IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 15, 2019, accepted June 3, 2019, date of publication June 10, 2019, date of current version June 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2921936

A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics

ANDREA PINNA[1], SIMONA IBBA[1], GAVINA BARALLA[1], ROBERTO TONELLI[2], AND MICHELE MARCHESI[2], (Member, IEEE)

[1]Department of Electric and Electronic Engineering (DIEE), University of Cagliari, 09123 Cagliari, Italy
[2]Department of Mathematics and Computer Science, University of Cagliari, 09124 Cagliari, Italy

Corresponding author: Andrea Pinna (a.pinna@diee.unica.it)

This work was supported in part by Regione Autonoma della Sardegna, under projects ''EasyWallet''-POR FESR Sardegna 2014–2020 and ''CAFCha-Certification of AgriFood Chain''-POR FESR Sardegna 2014–2020.

ABSTRACT In this work, we perform a comprehensive empirical study of smart contracts deployed on the ethereum blockchain. The objective of the analysis is to provide empirical results on smart contracts features, smart contract transactions within the blockchain, the role of the development community, and the source code characteristics. We collected a set of more than 10 000 smart contracts source codes and a dataset of meta-data regarding their interaction with the blockchain from etherscan.io. We examined the collected data computing different statistics on naming policies, smart contract ether balance, number of smart contract transactions, functions, and other quantities characterizing the use and purpose of smart contracts. We found that the number of transactions and the balances follow power-law distributions and the software code metrics display, on average, values lower than corresponding metrics in standard software but have high variances. Focusing the attention on the 20 smart contracts with the topmost number of transactions, we found that most of them represent financial smart contracts and some of them have peculiar software development stories behind them. The results show that blockchain software is rapidly changing and evolving and it is no longer devoted only to cryptovalues applications but to general purpose computation.

INDEX TERMS Blockchain, code metrics, ethereum, smart contracts, solidity.

I. INTRODUCTION

The publication of the Ethereum white paper in 2014 [4] and the implementation of the Ethereum platform moved the blockchain technology [20] to the second generation. In fact, what this platform for decentralized applications proposed, was new and disruptive: a blockchain-based programmable Turing complete virtual machine to run software code written specifically for the blockchain environment [24]. Such software was originally conceived to take advantage of the blockchain features in order to automatically implement the constraints two parties can agree upon when they sign a contract in a trustless environment, so that the software code was named ''Smart Contract''. Nowadays, the initial concept has been largely extended so that Smart Contracts can be considered as general purpose software programs, as we show in our empirical analysis.

Smart Contracts (SCs for short) are small computer programs stored inside the Ethereum public ledger (or inside another blockchain) and associated to a particular blockchain address which references the SC software code.

Ethereum Smart Contracts are mainly written in Solidity, a programming language derived from Javascript, Python and C++, which allows to run programs on the blockchain infrastructure as decentralized applications. The Smart Contracts code is compiled and the corresponding *bytecode* is recorded into the blockchain and run by the Ethereum Virtual Machine (EVM). Virtually, SCs can perform any computational task standard programs can perform, but there are specific constraints that must be respected due to the decentralized structure of the blockchain and to the consensus protocol adopted by Ethereum, so that SCs display specific features and issues which are unknown in traditional software development. A typical example is the extraction of a pseudo-random number which should be replicated in all the blockchain nodes in order to obtain the same result [13].

Due to these specific features, this technology is having a great success and has paved the way for a new set of applications, yet to be fully exploited. Ethereum is the most important blockchain based platform in terms of number

The associate editor coordinating the review of this manuscript and approving it for publication was Sun Junwei.

of transactions. At time of writing the number of accounts stored in the blockchain is higher than sixty millions. The number of contract created in the blockchain is over fourteen million five hundred thousand[1] Contract accounts are used both to create decentralized applications and to create new digital tokens, looking to new business opportunities and to an easier way of funding (the ICO phenomenon [10], [11]). The byte-codes of contracts are always available, because they are recorded in the blockchain. However, byte-codes are not intelligible; in order to increase the trust of users, developers of decentralized applications may provide the source code of their contracts. Third party websites, like Etherscan.io, offer a verification service that makes Smart Contracts source code public. The overall success of decentralized applications presents practitioners and software engineers with new and specific challenges. In the scenario of a wide diffusion of the blockchain technology, Smart Contracts could represent the backbone for several future decentralized applications [12], [14], [15], [20].

Since blockchain is a newborn technology, the development of new decentralized applications could take advantage of a thorough analysis of what has been created up to now, with the aim of analyzing errors of the past and of improving software development best practices. By the end of 2017 the amount of Smart Contract source code freely available and the number of related transactions on the Ethereum blockchain reached a size which allows a systematic empirical and statistical study.

In this study we analyze some source code features and different Smart Contracts code measures, the evolution of the Solidity language, and other features relating Smart Contract source code to the transactions performed on the Ethereum blockchain. Such an empirical analysis would have been an impossible task just a few months before the time of our study because of the scarcity of Smart Contracts source code available deployed on the blockchain and for the contemporary scarcity of statistics related to the operations and interactions among Smart Contracts and the blockchain.

The purpose of our work is to empirically analyze and characterize the interaction between Smart Contracts and blockchain, in terms of software measures, of EVM compiler version, of developers practices, of Solidity language features and other peculiarities of the blockchain environment and to examine the main software characteristics of contracts written in solidity as well as their purposes. Furthermore, thanks to the availability of Smart Contracts written and deployed at different times, we analyzed some of the evolutionary features of the Solidity programming language and of the way developers write Smart Contracts.

Our study aims at understanding software features and metrics of Smart Contracts, in order to measure progress and performance during the evolution of the Ethereum blockchain technology in these first years.

[1] data from https://stat.bloxy.info.

To lead our research we performed an empirical study collecting the dataset of all Smart Contracts source codes available from Etherscan.io up to the beginning of 2018. We computed several software metrics on the entire dataset and identified the twenty most used Smart Contracts, in terms of blockchain transactions, representing a reduced set on which we performed a systematic and more detailed analysis, in terms of both functionality and development history. We identified some empirical indicators useful to characterize Smart Contracts from a statistical point of view. By means of these indicators we studied the usage of Smart Contracts in the Ethereum blockchain and their evolution over time.

Results lead us to observe an active developer community that constantly follows the evolution of the language that develops more and more specialized Smart Contracts and improves contracts already developed. In general code measures show that Smart Contracts have a limited number of lines of code which are well commented and that implement specific functionalities.

The remaining of the paper is organized as follows: Section II provides a selection of related work in the field of Smart Contract analysis and metrics applied to specific software categories. Section III provides a description of the Solidity language and of the Ethereum environment. Section IV describes the dataset and the results of the analysis in terms of contract name, compiler version, balance and transactions, and of the measure of source codes, such as the number of line of code, the number of contract declarations and the related size of the bytecode. Section V analyzes twenty Smart Contracts, selected from the dataset with the highest number of transactions. First it provides a description of each contract, then it describes the interaction of the development community in terms of number of versions and of reuse of code. Finally the section reports the results of the code analysis performed by means of volume and complexity code metrics. Section VI discusses the findings of this work, summarizing results and providing some considerations derived from them. Section VII concludes the paper.

II. RELATED WORKS

Research literature on blockchain in general and on Smart Contracts in particular, from a software development perspective is limited to the last few years. The development and the diffusion of ''Solidity'' as programming language for writing Smart Contracts on the Ethereum platform started very recently and the definition and implementation of the language and of its Virtual Machine on Ethereum (EVM) is still ongoing.

In this section we provide an overview of the more recent findings in the field with a glimpse to the specific domain of Smart Contracts programming and related topics already published in software literature.

Only very recently the research on software engineering and computer science paid particular attention to the blockchain technology and its specificities. In 2017, Porru *et al.* [18] underline the need of a new branch

of software engineering, and coined the term *BOSE* (Blockchain-oriented software engineering) to deal with this new technology. In this context, authors highlighted the need of new professional roles, new specialized metrics and new modeling languages in order to ensure security and reliability. They designed possible solutions proposing the directions for future specific steps of the BOSE.

Bartoletti and Pompianu [2] conducted a survey of Smart Contracts by studying their usage, development platforms and design patterns. Furthermore, they categorized the contracts by their application domain in order to understand the best convenient investment.

Tonelli *et al.* [22] analyzed more than 12000 certified Smart Contracts provided by Etherscan, along with Bytecode and ABI. Their results report that metrics are less variable than in traditional software systems because of the domain specificity. Furthermore in Smart Contract software metrics there are no large variations from the mean. All values are generally within a range of few standard deviations from the mean.

In order to define a specific Blockchain Software Engineering, Destefanis *et al.* [9] argue that Smart Contracts have a non-standard software life-cycle and therefore applications can hardly be updated or it is more difficult to release a new version of the software.

Wan *et al.* [23], in order to design efficient tools to detect and prevent bugs within the blockchain, performed an empirical study to understand the blockchain bug characteristics. They investigated the bugs frequency distribution manually examining 1108 bugs in eight open source blockchain.

Bragagnolo *et al.* [3] presented *SmartInspect*, a tool able to debug the code of a Smart Contract, addressing the lack of inspectability of a deployed code. In fact, once a Smart Contract is deployed, data are encoded and the source code cannot be redeployed. Authors proposed a solution by analyzing the contract state through a decompilation techniques and a mirror-based architecture without redeployed it.

Rocha *et al.* [8] implemented a tool to handle Smart Contract written in Solidity language, the solution is specifically designed for Pharo (a live programming environment based on Smalltalk code language).

Norvill *et al.* [17] used *Etherscan.io* in order to explore Smart Contracts and to analyze bytecode level metrics or to identify similarities between compiled pieces of code. They focused their attention on contracts compiled code, source code, and metadata such as the contract name.

The Smart Contracts are the basis for Initial Coin Offerings (ICO), the new means of crowdfunding centered around cryptocurrency in the blockchain development area. In this regard Fenu *et al.* [10] analyzed the quality and the software development management of 1388 ICOs in the 2017. Ibba *et al.* [11] they investigated on the ICO process analyzing a dataset obtained collecting data from specialized websites. They emphasized the advantages which Lean methodologies could lead both to the team organization and to stakeholders involvement.

In general the literature on Smart Contracts software features and in particular on the Solidity programming language is still limited and a comprehensive empirical analysis on a dataset of thousands Smart Contracts source codes and the metrics representing and characterizing their interaction and usage within the Ethereum blockchain has not been performed yet.

III. BACKGROUND

Our analysis takes into account a particular typology of software programs called Smart Contracts, written in a programming language specific for the EVM of the Ethereum blockchain environment, called *solidity*. In this section we provide a brief description of the Ethereum system and of Smart Contracts.

