call any privileged function, amongst which `kill()`. The kill function calls `suicide()`, which is now being replaced by `self-destruct`. The suicide function sends the remaining funds to the owner, destroying the contract and clearing its storage and code. Figure 3 shows the diagram of the functions and their dependencies for the Parity smart contract library defined at the address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`

Every call to the library will now return false and the multi-signature wallet contracts relying on the library contract code would get zero (with DELEGATE-CALL). The contracts still hold funds, but all the library code is set to zero. The multi-signature wallets are locked and the majority of the functionalities depend on the library which returns zero for every function call.

Indeed, after having killed the library contract, any other contracts depending on the killed library queried with

```
isowner(any_addr)
```

return `TRUE`, as a consequence of the delegate call made to a dead contract (the hacker tried this, to allegedly test the exploit).

The Ethereum Transaction that tracks the kill call is

```
0x47f7cff7a5e671884629c93b368cb18f58a
993f4b19c2a53a8662e3f1482f690
```

Wallets deployed before July 19 used a different library contract with a similar initWallet bug, but the library contract

[2]https://github.com/paritytech/parity/issues/6995#issuecomment-342409816

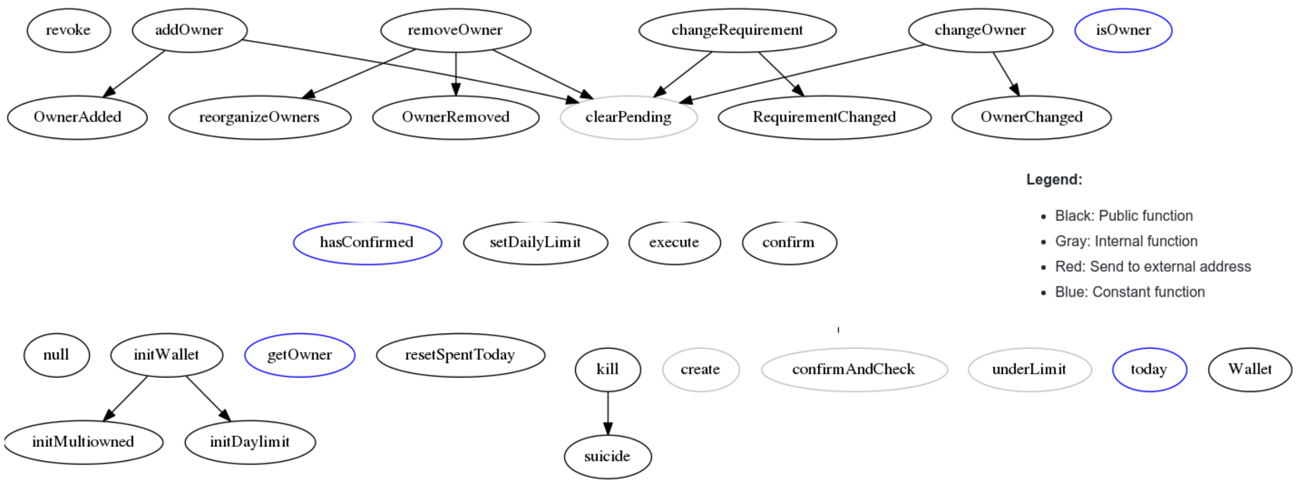


Fig. 3. Parity Wallet Dependency Graph

could not be taken over in a similar fashion as it had already been initialized by the developers at Parity.

The choice of defining the Wallet library as a contract instead of as a library, with the actual wallets making simple DELEGATE-CALLs to this linked smart contract, also needs to be confronted with the recommended practice of clearly defining libraries as such. Such a choice, makes the library contract behave more like a Singleton than a proper Library.

The only way to restore wallets' functionalities would be a *hard fork* in order to re-enable the library code. It is worth recalling that a hard fork would require an agreement by the majority of the miner community and their coordinated effort to develop an alternative branch of the blockchain, where the hack has basically never occurred. Although there have been cases in which this has been done, e.g. for the DAO attack [7], such a recovery strategy is an extraordinary event of extremely difficult realisation, which bears disastrous consequences - e.g. the cancellation of "happened" independent transactions - and that cannot currently be considered a routine error-correction practice.

VI. Best Practices That Could Have Helped

Smart Contracts security is an open research field [5]. The development of bug-free source code is still an utopia for traditional software development, after decades of analysis and development of engineering approaches. Error freedom is even more daunting for blockchain software development, which started less then a decade ago. In this section we discuss approaches that could have helped in mitigating the effects of the attack, drawing from accepted best practices in traditional software engineering. It is worth remarking here that vulnerabilities like the one leading to the Parity attack had been highlighted in literature, e.g. [8], a fact that strengthens even more our call for the adoption of standard and best practices in Blockchain Software Engineering.

A. Anti-patterns

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.[3] We have identified three anti-patterns in our case study that are responsible for the issue under analysis.

- The creation of a SM, that serves as full-fledged library, which is then left uninitialized.
- The creation of SMs that depend on external SM used as a library, and the address of such external library is hard-coded in the SMs and cannot be updated.
- The used SM library includes the definition of a public function that might call destroying functionality, such as *suicide*.

On the contrary, a pattern to be used is

- To allow SCs to re-address other SCs code whenever these are used as a library.

Such a strategy would also enable SCs to reference new libraries that have been deployed in more recent blocks. Furthermore, the strategy can be exploited for debugging purposes, refactoring, introduction of new features, and, in general, for purposes similar to versioning in traditional software engineering.

For example, faults could be so corrected in SC code and the corrected version of the same SC can be successively re-deployed and accepted by miners. Once provided with the address of the debugged contract, the very same contracts that were calling the faulty version can call the debugged contract. A similar scenario can be used for any issue resolution as in traditional versioning systems.

Such a solution applies to the present case study, in the hypothesis that the Wallet library address would have been saved into the private storage of each Parity wallet, and

[3]Definition from https://en.wikipedia.org/wiki/Anti-pattern. The term *anti-pattern* was coined in 1995 by Andrew Koenig.

managed through setter and getter methods, instead of being hard coded.

B. Testing

Testing smart contract is challenging and critical, because once deployed on the blockchain they become immutable, not allowing for further testing or upgrading. At present, to the best of our knowledge, there is not a testing framework for Solidity, like e.g. JUnit for the Java language, meaning that every smart contract has to be tested manually.

The nature of smart contracts introduces at least two complications to testing: an application may be critical and it is very difficult to update an application once deployed. As a result, it is desirable to use robust testing techniques. Manual test generation is likely to form an important component but inevitably is limited; there is a need for effective automated test generation (and execution) techniques.

Currently available options to test contracts are:

- Deploy the contract to live (the real) Ethereum main network and execute it. This costs about 5 minutes and real money to deploy and execute (slow, public(dangerous), $$$).
- Deploy the contract to the test-net Ethereum network (for developers usage) and execute it. This costs about 2 minutes to deploy and free ether (slow, public(dangerous), free(no real money))
- Deploy the contract to an Ethereum network (local) simulator and execute it. This costs about 3 seconds and is free (fast, private(nice!!), free), but limited interaction and no realistic test of network related issues.

There are many automated test generation approaches that might have value. A number of these use formal approaches, such as those based on symbolic execution (e.g. [9]), or search-based methods (e.g. [10]) to produce test cases that provide code coverage. It is unclear whether such techniques would have found the attack on Parity since the attack involved a sequence of operations and code coverage techniques typically aim to cover smaller structures, such as branches in the code[4]. Code coverage approaches may still have value but it appears that they are not sufficient on their own.

An alternative, and complementary, approach is to base test automation on a model; an approach that is typically called model-based testing (MBT) (see [11] for a survey). Some MBT techniques aim to cover the model but there are also more rigorous approaches that generate test cases that are guaranteed to find certain classes of faults (defined by a fault model) or that are guaranteed to find all faults if certain well-defined conditions (test hypotheses) hold [12]. Such approaches provide a trade-off: as one weakens the assumptions or widens the class of faults, the cost of testing increases. There is also the scope to utilize formal verification techniques in order to reason about the underlying assumptions made by these techniques or, indeed, to find faults.

[4]It is possible to require, for example, the coverage of paths but such approaches tend not to scale.

We have seen that smart contracts are state-based and so it would be natural to use state-based models in MBT, allowing the use of a wide range of test automation techniques. There appears to be potential for MBT approaches to find faults such as the one that resulted in the Parity attack. First, the process of producing a model might have led the developers to consider what happens if a user calls a library function without it being initialized. Second, the attack involved a particular (short) sequence of operations and state-based MBT techniques focus on the generation of such sequences. Naturally, the effectiveness of MBT depends on the model and also the fault model (or test hypotheses) used. An interesting challenge is to explore smart contracts and their faults in order to derive appropriate fault models or test hypotheses.

VII. Road-map to BOSE

The Parity wallet case study clearly showed that a Blockchain-Oriented Software Engineering (BOSE) [13], [14], [15], [16], [17] is needed to define new directions to allow effective software development. New professional roles, enhanced security and reliability, modeling and verification frameworks, and specialized metrics are needed in order to drive blockchain applications to the next reliable level. At least three main areas to start addressing have been highlighted by our analysis of a specific case of study:

- Best practices and development methodology
- Design patterns
- Testing

The aim of BOSE is to create a bridge between traditional software engineering and blockchain software development, defining new ad-hoc methodologies, fault analysis [18], patterns [19], [20], quality metrics, security strategies and testing approaches [21] capable of supporting a novel and disciplined area of software engineering.

VIII. Conclusions

In this paper, we presented a study case regarding the Parity smart contract library. The problem resulted from poor programming practices that led to the situation in which an anonymous user was able to accidentally (it is not clear if he did it on purpose) freeze about 500K Ether (150M USD on November 2017).

We investigated the case, analyzing the chronology of the events and the source code of the smart contract library. We found that the vulnerability of the library was mainly due to a negligent programming activity rather than a problem in the Solidity language.

The vulnerability was exploited by the anonymous user in two steps. First the attacker was able to become the owner of the smart contract library (because it was created and left uninitialized), then the attacker did nothing more than calling the initialization function. After that the suicide function was called, which killed the library, leading to the situation in which it was not possible to execute functionality on the smart contracts created with the library, because all the delegate calls ended up in the dead smart contract library. This case clearly demonstrated a need for Blockchain Oriented Software Engineering in order to prevent, or mitigate such scenarios.

The aim for BOSE is to pave the way for a disciplined, testable and verifiable smart contract software development.

REFERENCES

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

[2] "Ethereum foundation. the solidity contract-oriented language." https://github.com/ethereum/solidity., 2014.

[3] "A postmortem on the parity multi-sig library self-destruct," https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, 2017.

[4] "Ethereum foundation. ethereum original white paper." https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[6] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 706–719.

[7] D. Siegel, "Understanding the dao attack," http://www.coindesk.com/understanding-dao-hack-journalists, 2016.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[10] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 226–247, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.71

[11] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, 2009. [Online]. Available: http://doi.acm.org/10.1145/1459352.1459354

[12] M. Gaudel, "Testing can be formal, too," in *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, ser. Lecture Notes in Computer Science, vol. 915. Springer, 1995, pp. 82–96.

[13] A. Pinna, R. Tonelli, M. Orrú, and M. Marchesi, "A petri nets model for blockchain analysis," *arXiv preprint arXiv:1709.07790*, 2017.

[14] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: challenges and new directions," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 169–171.

[15] H. Rocha, S. Ducasse, M. Denker, and J. Lecerf, "Solidity parsing using smacc: Challenges and irregularities," in *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, ser. IWST '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:9. [Online]. Available: http://rmod.inria.fr/archives/workshops/Roch17a-IWST-SolidityParser.pdf

[16] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: Smart contract inspection technical report," Inria Lille-Nord Europe, Technical Report, Dec. 2017. [Online]. Available: http://rmod.inria.fr/archives/reports/Roch17b-TR-SmartInspect.pdf

[17] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu, "Smart contracts software metrics: a first study," *arXiv preprint arXiv:1802.01517*, 2018.

[18] M. Ortu, G. Destefanis, S. Swift, and M. Marchesi, "Measuring high and low priority defects on traditional and mobile open source software," in *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*. ACM, 2016, pp. 1–7.

[19] G. Destefanis, R. Tonelli, G. Concas, and M. Marchesi, "An analysis of anti-micro-patterns effects on fault-proneness in large java systems," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1251–1253.

[20] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, "Micro pattern fault-proneness," in *Software engineering and advanced applications (SEAA), 2012 38th EUROMICRO conference on*. IEEE, 2012, pp. 302–306.

[21] S. Counsell, G. Destefanis, X. Liu, S. Eldh, A. Ermedahl, and K. Andersson, "Comparing test and production code quality in a large commercial multicore system," in *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*. IEEE, 2016, pp. 86–91.

The ICO Phenomenon and Its Relationships with Ethereum Smart Contract Environment

Gianni Fenu, Lodovica Marchesi, Michele Marchesi and Roberto Tonelli
Dept. of Mathematics and Computer Science
University of Cagliari
Cagliari, Italy
fenu@unica.it, lodo.marchesi@gmail.com, marchesi@unica.it, roberto.tonelli@dsf.unica.it

Abstract—**Initial Coin Offerings (ICO) are public offers of new cryptocurrencies in exchange of existing ones, aimed to finance projects in the blockchain development arena. In the last 8 months of 2017, the total amount gathered by ICOs exceeded 4 billion US$, and overcame the venture capital funnelled toward high tech initiatives in the same period. A high percentage of ICOs is managed through Smart Contracts running on Ethereum blockchain, and in particular to ERC-20 Token Standard Contract. In this work we examine 1387 ICOs, published on December 31, 2017 on icobench.com website, gathering information relevant to the assessment of their quality and software development management, including data on their development teams. We also study, at the same date, the financial data of 450 ICO tokens available on coinmarketcap.com website, among which 355 tokens are managed on Ethereum blochain. We define success criteria for the ICOs, based on the funds actually gathered, and on the behavior of the price of the related tokens, finding the factors that most likely influence the ICO success likeliness.**

Index Terms—ICO; Initial Coin Offering; cryptocurrencies; Ethereum; Smart Contracts.

