Internet of Things 11 (2020) 100198

Contents lists available at ScienceDirect

Internet of Things

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

Ethereum smart contracts: Analysis and statistics of their source code and opcodes

Stefano Bistarelli[a], Gianmarco Mazzante[b], Matteo Micheletti[b], Leonardo Mostarda[b,*], Davide Sestili[b], Francesco Tiezzi[b]

[a] Mathematics and Computer Science department, University of Perugia, Italy
[b] Computer Science department, University of Camerino, Italy

ARTICLE INFO

Article history:
Received 24 February 2020
Accepted 6 March 2020
Available online 12 March 2020

Keywords:
Ethereum
Smart contracts
Opcodes statistics
Analysis of smart contracts

ABSTRACT

Smart contracts are programs that are used for verifying and enforcing the terms of an agreement. Ethereum is an extensively used platform that can be used to execute smart contracts. These can be defined by using high-level languages such as Solidity, Vyper and Bamboo that are compiled into low-level machine instructions, i.e., bytecode. These instructions define a Turing complete language and are often represented by using a readable format that is referred to as opcodes. Here, we present a novel, more comprehensive, study that aims at gaining a precise understanding on how programmers use the linguistic instructions supported by Ethereum. More precisely, the contracts' source code has been also analysed, in order to understand the high-level control structures and core instructions that are used for smart contract definitions, and how they are reflected on the opcode level. We have gathered ten of thousands of verified Ethereum smart contracts that have been written by using the Solidity language. This study can lay the groundwork for defining new formalisms and new domain specific languages (DSL). These can support the users in the development of decentralised applications (DAPPs).

1. Introduction

The use of smart contracts has been growing steadily for their tangible benefits. They rely upon the blockchain technology to ease the transfer of money, property, information or anything that people consider appropriate. Smart contracts have been successfully applied to IoT applications where things belong to different entities that have no mutual trust [1]. Smart contracts, differently from traditional contracts, are computable, this should improve their security and decrease the costs of enforcing the terms of the agreement. Ethereum [2] is amongst the most well-known platforms for smart contract definition. This is usually performed by using the Solidity [3] programming language, which is similar to JavaScript and C++. Other languages, such as Vyper and Bamboo, can also be used. Solidity smart contracts cannot be executed by the Ethereum Virtual Machine (EVM) directly, thus they are compiled into low-level machine instructions, i.e., bytecode. These instructions define a Turing complete language and are often represented by using a readable format that is referred to as *opcodes*. Smart contracts are executed over the blockchain-based distributed computing platform offered by Ethereum. A lot of research has been performed to analyse smart contract data from various perspectives. The authors in [4] analyse

* Corresponding author at: Computer Science department, University of Camerino, 62032 Camerino, Italy.
E-mail address: leonardo.mostarda@unicam.it (L. Mostarda).

https://doi.org/10.1016/j.iot.2020.100198
2542-6605/

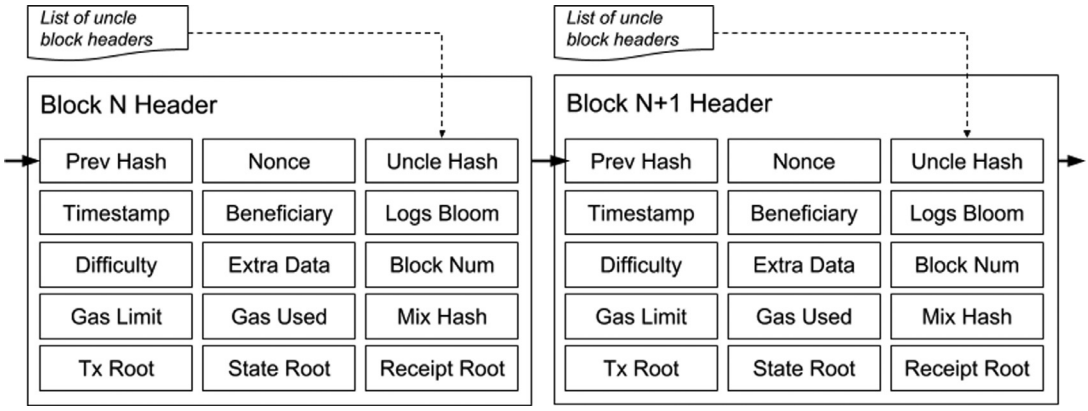

Fig. 1. Block data of the Ethereum blockchain.

smart contracts in order to detect zombie contracts. The empirical study that has been presented in [5] analyse the use of contracts by considering their application domain. Finally, other researches, such as those described in [6,7], inspect the contracts from economic, technical, and legal point of views.