A. THE ETHEREUM SYSTEM

Ethereum is a blockchain with an embedded Turing complete computing machine. Thus computer programs can be uploaded into the blockchain and executed on the nodes implementing the blockchain network on a peer-to-peer computer network. The nodes interact managing transactions which are the core concept for obtaining a correct and validated sequence of blocks recording and holding all the information. Identities are associated to accounts/addresses managed by a public-private key pair. A blockchain address is associated with the pair. The blockchain has associated a criptocurrency (the Ethers) in the network, which is used as an incentive for miners and so that the accounts can hold, send and receive criptovalue. Since the blockchain can perform computation, the account can also contain code, associated to a so called ''smart contract'' by means of the blockchain address which is determined at the time the contract is created. Transactions ensure that every change of state is recorded into the public ledger representing the blocks sequence. As a consequence accounts can be of two kinds: External Accounts, managed by the public-private key pairs, and Contracts Accounts, managed by the stored code. The Ethereum Virtual Machine (EVM) deals with the two kinds of account in the same way. Interactions among the parties are allowed by means of transactions, made by messages sent from one account to another and containing binary data (the so called ''payload''), and a certain amount of cryptovalue (Ethers). Transactions can be activated by the public-private key pair, sending a request in broadcast to the network nodes, or by Smart Contracts, within the same scheme, by executing the code stored in them. This working scheme describes the *interactions* affecting Smart Contracts within the blockchain analyzed in the present work.

If the account receiving the message is a Smart Contract then it executes the code with the payload as input data. Transactions can also create new contracts by means the operation called Smart Contract deployment, represented in Fig. 1, where the compiled code is passed in the payload of the transaction and permanently stored in the blockchain,

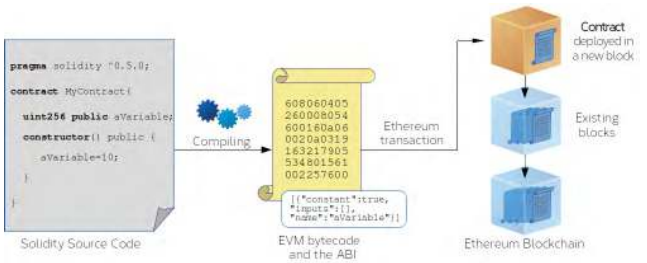

FIGURE 1. Deployment of an ethereum smart Contract in the blockchain.

Transactions require ''Gas'' consumption, to avoid infinite amount of computational work to be executed and Gas is payed in Ethers.

Other interactions may occur by Message Calls and Delegate Calls. These are messages sent by Smart Contracts and have a source, a target, a payload, an amount of Ethers, an amount of gas and return data. In Delegate Calls the code at the target address is executed in the context of the calling contract.

B. SOLIDITY SMART CONTRACTS

A Smart Contract is a computer program that aims to implement a logical sequence of steps according to some clauses and rules. In a conceptual level, Smart Contracts consist of three parts [21]:

- the code of a program that becomes the expression of a contractual logic;
- the set of messages which the program can receive, and which represent the events that activate the contract;
- the set of methods which activate the reactions foreseen by the contractual logic.

Smart Contracts run in a blockchain where contract transactions can be permanently recorded in a transparent environment and are immutable. Once the Smart Contract is deployed into the blockchain its code cannot be modified and the clauses introduced by the parties in the contract will obligatorily be respected because of the computational nature of the system, as for the execution of any software program.

There are different blockchains able to run programs implementing Smart Contracts. Even the Bitcoin blockchain supports a limited amount of software code that can be deployed using transaction in a blockchain address [1]. Other examples are Hyperledger Fabric [5], the Qtum platform [7] and the Achain platform [6].

Among all, the most popular is the Ethereum platform, the first blockchain specifically conceived to run Smart Contracts. The most popular programming language for Smart Contracts in Ethereum is ''Solidity''. In this platform it is possible to read some information that characterize each Ethereum transaction. In particular, Smart Contracts are activated by messages, that are Ethereum transactions executed by the message sender. Currently the Ethereum platform hosts the large majority of Smart Contracts.

As represented in Fig. 1, the process to deploy a Smart Contract into the Ethereum blockchain is composed by three phases. The first phase consists in the code writing in Solidity language; the second consists in the code compiling, that can be executed in a local environment (i.e. the remix environment[2]) to convert the script in the EVM *bytecode* [19]; and finally the last phase consists in creating a transaction in the blockchain, that actually deploys the contract. At the moment of the deployment, the blockchain assigns an address to the Smart Contract. Accessing to that address it is possible to visualize some data of the Smart Contract like its address, its balance, and its Application Binary Interface (ABI).

In order to avoid the possibility of EVM overload, the execution of Smart Contract functions (when they involve changes to blockchain records) lead to a cost in terms of cryptocurrency. In particular, to each low level operation is associated a computational cost (defined in units of *Gas*) [24]. The price in Ether of a unit of Gas is not fixed but follows the free market rules.

Solidity is a *contract-oriented*, high-level language whose definition was influenced by Object Oriented (OO) languages like Python, C++, and especially by JavaScript.

It is a typed programming language and supports traditional types such as integer, string, array, as well as structures, associative arrays, and enumerations.

Moreover Solidity has a specific type, the *address*, that identifies users and other contracts. Each contract variable can be interpreted as a record of a database which can be queried and modified by calling functions of the code that manages the database. The set of variables and their associated values represent the state of the contract. Smart Contracts functions can be externally called by means of blockchain transactions. In order to make the development more modular, specific function *modifiers* can be defined and associated to different functions, for instance to perform checks in a declarative way.

Recently different mainstream integrated development environments (IDE's) appeared for supporting solidity code development, as for example IntelliJ IDEA, developed by JetBrains and Visual Studio Code, developed by Microsoft. We used the Intellij-Solidity plugin[3] to read and compare contracts source codes.

On the contrary there is still a lack of specific tools for analyzing Solidity source code metrics, so that we recurred to the similarity of Solidity with Javascript and C++ for the analysis of Solitidy source codes metrics. In fact, an exploratory evaluation of the features of Smart Contracts source codes can be performed using metrics and methodologies obtained adapting existing tools and designed for similar languages.

IV. ANALYSIS OF THE SMART CONTRACTS DATASET

We performed an empirical study on 10174 Smart Contracts, deployed in the Ethereum blockchain and validated using the Etherscan validation service. Our dataset includes all Smart

[2]Available online at `https://remix.ethereum.org/`
[3]https://plugins.jetbrains.com/plugin/9475-intellij-solidity

Contracts uploaded until the beginning of 2018. The analysis considers two information sets at different levels.

The first set characterizes the contract with respect to the blockchain environment and to the interactions with it. It is a set of parameters associated to, and defining the contract state, which can be time varying. It consists of a list containing descriptive information of each Smart Contract. In particular, it contains the Ethereum address, the contract name, the number of transactions performed up to data, the compiler version and the balance of each Smart Contract verified in Etherscan. We extracted all the information from both the source code and by browsing the Ethereum blockchain transactions related to each contract, starting from the list of verified Smart Contract source codes provided by etherscan.io.[4]

The second set characterizes software code, is fixed, and can be viewed as independent from the blockchain environment. It consists of a collection of 10174 ''.sol'' files containing the contracts source code as extracted from the Etherscan website. In fact, Etherscan provides a descriptive page for each contract as well as the source code in separated frames. We extracted the source code from the contract page implementing an R script. Given a contract address, the script loads the html of the contract page, recognizes the start and the end of the source code, extracts and saves it in plain text. The size of the source codes dataset is about 100 MB.

Our empirical study first examines the two sets independently, then compares the information collected on both.

We first analyzed the parameters that characterize the Smart Contracts in the blockchain, aiming to provide statistical information of features like the name usage, the compiler version, the number of transactions, and the balance of contracts.

In the second part we characterized Smart Contracts source codes, also by means of a statistical analysis. In particular, we computed a set of code metrics for each Smart Contract in the dataset and present the statistics characterizing the entire dataset.

A. SMART CONTRACTS PARAMETERS: ANALYSIS

We evaluated the main parameters and metadata that describe every Smart Contract in our dataset. Specifically, we focused our attention on the *contract name*, the *compiler version*, the *number of transactions*, and the *contract balance* in Ether.

We chose to analyze the list of contract names in order to evaluate if the ethereum developers community uses specific names for specific functionalities or whether the contract name does not have particular meaning, since the contract name is the analogous of the Class-Name in OOP.

The analysis of the compiler versions allow us to understand if developers follow the continuous updating of the language specifics, released in order to fix bugs and to provide new and optimized functionalities.

[4]List available at https://etherscan.io/contractsVerified

The contract balances and the number of transactions are two series of values characterizing contracts in terms of usage, popularity, and in terms of funds inserted into 'that' account. We obtained both a snapshot of the interaction of each contract in the blockchain and an overall statistics on their values. The number of transactions is the total number of transaction that a contract receives and sends from *normal* accounts (owned by users). This number does not include transactions sent between contracts (called internal transactions).

All these data are public available for each Smart Contract deployed in the Ethereum blockchain and verified by Etherscan.

1) CONTRACT NAME

In the Etherscan platform, Smart Contracts are characterized by a Contract Name. According to Etherscan specification, the Contract Name must match the *ContractName* in the source code that is deployed into the blockchain. See for instance the contract Crowdsale in Appendix A or the contract KittyCore in appendix C.

So we refer to *Contract Name* either as the name in Etherscan which identifies the solidity file containing the source code or to the keyword inside the solidity file where, for each file, there may be different contracts. In facts, according to the language syntax, the keyword *contract* substitutes the keyword *class*, but a contract has features similar to a class. For example a contract can be represented as a structure that includes a set of variables and a set of functions (these can be public or private). But the similarity is far to be complete: class code can be called from other classes in OOP and methods can be called using methods and class names. Classes can be statically coupled when a class resources to code of another class in the system. Class names are also chosen according to good programming practices where the name reflects also class functionalities and purpose (eg. the ''rectangle'' class, the ''point'' class). On the contrary, some of these features are lost in Solidity Smart Contracts and so does the semantic of the name. The contract name looses any ''architectural compiling design'' meaning and its methods or functions, its functionalities, are called by mean of blockchain transactions.

As a consequence different Smart Contracts may hold the same name and contain completely different code, or two different Smart Contracts can be two slightly different versions of a same contract, or they may be the very same contract deployed many times for testing purposes, or again they can consist in part of code existing in one project and reused in another (eg the ''token'' contract, ERC20 compliant contracts) and so on. So it is of particular interest the analysis of contract name occurrences to understand how Solidity developers apply standard naming practices.

In our study we analyzed the collection of Contract Names in our dataset and we found that among the 10174 contracts (belonging to 10174 different addresses, only 6205 names differ.

TABLE 1. The 10 most used contract names.

Contract Name	Number
Crowdsale	213
Token	143
ERC20Token	138
ApprovedTokenDone	82
Presale	81
MyToken	66
TokenERC20	63
PreICOProxyBuyer	60
CrowdsaleToken	56
MultiSigWallet	54

More specifically, we found that:

- 4980 Smart Contracts have a unique Contract Name and are deployed only one time in the blockchain: there is no other address that holds a contract with the same name. Therefore there is no ambiguity, the contract is identified by the name.
- 1225 Contract Names are used more than once (from 2 to 213 times). So that there are very popular names where different blockchain addresses register many contracts with identical names, but also the same contract (with the same solidity code) multiple times.

Tab. 1 reports the list of the ten most used contract names and shows that some contract names (eg. *crowdsale*, *token*, *ECR20Token*) occur more than one hundred times.

The occurrence of the same contract name multiple times is due to at least three possibilities: contracts codes are identical and the very same contract is used many times in different accounts; contracts codes are similar for functionalities and code metrics, but the codes differ slightly, so these are a modification or an adaptation of the other; contracts are completely different in code and metrics and they only share the same name, because semantic has still a limited role in Smart Contracts software development.