I. INTRODUCTION

Recently, the cryptocurrencies phenomenon has become widespread, in terms of adoption, number of available currencies and market capitalization. Made possible by blockchain technology, which ensures trust, security, pseudo-anonimity and immutability through strong cryptography and a decentralized, peer-to-peer approach, a cryptocurrency can be easily dispatched from the initial owner to another person, in whatever part of the world, in matter of minutes and with no intermediary, whose behavior can also be modeled using a Petri Net approach [1]. These features make cryptocurrencies ideal also for crowfunding purposes, leading to the so called ICO phenomenon.

Initial Coin Offerings (ICO) are public offers of new cryptocurrencies in exchange of existing ones, aimed to finance projects, mostly in the blockchain development arena. Despite being totally unregulated, and even banned in several countries, the easiness of sending funds through blockchain transactions, and the hope to get very high returns even before the business initiative reaches the market – because ICO tokens are traded immediately on cryptocurrency exchanges – made the ICO phenomenon explode. In the last 8 months of 2017, the total amount raised by ICOs exceeded 4 billion US$, and overcame the venture capital funneled toward high tech initiatives in the same period [2]. ICOs are usually characterized by the following features: a business idea, typically explained in a white paper, a proposer team, a target sum to be collected, a given number of “tokens”, that is a new cryptocurrency, to be given to subscribers according to a predetermined exchange rate with one or more existing cryptocurrencies.

Nowadays, a high percentage of ICOs is managed through Smart Contracts running on Ethereum blockchain, and in particular through ERC-20 Token Standard Contract. Cloning an ERC-20 contract, it is very easy to create a new token, issue a given number of tokens, and trade these tokens with Ethers – the Ethereum cryptocurrency, which has a monetary value – according to a given exchange rate. The contract stores the addresses of the token owners, together with the amount of owned tokens, and allows transfers only if the sender shows the ownership of the private key associated to the address.

Since the ICO phenomenon had a boom starting from May-June 2017, only a few research reports, and almost no paper published on scientific journals, has appeared on the subject so far. We can just quote the working papers by Zetzsche et al. [2], and by Adhami et al. [3], that report analyses of ICO features. The former paper is focused on legal and financial risk aspects of ICOs, but its second section contains a taxonomy, and some data about ICOs that the authors claim are continuously updated. In the latter paper 253 ICOs are analyzed, starting from 2014 to August 2017, and the significance of some factors that influence the success of an ICO is studied. Recently, Subramanian [4] quoted the ICOs as an example of decentralized blockchain-based electronic marketplace. The main source of information about blockchains, tokens and ICOs is obviously the Web. Here we can find websites enabling to explore the various blockchains associated to the main cryptocurrencies, including Ethereum's one. We can also find websites giving extensive financial information on prices of all the main cryptocurrencies and tokens, and sites specialized in listing the existing ICOs and giving information about them. Often, these sites also evaluate the soundness and likeliness of success of the listed ICOs. X.Wang, F. Hartmann and I. Lunesu in their work “Evaluation of Initial Cryptoasset Offerings: the State of the Practice” [5] studied the emergent websites that provide evaluations of ICOs and identified 28 different websites as proper ICOs evaluation websites, out of 169 URLs found with a research over specific keywords using a websearch engine. Analysing the 28 results, using Google Trends we found the five most popular sites on ICOs: icobench, icorating, icoalert, icotracker and icodrops. We decided to use icobench.com mainly because it manages a large set of ICOs, indeed the biggest among these five and moreover it provides a set of API to automatically gather information on ICOs.

In this work we examine 1387 ICOs, published on December 31, 2017 on icobench.com website, gathering information relevant to the assessment of their quality and software development management [6], including data on their development teams. We also studied, at the same date, the financial data of 450 ICO tokens available on coinmarketcap.com website, among which 355 tokens are managed on Ethereum blockhain. We defined success criteria for the ICOs, based on the funds actually gathered, and on the behavior of the price of the related tokens, and studied the factors that most likely influence the ICO success likelihood.

We analyzed some key features of the ICOs, like their country, the kind of business they address, the team size, the ratings obtained by icobench.com site. We found that more than 1000 ICOs are managed on the Ethereum blockchain, mainly following ERC-20 standard. This causes a considerable stress on Ethereum blockchain, confirmed by the analysis of token transactions using the data gathered from ethplorer.io site. The total number of transfer transaction is above 16 million, and the total number of token holders is about 5.5 million. After performing a multivariate analysis of the factors influencing the success, we also found that the ratings of icobench.com site have a high probability to predict the success, as well as some of the countries of origin and the platform.

In the followings, Section II presents the methods used to gather the data, to evaluate ICO success and to analyze the data. Section III presents and discusses the results. Section IV concludes the paper.

II. Method

To perform a massive study of the ICOs characteristics and success factors, we need to gather ICO data from the Web, to establish what data are to be analyzed and how ICO success can be defined, and to analyze the data to draw facts about the factors that determine success. The following subsections give insight on these steps.

A. *Retrieving ICO Data from the Internet*

The main sources of data we used are of three kinds:

- data about the ICOs themselves, collected by icobench.com site;
- financial data about the ICO tokens traded on main cryptocurrency exchanges, collected by coinmarketcap.com website; these data include the address of the token contract on Ethereum blockchain;
- data about token transactions and holders directly collected from Ethereum blockchain using a blockchain explorer (ethplorer.io).

ICO data were massively collected from icobench.com, which kindly granted us permission to access their API calls. icobench.com is one of the main sites giving information about ICOs. As its name suggests, icobench.com also performs an analysis of each submitted ICO. As specified in their website, it follows a special rating methodology, based on a combination of: ICO profile rating given by its “Benchy” ICO analyzer robot and possibly ratings on the team, the vision and the product provided by a pool of indipendent experts. Benchy’s rating is based on an algorithm that uses more than 20 different criteria, described in icobench.com website.

Each ICO shown in the site is provided of a unique integer progressive identifier. We performed an API query for all of these numbers, gathering the whole icobench.com database, in json format. The ICO data include name, token symbol, description, rating, country, start and end dates of the crowfunding, financial data such as the total number of issued tokens and the percentage that is sold in the offer, initial price of the token, platform used, hard and soft cap (maximum and minimum number of tokens to sell), raised money (in US$) if the ICO has finished, data on the team proposing the ICO, main milestones and category. Some of these data, such as short and long description, and milestones are textual descriptions. Others are categorical variables, such as the country, the platform, the category (which can assume many values), and variables related to the team members (name, role, group). The remaining variables are numeric, with different degrees of discretization. Unfortunately, not all ICOs record all variables, so there are several missing data.

Financial data were collected from coinmarketcap.com website, which is one of the most popular sites giving almost real-time data on the quotation of the various cryptocurrencies in the world exchanges – an exchange is a website where it is possible to buy and sell cryptocurrencies against each others, and against standard currencies. It also has a specific “token” section giving information about the tokens (usually ICO tokens). This information included the address of the token contract in the related blockchain (usually Ethereum).

We gathered the needed information from this website using the Python scraping library “Beautiful Soup” [7]. For each listed token we recovered name, symbol, number of tokens, capitalization, Ethereum address (if it is a token managed on Ethereum blockchain), price series (daily closing price in US$, volume and market cap) in a given time interval.

Using the Ethereum address, when present, we query ethplorer.io publicly available APIs, gathering information about the total token supply, the number of token transfers, the number of token holders. Using this website, it is also possible to obtain information on each transaction, and each holder, but this is beyond the scope of this paper.

All the recovered data were stored in a database, linking data coming from different sources (icobench.com and coinmarketcap.com) through the name and symbol of the tokens. In some cases, name and/or symbol differ, so it was needed a more sophisticated matching procedure, using the Levenshtein distance [8] between names in the case the symbol is the same. The Levenshtein distance is a measure of similarity between string, also called “edit distance”. It can be rephrased as “the minimal number of insertions, deletions and substitutions to make two strings equal”. After a series of empirical tests, we chose as threshold value 0.6. Therefore if this distance is below 0.6, we assume a match. Anyway, we checked by hand these matches, and also other possible matches (same name, different symbol; same symbol, names with Levenshtein distance greater than 0.6).

B. *Defining ICO Success*

Since half of the year 2017, the number of ICOs launched on the market skyrocketed. However, only a fraction of them was able to gather an amount of money according to the needs and hopes of their proposers. Moreover, all successful ICOs after the end of the sale are quoted on some exchange, where they are traded against other currencies, usually Bitcoin. Quite often, in the past, the actual price of the tokens increases a lot after their quotation on the exchange, and this is one of the main attraction factors of ICOs – the fact that the token can be sold with a profit very soon, long before the realization of the business initiative behind the ICO.

In our analysis, we used a dichotomous variable to describe the ICO's success: successful or failed. This is the same approach of the paper by Adhami et al. [3]. They define an ICO as "successful" if it reaches at least the soft cap declared by its proposers. We decided to extend this definition because on one hand our data may lack the value of an ICO's soft cap, and on the other hand several ICOs include provisions allowing to go ahead with the ICO even in the case the soft cap is not reached, and this happens in many cases. Several ICOs are not eligible to be considered, typically because they lack data, or are still in progress. To assess whether an ICO is successful, the criteria we use are the following:

1. we regard as failed an ICO that raised less than 80.000 US$; we regard as undecided – and did not consider in the analysis – an ICO that raised between 80.000 and 200.000 US$; ICOs raising more than 200.000 US$ are considered successful, except in the case they fall in criterion 3. The selection of the thresholds is arbitrary; it was done not to create a clean break between successes and failures. In any case, the ICOs within this range are only 11. In future work the results might be parameterized according to the thresholds;

2. we do not consider ICOs ending in 2018, except the few ones that raised money in 2017 and were stopped; we do not consider ICOs that raised no money and that have no end date;

3. for ICOs with a token provided of a price series long enough in 2017 (at least prices in the whole month of December 2017), we considered as failed the ICOs with a market cap diminished by more than 75% since the beginning of their quotation; the market caps are computed as a moving average of 20 consecutive days, to filter out daily variations.

C. *Analysis of the Factors Influencing the Success*

Among the data associated to an ICO, we chose some factors that could possibly influence its success. These factors are the ratings obtained by icobench.com website, the country of origin, the team's size, the opening and closing date, number of tokens sold, the platform, the category and others.

To analyze our dataset we resort to multivariate statistical analysis for dicotomic dependent variables. In fact our target is to measure if and to which extent the collected variables contribute to the success or failure of an ICO project. Given the dicotomic nature of the target variable, success or not, simple regression analysis and predictive models cannot be applied directly.

We set to one the dependent variable in cases where the ICO has been successful and to zero otherwise. Success or failure can be so tested against the set of independent variables which is the set of variables we collected from icobench.com and the contribution of each variable to success or failure can be evaluated and compared with other variables.

The best suited model is the Logit model, where the logarithm of the odd ratio among success and failure is modeled through a multivariate linear analysis as a linear combination of the independent variables of interest. The model outputs the best fitting coefficients as well as the statistical significance of each variable with respect to ICO's success or failure. Other models could be used (eg. SVM or Naive Bayes) for classifying the ICOs but our purpose is, once it is known that an ICO is successful or unsuccessful according to the above described criteria, to determine which factors contribute or not, and to which extent, to success or failure. So we decided to use the Logit model.

In order to simplify our analysis we filtered the raw data for some variables and concentrated the analysis on part of them. Specifically, for the multivariate analysis we didn't consider the raised founds, which has been already chose as the discriminant variable for the success, the token which is simply a label, the type, whose values are mostly missing or mostly equals, and all other variables with many missing values.

According to the Logit model we define:

$$\ln \frac{P}{P-1} = a + \sum_{i=1}^{N} \beta_i x_i \qquad (1)$$

the Logit model where P represents the success probability, 1-P the failure probability, P/(1-P) the *odd ratio*, and the sum is a linear combination of all independent variables in the vector **x** with coefficients in the vector **beta**. We implemented the computation using the R packages 'lme4' and 'rms', using the general regression model provided by the 'lrm' function.

We targeted the level of significance of independent variables with respect to influencing ICOs success or failure and how much the single coefficients variation can affect the odd ratio against failure.

III. Results and Discussion

We gathered all ICOs listed on icobench.com website on 31/12/2017. Overall, they are 1387. We also gathered information on 450 tokens listed on coinmarketcap.com, and on 355 tokens managed on the Ethereum blockchain and whose data are reported on ethplorer.io site. We found that the success rate on the studied ICOs is about 35%. On these data we performed automated analyses to detect which factors influence an ICO's success, reported in the followings.

A. Descriptive Statistics of the ICO Data

Note that the order of coloring of the figures in every pie chart of this work is the same of the order of the related captions. We report in Fig. 1 the countries of origin of the ICOs. As you can see, USA and Russia Federation are the most active countries in proposing ICOs. The country of 139 ICOs is not declared in icobench.com; the countries with less than 4 ICOs are cumulated under the "Others" tag. We note that some relatively small countries, like Singapore, Switzerland, Estonia and Slovenia are very active in proposing ICOs.