This work presents a novel study about the usage of Ethereum opcodes and high-level Solidity constructs in a large set of real smart contracts deployed in the Ethereum blockchain platform. The objective is to understand how smart contract programmers use the linguistic instructions supported by Ethereum. We provide an analysis of smart contract opcodes in order to find out opcodes that play a crucial role in practice, and rule out the ones that are not practically relevant. In this paper we further enhance our analysis with the introduction of a new study on the Solidity smart contract source code. This allows the comprehension on how high-level linguistic constructs have been used in practice by programmers. To carry out this study, we have gathered ten of thousands of verified Ethereum smart contracts that have been written by using the Solidity language. Smart contracts are verified when a proof that they can be obtained by compiling a specific source code has been provided. We analyse the control-flow structures and core instructions that are used for smart contract definitions.

We have analysed the use of opcodes in order to extract useful information. For instance, we found out the existence of opcodes that do not appear in any smart contract deployed on the blockchain. Our analysis of smart contracts' source code, instead, provides important insights into the use of the Solidity language and, hence, sheds light on the nature of smart contracts written so far. For example, we found out that the majority of smart contracts do not use control loops, but have a rather simple linear structure, and that smart contracts rarely call other smart contracts. Our study hence supports the identification of a set of core features for the definition of new formalisms and domain-specific languages supporting the development of applications based on smart contracts. Formalisms can facilitate the development of formal techniques for verification.

The rest of this paper is organised as follows. Section 2 describes the background of Ethereum; Section 3 details the method that have been used for collecting data about smart contracts and describes the outcome of our analysis; Section 4 reviews the related work; Section 5 concludes the paper and outlines future work.

2. Background on Ethereum

The **blockchain** implements a ledger which records transactions between two parties in a verifiable and permanent way. The blockchain is shared and synchronised across various nodes (sometimes referred to as miners) that cooperate in order to add new transactions via a consensus protocol [8]. This allows transactions to have public witnesses thus making the ledger resistant to various malicious attacks (such as data modification). This paper focuses on the Ethereum [2] blockchain implementation which is to date the most popular one for developing smart contracts.

The consensus uses a modified version of Nakamoto algorithm [9] via transaction-based state transitions. The Ethereum platform is used for the implementation of the Ether cryptocurrency which can be transferred between accounts. A wallet stores the public and private keys (i.e., addresses) which are used to receive and spend Ether.

The Ethereum blockchain is composed of a growing list of blocks. Fig. 1 shows all data that are contained in a block. An Ethereum State is represented as a Merkle tree [10] and defines the current balances of all accounts plus some extra data. The field *state root* is the hash of the root node of the state tree after all transactions are executed and finalised. The *tx root* field is the root of the Merkle tree that contains all transactions that have been validated. A transaction can contain a transfer of Ether, an interaction with a smart contract or the management of a token. The *timestamp* field defines the time at which the block is added into the blockchain. *Block num* is a unique block identifier. Transaction receipts store the state after the execution of a transaction. All receipts are kept in an index-keyed trie [11]. The hash of its root is placed in the block header as the *receipt root*. The bloom filter [12] is created based on the transaction log information. This is reduced to 256 bytes hash and all hashes are in the block header as the *logs bloom*. Each block also includes the hash of the prior