A typical example of contracts sharing common names are contracts associated to ICOs [10]. The contract ''Crowdsale'', (see Appendix A) belongs to this category, since its code manages token crowdsales with different purposes and may be easily reused in different ICOs.

In general Smart Contracts with the same contract name, although belonging to different projects, have very similar functionalities and metrics.

Among the 213 Smart Contracts called *Crowdsale*, we found that six source codes are deployed at least twice. One of these codes[5] has 4 duplicates. This is a Smart Contract with the same bytecode and identical metrics that were subsequently memorized in the blockchain in four different addresses.

2) COMPILER VERSION

According to [8] any Smart Contract written in Solidity has a grammar that starts with the SourceUnit rule which contains instances of a pragma directive that declares the source file compiler version. It starts with the keyword ''pragma'' followed by an identifier, and then any combination of one or more characters until a semicolon terminates the row (see Fig. 2).

```
1 pragma solidity ^0.4.18;

3 contract Simple {
    ...
5   }
  ...
```

FIGURE 2. Example of definition of the pragma version. In the first row is specified that in the following will be used the version 0.4.18 of solidity.

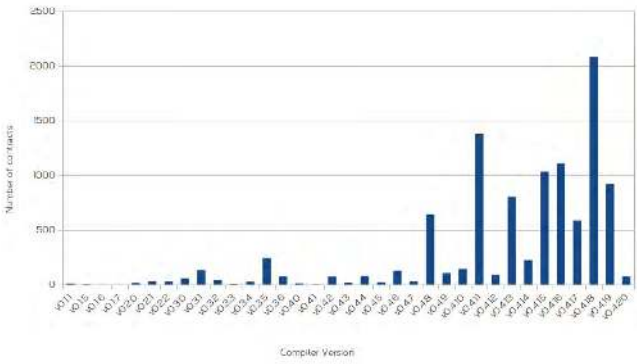

FIGURE 3. Histogram of the number of verified contracts per compiler version.

This declaration ensures that the contract does not suddenly behave differently with a new compiler version. In our dataset, the latest version of the compiler is the v0.4.20 and the most used version is the v0.4.18.

In fact Solidity is fast evolving and new features or functionalities of the language are introduced from time to time, rendering unstable the behavior of the code under different versions. Versions may be updated when a bug is discovered or new language constructs are needed and so on.

Fig. 3 reports the histogram of the number of verified contracts per compiler version. There are some specific cases that we consider useful to mention for our analysis. The only Smart Contract with compiler v0.1.6 is developed by Piper Merriam, the creator of Ethereum Alarm Clock (ECM) that allows users to schedule a contract call for a specified future block.[6]

There is only one contract[7] that uses the version v0.1.7. It is a Smart Contract developed by Gavin Wood, one of the Ethereum founders and the inventor of Solidity. V0.1.6 and v0.1.7 have been introduced in October and November 2015 respectively. Five versions have the first transaction verified on 24.03.2016, when the Etherscan service was launched.

In order to understand how fast the developers acknowledge the updating of the language, we collected the date of release of the documentation (generally available on Github) related to a new version of the pragma and compared it with

[5]See for instance the source code of the address 0xa1877c745628 21ff59ffc0bc999e6a2e164f4d87

[6]with address 0x07307d0b136a79bac718f43388aed706389c4588

[7]with address 0xbF35fAA9C265bAf50C9CFF8c389C363B05753275 and contract name *wallet*

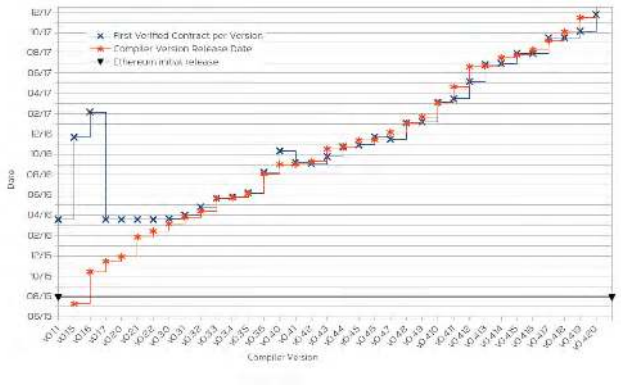

FIGURE 4. Date of release of compiler versions and the date of the first contract activation per compiler version.

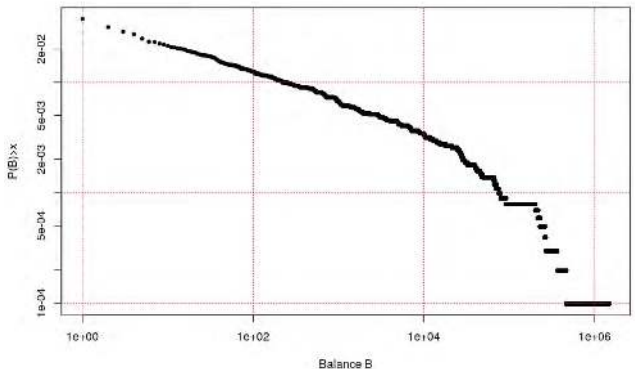

FIGURE 5. CCDF of the balance in Ethereum per contract.

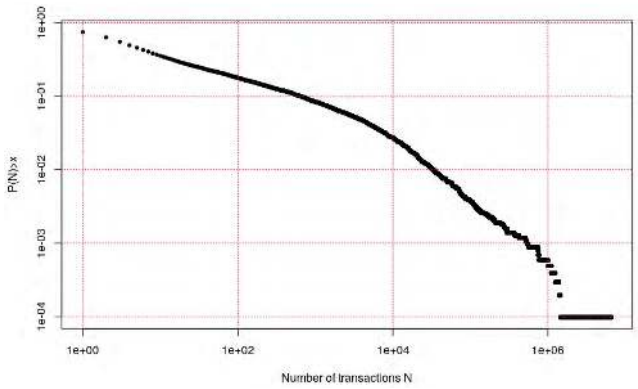

FIGURE 6. CCDF of the number of transactions per contract.

the date of the first transaction that involves a contract with the same version of pragma. In most cases, given a compiler version, the first transaction related to a Smart Contract presenting that pragma (or the first activation) has been executed the same day or a few days after the release of that version in Github (23 out of 34).

In the remaining cases, on the contrary, the documentation of the pragma version has been made available after the date of the first usage.

Figure 4 shows the history of compiler versions and the dates of the releases of new versions of the compiler (red dots) and of the first transaction to a Smart Contract characterized by the same compiler version. The Fig. 3 shows a net growth reflecting the growth in use of Smart Contracts in 2016 and 2017.

3) BALANCES AND TRANSACTIONS

Focusing on the Smart Contract balance, we found that a very few Smart Contracts collect the majority of the total balance of all Smart Contracts. In fact, the total balance of the 10175 Smart Contracts is about 4.64 millions Ether, but 80% of the total balance belongs to 10 Smart Contract alone, namely to less then 0.1% of the contracts accounts. In general Smart Contracts do not collect Ethers, except in the case they are *wallets*. A wallet is a Smart Contract realized to securely collect Ethers and could implements some functions such as the ''multiple ownership'' or the ''escrow''. Tab. 2 summarizes the information about these contracts.

Considering contract names in this table, most of them can be recognized as wallets. In order to investigate on the distribution of the wealth, we represented in Fig. 5 the distribution of the balance of the contracts in our dataset. The figure shows the Complementary Cumulative Distribution Function (CCDF) of the balance. The plot is in log-log scale and axes tags are in normal scale. The figure suggests a power-law distribution of wealth among the contracts so that most of the total wealth is held by a small fraction of contracts and conversely most of the contracts hold a very small balance.

Fig. 6 shows the CCDF of the total number of Smart Contracts transactions. Also this second distribution follows a power-law-like behavior until the values around 10^4 transactions.

Given the similarity of the two distributions, we computed the correlation among the two datasets. The resulting correlation coefficient is 0.026 stating that there is no correlation between the number of transaction of a Smart Contract and its balance. Despite the two distributions display similar features and show a tail, there is no simple general relationship among Smart Contract balance and number of transactions. In facts, as reported in Tab. 2, Smart Contracts with high balance may use a low number of transactions and vice-versa.

B. MEASURES ON SMART CONTRACTS SOURCE CODES

In this paragraph we describe the analysis performed on the contracts source codes, discuss the parameters under investigation and provide the results of the source code analysis. In order to analyze the contracts source code, we computed the values of the following code metrics, that can be divided in two groups. The first group represents the **Volume metrics**. The second group includes **Contract oriented metrics** which describe the logical size of the source code.

1) VOLUME METRICS

M1, *Lines of Code* (LoC) is the number of line of code excluding comments and blank lines. For comparison, we computed

TABLE 2. Smart Contract balance.

Address	Balance	ContractName	TxCount	% Tot balance
0xab7c74abC0C4d48d1bdad5DCB26153FC8780f83E	1500000,00	Wallet	243	32,33%
0xf0160428a8552ac9bb7e050d90eeade4ddd52843	466648,15	TokenSales	3512	10,06%
0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9	369023,14	MultiSigWallet	226	7,95%
0xa646e29877d52b9e2de457eca09c724ff16d0a2b	269419,35	MultiSigWallet	37	5,81%
0xcafe1a77e84698c83ca8931f54a755176ef75f2c	263522,66	MultiSigWallet	335	5,68%
0xbf4ed7b27f1d666546e30d74d50d173d20bca754	232267,28	WithdrawDAO	18771	5,01%
0x851b7f3ab81bd8df354f0d7640efcd7288553419	218474,70	MultiSigWalletWithDailyLimit	72	4,71%
0xB62EF4c58F3997424B0CCeaB28811633201706Bc	203467,99	Fundraiser	75	4,39%
0x16a0772b17ae004e6645e0e95bf50ad69498a34e	91780,96	MultiSigWallet	90	1,98%
0xa4dDd3977920796BFb14cA8d0FB97491fA72a11d	79431,47	RefundVault	26	1,71%

TABLE 3. Statistics on code metrics computed among 10174 contract source codes.

	Lines	M1: LoC	M2: CpL	M3: NDC	M4: NDF	Bytecode
Average	321.81	180.01	0.48	4.39	5.30	9448.99
Variance	110453.33	35452.37	0.19	13.92	49.33	47139522.84
Standard Dev.	332.35	188.29	0.44	3.73	7.02	6865.82
Median	206.00	117.00	0.42	3.00	3	7967.00
Min	0.00	0.00	0.00	0.00	0	57.00
Max	4240.00	2294.00	10.07	36.00	125.00	50607.00

```
contract Base {
  //...
}
contract Derived is Base() {
  //..
}
```

FIGURE 7. Contracts declaration in solidity.

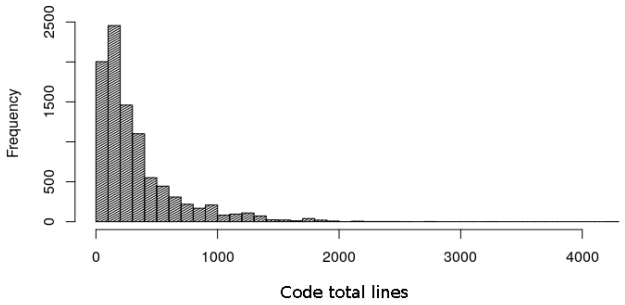

FIGURE 8. Histogram of the number of lines of code per source code.

also the total number of code lines (including blanks and comments).