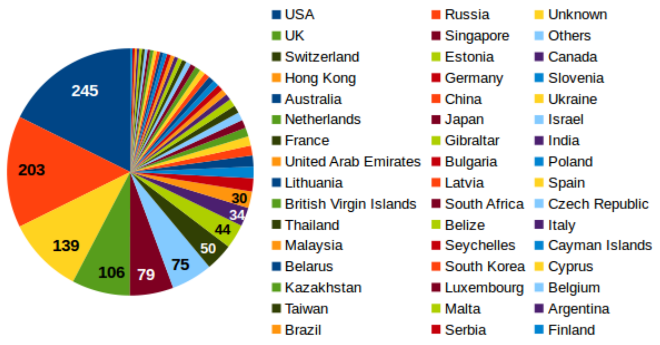


Fig. 1: The countries of origin of the ICOs.

Fig. 2 shows the main business category of the ICOs analyzed. Note that icobench.com allows to assign more than one category to an ICO. Here we report just the first category, which we assume is the most expressive of the ICO business target. Most ICOs declares themselves as "platforms" to perform decentralized business. 233 ICOs are new kinds of cryptocurrencies, whereas the remaining categories cover almost all business sectors.

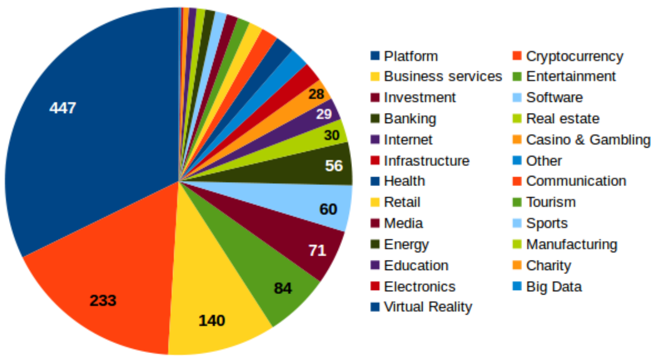


Fig. 2: The main categories of the ICOs.

The distribution of the overall ratings given to the various ICOs is reported in Fig. 3. All considered ICOs have a rating, in most cases given by the robot of icobench.com site. The ratings span between 0 and 5. In the figure we report the centered moving average of 3 rating values, to filter out the noise. As you can see, the distribution is quite regular, with a steady climbing of the rating from the value of about 1.2, until the peak at the value of 3.9; then a steep descent follows, with very few ratings equal or above to 4.5.

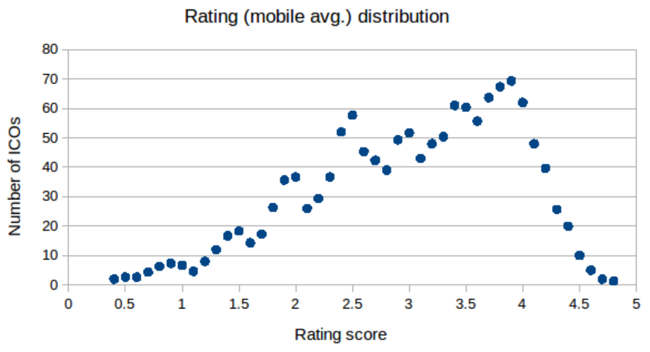


Fig. 3: The distribution of ICO ratings (mobile average of 3 rating scores).

In Fig. 4 we report the team size distribution, as declared by ICOs proposers. Note that, in this analysis, we consider the overall team, including business people and advisors, and not only the software development team. We have what looks like an unimodal distribution, with a peak around 7-8 people. Note that in some cases the ICO team is composed just by the business and marketing people who developed the business idea – the developers will be hired only in the case of ICO success. When the software developers are part of the team, they typically account for a percentage between 20% and 50%. When a team is very large, this means that it include many advisors, who contribute suggestions but are not really involved in the ICO operations.

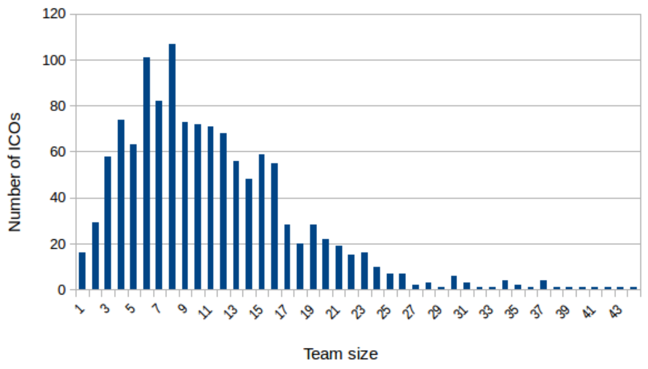


Fig. 4: The team sizes of the studied ICOs.

Fig. 5 shows the platforms used to deliver and manage the token or coin offer. As you can see, Ethereum is by far the most used platform. 193 ICOs do not declare their platform, and Waves is the second most popular platform, chosen by 67 ICOs. There are many other platforms or approaches that were used to deliver the tokens, but overall they cover only 54 cases. We also analyzed the smart contract standard used to manage the tokens. In 787 cases it is ERC-20 on Ethereum, in 581 cases it is not explicitly declared, and in 12 cases it is the new standard ERC223, which is an evolution of ERC-20. One ICO mentions the NEP5 standard, which is the equivalent of ERC-20 for NEO blockchain. The Ethereum standard ERC-20 for token management was developed in 2015. It defines a set of rules that a contract carried out with an Ethereum token has to implement [9]. The standard ensures the interoperability of the assets, making them more useful. Various implementations of ERC-20 written in Solidity language are freely available. In

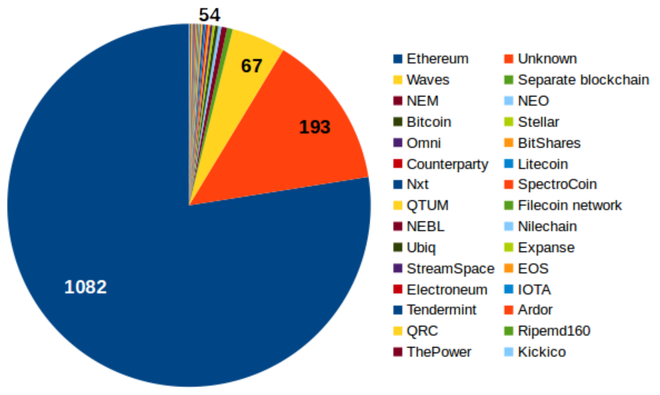


Fig. 5: The platforms used to manage the ICO token offer.

deeper detail, ERC-20 standard includes the following functions:

- totalSupply(): return the total number of tokens.
- balanceOf(address owner): return the number of tokens available to the given owner.
- allowance(address owner, address spender): return the amount of tokens approved by the owner that can be transferred to the spender's account.
- transfer(address to, uint tokens): transfer the balance from token owner's account to 'to' account.
- approve(address spender, uint tokens): token owner approves for 'spender' to call transferFrom() function to transfer 'tokens' from the token owner's account.
- transferFrom(address from, address to, uint tokens): transfer 'tokens' from the 'from' account to the 'to' account.

From a software engineering perspective, it is worth noting that the number of ICOs relying on Ethereum blockchain for the delivery and management of their tokens is impressive; this denotes the great capacity of the Ethereum system to withstand huge workloads. On December 31 2017, this number was equal to 1082, steadily growing by the day. The overall value of all tokens overcomes 30-40 billion US$ at the present evaluation! Despite this load, Ethereum public blockchain looks

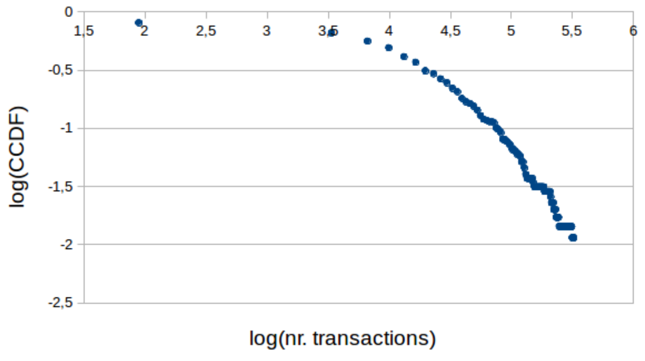


Fig. 6: CCDF of the transfers count of the considered ERC-20 tokens on the Ethereum blockchain.

performing quite well. Only in mid 2017, Bancor and Status ICOs clogged Ethereum network, making it unusable for some hours.

TABLE I. STATISTICS OF NUMBER OF TRANSFERS AND HOLDERS OF 355 ERC-20 ETHEREUM TOKENS.

Data	***mean***	***median***	***st. dev.***	***min***	***max***
# of transfers	46076	13186	132586	89	1311959
# of holders	15515	2872	76938	19	959205

We analyzed the number of transfer transactions and of the token holders for all 355 tokens managed on Ethereum using ERC-20 standard, that also enables websites like ethplorer.io to easily gather and show relevant data. Table I shows the main statistics. As you can see, both data series have mean much higher than median, a high standard deviation and very large maximum values. This is a typical behavior of fat-tailed distributions. Consequently, we analyzed the distributions of these data, which are shown in Figs. 6 and 7 in the form of complementary cumulative distribution function (CCDF), in log-log format. Both distributions tend to follow a straight line in the right of the plot, which is the typical characteristic of power-law distributions.

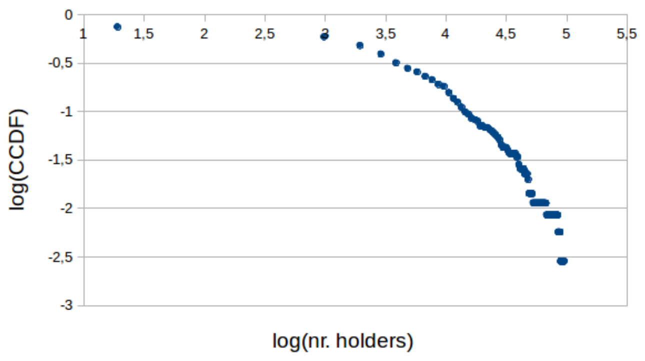


Fig. 7: CCDF of the holders count of the considered ERC-20 tokens.

B. *Multivariate Analysis of the Factors Influencing the Success*

We eliminated the ICOs still in progress, whose end date was in 2018, except for 5 ICOs that raised a significant amount of money in 2017, and were closed in advance. The ICOs ended within 2017 are 971. We also excluded the ICOs with no raised money according to icobench.com, and with no end date. We assumed they are ICOs still in progress, registered on the site, but whose end date is still to be determined – or even abandoned ICOs. The considered ICOs were thus reduced to 712. Among these ICO's tokens, only 215 are quoted on exchanges and their financial data are reported on coinmarketcap.com site. We were able to assess the third success criterion only for these tokens.

In order to perform the multivariate analysis we started including all the variables but those already excluded according to the described methodology. Those included in the full model are: rating, rtTeam, rtVision, rtProduct, rtProfile, country, platform, team size and finally category.

We briefly report the most significant values in Table II.

TABLE II. MOST INFLUENCING COEFFICIENTS IN THE LOGIT MODEL.

Independent Variable	Coeff.	S.E.	Wald Z	Pr(>|Z|)
rating	1.6990	0.442	3.84	0.0001
platform = Ethereum	-0.9867	0.2786	-3.54	0.0004
country = Slovenia	1.6493	0.6830	2.41	0.0158
category = Software	1.1393	0.5329	2.14	0.0325
country = USA	0.7895	0.3816	2.07	0.0385
rtProfile	-0.7512	0.3858	-1.95	0.0515
rtVision	-0.3291	0.2021	-1.63	0.1033
country = Israel	1.2882	0.7912	1.63	0.1035
country = China	1.1639	0.7228	1.61	0.1073
category = CasinoGambling	1.0980	0.6861	1.60	0.1095
rtProduct	0.3379	0.2153	1.57	0.1167
category = Businessservices	-0.7462	0.5015	-1.49	0.1367
team.size	-0.0225	0.0161	-1.40	0.1614
country = Singapore	0.6228	0.4644	1.34	0.1799
country = UK	0.5578	0.4475	1.25	0.2126
platform = Waves	-0.5726	0.5039	-1.14	0.2558

Table II reports the Logit model coefficients, their standard errors, the Wald normalized Z value and the relative p-value for all the independent variables in the model, sorted according to an increasing p-value, up to the case of the ‘Wave’ platform, which we retained since it is the only one that can be compared with the most common platform ‘Ethereum’.

The results show that the most significant variable is the 'rating' as reported by icobench.com in a scale between 0 and 5. In particular the relative p-value in the full model is 0.0001 and the coefficient value and the Z-Wald value are 1.6991 and 3.84 respectively. This means that the model identifies the variable relevant for influencing ICO's success according to the described criteria and that, in particular, a unit increase of the 'rating' carries a factor of about five in favor of the odd ratio, meaning that the odds are shifted of a consistent amount for each unitary increase of the 'rating' provided by icobench.com. The other way round, icobench.com rating system is a reliable indicator of the possible success of the ICO and, consequently, of the quality of the ICO project.

The other interesting variable related to icobench.com is the rtProfile with a 95% significance level (p-value 0.0515) and coefficient and Z-Wald of -0.7512 and 0.3858. This means that a unit increase of this index, which is in the range 0 to 5, multiplies the odd ratio of a factor of about 0.4. This indication is in agreement with the previous one, since the rtProfile is automatically assigned by a robot on the basis of a combination of values of the other four icobench.com indexes and is mainly influenced by the rating value, and coincides with it in the cases where the other indicators are missing.