block in the blockchain, *prev hash*, linking the two. This ensures the integrity of the previous block, all the way down to the first block. The hash is calculated on the header fields by using the Ethash [2] algorithm. More precisely, every time a new hash block is calculated, the Ethereum nodes perform a mining process (also referred to as proof-of-work) where they must find a nonce value that produces a desired hash. This is a hash that contains a certain amount of initial zeros. The number of zeros, defined by the *difficulty* field, establishes the computational effort that is needed to find the hash. When a node finds the nonce (i.e., the block is solved), the consensus is used to spread the solution to other nodes. These verify the solution proposed and add the block to the blockchain. Sometimes separate blocks can be produced concurrently. This creates a temporary fork of branches. The Ghost protocol [13] is used in order to select one of the branches leaving the others as orphan ones (the orphan branches are stored in the *uncle hash* field). All Ethereum transactions cost a certain amount of **gas** [13], which corresponds to a tiny fee required by the blockchain infrastructure to process the transaction. The *gas limit* is the maximum amount of units of gas a user is willing to spend on a transaction. The block gas limit sets an upper limit to the *gas used* block field. This is the sum of all gas spent by the transactions of the block.

One of the main feature of Ethereum is its Turing-complete scripting language, which allows the definition of smart contracts. These are small applications that are executed on the top of the whole blockchain network. The code of an Ethereum contract is written in a low-level, stack-based bytecode language, i.e., the Ethereum Virtual Machine (EVM) code. Opcodes are a human-readable representation of bytecodes. A comprehensive collection of Ethereum bytecodes and opcodes is described in the Ethereum yellow paper [2]. Smart contracts can be written by using high-level languages that are compiled down to EVM bytecodes. The most popular language for writing smart contracts is **Solidity**. There exist two different compilers for compiling Solidity smart contracts. One is written in Javascript and is called `solc-js` while the other is written in C++ and is called `solc` [14]. This is the official compiler thus the one that is taken into account in our study.

A smart contract is deployed onto the blockchain by using a transaction. A gas limit and a gas price are always associated to the smart contract execution. The former is the maximum amount of gas a user is willing to pay for the smart contract execution while the latter defines the conversion rate of gas to the Ether cryptocurrency. Higher conversion rates will encourage the selection of contracts by the miners. Miners locally execute the smart contract bytecode. Each instruction costs a certain amount of gas, the total amount paid for the contract is proportional to the instructions that are executed. A smart contract always gets a unique contract address when is successfully added to the blockchain. A users can execute a contract by generating a transaction that contains the contract address, the gas limit and the contract method to be called. The output of the execution, the gas used and other information are written into a block. It is possible that the gas assigned to a transaction is not enough in order to end the computation. In this case, the execution is terminated with an exception, and the chain state is reverted to its initial state (before the computation started). Notably, the contract execution must be paid although its execution did not successfully end. This prevents resource-exhaustion attacks [9].

A **Block explorer** or simply **Explorer** is a tool used in order to gain access to information that regards online blockchain transactions. Some block explorers allow users to verify smart contracts that are deployed on the blockchain. Explorers can enable types of blockchain analyses and permit the *contracts verification*. This process is composed of the following three steps:

1. the author registers the compiled code of the smart contract into the blockchain;
2. the author publishes the original smart contract source code and the compiler version into the block explore;
3. the explorer flags the contract as verified when the compiled code can be obtained from the related source code.

It is worth mentioning that the smart contract verification process cannot be performed by solely using the blockchain. This does not store any Solidity source code nor compiler information but only the bytecode.

3. Analysis of opcodes and smart contracts

This section details the experimental settings that have been used in order to collect data of Solidity smart contracts. The section concludes with a description of the data analysis and its results.

3.1. Experimental setup

Etherscan [15] has been used to collect data about smart contracts. Although different block explorers, such as Ethplorer [16] and Blockchair [17] are available, Etherscan is the only explorer that allows the verification of smart contracts. Our study considers the following smart contract information: (i) the Ethereum unique address of the contract; (ii) the opcodes that are used inside the smart contract; (iii) the version of the Solidity compiler that is used for compiling the smart contract; (iv) all dates where one or more smart contracts have been verified.

The contracts taken into account by our study are those that have been verified between October 2016 and May 2018. This is the date where our data collection ended. Very few contracts had been verified before October 2016 thus they have not been considered.