M2, *Comments per line* (CpL) is the ratio between lines of comment and lines of code.

2) CONTRACT ORIENTED METRICS

M3, *Number of Declared Contract* (NDC) is the number of contracts (the equivalent of classes in OO languages) declared in the source code. In solidity the declaration of a contract type is defined with the keyword *contract*. A contract can inherit from other contracts declared in the source code and can instantiate contracts, as described in Appendix A. Fig.7 shows the declaration of two contracts. The contract *Derived* inherits functions and variables from the contract *Base*. A solidity source code can contain several contract declarations and a contract implementation can use or inherit the other contracts in the source code. The deployment of a Smart Contract involves one contract definition at time.

M4, *Number of Declared Functions* (NDF) is the number of functions declared in the source code.

Furthermore, we measure the size of the bytecode of each contract. The bytecode is the result of the compiling operation and its length depends on the content of the source code, on the version of the compiler and on the compiling optimizations.

Table 3 reports different statistics: the averages, variances, standard deviations, medians, minima, and maxima values for each metric.

The table shows that all the metrics display features typical of a tail distribution. They have high dispersion around the mean, with values of standard deviation comparable or even higher than the median. Such phenomenon is typically related to the presence of statistical units with very large values of the metric which contribute to rise the value of the average with respect to the median. The maximum values are an order of magnitude larger than the average, indicating the presence of outliers. The shortest bytecode has a length of 57 bytes. Considering the maximum values, the longest source code has a length 10 times longer that the average value in the dataset. The same can be said for metrics M1 (LoC) and M3 (NDC). The largest Bytecode is about five times the average. Max values of M2 (CpL) and M4 (NDF) are much higher than the average value.

In order to represent the distribution of metrics values in the dataset we plot the histograms for the numbers of lines of code, of the number of contract declarations per file (NDC) and of the size of the bytecode. Fig. 8 shows the histogram of the number of lines of code. Each bin is large one hundred units. The mode of the distribution is between 100 and 200 lines of code.

TABLE 4. Matrix of the cross-correlation coefficients between metrics and indicators computed among 10174 contracts.

	Line Total	Bytecode	M1: LoC	M2: CpL	M4: NDF	Tx Count
Line Total	1	0,52	0,95	0,22	0,91	0,02
Bytecode	0,52	1	0,56	0,03	0,52	0,02
LoC	0,95	0,56	1	0,02	0,91	0,02
CpL	0,22	0,03	0,02	1	0,12	-0,01
NDF	0,91	0,52	0,91	0,12	1	0,02
Tx Count	0,02	0,02	0,02	-0,01	0,02	1

FIGURE 9. Histogram of the number of contract declaration per source code.

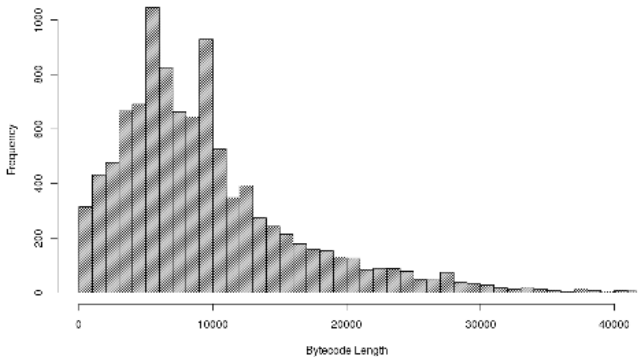

FIGURE 10. Histogram of the length of bytecodes in byte.

Fig. 9 shows the number of occurrences of the discrete values of the NDC, i.e the number of contract declarations per source code file. In this case the bin size is set to one. The mode of the number of contract declarations per file is 1 since source codes with more than 15 contract declarations are rare. These two graphs show a fast decreasing of values, characterized by a long tail.

Fig. 10 provides the histogram of the size of contracts bytecodes. Each bin is large 1000 bytes. This graph presents a normal-like distribution. The mode is between 6000 and 7000 byte.

In order to investigate if and how code metrics influence each other, we computed the cross correlation matrix. Tab 4 reports the results of the cross correlation coefficients between code metrics, including the length of the bytecode and the number of transactions of each contract, that will be discussed later. The highest correlation coefficient, which is trivial, is between the metric M1 (LoC) and the total number of lines. Also the M4 (NDF) has a high correlation coefficient with the LoC and the total number of lines. The M2 (CpL) is not correlated with the length of the code or with the M4 (NDF). This means that the number of comments on the code is heterogeneous and, in general, not proportional to the length of the source code.

The length of the bytecode is only moderately correlated both with the code length and with the number of declared functions. In addition, the number of transaction that involve a Smart Contract is not correlated with any code metric. This means that, for instance, highly used Smart Contracts have very different source codes lengths. In the following we will confirm this results with a further analysis.

V. DETAILED ANALYSIS OF THE TOP 20 USED SMART CONTRACTS

In this section, we present a detailed analysis of the twenty Smart Contracts with the largest number of transactions (Tx count). Tab. 5 lists these contracts. That can be classified according to their typology [2] in five categories: Wallet, Financial, Game, Library, and Notary. Wallet contracts are characterized to be deposits of ether and they usually have a high balance. Financial contracts aim to provide functions useful to manage financial goods such as tokens. Game contracts implement lotteries and digital collections. Library contracts are developed and deployed to provide functionality useful for other contracts (i.e maths libraries). Finally, Notary contracts take advantage on the blockchain characteristics to record agreements between parts.

In the following we provide a short description for each of the 20 most used Smart Contracts.

A. SMART CONTRACTS DESCRIPTION

1) ETHERDELTA

It is tagged as etherdelta_2 on Etherscan and is the Smart Contract executed to store and transfer tokens with Ethereum wallets, in the cryptocurrency exchange EtherDelta.[8] EtherDelta is in fact one of the most used decentralized trading platform for Ethereum and manages ERC20 compatible tokens. In order to trade on EtherDelta a user must create a wallet or use an existing wallet which interacts with this Smart Contract.

[8]https://etherdelta.com

TABLE 5. List of the twenty smart contracts under examination.

#	Address	Tx count	Contract name	Project	Typology	Standard
1	0x8d12a197cb00d4747a1fe03395095ce2a5cc6819	6562254	Etherdelta	Ether Delta	Wallet	
2	0x03df4c372a29376d2c8df33a1b5f001cd8d68b0e	1450979	Bitocinereum	Bitcoinereum	Finantial	ERC-20
3	0x06012c8cf97bead5deae237070f9587f8e7a266d	1390301	KittyCore	CryptoKitties	Game	ERC-721
4	0xE94b04a0FeD112f3664e45adb2B8915693dD5FF3	1241476	ReplaySafeSplit	BitTrex	Library	
5	0x6090a6e47849629b7245dfa1ca21d94cd15878ef	1103826	Registrar	Ethereum Name Service	Notary	
6	0x86fa049857e0209aa7d9e616f7eb3b3b78ecfdb0	1005345	Dstoken	EOS	Financial	ERC-20
7	0xa3c1e324ca1ce40db73ed6026c4a177f099b5770	768834	Controller	BitTrex	Library	
8	0xd26114cd6ee289accf82350c8d8487fedb8a0c07	750295	OMGToken	OmiseGO	Financial	ERC-20
9	0xf230b790e05390fc8295f4d3f60332c93bed42e2	742060	TronToken	TRON	Financial	ERC-20
10	0x93e682107d1e9defb0b5ee701c71707a4b2e46bc	557685	MCAP	MCAP Labs	Financial	ERC-20
11	0xa74476443119a942de498590fe1f2454d7d4ac0d	535935	GolemNetworkToken	Golem Network	Financial	ERC-20
12	0xb1690c08e213a35ed9bab7b318de14420fb57d8c	514167	SaleClockAuction	CryptoKitties	Game	
13	0xaBbb6bEbFA05aA13e908EaA492Bd7a8343760477	421810	ReplaySafeSplit		Library	
14	0xd0a6e6c54dbc68db5db3a091b171a77407ff7ccf	370266	EOSSale	EOS	Financial	ERC-20
15	0x744d70fdbe2ba4cf95131626614a1763df805b9e	296699	SNT	Status Network	Financial	ERC-20
16	0x9a642d6b3368ddc662CA244bAdf32cDA716005BC	296623	HumanStandardToken	QTUM	Financial	ERC-20
17	0xb97048628db6b661d4c2aa833e95dbe1a905b280	279107	PayToken	TenXPay	Financial	ERC-20
18	0xece701c76bd00d1c3f96410a0c69ea8dfcf5f34e	269043	Etheroll	Dice	Game	ERC-20
19	0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	253559	ReplaySafeSplit		Library	
20	0x0D8775F648430679A709E98d2b0Cb6250d2887EF	213364	BAToken	Brave Browser	Financial	ERC-20

Bitcoinereum[9] is the fist Bitcoin-like mineable Ethereum ERC20 Token and, through the Bitcoin Supply mechanism, enables a bitcoin-like currency to run on the ethereum blockchain. To bring the Bitcoin supply mechanism into Ethereum, Bitcoins enter the Ethereum blockchain in form of ERC20 tokens.

KittyCore and **SaleClockAuction** are two Smart Contracts belonging to one of the most popular applications of Ethereum blockchain, CryptoKitties, the game in which users can buy, sell, and breed cartoon kittens. The application was launched on November 28th 2017, and in a little more than a month these two contracts (out of a total of 17 Smart Contracts developed in this project) have been responsible for the 6,2% of all transactions on the ethereum network.

2) REPLAYSAFESPLIT

In the set of 20 top used Smart Contracts, the contract name *ReplaySafeSplit* appears three times. The functionality of these three Smart Contracts are very similar: they are used to split Ether funds in several addresses and protect against replay attacks between Ethereum Classic (ETC) and Ethereum (ETH). As a result of the hard fork of the Ethereum network (on July 20th, 2016), holders of an ETH fund prior to the 1920000 block ended up with two funds on the same address and therefore found themselves having ETH and ETC in equal quantities: the ETHs on the *support-dao-fork* network and ETC on the *oppose-dao-fork* network. The two coins are still linked to each other: a move of ETHs moves also ETC and vice versa. ReplaySafeSplit is used to separate ETH pre-forks on two new and different addresses, one specific for ETH post-fork and another one specific for ETC. ReplaySafeSplit recalls the fork oracle Smart Contract.[10] A specific version (labeled Bittrex_2[11] on Etherscan) is used on the Digital Currency Exchange Bittrex (https://bittrex.com/) with the same capabilities.

3) REGISTRAR

It is one of two Smart Contracts that compose the core of the Ethereum Name Service[12](ENS), an extensible naming system based on the Ethereum blockchain. *Registrar* owns a domain and, according to the rules written in the contract, issues subdomains of that domain to users. For each domain and subdomain Registrar memorizes the owner (an external account, typically a user or another Smart Contract), the resolver and the time-to-live for all records.