The rtVision as well has some incidence on the success of the ICO, having a p-value around 0.1 and contributing to changing the odd ratio of a factor 0.7 for each unit incremement.

Some interesting results concern the countries, the category and the platform. For the latter the topic case is the Ethereum platform. Data analysis shows that the Ethereum platform has high significance level (p-value 0.0004) and changes the odd ratio of a factor of about 0.4, on average, with respect to the other platforms.

It has to be noted that when considering data on platforms 'per-se' there are many spurious data, namely those where a given platform appears only once or in a very few cases.

In these particular cases, even if they are not at all statistically significant, the odd ratio is exceptionally high or low, meaning that the variable automatically means success or failure, given that they appear only twice or three times with always success or failure.

Not considering the spurious cases of platforms appearing one or very few times the only comparison can be made with Waves, another quite common platform, which, on the contrary, does not appear to provide a significant contribution to the ICO's success.

For what concern the countries the best ones where to start an ICO are Slovenia and USA. Good places are also, but to a less extent, Israel and China. In particular, Slovenia and USA have a good statistical significance, of about 0.016 and 0.038 respectively, and they contribute to the odd ratio in favor of success of about 5.2 and 2.2 respectively. The other countries have less statistical significance in determining the ICO’s success.

Finally the category which positively contributes to the ICO’s success is ‘software’, with a p-value of 0.0325 and a relative contribution to the odd ratio of 3.1. Other categories which in principle could be interesting are ‘gambling’ and ‘business’, but with a much lower statistical significance.

The analysis also shows that ‘team size’ does not seem to count for determining success or failure of the ICO project. Since we performed the analysis with the basic model using all available numerical data, we also checked the possible contribution of ‘team size’ to success or failure gathering into different categories the team’s size, making different choices for the categories (small, medium, large team’s size or even 5 different categories), but also this analysis confirms that the variable does not count for ICO’s success or failure.

IV. CONCLUSIONS

In this work we examined 1387 ICOs, from icobench.com website, also gathering information from other source. Financial information about the prices of the ICO tokens was obtained from coinmarketcap.com site, and transaction information coming from Ethereum blockchain was obtained from ethplorer.io site. An intial analysis gave insights on some key features of the ICOs, showing the countries they come from, the addressed business field, the team size and the software platforms used to manage the ICO tokens. Unsurprisingly, we found that most ICOs are managed on Ethereum blockchain, using ERC-20 standard. To this purpose, we also found that the distribution of token transfer transactions and token holders follow a fat-tailed distribution, resembling a power-law in the tail. This kind of distribution is very common in technological and financial data. Its meaning is that, though there are many tokens managed on the blockchain, only a few of them account for most of the workload applied to the blockchain.

Subsequently, we performed a multivariate analysis to assess the factors that can influence the success of an ICO. To this purpose, we divided the considered ICOs in two categories – successful and failed. The analysis showed that there are some factors that are correlated to an ICO success. They are the country of origin – it looks that ICOs coming from Slovenia and USA, and, to a less extent, Israel and China, are more prone to have success. Most other countries do not bear significance. The team size does not seem to be relevant to the success. A high overall rating on icobench.com site, on the other hand, looks quite correlated to the success of an ICO, though this looks mainly due to the robot's advice rather than to the human experts' advices. Finally, managing the ICO token on Ethereum blockchain looks another success factor.

Future work will regard gathering ICO data also from other sources, to double check their validity, and to perform a deeper analysis of token transactions on Ethereum blockchain, also to relate blockchain activity to price and volume information of the token.

Acknowledgment

We thank icobench.com administrators who granted us the permission to fully access their API interface.

References

[1] A. Pinna, R. Tonelli, M. Orrú, M. Marchesi, “A Petri Nets Model for Blockchain Analysis”, The Computer Journal, January 25 2018, available at: https://doi.org/10.1093/comjnl/bxy001

[2] 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”, Univ. Luxembourg Law Working Paper No. 11/2017, available at SSRN: https://ssrn.com/abstract=3072298, November 2017.

[3] S. Adhami, G. Giudici, S. Martinazzi, “Why Do Businesses Go Crypto? An Empirical Analysis of Initial Coin Offerings”, October 20 2017, available at SSRN: https://ssrn.com/abstract=3046209

[4] H. Subramanian, “Decentralized Blockchain-Based Electronic Marketplaces”, Comm. ACM, vol. 61, pp. 78-84, January 2018.

[5] X.Wang, F. Hartmann and I. Lunesu, “Evaluation of Initial Cryptoasset Offerings: the State of the Practice”, 1st Int. Workshop on Blockchain Oriented Software Engineering, IWBOSE 2018, Campobasso, Italy, March 20 2018.

[6] S. Porru, A. Pinna, M. Marchesi, R. Tonelli, ”Blockchain-oriented software engineering: Challenges and new directions”„ Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 201730 June 2017, Article number 7965292, Pages 169-171.

[7] V.G. Nair, "Getting Started with Beautiful Soup", Packt Publishing, Birmingham - Mumbay, 2014.

[8] G. Navarro, “A Guided Tour to Approximate String Matching”, ACM Comp. Surv., vol. 33, pp. 31-88, March 2001.

[9] F. Vogelsteller, V. Buterin, “ERC-20 Token Standard”, available online at: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md

Evaluation of Initial Cryptoasset Offerings: The State of the Practice

Felix Hartmann
Free University of Bozen-Bolzano, Italy
hartmannfelix00@gmail.com

Xiaofeng Wang
University of Bozen-Bolzano, Italy
xiaofeng.wang@unibz.it

Maria Ilaria Lunesu
University of Cagliari, Italy
ilaria.lunesu@diee.unica.it

***Abstract*—Initial Cryptoasset Offering (ICO), also often called Initial Coin Offering or Initial Token Offering (ITO) is a new means of fundraising through blockchain technology, which allows startups to raise large amounts of funds from the crowd in an unprecedented speed. However it is not easy for ordinary investors to distinguish genuine fundraising activities through ICOs from scams. Different websites that gather and evaluate ICOs at different stages have emerged as a solution to this issue. What remains unclear is how these websites are evaluating ICOs, and consequently how reliable and credible their evaluations are. In this paper we present the first findings of an analysis of a set of 28 ICO evaluation websites, aiming at revealing the state of the practice in terms of ICO evaluation. Key information about ICOs collected by these websites are categorised, and key factors that differentiate the evaluation mechanisms employed by these evaluation websites are identified. The findings of our study could help a better understanding of what entails to properly evaluate ICOs. It is also a first step towards discovering the key success factors of ICOs.**

***Index Terms*—Initial Cryptoasset Offerings, Initial Coin Offerings, ICO, ICO evaluation, Information Richness, Evaluation Transparency**

I. Introduction

Few phenomena in the last couple of years can be considered as disruptive and grab the attention of the global audience as cryptocurrency, blockchain technology behind, and a closely related fundraising mechanism - Initial cryptoasset offering (ICO), also often called Initial Coin Offering or Initial Token Offering (ITO). ICO is a major and disruptive trend in the financing of new cryptocurrency and blockchain startups, and is compared to Initial Public Offerings (IPOs) and Venture Capitals (VCs) as a state-of-the-art strategy to finance new ventures. The first ICO dates back to 2013, but ICOs have experienced an exponential growth in the last two years, outcompeting and outperforming VC in the financing of cryptocurrency and blockchain startups in the second quarter of 2017 for the first time.

ICO offers to the public a fraction of ownership in a new digital (mostly blockchain) project in the form of tokens/coins. Even though often described as a hybrid between a grant and an investment and sharing similar traits with IPOs, ICOs have distinct features. They consist in the sale of a stake in a project with the aim to raise funds at an early stage of new venture development [1]. In the beginning of 2018 nevertheless we start to observe (more) established companies launching ICOs (e.g. Kodak and Telegram) to explore this new funding opportunity. The stake (tokens/coins) could represent an utility, asset, commodity, currency or collectible. A main distinction to other forms of early stage venture investments is the fast liquidity cryptoassets can gain on online secondary markets (centralized/decentralized exchanges).

The number of ICOs has increased drastically in the last few years. Unfortunately, the amount of frauds and scams are also amounting largely due to the emergent nature of ICOs and a lack of laws and regulations on this brand-new fundraising mechanism [2]. Stories such as founders disappearing with the large amount of money collected after ICOs are closed are eye-catchy meanwhile worrying for genuine investors and stakeholders[1]. However, conducting due diligence on an expanding number of ICOs would be extremely time-consuming and costly, not only for a layman investor, but also for well-informed analysts equipped with in-depth knowledge about blockchain and ICOs.

As a solution to this issue, online ICO evaluation platforms are emerging and gaining the increasing visibility in online media and forums[2]. What remains unclear, in addition to some other aspects, is how these websites are evaluating ICOs, how reliable and credible their evaluations are, and consequently how useful they are for investors taking investment decisions. Based on this observation, we aim at providing a better understanding of ICO evaluation websites and the evaluation mechanisms behind them. To start with, in this study we have attempted to answer the following research question:

RQ: How are initial cryptoasset offerings evaluated in practice?

To answer the research question, we have studied a set of websites that not only list ICOs but also provide evaluations on them. In total we have identified 28 websites that evaluate ongoing and upcoming ICOs. We have analysed the ICO information covered by these websites and the evaluation mechanisms applied by them to produce evaluation results. The contribution of our study will be an overview of the state of the practice on how ICOs are evaluated, which can lead to the discovery of key information that should be

[1]read the story at https://www.cnbc.com/2017/11/21/confido-ico-exit-scam-founders-run-away-with-375k.html

[2]e.g., https://medium.com/@Demien/top-ico-analysis-websites-1d936f965101, https://bitcointalk.org/index.php?topic=2180784.0;all

considered when evaluating an ICO, as well as how to evaluate such information. In this paper we report the initial findings obtained in our study.

The remaining part of the paper is organised as below. Section 2 explains ICOs in more detail and reviews the exisiting studies on ICOs and their evaluation. In Section 3, we elaborate on how we have collected the set of ICO evaluation websites, following a clearly defined process. Section 4 reports the initial analysis results of these websites, which are further discussed in Section 5. Section 6 concludes the paper with reflecting on the limitations of the study and follow-up research that needs to be done to produce deeper insights on ICOs and their evaluation.

II. Background and Related Work

A. Introduction to ICO

The multiple applications that the blockchain technology underpins [3] [4] [5] give the birth of ICOs, which provides the native support for managing ICOs by Smart Contracts [6] and for raising founds through tokens sold in a cryptovalue. Several definitions of ICOs can be found in the very recent literature. We introduce them by directly extrapolating from the set of papers published in 2017 about ICO fundraising method, ICO comparison, ICO characteristics and so on.

According to Conley et al. [7], ICOs are a vehicle for startups to raise early capital. The tokens of these sales are intended to cover a widely varied set of roles on different platforms.

Similarly, Abgaryan et al. [2] consider ICO a fundraising model largely implemented by startups based on the blockchain ecosystem. The ICO mechanics represents a process of emission of cryptographic tokens to be distributed to the funders and contributors.

ICOs may be defined as open calls for funding promoted by project initiators. The main definition of an ICO is very similar to that of crowfunding [8]. A cryptoasset could represent either a utility or a security. If it provides a utility on a platform e.g. to gain access to products or services, the cryptoasset can be seen as a utility. The ICO in this case shares similarity to the donation-based crowdfunding mechanisms. On the other hand many projects are offering cryptoassets without direct utility. The cryptoasset in this case can be seen as a security in the first place. The ICO therefore is more similar to the investment crowdfunding mechanisms.

However ICO takes the crowdfunding concept to a whole new level. It is claimed as the most efficient means of financing entrepreneurial initiatives in the history of capital formation. ICOs minimize transaction costs and democratize finance while disintermediating banks [9]. Entrepreneurs organize token sales directly. A token is a representation of a value unit. In a token sale, the company provides tokens or coins for investors to buy. The money invested to buy the tokens is the funding amount. Considering that this is an unregulated space, the token sales function is out of most national laws or economic regulations. In this manner entrepreneurs can design any model of token sales. Tokens bought in a crowd sale are often markable (like stocks). This allows traders to sell-buy early and do not have to face a lock-in period.

Some ICOs have also a pre-sale phase, or so-called pre-ICO, which represents a preliminary offer made to (selected) investors.

There are two main channels of an ICO implementation:

- through the websites of the blockchain projects,
- through ICO hosting platforms (e.g., KICKICO). Thus, the main parties involved in the ICO processes are investors, blockchain startups and ICO hosting platforms.

The advantages of ICO as a fundraising mechanism for startups (especially early stage startups) are evident in comparison to traditonal funding mechanisms. VC is a traditional way to fund a startup, but it is very difficult to convince venture capitalists to invest on startups with unproven track records or composed of inexperienced teams. ICOs could disrupt the venture capital fund business model, which plays a key role in financing innovative startups. ICOs' advantage compared with other investment instruments used by venture capital funds is represented by their capability to form highly cost-effective and highly liquid assets primarily on the emerging crypto marketplaces characterized by their economic dynamics. Especially the fast liquidity of ICO investments differ from the traditional equity funding model. In fact, given the competitive elements of ICOs, the venture capital industry is investigating ways to participate in the ICO market. Venture capital funds increasingly try to capitalize on the opportunities presented by ICOs. The disruption of legacy finance by ICOs has triggered attempts by venture capital funds to capitalize on the source of disruption within the existing business model to benefit from its advantage.

IPO is another traditional way to raise funds but too expensive and unavailable for startups especially due to the lack of revenue and userbase. An ICO, instead, is easy to implement for (blockchain) startups. It raises money quickly and the money comes with few strings attached. Unlike IPOs, where companies sell stocks via regulated exchange platforms and are linked to ownership rights (e.g. voting rights) and investor protection, ICOs sell tokens/coins on unregulated (de-)centralized platforms that do not confer direct ownership rights and investor protection. Risks and rewards of tokens differ from those of equity and other securities [1].

Since often ICOs are seen as unregulated security offerings for raising capital, many stakeholders are asking for restrictive regulations. Others however argue that governments should create an innovation sandbox before implementing too restrictive regulations. Restrictive regulations could lead to a massive outflux of intellectual properties, expertises and capital to countries with less restrictive regulations.

B. Related Work

As far as the authors are aware of, there are no previous studies that have investigated the evaluation of ICOs. In this subsection we have reviewed previous studies on ICOs, considering them related work in a broad sense.

Dell'Erba [1] offers a deep study on ICOs. He focuses on their evolution starting from IPOs and crowdfunding, showing how they work and their increasing relevance. In contrast, Conley[7] offers a broad overview of aspects that it needs to take into account for designing a success token. He explains the monetary theory, financial economics, and game theory implications on the token creation process.

ICO mechanism has been studied by Abgaryan et al.[2] who identified five main characteristics that set ICO apart from other fundraising mechanisms. They introduced Aimwise, a platform designed to face up these problems through decentralization of ICO hosting. In the same vein, Barsan[10] proposes a study where ICOs are defined as a venture capital-raising tool for startups projects and blockchain applications that try to escape the constraints of regulation. ICOs are indeed seen as an alternative form of crowdfunding emerged outside the traditional financial sector and mostly finance projects on a public blockchain.

ICO process has also been studied by Kaal et al.[9] who present an overview about new practices, risk factors and red flag about the ICO realm. In particular they highlighted differences with ICOs, IPOs and crowdfunding, arguing also the importance of ICO roadmap and market features.

Yadav[11] conducts an explorative research to investigate signals for an ICO and come up with key themes via semi-structured interviews.The resulting outcomes could then be used as a basis to improve the survey research study aiming to gather more diversified research data.

Robinson[12] argues about the coming decentralized world, providing an overview of both public blockchain technology as well the Ethereum platform. A particular attention is paid to the foundational understanding of how the blockchain works, and the role ICOs play in this new economic ecosystem.

The ICO success rate and success factors start gaining attention of researchers. Adhami et al.[8] provide the first comprehensive description of the ICO phenomenon as well as the determinants of token offerings, analyzing a sample of 253 campaigns occurred from 2014 to August 2017. The results show a success rate equal to 81.0%; the project objectives behind these ICOs are mainly related to fintech services, to the development of a blockchain, or to the issuance of new cryptocurrencies. The authors point out that, according to their e "white paper" is not an influential factor for the ICO success. Instead the code availability, even if only partial, could influence positively ICO evaluation by investors. The market of ICO share several aspects of crowdfunding realm, including low protection of contributors and limited set of available information. The more evident differences between crowdfunding and ICO is represented by the fact that crowdfunding portals collect fiat money through traditional payment channels (banks, credit cards). ICOs, instead, offer tokens and rely on blockchains that act as the clearing house, out of any centralized control. Without a centralized platform (that usually goes through a selection process), the likelihood of an ICO success can only rely on factors related to projects. Indeed there is no proven evidence yet suggesting that a specific platform for ICOs could increase or rationalize fundraising volumes.

Flood et al.[13] have also studied the facets that brought a ICO to be successful. In order to avoid scams, they suggest to pay attention to three main facets of the project underlying an ICO: be a response to a real problem and blockchain is a logical solution, build confidence and trust in solution and team, and be in accordance with national/international legal restrictions.

The aspects of ICOs to take into account are many: regulation, technologies and practices, implications, market factors, etc.. In his study mainly addressing government regulatory authorities, banking sector and stock markets, Venegas [14] identifies some factors that affect cash flows in decentralized applications, to enhance the understanding on whether decentralized organizations are perceived as exactly "trustless entities", or, if investors are rather forced to "trust in the design". He presents a case study and explores risks, costs analysis and network correlation.

As shown in their work, Enyi et al. [15] perform an analysis from the perspective of which rules might be applicable to cryptocurrencies in relation to an ICO. They illustrate also rules that regulators may apply to cryptocurrencies in the context of an initial coin offering, this indicates that regulatory standards are still missing. From this angle, the lack of a regulatory framework may pose a major threat for those investors who have to conduct highly complex cryptofinance investments. Without a legal framework, they are lacking statutory protection emanating for their risky investments.

III. Research Approach

The nature of this study is exploratory due to the newness of the research phenomenon and as a consequence of limited literature on the evaluation of ICOs. To answer the research question, we decided to study the emergent websites that provide evaluations of ICOs. The research approach is composed of two major parts: data collection and data analysis.

A. Data Collection Steps

To identify what are those websites that provide evaluation of ICOs, we decided to use Google search engine. The following data collection steps as it is shown by Bajwa et al. in [16] were followed:

Step 1: Define and refine search keywords: The first step of the data collection was to define the search keywords that can be used to capture ICO evaluation websites. Based on the main research question, we have brainstormed the initial set of search keywords. Meanwhile, a domain expert has provided an initial list of ICO tracking and rating websites that he was aware of. Several trial searches were conducted, and the search results were observed against the initial list to see if the search keywords could capture the listed websites. The search keywords were refined several times based on the feedback from the trials. The final search keywords used in the study is as below (as a valid search string within the 32 keywords limit imposed by Google search engine:

("ICO" OR "initial coin offering" OR "ITO" OR "initial token offering" OR "token sale" OR "cryptocurrency crowdsale") AND ("curated" OR "rating" OR "benchmark" OR "evaluation" OR "monitor" OR "rank" OR "analysis" OR "analytics" OR "track" OR "list" OR "report" OR "review" OR "alert")

Step 2: Apply the defined search keywords to Google search engine: This step was conducted using Safari browser on the laptop of one of the authors. To avoid the potential influence of search history and personal information on the search results, before starting the search process, this author deleted the search history in the browser, cleared browser cache, and made sure that she logged out from her personal Google account. In the Google search setting (google.com/preferences), all options that could potentially influence on search results were turned off.

The search was conducted on the 28th of December, 2017. 169 results were actually shown and accessible, since Google search engine omits automatically the results that it considers similar or duplicates.

Step 3: Export search results: Since it was crucial that multiple researchers could work on the dataset and crosscheck the data analysis results, the search results needed to be exported and shared. Therefore, the author who conducted the Google search installed SEOQuake plugin to her Safari browser, which allowed the export of the search results (in the format of URLs) into an Excel file. We will share the excel file that contains the collected 169 URLs on Google Drive[3].

B. Data Analysis Steps

The following steps were conducted:

Step 1: Decide the relevance of the URLs: Even though the search keywords were tested and refined to capture the most relevant websites related to ICO evaluation, it was still needed to review all the collected URLs manually to decide which URLs represented a valid ICO evaluation website. The criteria to decide if a website can be considered an ICO evaluation website are:

- it contains the information of a set of past, ongoing or upcoming ICOs;
- it provides some form of evaluation on the listed ICOs in the format of ranking, scoring, rating, etc..

Through this step, we have identified 28 ICO evaluation websites that were analysed in the next steps.

Step 2: identify the information relevant to evaluate ICOs: For each ICO evaluation website identified in *Step 1*, the information about each ICO that is listed on the website was identified, such as project description, team information, social media channels, etc.

The result of Step 2 is a comprehensive list of information about an ICO that the identified ICO evaluation website considered relevant.

Step 3: Identify different evaluation mechanisms applied by the identified ICO evaluation websites: The focus of this step is to understand what are the key factors that differentiate the identified ICO evaluation websites, in order to unveil different evaluation mechanisms employed by different websites.

The result of this analysis step is a list of key factors that can be used to construct the evaluation strategy of ICOs.

[3] $https : //drive.google.com/file/d/1lIZUk - yMP4pmXhzJxQ5wErEonRjS_B43/view?usp = sharing$

IV. Findings

Following the data analysis process described in Section III, 28 websites out of the 169 collected URLs have been identified as proper ICO evaluation websites, as shown in Table I. The rest are either websites that only provide basic ICO information without evaluation, such as ICO TRACKER (https://icotracker.net/), or COINTELEGRAPH (https://cointelegraph.com/ico-calendar), or links to social media channels such as Twitter and Reddit. Some websites are simply not relevant to this study even though they were captured by the defined search keywords.

A. Key Basic ICO Information

Table II is a summary of the key basic information about an ICO that an evaluation website could collect, generally provided by a startup that requests its ICO to be evaluated by a given evaluation website. It is not clear (and it is not possible to understand through the data collected in this study) if due diligence has been done to check the correctness of the provided information.

As shown in Table II, to be able to understand an ICO and evaluate it properly, it is not sufficient to have only the information about the ICO itself and the token offered (see the information items listed under the "ICO Information" and "Token Information" categories in Table II). The majority of the 28 websites also list the information about projects and the teams behind them.

One key piece of information about a startup project is the white paper, which most ICO evaluation websites would provide a link to. However, another piece of key information, the roadmap (or sometimes called milestones) of a project, has received much less attention and only about half of the 28 websites include this information when they introduce and evaluate an ICO.

The importance of the team working on a startup project that runs an ICO is highlighted by the fact that 26 out of 28 websites list team members and their expertise as key information of each ICO. In comparison, much less websites have considered to include information beyond the core team, including the advisory team, the partners, and the community built by the project.

One set of information that has received significantly less focus is "Technical Information" about a startup project that runs an ICO. Despite the fact that the majority of the 28 websites provide links to software repository (typically hosted by Github), only one website (ICO Transparency Monitor) includes specific information about smart contracts. Other technical information regarding the underlying blockchain infrastructure that a startup project builds upon is not covered by any of the identified evaluation website.

TABLE I
A LIST OF ICO EVALUATION WEBSITES IDENTIFIED IN THIS STUDY

No.	Name	URL	Centralised or crowd-based evaluation	Evaluation result
1	ICObench	http://icobench.com/	crowd-based	numeric score
2	ICO Transparency Monitor	http://icomonitor.io	crowd-based	numeric score
3	ICOrating	http://icorating.com	centralised	categorical rating + numeric score
4	cyber fund	http://cyber.fund/radar	crowd-based	symbol rating
5	ICO bazaar	http://icobazaar.com	crowd-based	symbol rating
6	ICO Hot List	https://www.icohotlist.com/	centralised	categorical rating
7	ICO drops	https://icodrops.com	centralised	categorical rating
8	CoinSchedule	https://www.coinschedule.com/	centralised	categorical rating
9	Cryptorated	https://cryptorated.com/ico-reviews/	centralised	numeric score
10	ICOWATCHLIST	https://icowatchlist.com/	centralised	categorical rating
11	CrushCrypto	https://crushcrypto.com/ico-analysis/	centralised	assessment report
12	TOKENTOPS	https://tokentops.com/rating/	crowd-based	symbol rating + numeric score
13	TOP ICO LIST	https://topicolist.com/	centralised	categorical rating
14	ListICO.io	https://www.listico.io/	centralised	symbol rating + numeric score
15	ICORanker	https://www.icoranker.com/	centralised	categorical rating + numeric score + assessment report
16	BitcoinX	http://www.bitcoinx.com/ico-list/	centralised	categorical rating
17	DIGRATE	https://digrate.com/icos/rated	centralised	categorical rating + assessment report
18	Foundico	https://foundico.com/	centralised	numeric score
19	Block discover	http://www.blockdiscover.com/category/icos/	centralised	assessment report
20	Wiser ICO	https://wiserico.com/	crowd-based	numeric score
21	Foxico	https://foxico.io/	centralised	numeric score + assessment report
22	VerifiedICOs	https://www.verifiedicos.com	centralised	judgement
23	ICO Jury	https://icojury.com	crowd-based	symbol rating+numeric score
24	CryptoMoon	https://docs.google.com/spreadsheets/d/1js-N4uFteHPAYMAZJRPajDhOkhVCE-iwHdPnxPtuftU/edit#gid=1430929598	centralised	numeric score
25	Picolo Research	https://picoloresearch.com/	centralised	symbol rating+categorical rating
26	GLOBALHALO	https://globalhalo.com/category/ico-reviews/	centralised	assessment report
27	ICOlink	https://icolink.com/icos-list.html	crowd-based	symbol rating+numeric score
28	ICOmarks	https://icomarks.com/ico	centralised	numeric score

TABLE II
A LIST OF BASIC INFORMATION ABOUT AN ICO COVERED BY THE SELECTED EVALUATION WEBSITES

Category	Information Item
Project Information	Description
	Whitepaper
	Roadmap
	Video
	Website
	Social Media Channels
Team Information	Team members
	Advisors
	Partnership
	Community
Token Information	Token Symbol
	Token Type (Asset Class)
	Platform
	Total Token
	Tokens for sale
	Token distribution
	Initial token price
ICO Information	Starting Date
	Ending Date
	Status
	Soft Cap
	Hard Cap
	Accepted currencies
	Pre-ICO
Technical Information	Github repository
	Smart contract

Overall, how well an ICO evaluation website is providing key basic information about an ICO can be understood by what we term as “information richness”, which can be measured by how many key basic pieces of information are covered and how detail each piece of information is. In Figure 1, the horizontal axis indicates the information richness of the 28 websites studied, from low to high.