DSToken (labeled EOSTokenContract) and **EOSSale** (labeled EOSCrowdsale) are Smart Contracts of the famous Infrastructure for Decentralized Applications EOS[13] that introduces a blockchain architecture designed to allow the vertical and horizontal scaling of decentralized applications. EOSTokenContract is in fact the token of the EOS ICO that aims to finance *block.one*, the platform that, based on scalability, flexibility and usability criteria, intends to make the blockchain technology accessible to businesses which, in this way, can memorize Smart Contracts on blockchain. EOS tokens are ERC-20 compatible tokens distributed on the Ethereum blockchain under a related ERC-20 Smart Contract. EOSTokenContract handles all the logic of ownership and transfers; Instead, EOSCrowdsale manages all the logic of contributions, periods and claiming.

4) CONTROLLER

It is one of the two Smart Contracts that implements the core of Bittrex (the other one is ReplaySafeSplit, as previously described) and manages the exchange of cryptocurrency. The main function of Controller is MakeWallet that is used to create ETH wallets and has control functions of owner and destination.

[9] http://www.bitcoinereum.com/

[10] Having address *0x2bd2326c993dfaef84f696526064ff22eba5b362*

[11] Having address *0xE94b04a0FeD112f3664e45adb2B8915693dD5FF3*

[12] https://ens.domains/

[13] https://eos.io/

OMGToken (labeled OmiseGoToken). It is the token of OmiseGO (OMG),[14] currently one of the most famous cryptocurrencies of the ICO market which aims to simplify and make cryptocurrency transactions almost instantaneous. OMG is a public Ethereum-based financial technology for use in mainstream digital wallets. At the same time it is an e-wallet and payment platform acting through assets and crypotocurrencies. The advisors of OMG are almost all from the Ethereum foundation. OMGToken is an ERC20 basic token on Ethereum. Once the OMG blockchain is created, the OMG tokens are transferred to this new blockchain.

TronToken (TRX) is the token of the TRON ecosystem.[15] It is the blockchain-based decentralized protocol and open-source platform that aims to construct a global free content entertainment system and provides functions of credit sharing and payment for many services such as online casinos, mobile games, live shows, social networks. It is based on an ICO and is a ERC20 standard Ethereum token. Starting from December 2017 it is the second most used token with market capitalization that rose from $477 million to $3 billion just within 5 days (from December 13 to December 18).

MCAP[16] uses the ERC 20 protocol for peer-to-peer transactions and is the token of MCAP Labs ecosystem. Its ICO was launched by BitcoinGrowthFund (BGF) with the aim to invest in the mining of cryptocurrencies, especially Bitcoin. The algorithms developed by BGF identifies which cryptocurrency must be mined at any time to maximize profit. The Smart Contract has five functions: *mcap* to initialize contracts with initial supply tokens to the creator of the contract; *transfert* that sends coins; *approve* which allows another contract to spend some tokens in the owner behalf; *approveAndCall* that in a single transaction approves and communicates the approved contract and finally *transfer*, called from a contract that attempts to obtain the coins.

5) GOLEM NETWORK TOKEN (GNT)

It is the token of the Golem Network project,[17] a decentralized distributed network of computers in which users can sell and buy computing power. Through Golem Network users can decentralize all the tasks thanks to the computer of another user connected to the network, or sell the computing power of their own computer to help those who need it. The GNT Token is partially-ERC20-compliant because it does not implement the *approve*, *allowance*, and *transferFrom* functions, and the *Approval* event. On the Ethereum blockchain, the crowdfunding start block is 2607800 and it was launched in the 11th November 2016. The main functions of this Smart Contract are: management of payments for resource usage and remuneration for software developers; submitting of deposits by providers and software developers and participation in the process of software validation and certification.

SNT (labeled StatusTokenContract) is the token of Status Network,[18] an open source messaging platform that includes a DApps browser, a messenger, a wallet, and can be described as a mobile operating system to access Ethereum from anywhere. It is therefore a peer-to-peer messaging app without central server to store private data or conversation. Status Network aims, through the use of blockchain technology, to remove centralized third-party applications or middlemen in the people communications. The entire project combines 10 Smart Contracts. SNT is a ERC20-compliant token and derives from the MiniMe Token[19] that allows for token cloning (forking). SNT has a modular architecture and is used to power the Status Client, including some fundamental utilities such as a Decentralized Push Notification Market, the Governance of the Status client, Username Registration using ENS, and so on.

HumanStandardToken (labeled QtumTokenContract) is the token of the Qtum project,[20] a Value Transfer Protocol (VTP) blockchain. Qtum is therefore a Smart Contract ecosystem for businesses that want to run decentalized apps blockchain-based, executable on mobile devices. The aims is to turn any human-readable agreements into a Smart Contract. Qtum uses Bitcoin's UTXO model in order to allow the contact execution also on mobile devices.

HumanStandardToken is a ERC20-compliant and includes 3 contracts called *Token*, *StandardToken* and *HumanStandardToken*. The contract Token modifies ERC20 base standard in the totalSupply function because a getter function for the totalSupply is automatically created.

PayToken (labeled TenXContract) is the token of the TenXPay (TENX)[21] project that aims to solve one of the major problems of the cryptocurrency market: how you spend cryptocurrencies in the real world. It is a portfolio-bank based on cryptographic assets with a debit card. With an encryption-protected off-line multi-asset instant transaction network, the service supports unlimited cryptographic assets (initially only supports ETH, ERC20, DASH and BTC). Users can choose which cryptographic asset to use for payment by debit card and ATM withdrawals. The contract calls a function named MakeWallet. PayToken is a ERC20-compliant token. Users can store PayToken in any ERC20-enabled wallet.

Etheroll (labeled *Etheroll_old_3*) is a Smart Contract of the Ethereum Dice game project and is used to place bets on dice games using Ethers with no deposits or sign-ups. The dice rolls are random and cryptographied in a secure way, thanks to the Ethereum blockchain. In order to obtain the final results of dices, the Etheroll smart-contract invokes the API of Random.org,[22] performs sha3() encryption on its result and on IPFS address of the TLSNotary proof. In the

[14] https://omisego.network/
[15] https://tronlab.com/en.html
[16] https://bitcoingrowthfund.com/mcap)
[17] https://golem.network/
[18] https://blog.status.im
[19] https://github.com/Giveth/minime
[20] https://qtum.org/
[21] https://www.tenx.tech/
[22] https://api.random.org/json-rpc/1/invoke

following we will provide more detailed information of this Smart Contract.

BAToken (labeled BatTokenContract - BAT) is the token of the new Brave browser, created by Brendan Eich, creator of Javascript and cofounder of Mozilla. Users are paid in digital currency to view advertising or to click on the advertising banners. BAT is ERC20-compliant.

Most of the smart contracts listed in Tab.5 are financial contracts, and the description highlight the economic interest behind the contract. We found that the several of the described projects makes use of an ICO to fund, and consequently promote, the business idea. These projects are Etherdelta, EOS, OmiseGo, TRON, MCAP, Golem, Status, Qtum, TENX, Etheroll, and Brave Browser. One of the success factors of an ICO is the team size and its composition [10], [11]. So, projects which resort to an ICO are more likely supported by a convincing and well-formed development team.

B. SMART CONTRACTS USAGE INDICATORS

In this section we define empirical indicators useful to describe Smart Contracts usage from a statistical point of view. We identified various *usage indicators* characterizing how and to which extent Smart Contracts code is called or used in the applications of the Ethereum blockchain. We divided the usage indicators in two groups. A first group, characterizing ***blockchain interaction***, describes the occurrences in the blockchain of contract-related operations. It contains the following indicators.

I1, *Number of transactions* (Tx Count): the overall number of transactions (both in input and in output) involving the contract.

I2, *Transactions per day* (Tx/day): the number of transactions normalized with respect to the days of activity (DoA namely the elapsed time in days between the contract creation and its last transaction).

The selected indicators can be easily extracted from the blockchain data and offer a snapshot of the impact that the contract had on the blockchain.

A second group, *developers' interaction*, includes indicators describing the evolution of a contract in terms of its development history and of its reuse to create new contracts. It contains the following indicators.

I3, *Number of Deployments*: counts the total number of contract versions deployed in the Ethereum blockchain and verified using the etherscan service (consider that each deploy involves a cost in Ether). We compared this indicator with the total number of contracts having the same name.

I4, *Number of versions*: counts the number of versions of a Smart Contract which are used within the same project. This indicator consider only versions of the contract that have been active in a certain period of time and it does not count contracts with a low number of transactions (less than 100). It indicates a continuous activity of development.

I5, *Number of code reuse*: counts the number of new contracts created reusing another Smart Contract source code belonging to a different project. As the previous indicator, we excluded from this analysis contracts having a low number of transactions (less than 100).

We also take into account the balance of the Smart Contracts (i.e. the amount in Ether associated to the contract address), but we don't consider it as a good usage indicator because it increases and decreases over time, and, furthermore, only few contracts are used as a deposit of Ether (see subsection IV-A.3).

Tab 6 reports the values of the usage indicators for the twenty Smart Contracts analyzed together with the compiler version. Results show that these contracts are involved every day in a large number of transactions and have a null balance in most of the cases. On the other hand, the indicator value describe the heterogeneity in the usage of these Smart Contracts in terms developers' interactions.

1) BLOCKCHAIN INTERACTION

The twenty Smart Contracts chosen have the highest value of I1 (TxCount), namely the total number of transactions. A transaction that involves a Smart Contract is also called *message* and contains the instructions needed to execute a function of the contract. It involves a change of blockchain data (i.e its state). Consider that every blockchain change has a cost, that accounts for the computational effort needed to execute the transaction. These selected contracts are those that have involved many changes of state of the Ethereum blockchain.

On Tab 6 the contract *BAToken*, in position twenty,has a number of transactions over seventy times higher than the average value of the complete dataset which is 3019. The first contract, *Etherdelta*, has been involved in a transaction twenty thousand times more than the average usage. In total, these contracts are about 0.2% of the total set but are involved in about 61.3% of the total number of transactions. These numbers are in line with the distribution of the number of transaction previously reported in Fig 6. The enormously larger usage of this subset of Smart Contracts, explains the presence of a strong tail in the statistical distribution reported in Fig 6 and justifies our choice of examining in detail the most used Smart Contracts.

The values of I2 (Tx/day) are a normalization of the values of the indicator I1, obtained dividing it by the effective usage time. This allows us to compare Smart Contracts in terms of frequency of interactions, even when they have been deployed in the blockchain at different times.

The number of usage days is the number of days between the first and the last transaction of the contract. All the contracts under examination are characterized by a high value of I2, from a minimum of 479.32 up to 42130.33 transactions per day. Contracts with a high value of I2 can be considered either needful contracts in the Ethereum ecosystem, or contracts that have had a extraordinary popularity in their activity period.

It is relevant the case of the contract *KittyCore*, that, as described before, is a decentralized game. Considering the value of the indicator I2 of that contract, that is the highest value of transaction rate among the twenty selected contracts,

TABLE 6. Contract usage indicators.