B. Key Factors of an ICO Evaluation Mechanism

After reviewing how ICOs are evaluated by each of the 28 websites, a set of key factors emerged, which can be used to classify and differentiate the evaluation mechanisms employed by different ICO evaluation websites. These key factors are reported below.

1) Transparency of evaluation process: To understand if the evaluation given to an ICO is reasonable and trustable, an evaluation process needs to be described in sufficient detail and open to scrutinize. The transparency of an evaluation process is one of the key factor to understand how an ICO evaluation website works to provide useful guidance to investors and other stakeholders.

Among the 28 websites analysed, only 6 provided different levels of detailed descriptions of which key information is taken into evaluation, how they are evaluated, and how the final conclusion about and ICO has been achieved. In Figure

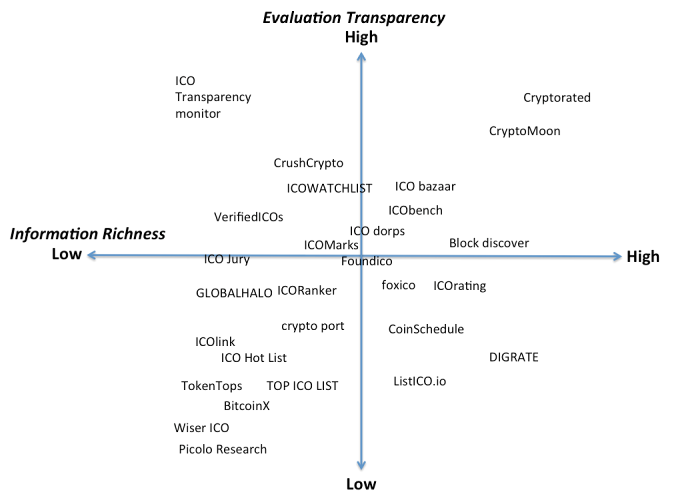


Fig. 1. ICO evaluation websites: information richness vs. evaluation transparency

1, the vertical axis indicates different levels of transparency of evaluation process the 28 websites demonstrated, from low to high. Among them, ICO Transparency Monitor, Cryptorated and CryptoMoon have provided in-depth details on how they evaluate each ICO.

2) Centralised vs. Crowd-based evaluation: Another key factor that differentiates the evaluation mechanisms behind the 28 websites is who is evaluating ICOs. Many websites rely on the task force in house and provide evaluation from their own teams, which is what we call "centralised evaluation". In contrast, interestingly, 8 websites choose to rely on the wisdom of the crowd and outsource evaluation to a larger pool of evaluators. In their cases, one ICO typically receives multiple evaluations from different evaluators and a final evaluation is achieved through averaging different individual evaluations. ICObench is a representative case of crowd-based evaluation.

3) Level of clarity of evaluation result: In the end, the users of an ICO evaluation website need to understand if an ICO is a trustable and worthy subject to invest or engage in other manners. More clear the evaluation results, less effort is needed from the users to make decisions (of course the premise is that the website provides trustable evaluation results, which links to the transparency factor). The analysis of the 28 websites revealed different types of representation of evaluation results, ranging from more vague, subjective opinions of the evaluators in the format of text report or review (e.g., Block discover, GLOBALHALO) to less ambiguous, quantifiable scores/rating/ranking (e.g., ICOmarks, CryptoMoon).

V. Discussion and Conclusion

In this paper we have presented the initial findings from an exploratory study of 28 ICO evaluation websites, in order to understand how ICOs are evaluated in practice. It is shown through the analysis that how ICOs are evaluated varies from website to website, in terms of the amount of key information provided about an ICO, the transparency level of the evaluation process, the source of evaluation and final representation of evaluation result. The findings reported in the paper are descriptive and qualitative in nature given the early phase of our study. In the the next phases of the study we will define and measure key factors of ICO evaluation in a quantitative manner. An additional analysis that could be conducted is to choose a set of ICOs that have been evaluated on multiple websites listed in Table I and see if the evaluations are consistent across different websites.

One aspect regarding ICOs that has received little attention from the identified evaluation websites is the technical information regarding the project behind an ICO, such as the underpinning blockchain system, the consensus mechanism employed by the system, the deeper analysis of software repository and smart contract to assess their quality, etc.. Of course this demands much bigger effort from evaluators of an ICO, but the added value to the evaluation result may justify the effort spent and pay off eventually.

Since not all the identified websites make explicit how they evaluate ICOs, in the next steps we need to conduct further selection, and study those evaluation websites that provide detailed information on their evaluation processes. This is a useful analysis to provide deeper insights on how different evaluation mechanisms working in practice to produce evaluation results.

ICO is at its top point of the hype curve, and there is a very high amount of volatility primarily due to a lack of proper due diligence. ICO evaluation websites could help to elevate this situation to certain extent. The goal of this research is to provide a better understanding of the state of the practice of ICO evaluation. Two main contributions of the study are: 1) Key information about ICOs collected by these websites are categorized, and 2) key factors that differentiate the evaluation mechanisms employed by these evaluation websites are identified. The findings of our study could help a better understanding of what entails to properly evaluate ICOs. It is also a first step towards discovering the key success factors of ICOs.

The research in its current form is by no means conclusive. More in-depth analysis of ICO evaluation websites is desired. However the primary aim is to open gates for future research in the domain of initial cryptoasset offerings. Several next steps have been laid out as argued previously, in order to produce more meaningful and significant results in this highly innovative and growing space.

References

[1] M. Dell'Erba, "Initial Coin Offerings. A Primer," *SSRN Electronic Journal*, 2017. [Online]. Available: file:///C:/Users/Ilaria/Downloads/SSRN-id3063536.pdf https://www.ssrn.com/abstract=3063536

[2] A. Abgaryan, L. Liu, T. Menteshashvili, S. Abgaryan, and C. Gao, "Aimwise: Decentralized ico servicing platform," 2017.

[3] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 2017, pp. 557–564.

[4] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, “Blockchain-oriented software engineering: challenges and new directions,” in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 169–171.

[5] A. Pinna, R. Tonelli, M. Orrú, and M. Marchesi, “A petri nets model for blockchain analysis,” *arXiv preprint arXiv:1709.07790*, 2017.

[6] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, 2014.

[7] J. P. Conley *et al.*, “Blockchain and the economics of crypto-tokens and initial coin offerings,” Vanderbilt University Department of Economics, Tech. Rep., 2017.

[8] S. Adhami, G. Giudici, and S. Martinazzi, “Why do businesses go crypto? an empirical analysis of initial coin offerings,” 2017.

[9] W. Kaal and M. Dell’Erba, “Initial coin offerings: Emerging practices, risk factors, and red flags,” 2017.

[10] I. Barsan, “Legal challenges of initial coin offerings (icp),” 2017.

[11] M. Yadav, “Exploring signals for investing in an initial coin offering (ico),” 2017.

[12] R. Robinson, “The new digital wild west: Regulating the explosion of initial coin offerings,” 2017.

[13] J. Flood and L. Robb, “Trust, anarcho-capitalism, blockchain and initial coin offerings,” 2017.

[14] P. Venegas, “Initial coin offering (ico) risk, value and cost in blockchain trustless crypto markets,” 2017.

[15] J. Enyi and N. Le, “The legal nature of cryptocurrencies in the us and the applicable rules,” 2017.

[16] S. S. Bajwa, X. Wang, A. N. Duc, and P. Abrahamsson, “failures to be celebrated: an analysis of major pivots of software startups,” *Empirical Software Engineering*, vol. 22, no. 5, pp. 2373–2408, 2017.

Checking Laws of the Blockchain with Property-Based Testing

Alexander Chepurnoy*, Mayank Rathee†
* Ergo Platform and IOHK Research
Sestroretsk, Russia
† Department of Computer Science and Engineering, Indian Institute of Technology (Banaras Hindu University)
Varanasi, India

***Abstract*—Inspired by the success of Bitcoin, many clients for the Bitcoin protocol as well as for alternative blockchain protocols have been implemented. However, implementations may contain errors, and the cost of an error in the case of a cryptocurrency can be extremely high.**

We propose to tackle this problem with a suite of abstract property tests that check whether a blockchain system satisfies laws that most blockchain and blockchain-like systems should satisfy. To test a new blockchain system, its developers need to instantiate generators of random objects to be used by the tests. The test suite then checks the satisfaction of the laws over many random cases. We provide examples of laws in the paper.

I. Introduction

A blockchain-based cryptocurrency system consists of a set of policies and protocols, such as the consensus protocol, the monetary policy for token emission, the rules for transaction processing, a peer management protocol and network communication protocols. A *node*, or a *client*, is a software implementation of the protocols. Usually not all the details of all the protocols are rigorously specified. Instead, typically there is a reference client implementation that acts as the definition of the protocols for other implementations. A notable exception here is Ethereum, where the Yellow Paper [1] tries to define all the details of a client implementation. In Bitcoin, however, the reference implementation Bitcoin Core is considered to be standard, and any alternative implementation is expected to reproduce its behavior, and even its bugs [2].

Repeated testing of even the most carefully written and designed system is crucial to expose hidden vulnerabilities in the developed system which might miss the eye of the developers. Such tests should be performed regularly in order help ensure reliability, security and performance of the system. Furthermore, in the case of Bitcoin and other cryptocurrencies, an error in a client implementation could be utterly costly and hard to fix. For example, the famous value overflow bug in Bitcoin [3] caused a fork of more than 50 blocks (more than 8 hours) and required a soft fork (for the majority of miners to upgrade) to be fixed. With an increasing demand for the development of more clients for existing as well as new alternative blockchain systems and cryptocurrencies, such costly bugs can be expected to become more common and problematic and testing can be expected to become one of the most important parts of the software development lifecycle for blockchain systems.

This paper addresses these trends by:

1) proposing a generic property-based testing framework that can be easily plugged into the implementation of a concrete client;
2) describing some of the essential properties which ought to hold true for any cryptocurrency implementation.

A. Property-Based Testing

In this section, we give a formal definition of a property followed by a discussion on property-based testing in contrast to conventional testing methodologies.

Within the scope of a data domain $\mathbb{D}$, a property can be seen as a collective abstract behavior which has to be followed by every valid member of the data domain. More precisely, a property is a predicate $P : \mathbb{D} \to \{true, false\}$ and it is desirable that it be *valid*:

$$\forall X \in \mathbb{D}, P(X) = true$$

To illustrate, an example of a property P over the domain of pairs of strings $\mathbb{S} \times \mathbb{S}$ is shown below:

$$P((s_1, s_2)) = \#(s_1 :: s_2) > \#s_1$$

where :: denotes string concatenation and $\#s$ denotes the length of string s. This property is false for any (s_1, ε), where ε is the empty string. Therefore, it is not valid.

In contrast to conventional testing methods, where the behavior of a program is only tested on some pre-determined cases, property-based testing [4] emphasizes defining properties and then testing their validity against randomly sampled data points. As property-based testing uses a small number of randomly sampled data points, it still provides only an approximate answer to the question of whether a property is satisfied on all data points. However, it may provide more confidence than conventional unit testing, because the randomly sampled data points may cover problematic cases that were not foreseen by the developers. There are various popular libraries available for property testing including QuickCheck for Haskell [5], JUnit-QuickCheck for Java [6], theft for C, ScalaTest [7] and ScalaCheck [8] for Scala.

Property-based testing is also advantageous when testing an application developed on top of a general framework, as is the case of blockchain systems developed on top of Scorex, because the framework may provide pre-implemented properties that the application should satisfy and the application developer just needs to implement application-specific generators of random data points.

B. The Scorex Framework

The idea of a modular design for a cryptocurrency was first proposed by Goodman in the Tezos position paper [11]. The paper (in Section 2) proposes to split a cryptocurrency design into the three protocols: network, transaction and consensus. In many cases, however, these layers are tightly coupled and it is hard to describe them separately. For example, in a proof-of-stake cryptocurrency a balance sheet, which representation is heavily influenced by a transaction format, is used in a consensus protocol.

Plenty of modular open-source frameworks were proposed for speeding up development of new blockchain systems, including: Sawtooth [12] and Fabric [13] by Hyperledger, Exonum [14] by Bitfury Group, and Scorex [15] by IOHK. We have chosen Scorex, because it has finer granularity. In particular, in order to support hybrid blockchains as well as more complicated linking structures than a chain (such as Spectre[16]), Scorex does not have the notion of blockchain as a core abstraction. Instead, it provides a more general abstract interface to a *history* which contains *persistent modifiers*[1]. The history is a part of a *node view*, which is a quadruple of $\langle$*history*, *minimal state*, *vault*, *memory pool*$\rangle$. The node view is updated whenever a persistent modifier or a transaction is processed. The minimal state is a data structure and a corresponding interface providing the ability to check the validity of an arbitrary persistent modifier for the current moment of time with the same result for all the nodes in the network having the same history. The minimal state is to be obtained deterministically from an initial pre-historical state and the history. The vault holds node-specific information, such as a user's wallet. The memory pool holds unconfirmed transactions being propagated across the network by nodes before their inclusion into the history. Such a design, described in details in Section II, gives us the possibility to develop an abstract testing framework where it is possible to state contracts for the node view quadruple components without knowing details of their implementations.

C. Our Contribution

This paper reports on the design and implementation of a suite of abstract property tests which are implemented on top of the Scorex framework to ease checking whether a blockchain client satisfies the specified properties. A developer of a concrete blockchain system just needs to implement generators of random test inputs (for example, random blocks and transactions for a Bitcoin-like system), and then the testing system will extensively check properties against multiple input objects. We have implemented 59 property tests, and integrated them into a prototype implementation of the TwinsCoin [17] cryptocurrency.

[1]In a blockchain-based cryptocurrency, the blockchain can be seen as the history and every block can be seen as a persistent modifier.

D. Related Work

Verification and testing of software systems [18] is an integral part of a software development lifecycle. Immediately after the implementation of the software, and before its deployment, it has to be verified and tested extensively enough to ensure that all the functional requirements have been properly met. Over the course of time, both testing and verification methods have been becoming increasingly formal, sophisticated and automated.

Formal verification usually involves constructing an abstract mathematical model (a.k.a. specification) of the system's desired behavior. From a logical perspective, the specification can be regarded as a collection of properties that ought to hold for the system, although often the specification is not described directly in logical form, but rather using various mathematical modeling frameworks, such as finite state machines [19], Petri Nets, process algebra and timed automata [20]. Once both the specification and the system are ready, the actual verification that the program satisfies the specification can be attempted. If successful, the verification proves that the properties of the specification are valid for the program. This is a starkly stronger result than what can be achieved through testing, where the properties are only checked on a few samples. However, full verification is hard to achieve automatically, and expensive to do manually or interactively.

Testing (either conventional or property-based) remains a less costly and hence more prevalent approach. Since a software program is developed at module or class level and is integrated with other modules or classes along the development cycle, testing is done at unit level, integration level and system level [18], before the software is deployed. End-to-end testing [21] is also performed, usually after system testing, to validate correct flow spanning different components of the software in real world use cases. Unit tests target individual modules, methods or classes and have a small coverage compared to integration tests which aim towards checking the behavior of modules when combined together. The two main approaches to unit testing are black box testing and white box testing. The former one focuses on designing test instances without looking inside the code or design, in other words, the black box testing focuses only on the extensional functionality of the unit under testing, while the white box testing approach is more inclined towards code coverage (i.e. ensuring that test instances execute as many different paths of the code as possible).

Although initially white box testing was considered only as a method for unit testing, recently it has emerged as a popular method for integration testing as well. Integration testing is usually done by one or a combination of the following approaches:

1) *Big-Bang approach:* all the components are integrated together at once and then tested. This method works well for comparatively smaller systems, but is not well suited for larger systems. One obvious disadvantage is that the testing can only begin after all the individual components have been built.

2) *Top-Down approach:* the modules at upper level are tested first and then we move down until we test the lowest level modules at the end. Since lower level modules might not be developed when the upper ones are being tested, stubs are used in place of such modules. The stubs try to simulate behaviour of the modules not yet implemented.
3) *Bottom-Up approach:* in contrast to the top-down approach, here the lower level modules are tested first and then we iteratively move upwards in the hierarchy until we reach the highest level module. Now as we are testing lower level modules first, stubs are used to simulate the behaviour of not yet implemented higher level modules, in case any sibling interaction is required.
4) *Sandwich approach:* this combines the Bottom-Up and Top-Down approaches.

Going beyond conventional unit testing methods, which do not take any input parameters, parameterized unit tests [22] are generalized tests that have an encapsulated collection of test methods whose invocation and behaviour is controlled by a set of input parameters giving more flexibility and automation to unit testing as a whole.

The final full scale testing that a software product undergoes is called the system testing, which includes tests like security test, compatibility test, exceptions handling, scalability tests, stress tests and performance tests.

Stress tests are particularly important for electronic payment systems, even conventional ones that are not based on cryptocurrencies. Visa, for instance, performed an annual stress test in 2013 to prepare their VisaNet system for the peak traffic of the upcoming holiday season. The test results showed that the system was able to handle 47,000 transactions per second, a 56% improvement compared to the system's capacity in the previous year. Within cryptocurrencies, the Bitcoin network experienced a spam campaign called *"stress test"* [23], which caused the network's performance to degrade and essentially resulted in a denial-of-service attack [2]. The intention behind this campaign was to expose vulnerabilities of the network, particularly when facing spam attacks. The maximum transaction verification rate of a network under spam can be improved through clustering of transactions to differentiate spam and genuine transactions [23] or through $UTXO$-cleanup transactions, a new special type of transaction created by miners to combine spam transactions together, thereby reducing the $UTXO$ set size and the impact of the spam attack on the network.

E. Structure of the Paper

In Section II we explain the architecture of the Scorex framework. We then describe our approach to property-based testing of blockchain system properties and present many examples of blockchain property tests in Section III. Finally we state our conclusions in Section IV.

[2]a cyber-attack on a system where the attack makes the system's resources unavailable or degrades their quality to a point where it becomes difficult or sometimes impossible for honest users to avail the resource

II. Scorex Architecture

Scorex is a framework for rapid implementation of a blockchain protocol client. A client is a node in a peer-to-peer network. The client has a local view of the network state. The goal of the whole peer-to-peer system [3] is to synchronize on a part of local views which is a subject of a consensus algorithm. Scorex splits a client's local view into the following four parts:

- *history* is an append-only log of *persistent modifiers*. A modifier is persistent in the sense that it has a unique identifier, and it is always possible to check if the modifier was ever appended to the history (by presenting its identifier). There are no limitations on a modifier structure, besides the requirements to have a unique identifier and at least one parent (referenced by its identifier). A persistent modifier may contain transactions, but this is optional. A transaction, unlike a persistent modifier, has no mandatory reference to its parents; also we consider that a transaction is not to be applied to the history and a minimal state (described below). If a modifier is applicable to a history instance and so could be appended to it, we say that the modifier is *syntactically* valid. As an example, in a Bitcoin-like blockchain the history is about a chain of blocks. A block is syntactically valid if its header is well-formed according to the protocol rules, and current amount of work was spent on generating it. However, a syntactically valid block could contain invalid transaction, see a notion of semantic validity below. We note that there are alternative blockchain protocols with multiple kinds of blocks, microblocks, paired chains, and so on, that is why we have chosen abstract notions of a persistent modifier and a history, not the block and the blockchain.
- *minimal state* is a structure enough to check semantics of an arbitrary persistent modifier with a constraint that the procedure of checking has to be deterministic in nature. If a modifier is valid w.r.t minimal state, we call it a *semantically* valid modifier. Thus, in addition to syntax of the blockchain, there is some stateful semantics, and minimal state takes care of it. That is, all nodes in the system do agree on some pre-historical state S_0, and then by applying the same sequence of persistent modifiers $m_1, \ldots, m_k$ in a deterministic way, all the nodes get the same state $S_k = apply(\ldots apply(apply(S_0, m_1), m_2), m_k)$ if all the $m_1, \ldots, m_k$ are *semantically* valid; otherwise a node gets an error on the first application of a semantically invalid persistent modifier. From this more abstract point of view, the goal of obtaining the state S_k is to check whether a new modifier m_{k+1} will be valid against it or not. Thus the minimal state has very few mandatory functions to implement, such as $apply(\cdot)$ and $rollback(\cdot)$ (the latter is needed for forks processing).

[3]concretely, its honest nodes. For simplicity, we omit a notion of adversarial behavior further.

- *vault* contains user-specific information. For a user running a node, the goal to run it is usually to get valuable user-specific information from processing the history. For that, the vault component is used which has the only functions to update itself by scanning a persistent modifier or a transaction and also to rollback to some previous state. A wallet is the perfect example of a vault implementation.
- *memory pool* is storing transactions to be packed into persistent modifiers.

The history and the minimal state are parts of local views to be synchronized across the network by using a distributed and decentralized consensus algorithm. Nodes run a consensus protocol to form a proper history, and the history should result in a valid minimal state when persistent modifiers from the history are applied to a publicly known prehistorical state.

The whole node view quadruple is to be updated atomically by applying either a persistent node view modifier or an unconfirmed transaction. Scorex provides guarantees of atomicity and consistency for the application while a developer of a concrete system needs to provide implementations for the abstract parts of the quadruple as well as a family of persistent modifiers.

A central component which holds the quadruple <*history, minimal state, vault, memory pool*> and processes its updates atomically is called a *node view holder*. The holder is processing all the received commands to update the quadruple in sequence, even if they are received from multiple threads. If the holder gets a transaction, it updates the vault and the memory pool with it. Otherwise, if the holder gets a persistent modifier, it first updates the history by appending the modifier to it. In a simplest case, if appending is successful (so if the modifier is syntactically valid), the modifier is then applied to the minimal state. However, sometimes a fork happens, so the state is needed to be rolled back first, and then a new sequence of persistent modifiers is to be applied to it.

As an example, we consider the cryptocurrency Twinscoin [17], which is based on a hybrid proof-of-work and proof-of-stake consensus protocol. Scorex has a full-fledged Twinscoin implementation as an example of its usage. There are two kinds of persistent modifiers in Twinscoin: a proof-of-work block and a proof-of-stake block. Thus the blockchain is hybrid: after a Proof-of-Work block it could be only a Proof-of-Stake block, and on top of it there could be only a Proof-of-Work block again. Thus a TwinsCoin-powered blockchain is actually two chains braided together. Only Proof-of-Stake block could contain transactions. Such complicated design makes Scorex a good framework to implement the TwinsCoin proposal. Unfortunately, TwinsCoin authors made only some particular tests. We got working tests for the Twinscoin client just by writing generators for transactions and persistent modifiers.

It could be the case that in a decentralized network two generators are issuing a block at the same time, or in the presence of a temporary network split different nodes are working on different suffixes starting with the same chain, or an adversary may generate blocks in private and then present them to the network. In short, a fork could happen. This is a normal situation once majority of block generators are honest (see [24] for formal analysis of the Bitcoin proof-of-work protocol).

Processing forks in a client could be a complicated issue, making testing of this functionality important. We proceed by describing the way in which forking is implemented in Scorex. When a persistent modifier is appended to a history instance, the history returns (if the modifier is syntactically valid) *progress info* structure which contains a sequence of persistent modifiers to apply as well as a possible identifier of a modifier to perform rollback (for the minimal state, vault, memory pool) to before the application of the sequence. By such a realization of the interfaces, Scorex allows history to be non-linear (for example, it could be a block tree), but other components of the node view quadruple have sequential logic. For efficiency reasons, the minimal state is usually limited in maximal depth for a rollback, so the rollback could fail (this situation is probably unresolvable in a satisfactory way without a human intervention).

III. Property-based testing of a blockchain client

In this section we report on our approach to generalized exhaustive testing of an abstract blockchain (or blockchain-like) protocol implementation. For extensive testing, we test history, minimal state and memory pool components separately, and also do thorough checks for node view holder properties.

In total, we have implemented 59 property tests. They are using random object generators described in Section III-A. Most of the tests are relatively simple, others could check complex functionalities where several components are involved. We provide many examples in Section III-B.

A. Generators

We recall that (unlike unit tests, for example), property-based tests are checking not an output of a functionality under test against a concrete input, but rather a relation between input and output values for an arbitrary input value. Thus, in order to run a property-based test, an instance of an input value is needed. To be able to obtain it, a property-based test is supplied with a random input generator, which provides a random input domain element upon request. For our testing framework, a developer of a concrete protocol client needs to provide implementation for generators of the following types:

- a syntactically valid (respectively, invalid) modifier, which is valid (respectively, invalid) against given history instance
- a semantically valid (respectively, invalid) modifier, which is valid (respectively, invalid) against given minimal state instance. The modifier could be syntactically invalid
- a totally, so both semantically and syntactically, valid modifier. Respectively, a sequence of totally valid modifiers
- a transaction

- history instance, for which it should be possible to generate a syntactically valid modifier
- minimal state instance, for which it should be possible to generate a semantically valid modifier
- vault instance
- node view holder instance, for which it should be possible to generate a totally valid modifier

As an example, for the TwinsCoin implementation we provide concrete implementations for all the generators mentioned above. To generate a syntactically valid modifier, we generate a Proof-of-Work block if a previous pair of <*Proof-of-Work block, Proof-of-Stake block*> is complete, otherwise we generate a new Proof-of-Stake block. We recall that in TwinsCoin transactions can be recorded only in Proof-of-Stake blocks. A minimal state in the TwinsCoin implementation, similarly to Bitcoin, is defined as a set of current unspent transaction outputs. In order to generate a semantically valid modifier, we generate a Proof-of-Stake block including transactions based on unspent transaction outputs. A totally valid modifier generator, based on given history as well as minimal state instances, produces either an empty Proof-of-Work [4] or a semantically valid Proof-of-Stake block, depending on the last block in the history (in order to generate the modifier which is also syntactically valid).

Interestingly, we implicitly define some properties via generators. In particular, the existence of a generator for a totally valid modifier for any given correct history and valid minimal state instances assumes that it is always possible to make a progress in constructing a blockchain. To the best of our knowledge, the need of this property to be hold was first stated in the formal model of the Bitcoin protocol by Garay et. al. [24] (see "Input Validity" definition in Section 3.1 of the paper [24]).

B. *Examples of Properties Tests*

To explain our approach to the testing of a client in details, in this section we provide some examples of property tests which are valid for most blockchain-based systems. We have grouped the tests based on their similarity.

1) *Memory Pool Tests.*
   Memory Pool (or just mempool in the Bitcoin jargon) is used to store unconfirmed transactions which are to be included into persistent modifiers. The following tests are used to check some general properties of a memory pool which every blockchain client should pass.
   - *A memory pool should be able to store enough transactions:* in TwinsCoin implementation, we are testing that the memory pool which is empty before the test should be able to store a number of transactions up to a maximum specified in settings.

[4]unlike Bitcoin, Twinscoin does not have a notion of a coinbase transaction rewarding miner, instead, block generator's public key is included into the block directly.