#	Contract name	I1: Tx count	I2: Tx/day	Balance	DoA	I3: NoD / Tot	I4: NoV	I5: RoC	Compiler
1	Etherdelta	6562254	20191.55	4.61e+14	325	8 / 8	5	0	v0.4.9
2	Bitcoinereum	1450979	17694.87	0	82	1 / 1	1	0	v0.4.17
3	KittyCore	1390301	42130.33	70.9	33	1 / 1	1	0	v0.4.18
4	ReplaySafeSplit	1241476	2369.23	0	524	6/6	1	0	v0.3.5
5	Registrar	1103826	4580.19	0,	241	3 / 11	2	0	v0.4.10
6	DSToken	1005345	5376.18	1373.27	187	10 / 10	1	6	v0.4.11
7	Controller	768834	5571.26	0	138	4 / 16	2	0	v0.4.11
8	OMGToken	750295	4191.59	0	179	1 / 1	1	0	v0.4.11
9	TronToken	742060	5936.48	0	125	1 / 1	1	0	v0.4.16
10	MCAP	557685	2523.46	0	221	3 / 4	1	0	v0.4.11
11	GolemNetworkToken	535935	1201.65	0	446	3 / 3	1	0	v0.4.4
12	SaleClockAuction	514167	15580.82	7.09	33	1 / 1	1	0	v0.4.18
13	ReplaySafeSplit	421810	876.94	0	481	6 / 6	1	0	v0.3.5
14	EOSSale	370266	1969.50	0	188	1 / 1	1	0	v0.4.11
15	SNT	296699	1521.53	0	195	1 / 1	1	0	v0.4.11
16	HumanStandardToken	296623	1694,99	0	175	44 / 44	1	11	v0.4.10
17	PayToken	279107	1409.63	0	198	1 / 1	1	0	v0.4.11
18	Etheroll	269043	1724.63	0.79	156	20 / 21	4	0	v0.4.10
19	ReplaySafeSplit	253559	479.32	0	529	6 / 6	1	2	v0.3.5
20	BAToken	213364	987.80	0	216	1 / 1	1	0	v0.4.10

we found that this contract is involved in about 30 transactions per minute. It is associated to the contract *SaleClockAuction* that also has a very high value of I2.

In Tab. 6 we reported the Days of activity (DoA) for each contract. The contract *KittyCore* counts only 33 DoA. The longest-running contracts are the *ReplaySafeSplit* family (all exist and have been used for more than a year), followed by *Etherdelta* and *Registrar*.

For what concerns the balances, only five out of twenty Smart Contracts have non null balances and only two have significantly high balances. *Etherdelta* and *Dstoken*, as already described, the former is a popular wallet, the latter is a financial token born to fund a nascent blockchain. The analysis shows that most of the twenty Smart Contracts do not collect ether inside.

2) DEVELOPERS' INTERACTIONS: VERSIONS AND REUSE OF CODE

We examined the interaction of developers with Smart Contracts through the blockchain. According to the indicators defined, we analyze the number of deployments (I3), the number of versions (I4) and the number of times of code reuse (I5) for each Smart Contract.

One of the objectives of our empirical study is to investigate if Smart Contracts have been implemented thorough a code development process. For this reason, we checked the history of each contract, examining the presence of past versions and if improved versions have been deployed into the Ethereum blockchain.

We started our investigation filtering the dataset by the *contract name*, and then, since different Smart Contracts can have the same name, by means of an accurate analysis of the lines of code, we extracted the set of contracts referable to the same development history. The analysis of the source code allows to identify different contracts holding the same name. These contracts have been analyzed as different contracts. For computing the indicator I3 (Number of Deployments), we consider the number of contracts referable to the same source code. In Tab 6 we reported the I3 indicator (NoD) together with the number of contracts with the same name (Tot). We defined as a "new *version*" of a previous Smart Contract each new Smart Contract that once uploaded in the blockchain replaces the previous one in terms of blockchain interactions. The new version could contain code changes. We reported the values of the usage indicator I4 (Number of Version) in Tab 6 as NoV.

Finally, for the indicator of Reuse of Code (I5) we considered as *reuse* of the code of a Smart Contract the cases where the source code of the contract is used to implement a very similar contract that has the same name but is referable to a different project (for instance to implement a new token). The number of reuse of code is reported in Tab 6 as RoC. Evaluating version or reuse of code we did not consider those Smart Contracts which have a low number of transactions (less than one hundred), because they are rarely operational and could be only tests. So, they do not represents properly a new version or a reuse. In the following we analyze some contracts, focusing on the developers interaction, namely in terms of versions and reuse of code.

EtherDelta has eight contracts with the same name. It has five different versions, and the last one created is the most active. Two of the old versions are still used but they have a low number of transactions, about one per week. We notice that both of them have a lower amount of Ethers than the first one, therefore we can suppose that these Smart Contracts were used only by the contract developers and not by final users.

ReplaySafeSplit has three different Smart Contracts in the top twenty and all of them are active and are involved, on average, in a transaction every five minutes. By analyzing the

TABLE 7. Code metrics results in the twenty selected source codes.

Contract Name	M1: LoC	M2: CpL	M5: LpF	M3: NDC	M4: NDF	M6: MCC	M7: SCC
Etherdelta	232	0.18	5	7	32	4	60
Bitcoinereum	132	0.03	7	1	17	5	23
KittyCore	971	0.74	8	16	69	7	82
ReplaySafeSplit	20	0.2	7	2	2	3	4
Registrar	324	0.64	6	3	29	5	58
Dstoken	376	0.12	4	9	51	4	64
Controller	170	0.01	4	6	23	4	32
OMGToken	185	0.71	7	10	25	2	36
TronToken	131	0.03	7	1	17	5	23
MCAP	62	0.4	8	2	6	5	15
GolemNetworkToken	161	0.49	11	3	11	6	42
SaleClockAuction	300	0.68	7	6	21	2	23
ReplaySafeSplit	28	0.43	4	3	2	5	6
EOSSale	584	0.13	5	11	68	4	87
SNT	850	0.8	9	14	57	4	94
HumanStandardToken	70	0.93	6	3	8	2	11
PayToken	253	0.87	6	10	31	3	39
Etheroll	1112	0.41	4	5	52	3	64
ReplaySafeSplit	17	0.24	3	2	2	3	4
BAToken	129	0.31	8	4	12	7	30
Average Values	305.35	0.4175	6.3	5.9	26.7	3.85	39.5

code and the project's history, we found that they have a different usage for different projects, as explained in subsection V. We classified them as reuse of Smart Contract code, as reported in row 19 in Table 7. The other two (rows 4 and 13) are the examples of the reuse of code of the aforementioned Smart Contract and they do not have new versions and are not reused.

We found eleven contracts named *Registrar*. Analyzing these contracts we found that only 3 source codes can be evaluated in terms of the indicator I3, and the remaining have a completely different code. In addition, the discarded Smart Contracts have been involved in less than 10 transaction and these were probably tests. Only one out of three is active and belongs to the top twenty. We found one old version of this contract, and no reuse of code.

DSToken has ten records and we found six reuse of code, each of which derives from the Smart Contract in table 5, line 6. To confirm this, we checked on Etherscan that different labels are associated to these (still active) six addresses.

By considering a transaction number higher than one hundred and with reference to Etherscan, we found for *HumanStandardToken* 11 documented reuse of code. The remaining contracts are not evaluated in terms of reuse or new version of code, given the low number of transactions (in the order of units). However the code in these contracts has the same functionalities. We can state that, among those investigated, these two Smart Contracts are undoubtedly the most popular in terms of reuse of code because they were used as a reference for different projects.

The *Controller* has two versions related to the project Bittrex, as mentioned in subsection V. We detected a situation similar for *Registrar*: only 4 Smart Contracts belong to the project analyzed, the remaining 12 have a number of transactions in the order of units and a source code completely different. To be more precise we respectively found three couples of Smart Contracts and 6 Smart Contracts with the same source code or a different version of this.

Considering the records of *MCAP* we did not find different versions or reuse of code. One of the records has a different code and the other two are probably tests because of the low number of transactions (in the order of units). Similarly, *GolemNetworkToken* has no new versions or reuse of code.

Finally, all but one of the Smart Contracts named *Etheroll* found in the dataset are related to the same project. We considered four of these as different versions and the remaining as tests because of the low number of transactions. Actually, the previous versions are not used. Anymore this phenomena, in terms of source code improvement, is similar to the *EtherDelta* case. Referring to Table 6 the *Etheroll* Smart Contract in line 18 is no longer used and it has been replaced by the current active version.[23]

We also analyzed the declared pragma version. We found that in cases of different versions of the same Smart Contract, the pragma version of newer versions is generally updated with respect to the previous one, but not always corresponds to the most updated version of the language. There is only one case, the Etheroll Smart Contract,[24] that does not update the pragma version with respect to the previous one (v04.10) even if the next version (v04.11) has already been released.

[23] Having address 0xD91E45416bfbBEc6e2D1ae4aC83b788A21Acf583
[24] Having address 0xece701c76bd00d1c3f96410a0c69ea8dfcf5f34e

In all cases of Smart Contract updating, the developers have deployed the new version in the blockchain, supporting the related costs.

C. CODE METRICS

For every Smart Contract source code listed in Tab. 5 we computed the code metrics described in Section III and the following additional code metrics.

M5, *Lines of code per Function* (LpF): it is the average number of lines of code written to implement a function.

M6, *Max cyclomatic complexity* (MCC): it is the max value of the McCabe cyclomatic complexity among the cyclomatic complexities of all functions in the contract.

M7, *Sum of cyclomatic complexities* (SCC): it is the sum of the McCabe complexity of each function in the source code. That value depends on the number of function in the contract. The average cyclomatic complexity in a Smart Contract source code is equal to the division between the values of M7 and M4.

The last two metrics are **Complexity Metrics**. In particular, the cyclomatic complexity measures the number of linearly independent paths through a function in the source code. We computed the cyclomatic complexity according to McCabe definition [16] and using a commercial software.[25]

We report in Tab. 7 the resulting values for the metrics from M1 to M7 computed for each Smart Contract source code belonging to the selected set.

Results in Tab. 7 allow us to compare the value of metrics from M1 to M4 for the overall set of contracts presented in Tab. 3 with those of the top twenty contracts having the higher number of transactions and representing the contracts having the larger interactions in the blockchain.

We found that the source codes of the top twenty contracts have, on average, a value of M1 (LoC) equal to 305.35, that is well higher, on average, than the value of M1 computed on the full dataset (180.01 lines). In particular, exactly half of the source codes have a value of M1 higher than 180. Results confirm that the number of transactions and the number of lines of code are not correlated.

Analyzing M2 in Tab 7, namely the number of comments per line of code, we can observe a high variability of the results. Values ranges from one line of comment every one hundred lines of code to about one line of comment per one line of code. On average, there are 0.41 comments per line of code and this number is a little lower than the average value of the full dataset (0.49).

Considering the number of declared contracts measured by M3, and the number of declared functions measured by M4, we can observe, on average, higher values of declarations in comparison with the global results. In particular the average value of M3 is 5.90 (the average value of the full dataset is equal to 4.39) and M3 values range between a minimum of 1 to a maximum of 16, and half of the twenty Smart Contracts have M3 greater than 4. High values of M3 means that source codes of Smart Contracts are written exploiting the inheritance mechanism. The source code of KittyCore (having the maximum number of declared contracts) is a typical example of systematic use of the inheritance. The structure of this contract is reported in Appendix C.

The average value of M4 is 26.70, and it is about five times the average value of the full dataset (that is equal to 5.30). This means that, on average, the twenty selected contracts implement a higher number of functions. In facts, 17 out of 20 declare more than 5 functions. Values of M4 have minimum 1 and maximum 69. The highest values of declarations is related to the contract in third position, namely *KittyCore* that implements a large number of functionalities. High values of declarations characterize also some tokens (i.e *Dstoken, EOSSale, SNT*). These contracts improve the functionalities defined in the ERC-20 standard, by adding specific and customized features.

Analyzing the results of the metric M5 (Number of lines per function), computed only for the set of twenty contracts, we can observe that the functions have, on average, 6.30 lines. The contract GolemNetworkToken has the largest value. Considering the ERC-20 compliant contracts, the variability of the functions length suggests that tokens are not all implemented in the same way.

Considering the cyclomatic complexity metrics, M6 and M7,we can observe that the majority of the source codes has a maximum complexity (M6, MCC) lower than four (or in other words, it is hard to find functions with cyclomatic complexity greater than 4). Both the source code of *KittyCore* and the source code of *BAToken* have a function characterized by the highest value of cyclomatic complexity equal to seven. See Appendix C for the function of *KttyCore* which has the maximum cyclomatic complexity. The lower value of M6 is equal to 2 and characterizes the contracts *OMGToken, SaleClockAuction* and *HumanStandardToken*.

Finally, considering M7, namely the sum of the cyclomatic complexity of each function declared in the source code, the three most complex contracts belong to *SNT, EOSSale, and KittyCore*. The three versions of *ReplaySafeSplit* are characterized by a very low value of M7 because of the low number of functions. The codes reported in Appendix B shows that this contract has only two functions.

D. ANALYSIS OF RESULTS

Tab 8 shows the correlation matrix between metrics computed among the twenty selected contracts. As we can expect, metrics values of M1, M3, M4 and M7 are mutually correlated and this means that the longer is the code, the more complex is the program. In particular, the sum of the cyclomatic complexity (M7) and the number of declared functions (M4) have a correlation factor that represent a strong linear relationship of the ratio between the two metrics. The values of M5 and M6 have a average-high value of the correlation coefficient.

[25]We computed the cyclomatic metrics using *Understand*, that is a *scitools* software. These cyclomatic metrics are described in https://scitools.com/support/cyclomatic-complexity/

TABLE 8. Cross Correlation Matrix of source code metrics.

	M1 LoC	M2 CpL	M5 LpF	M3 NDC	M4 NDF	M6 MCC	M7 SCC
M1 LoC	1.00	0.25	0.03	0.68	0.88	0.14	0.82
M2 CpL	0.25	1.00	0.34	0.40	0.12	-0.17	0.15
M5 LpF	0.03	0.34	1.00	0.09	-0.04	0.59	0.13
M4 NDC	0.68	0.40	0.09	1.00	0.83	0.13	0.78
M5 NDF	0.88	0.12	-0.04	0.83	1.00	0.23	0.94
M6 MCC	0.14	-0.17	0.59	0.13	0.23	1.00	0.36
M7 SCC	0.82	0.15	0.13	0.78	0.94	0.36	1.00

TABLE 9. Correlation coefficients between usage indicators and code metrics.

	I1 Tx count	I2 Tx/day	I3 NoU
M1 LoC	-0.05	0.35	0.00
M2 CpL	-0.25	0.05	0.26
M5 LpF	-0.11	0.14	-0.28
M3 NDC	0.06	0.41	-0.21
M4 NDF	0.08	0.39	-0.13
M6 MCC	0.13	0.39	-0.40
M7 SCC	0.18	0.31	-0.18

The metric M2 does not present particular correlations coefficient.

In order to analyze the relationship between source codes and the use of the contracts, we analyzed if results of the applied metrics are correlated with the usage indicators. We studied if and how the two analysis are correlated computing the correlation matrix in Tab. 9 that reports correlation coefficients computed for the twenty selected contract.

We discovered that the there is no particular connection between these analysis. In particular, the indicator I1 (Tx count) is weakly correlated with all the other metrics. The indicator I2 (Tx/day) shows a interesting moderate correlation with the metrics that describe size and complexity of the source code (M1, M3, M4, M6 and M7).

VI. DISCUSSION AND RESULTS

Results of this empirical study provide a global overview of the world of Smart Contracts. This world can be described as very active in the usage of the blockchain, heterogeneous in the typologies and in the code features, and supported by an interactive and reactive development community. Our research leads to several outstanding findings we summarize below.

Result 1 (Impact of Solidity Language Evolution on Smart Contracts Development): We found that the Smart Contract developers' community follows and constantly adhere to the evolution of the Smart Contract programming language, Solidity. The reasons are probably found in the need to develop, already from the beginning, efficient and secure Smart Contracts. In fact, the update of a Smart Contract for bug fixing or for adding new functionalities consists in the deployment of a new Smart Contract in the blockchain and, in parallel, on the disposal of the old one, since there is not possibility to update or modify the source code once the contract is deployed on the Blockchain, as instead occurs for traditional software. The creation of a Smart Contract leads to a cost in Ether that depends on the dimension of its bytecode.

Result 2 (Smart Contracts Purposes): By analyzing the purposes of various Smart Contracts we found that developers have overtaken the concept of "parties' agreements" that characterizes the first era of Smart Contracts. In facts they created several typologies of decentralized applications, ranging from games to utility tokens. We also found that just a few percent of the total Smart Contracts are used to deposit ether.

Result 3 (Reuse of Code): We found strong evidences on the practice and importance of code reuse. Thanks to the availability of thousands Smart Contracts source codes, developers start from already implemented contracts to create new and more efficient applications, or updated and customized versions of former Smart Contracts. In addition, source codes are generally well commented, and this helps new developers to understand their contents.

Result 4 (Contract Name Relevance): The "contract name" of a deployed Smart Contract could cause some misunderstanding. We discover that some specific names are commonly used even if in general are associated to very different source codes with different purposes. We can conclude that contract name is not representative of the contract's purpose and code.

Result 5 (Interaction of Deployed Smart Contracts With the Blockchain): We analyzed the interaction of deployed Smart Contracts with the blockchain by means of usage indicators and we discovered that the number of transactions follow a power-law distribution. Since we found that the balances of the corresponding addresses follow a power-law distribution too, we computed the correlation among the two datasets finding no relationship between them. Indeed, Smart Contracts with high balance may use a low number of transactions and conversely Smart Contracts with very many transactions may have a low balance.

Result 6 (Balance and security connection): As stated before, we found that the distribution of the wealth overtakes the Pareto law because the wealth is strongly centralized on very few contracts (about the 90% belongs to twenty out of over ten thousands deployed Smart Contracts). This is related to the variety of typologies of Smart Contracts. In particular, most of the wealth belongs to contracts of the type *wallet*, which are responsible of the management and protection

of high amounts of cryptocurrency and consequently are more security critical. Our analysis thus suggests that a great advance in security with respect to cryptocurrency management and storage can be achieved focusing on the code quality or vulnerability analysis of just a reduced fraction of the total number of Smart contracts deployed on Ethereum, since only a little fraction of them holds and manages the largest part of cryptovalues.

Result 7 (Source code analysis and code metrics): The results of the analysis of the Source Codes give us a picture of a collection of Smart Contracts with heterogeneous features. These are characterized by code metrics that on average do not assume high values (for instance, the average number of lines of code is about one hundred and eighty lines), but have relatively high variances. This reveals that software development for Smart Contracts is highly heterogeneous reflecting the fact that many deployed contracts are probably only prototypes or trials or that many inexperienced developers deploy contracts on Ethereum without the adoption of a structured programming approach.

Source codes present, on average, four contract declaration, revealing the use of the inheritance or the recursion to already deployed contracts. The cross correlation analysis shows us that the Smart Contract bytecode (that can be seen as the payload of the transaction with which the Smart Contract is deployed) is mildly correlated to the number of lines of code and to the number of declared functions. We also found that the number of transactions is not correlated with source code features. This means that the most used Smart Contracts are not necessarily those well or better written. This renders even more critical the analysis of metrics and vulnerabilities of Solidity since it may easily occur that low quality or vulnerable code may be repeatedly and frequently used in exchanging goods or wealth across the Blockchain addresses or that such vulnerable code may be reused many times.

Result 8 (Detailed Analysis of Features in the Most Used Smart Contracts): We analyzed in detail a subset of twenty Smart Contracts, namely those involved in the highest number of transactions.

Result 8.1. Smart Contract Purposes: We discover that they are mainly financial Smart Contracts (that implement a token compatible with the standard ERC20). We found also wallet, library, notary, and game Contracts. The majority of the contracts in this subset belongs to projects funded using an ICO, fact that emphasize the role of the development team and justifies the use of ERC20 tokens.

Result 8.2. Definition of Empirical Usage Indicators to Analyze Smart Contract Activity: To characterize the activity behind these twenty Smart Contracts we define five usage indicators. These characterize both the interaction with the blockchain in terms of number of transaction and number of transaction per day, and the activity of the development community in terms of number of uploads, number of versions and number of reuse of code. We discovered that the game contract ''KittyCore'' has had, from the beginning, a high interaction with the blockchain. We also found that some contracts are still active in the blockchain after years, as it is the case of the contracts called ReplaySafeSplit. Regarding the number of uploads and the number of ''reuses of code'', we discovered that eleven contracts have a development story behind it. In facts, these contracts are the results of a continuous improvement and of the related replacement of the old versions. In four cases we can report the release of a new version of the contract. We found that the source code of four contracts was reused to develop new Smart Contracts.

Result 8.3. Source Code Metrics: Our empirical analysis shows that the statistics of the subset of the twenty most used Smart Contracts differ from the statistics computed on the total set. In particular, these contracts are, on average, longer (about three hundred lines of code) and define and contain five times more functions than average. In addition, we discover that these two metrics are strongly correlated with the sum of the McCabe cyclomatic complexity computed for the functions in the source codes, namely they are more complex than average contracts. Finally, we computed the correlation coefficient between source code metrics and usage indicators. Our results reveal that the number of transactions per day (that represents the frequency of usage of a Smart Contract) is moderately correlated with the number of lines of code, with the number of declared functions, and with the cyclomatic complexity of the source code.

VII. CONCLUSION

This work presents the setup, the analysis and the results of an empirical study on a set of Ethereum Smart Contracts and on their source code. We acquired a dataset of 10174 source codes, published by the beginning of 2018, and we statistically analyzed and characterized the overall dataset by mean of different software measures. Our empirical study examined the dataset from several points of view, like the use and the evolution of the Solidity compiler version and the related Solidity constructs, the number of interactions and transactions among Smart Contracts and Blockchain, the purpose and the naming practices for the Smart Contracts, the code reuse. Our empirical study is devoted to a complete characterization of the body of information available from metadata recovered by the analysis of Smart Contracts source code and by various information related to the Blockchain environment. The study contributes to understanding the interaction between Smart Contract and Blockchain and to the knowledge of the main characteristics of contracts written in Solidity. It also provides a description of the Ethereum Smart Contracts as elements of a system that is very active in the usage of the Blockchain, heterogeneous in the typologies and in the code features, and supported by an interacting and reactive development community.