   - *Filtering of valid and invalid transactions from a memory pool should be fast:* we got an impression from running the Twinscoin client that memory pool probably spends too much time on filtering out a transaction. To be certain about that we have implemented a test which is checking that an implementation of memory pool is able to filter out a transaction reasonably fast. As processing time is platform-dependent, the test during its instantiation is measuring time to calculate 500,000 blocks of SHA-256 hash. Time to filter out the transaction should be no more than that. We found that the Twinscoin implementation was really inefficient about filtering.
   - *A transaction successfully added to memory pool should be available by a transaction identifier:* the purpose of this test is to ensure that once a transaction is added to the memory pool, it indeed is available by a transaction identifier. The test simply adds the transaction to the memory pool and then query the transaction by its identifier. The initial transaction is the only correct result of the compound operation.

2) *History Tests.*
   A history is an abstract data structure which records all the persistent modifiers ever appended to it. We recall that the blockchain structure in the Bitcoin protocol is the example of a history implementation. Since history is an integral part of a node view, it is important to check if an implementation of history acts correctly. A consensus protocol aims at establishing common history for all the nodes on the network.
   A persistent modifier is the main building block of a history and is used to update the history and a minimal state. As soon as a valid modifier got appended to history, the whole node's local view is being changed in the sense that the history is updated, possibly along with the minimal state.
   We have many tests to test history, some examples are provided below.
   - *A syntactically valid persistent modifier should be applicable to a history instance and available by its identifier after that:* by definition, a syntactically valid persistent modifier should be applicable to a history instance, and then, once applied to the history, it should be available by its unique identifier. The importance of this test comes from the fact that it is of utmost importance for the client implementation to tell the difference between the modifiers that have been appended to the history from those that have not been added. For this purpose, the unique identifier of the modifier can be used to query the history to know whether the

modifier has been added to the history of not.

- *A syntactically invalid modifier should not be able to be added to history:* a syntactically invalid modifier should not be applicable to a history instance. The test first checks whether application of the modifier returns an indication of an error. Then the test checks that the modifier should not be available by its identifier.

- *Modifier not ever appended to a history instance should not be available by request:* when the persistent modifier which was never appended to the history, is queried from the history, it should always return empty result which shows that the invalid modifier has not been added to the history.

- *After application of a syntactically invalid modifier to a history instance, it should not be available in history by its identifier:* a syntactically invalid modifier is one which is inconsistent with the present view of history. Only syntactically valid modifier is eligible to be applied to history. To check how the history is filtering out invalid modifiers, we propose this test. We generate a random invalid modifier and attempt to add it to history. Since it is invalid, history should not add it and hence, it should not be available by identifier when queried from history. Some examples of generated invalid modifiers are ones with false nonces which do not satisfy the puzzle and ones with non-valid signatures.

- *Once a syntactically valid modifier is appended to history, the history should contain it:* this test ensures that if valid modifiers are correctly appended to the history, then they should be then available by their respectable identifiers. Also, the test is checking that the history is indicating success during the application.

- *History correctly reports semantic validity of an identifier:* a history instance should be able to indicate semantic validity of a persistent modifier. If the modifier is not appended to the history yet, the history should return on request that semantic validity of the modifier is not known. The same result should be returned once the modifier is appended, but semantic validity status is not provided by the node view holder (after applying the modifier to the minimal state). Once semantic validity status is provided, the history should return it by request. The test checks all the options, simulating node view holder with a stub.

3) *Minimal State Tests.*
Tests for the minimal state component are checking application of a semantically valid (respectively, invalid) persistent modifier, and also rollback functionality. In case of a better version of history (a fork) found, a rollback has to be performed which essentially rolls the system back to a common point (from which forks are started). Known examples of rollbacks performed in the Bitcoin network are recovery from the SPV mining issue [25], and also recovery from the arithmetic overflow bug [3].

- *Application and rollback should lead to the same minimal state:* in this test, a semantically valid persistent modifier m is generated and applied to a current state S. Due to this application of the modifier, a new minimal state S' is to be obtained from S. After the modifier is applied successfully to the history, a rollback is performed to take the system back to state S from the current state S'. The test now checks that the state to which the system comes after the rollback is indeed the state S by checking an identifier of the new state after rollback is the same as the identifier of the original state.

  We now use this test to explain how we generate a semantically valid modifier for the Twinscoin client. Before proceeding, we define the structure of a transaction t. A simple transaction is usually represented as the map $T : UTXO \rightarrow UTXO$, where $UTXO$ is the set of all the unspent transaction outputs or *boxes*. A box can be considered as a tuple $(pubkey, amount)$ where $pubkey$ is the public key of the account of the node to which this box belongs to and $amount$ refers to the monetary (in case of cryptocurrencies) amount which this box holds in the name of the $pubkey$ in the first half of the tuple. A transaction uses some ($\geq$ 0, 0 in the case of the rewarding transaction which is present at the end of each block and which rewards the miner) unspent boxes and generates new boxes with a constraint that sum of the amount of all the boxes used in the transaction is equal to the sum of amount of the boxes output by the transaction except the rewarding transaction for which this constraint doesn't apply. Once the new boxes have been added to the UTXO set, the old boxes which were input to the transaction are removed from the UTXO set to prevent double spending. This addition and removal of boxes from $UTXO$ set has to be done atomically in order to avoid inconsistencies in the system. Suppose a node A wants to send x amount to a node B, then the transaction for this purpose will use some boxes with As public key on them which sum up to an amount $y \geq x$ and output the

boxes b_1 and b_2 such that $b_1 = (B, x)$ which has Bs public key and an amount x whereas $b_2 = (A, y-x)$ will have As public key and an amount of $y - x$, if $y > x$. Now B has received a new box b_1 which belongs to him and this box now sits inside the set $UTXO$ until the point when B uses this box an one of the inputs to a future transaction. Readers should note that for more readability we will represent a transaction T by a tuple ([$in_1, in_2, ...$], [$o_1, o_2, ...$]) where in_i denotes input boxes to the transaction and o_i denotes the output boxes of the transaction.
Along with checking that the ids are same, it also checks that the components of the new state are also rolled back and not just the id number got rolled back. For this, we generate the modifier m in the following way:

– Generate a pair of transactions (t_1, t_2) where $t_1 = ([b_1], [b_2])$ and $t_2 = ([b_2], [b_3])$. This notation means that the first transaction t_1 uses a box b_1 as its input and then outputs a box b_2 which is then used by the second transaction as its input. In the above setting, we select the first input box b_1 randomly from the set $UTXO$ and finally output the box b_3 from t_2 which also just generates a random box (b_3). The generated transaction pair has to be valid in the sense that it should only use valid unspent boxes from $UTXO$ and satisfy the constraint that the sum of amounts of all the input boxes should be equal to the sum of amounts of the output boxes. The main caveat here is that the second transaction of the pair should use the output box of the first transaction of the pair.
– Now we generate a pair of modifiers (m_1, m_2) and include both of these transactions from the pair above in the respective modifiers.

Once the custom modifiers are generated, m_1 (first half of the pair) is appended to the history and the system moves from state s_1 to the state s_2. As mentioned before, the transaction t_1 from the pair uses a random box b_1 from the $UTXO$ of state s_1 and when the system moves to the state s_2, the $UTXO$ gets added with the box b_2 and b_1 is removed from the set. Once the state change happens, we append m_2 (second half of the modifier pair) to history progressing the system to state s_3. Since m_2 contains the transaction t_2 which takes as input b_2, when the system moves to s_3, b_2 is removed from $UTXO$ and b_3 is added.

Now it becomes clear why we generate pairs of transactions and modifiers in the way defined above. Finally, we perform a rollback from state s_3 to s_2 which should mean that once the rollback is successful, the box b_2 should come back to the set $UTXO$ and should be available by id whereas the box b_3 should now not be present inside the $UTXO$ set anymore. Both of these checks tell us that the rollback was performed correctly and the system indeed came back to the previous state with all its components. The reason that we generated the pairs of transactions above is because it helps us in easily checking by id if b_2 has returned to the $UTXO$ set since we generated b_2 ourselves and know its id already. This makes testing easy and transparent.

- *Application of a valid modifier again after a rollback should be successful:* as the previous test aimed at checking that the components of a state are recovered after a rollback happens, it would be quite wrong to think that it should be the only test that is necessary to check if the rollback system performs as expected. It is also equally important that after rollback the system performs normally, as it would perform if the rollback would have never happened. To check this property to certain degree, we propose this test. It checks that after the rollback has happened the system becomes stable again and any new valid modifier which is now added to the history is actually recorded and hence should be available if queried from history. This test ensures that after recovering from a rollback the system performs normally and can resume its functioning without any issues. It hence ensures that a continuity is maintained after a rollback.

- *Double application of a semantically valid modifier should not be possible:* this test checks that a semantically valid persistent modifier should not be added more than once. For example, if in Bitcoin a block could be successfully applied twice to the validation state (UTXO set), all the transaction inside the modifier will be double spent. We argue that an implementation of a blockchain system should prevent addition of a semantically valid persistent modifier twice in general. In this test, we generate a semantically valid modifier, then append it to a generated minimal state once, and on the second application of the modifier again to the minimal state an error should be returned.

4) *Node View Holder Tests.*
As was mentioned in Section II, the node view holder is the central component of a blockchain client which is responsible for atomically updating the quadruple <*history, minimal state, vault, memory pool*>. The update could be triggered by whether a persistent modifier of a transaction coming in. Below we provide some implemented tests for the node view holder.
- *A totally valid persistent modifier should*

successfully update the minimal state and the history: we recall that a totally valid modifier is a persistent modifier which is valid for both the history and the minimal state, so it is applicable to both of them. In this test we are sending a random totally valid persistent modifier to the node view holder component and then we are checking that history contains it and the version of the minimal state is equal to the modifier's identifier.

5) *Forking tests.*
At the moment we have two tests for forking. In both of them we apply random number of totally valid modifiers in the first place and remember last block which we call the common block. One test then is applying two totally valid modifiers which have the common block as parent. The test checks that after the application the history contains whether one of these modifiers, and version of the minimal state is equals to identifier of whether one of the two modifiers. The logic behind such a check is that we do not know whether an implementation of a node view holder will make switching from one prefix (of length one) to another (of the same length), but anyway the general property should hold. Another test first generates a sequence of totally valid persistent modifiers of length 2, applies it to the common block, then the test generates a sequence of totally valid persistent modifiers of length 4 starting from the common block, and applies the longer sequence. The test checks that switching takes place, so minimal state version equals to the identifier of the last block in the longer sequence, and the history contains the identifier in the current modifiers which do not have ancestors (for a blockchain, there is one such a modifier, for a block tree, there could be more than one modifiers returned). With the help of the forking tests we have found few errors in the Twinscoin implementation.

IV. Conclusion

In this paper we propose to improve quality of blockchain protocol implementations via exhaustive property-based testing. For generic abstract modular Scorex framework, we have implemented a suite of property-based tests. The suite consists of 59 tests checking different properties of a blockchain system. To run the suite against a concrete blockchain protocol client, developers of the client need to provide generators for random objects used by the protocol. The suite is checking properties against the implementation by using random samples. We used Twinscoin implementation provided with Scorex as an example of a concrete blockchain using our testing kit. In the paper we provide many examples of the tests.

Acknowledgments

Authors would like to thank Bruno Woltzenlogel Paleo for his valuable comments and corrections.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014, available at https://ethereum.github.io/yellowpaper/paper.pdf.
[2] P. Todd, "The difficulty of writing consensus critical code: the sighash_single bug," available at https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2014-November/006878.html.
[3] B. Wiki, "Value overflow incident," available at https://en.bitcoin.it/wiki/Value_overflow_incident.
[4] D. Ron, "Property testing," *COMBINATORIAL OPTIMIZATION-DORDRECHT-*, vol. 9, no. 2, pp. 597–643, 2001.
[5] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.
[6] T. Jung, "Quickcheck for java," 2015.
[7] B. Venners, "Scalatest," 2009.
[8] R. Nilsson, "Scalacheck: the definitive guide," 2014.
[9] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Formal Description Techniques VII*. Springer, 1995, pp. 197–211.
[10] M. H. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs: A survey," *Microelectronics journal*, vol. 39, no. 12, pp. 1395–1404, 2008.
[11] L. Goodman, "Tezos: A self-amending crypto-ledger," available at https://www.tezos.com/static/papers/position_paper.pdf.
[12] H. website, "Hyperledger sawtooth," available at https://hyperledger.org/projects/sawtooth.
[13] ——, "Hyperledger fabric," available at https://hyperledger.org/projects/fabric.
[14] B. Group, "Exonum - a framework for blockchain solutions," available at https://exonum.com/.
[15] IOHK, "Scorex 2.0 code repository," available at https://github.com/ScorexFoundation/Scorex.
[16] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol," Cryptology ePrint Archive, Report 2016/1159, 2016, http://eprint.iacr.org/2016/1159.
[17] A. Chepurnoy, T. Duong, L. Fan, and H.-S. Zhou, "Twinscoin: A cryptocurrency via proof-of-work and proof-of-stake," Cryptology ePrint Archive, Report 2017/232, 2017, http://eprint.iacr.org/2017/232.
[18] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
[19] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.
[20] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.
[21] W.-T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE, 2001, pp. 166–171.
[22] N. Tillmann, J. de Halleux, and T. Xie, "Parameterized unit testing: Theory and practice," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2. IEEE, 2010, pp. 483–484.
[23] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, "Stressing out: Bitcoin "stress testing"," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 3–18.
[24] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 281–310.
[25] "Spv mining." [Online]. Available: https://bitcoin.org/en/alert/2015-07-04-spv-mining

Author Index