We enrich the research providing more explicit knowledge about the Ethereum Smart Contract domain gathering eight relevant empirical results.

Future works should consider to analyze a higher number of Smart Contracts (taking into account the set of Smart

Contracts without available source code), further and specific code metrics (i.e to evaluate eventual code optimization in order to limit the Ethereum *gas* consumption or to measure the use of libraries and the interaction with already deployed contracts), other usage indicators (such us the internal transactions and the interaction between deployed contracts) and a wider analysis of correlation.

APPENDIX
SAMPLE OF SMART CONTRACTS SOURCE CODES
A. CROWDSALE

A portion of the source code of the contract Crowdsale deployed at the address:
0xa1877c74562821ff59ffc0bc999e6a2e164f4d87. This Smart Contract is named "Crowdsale".

The source code includes two contract definition. The first contract is *token* and the second is *Crowdsale*. The contract token is an interface. In a interface, all functions are only declared but not implemented. The contract *Crowdsale* declare an "instance" of the contract *token* called *tokenReward* and assign to it the contents of an already deployed Smart Contract. In the source code, the instance of a contract can be used to execute its functionalities.

```
pragma solidity ^0.4.8;
contract token {
    function transfer(address receiver, uint
    amount){ }
}

contract Crowdsale {
    uint public amountRaised;
    uint public resAmount;
    uint public soldTokens;
    mapping(address => uint256) public balanceOf;

    // initialization
    ...
    token public tokenReward =
      token(0
    x2Fd8019ce2AAc3bf9DB18D851A57EFe1a6151BBF);
    /*addressOfTokenUsedAsReward*/
  ...
}
```

B. REPLAYSAFESPLIT

The original source code of ReplaySafeSplit. This code has the lowest sum of cyclomatic complexity belong the set of the twenty most used Smart Contracts.

Its source code is available on etherscan.io at the address:
0xE94b04a0FeD112f3664e45adb2B8915693dD5FF3

```
contract AmIOnTheFork {
    function forked() constant returns(bool);
}
contract ReplaySafeSplit {
    // Fork oracle to use
    AmIOnTheFork amIOnTheFork =
        AmIOnTheFork(0
    x2bd2326c993dfaef84f696526064ff22eba5b362);
    event e(address a);
    // Splits the funds into 2 addresses
    function split(address targetFork, address
    targetNoFork) returns(bool) {
        if (amIOnTheFork.forked() && targetFork.
    send(msg.value)) {
      e(targetFork);
            return true;
        } else if (!amIOnTheFork.forked() &&
    targetNoFork.send(msg.value)) {
      e(targetNoFork);
            return true;
        }
        throw; // don't accept value transfer,
    otherwise it would be trapped.
    }

    // Reject value transfers.
    function() {
        throw;
    }
}
```

C. KITTYCORE

Contract declaration and the listing of the function *isValidMatingPair*. This function has the higher cyclomatic complexity. The contract name of the deployed contract corresponds with the name of the last contract declaration. The last contract inherits most of the contracts declared above. The function *isValidMatingPair* is the function which has the highest cyclomatic complexity among the contract. The complete source code is available on etherscan.io at the address:
0x06012c8cf97BEaD5deAe237070F9587f8E7A266d

```
pragma solidity ^0.4.11;

contract Ownable {...}
contract ERC721 {...}
contract GeneScienceInterface {...}
contract KittyAccessControl {...}
contract KittyBase is KittyAccessControl {...}
contract ERC721Metadata {...}
contract KittyOwnership is KittyBase, ERC721 {...}
contract KittyBreeding is KittyOwnership {
  ...
   function _isValidMatingPair(
  Kitty storage _matron,
  uint256 _matronId,
  Kitty storage _sire,
  uint256 _sireId
  )
  private
  view
  returns(bool)
  {
   // A Kitty can't breed with itself!
   if (_matronId == _sireId) {
    return false;
   }

   // Kitties can't breed with their parents.
   if (_matron.matronId == _sireId || _matron.
    sireId == _sireId) {
    return false;
   }
   if (_sire.matronId == _matronId || _sire.sireId
    == _matronId) {
    return false;
   }

   // We can short circuit the sibling check (
    below) if either cat is
   // gen zero (has a matron ID of zero).
   if (_sire.matronId == 0 || _matron.matronId ==
    0) {
    return true;
   }
```

```
41      // Kitties can't breed with full or half
          siblings.
        if (_sire.matronId == _matron.matronId || _sire
          .matronId == _matron.sireId) {
43        return false;
        }
45      if (_sire.sireId == _matron.matronId || _sire.
          sireId == _matron.sireId) {
          return false;
47      }
      return true;
49    }
      ...
51  }
    contract ClockAuctionBase {...}
53  contract Pausable is Ownable {...}
    contract ClockAuction is Pausable,
        ClockAuctionBase {...}
55  contract SiringClockAuction is ClockAuction {...}
    contract SaleClockAuction is ClockAuction  {...}
57  contract KittyAuction is KittyBreeding {...}
    contract KittyMinting is KittyAuction {...}
59  contract KittyCore is KittyMinting {...}
```

ACKNOWLEDGMENTS

The authors would like to thank the team of etherscan.io from which the authors extracted our source code dataset.

REFERENCES

[1] M. Bartoletti, T. Cimoli, and R. Zunino, "Fun with bitcoin smart contracts," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*, Oct. 2018, pp. 432–449.
[2] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.* Cham, Switzerland: Springer, Nov. 2017, pp. 494–509.
[3] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: Smart contract inspection technical report," Inria, Lille, France, Tech. Rep., Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01671196
[4] V. Buterin, "Ethereum white paper," Ethereum.org, Tech. Rep., 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper
[5] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Proc. Workshop Distrib. Cryptocurrencies Consensus Ledgers*, Jul. 2016, pp. 1–4.
[6] T. Cui *et al.*, "Achain blockchain whitepaper," Achain, China, Tech. Rep., 2017. [Online]. Available: https://www.achain.com/documents/Whitepaper.pdf
[7] P. Dai, N. Mahi, J. Earls, and A. Norta. (2017). *Smart-Contract Value-Transfer Protocols on a Distributed Mobile Application Platform*. [Online]. Available: https://qtum.org/uploads/files/cf6d69348ca50dd985b60425ccf282f3.pdf
[8] H. Rocha, S. Ducasse, M. Denker, and J. Lecerf, "Solidity parsing using SmaCC: Challenges and irregularities," in *Proc. 12th Int. Workshop Smalltalk Technol. (IWST)*, Sep. 2017, Art. no. 2.
[9] G. Destefanis, A. Bracciali, M. Marchesi, M. Ortu, R. Tonelli, and R. Hierons, "Smart contracts vulnerabilities: A call for blockchain software engineering?" in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 19–25.
[10] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, "The ICO phenomenon and its relationships with ethereum smart contract environment," 2018, *arXiv:1803.01394*. [Online]. Available: https://arxiv.org/abs/1803.01394
[11] S. Ibba, A. Pinna, G. Baralla, and M. Marchesi, "ICOs overview: Should investors choose an ICO developed with the lean startup methodology?" in *Proc. Int. Conf. Agile Softw. Develop.* Cham, Switzerland: Springer, May 2018, pp. 293–308.
[12] S. Ibba, A. Pinna, M. Seu, and F. E. Pani, "CitySense: Blockchain-oriented smart cities," *Proc. Sci. Workshops*, New York, NY, USA, May 2017, Art. no. 12.
[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269.
[14] K. Mannaro, A. Pinna, and M. Marchesi, "Crypto-trading: Blockchain-oriented energy market," in *Proc. AEIT Int. Annu. Conf.*, Sep. 2017, pp. 1–5.
[15] K. Mannaro, G. Baralla, A. Pinna, and S. Ibba, "A blockchain approach applied to a teledermatology platform in the sardinian region (italy)," *Information*, vol. 9, no. 2, p. 44, Feb. 2018.
[16] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[17] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in ethereum," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul./Aug. 2017, pp. 1–6
[18] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: Challenges and new directions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, Piscataway, NJ, USA, May 2017. pp. 169–171.
[19] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF CON*, vol. 25, p. 11, Jul. 2017.
[20] M. Swan, *Blockchain: Blueprint for a New Economy*. Newton, MA, USA: O'Reilly Media, 2015.
[21] N. Szabo, "The idea of smart contracts," Satoshi Nakamoto Institutue, Tech. Rep., 1997. [Online]. Available: https://nakamotoinstitute.org/the-idea-of-smart-contracts/
[22] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu, "Smart Contracts Software Metrics: A First Study," Feb. 2018, *arXiv:1802.01517*. [Online]. Available: https://arxiv.org/abs/1802.01517
[23] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 413–424.
[24] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, Apr. 2014.

ANDREA PINNA received the B.S. and M.S. degrees in electronic engineering and the Ph.D. degree in computer engineering from the University of Cagliari, in 2012 and 2018, respectively, where he has been a Research Fellow, since 2018.

His research interest concerns the study of blockchain technology and its applications. His topics of interest include the study of smart contracts, the engineering aspects in the development of decentralized applications, and the enhancement of the software sustainability thanks to the blockchain technology. He also dealt with the study of data stored inside blockchain of network features and users' behaviors.

SIMONA IBBA received the B.S. and M.S. degrees in electronic engineering and the Ph.D. degree in computer engineering from the University of Cagliari, in 2019.

Her scientific research activities are focused on the study of blockchain-based software and in particular on the application of the agile methodologies in blockchain software development. Her interests also include the knowledge management development and in particular the knowledge representation design, strategies for knowledge management, and the study of taxonomies, folksonomies, digital libraries, and scholarly literature.

GAVINA BARALLA received the master's degree in electronic engineering from the University of Cagliari, in 2012, where she is currently pursuing the Ph.D. degree in electronic and computer engineering.

Her research interests include knowledge management referred to semantic web, use of taxonomies, ontologies, linked data, blockchain technology, and smart contracts.

ROBERTO TONELLI received the Ph.D. degrees in physics and in computer engineering, in 2000 and 2012, respectively.

He is currently a Temporary Researcher and a Professor with the University of Cagliari, Italy. The main topic of his research has been the study of power laws in software systems within the perspective of describing software quality. Since 2014, he has been extended his research interest to the blockchain technology. His research interests are widespread and multidisciplinary.

MICHELE MARCHESI received the degree in Electronic engineering from the University of Genova, in 1975. He has been a Full Professor with the Faculty of Engineering, University of Cagliari, since 1994. Since 2016, he has been a Full Professor with the Department of Mathematics and Computer Science, University of Cagliari, where he teaches software engineering course.

He has authored over 200 international publications, including over 70 in the magazine. He has been one of the first in Italy to deal with OOP, since 1986. He was a Founding Member of TABOO, the Italian association on object-oriented techniques. He has also worked on object analysis and design, UML language and metrics for object-oriented systems since the introduction of these research themes. In 1998, he was the first in Italy to deal with Extreme Programming (XP) and agile methodologies for software production. He organized the first and most important world conferences on Extreme Programming and Agile Processes in Software Engineering, Sardinia, from 2000 to 2002. Since 2014, being among the first in Italy, he has extended his research interest to blockchain technologies, obtaining significant results in the scientific community.