A Java program, (source code available at: https://github.com/GianmarcoMazzante/opcodeSurv), has been developed in order to scan the Etherscan web pages of *verified smart contracts*. The scanning is used to retrieve the addresses of all verified contracts. A smart contract address can be given as an input to the Etherscan API that returns the smart contract source in opcodes format. This output is then analysed by our Java tool, which stores in a JSON file (see Fig. 2 for details)

```
0x0000000000b3F879cb30FE243b4Dfee438691c04#code v0.4.16 [{"ADD":131},{"SHA3":30},{"CALL":1},{"S
0x000000002647e16d9bab9e46604d75591d289277#code v0.4.20 [{"ADD":70},{"SHA3":32},{"REVERT":38},{
0x000000002bb43c83ece652d161ad0fa862129a2c#code v0.4.20 [{"CALLDATACOPY":13},{"ADD":203},{"SHA3
0x000000005fbe2cc9b1b684ec445caf176042348e#code v0.4.20 [{"CALLDATACOPY":2},{"ADD":69},{"SHA3":
0x00000b233566fcc3825f94d68d4fc410f8cb2300#code v0.4.21 [{"CALLDATACOPY":3},{"ADD":84},{"SHA3":
0x0006157838d5a6b33ab66588a6a693a57c869999#code v0.4.12 [{"DUP1":1},{"JUMPDEST":1},{"REVERT":1}
0x000621424c60951cb69e9d75d64b79813846d498#code v0.4.18 [{"ADD":38},{"SHA3":20},{"CALL":3},{"SW
0x00138bd67465733ae1bedfd2105a6ffa83af97d8#code v0.4.18 [{"CALLDATACOPY":9},{"ADD":229},{"SHA3"
0x0014966c7439d8c10aaf0204036198236d9b0912#code v0.4.19 [{"ADD":80},{"SHA3":6},{"REVERT":60},{"
0x0016e71c7ced04b51a1fd8bb5c36d9e0cee9e1bb#code v0.4.20 [{"CALLDATACOPY":1},{"ADD":130},{"SHA3"
0x001b6e5c7322899355eb65486e8cbb7dbbf19127#code v0.4.18 [{"CALLDATACOPY":1},{"ADD":131},{"SHA3"
0x001ea8150f4965195e10e5b5568047e1555a6dcd#code v0.4.21 [{"ADD":93},{"SHA3":24},{"CALL":1},{"SW
0x001f0aa5da15585e5b2305dbab2bac425ea71007#code v0.4.19 [{"ADD":146},{"SHA3":40},{"SWAP1":250},
0x00214120d3469a74ca586bc9557c0ff8fb09b157#code v0.4.24 [{"ADD":158},{"SHA3":31},{"REVERT":51},
0x0022ee765799c1f836a36612b8c62be098fd0bbb#code v0.4.21 [{"ADD":83},{"SHA3":18},{"SWAP1":101},{
0x0023291920145638 90ffb66b483c29dd6607c419#code v0.4.21 [{"CALLDATACOPY":1},{"ADD":141},{"SHA3"
0x0027449bf0887ca3e431d263ffdefb244d95b555#code v0.4.20 [{"ADD":28},{"SHA3":1},{"CALL":1},{"SWA
0x002842529757eab873cce9c251087e1b993f9200#code v0.4.18 [{"ADD":11},{"SHA3":3},{"CALL":2},{"SWA
0x002c74b0cd9354ba5464db519cae76d7cfaa070a#code v0.4.20 [{"ADD":30},{"SHA3":22},{"SWAP1":110},{
0x002f06abe6995fd0ea4be011c53bfc989fa53ce0#code v0.4.17 [{"ADD":36},{"SHA3":45},{"REVERT":61},{
0x0033B3926db9C84a4dB255c15b995D50FfF0A568#code v0.4.11 [{"ADD":13},{"SHA3":19},{"SWAP1":72},{"
0x0033fb5561719b8b697b604466d6d39308c58191#code v0.4.16 [{"CALLDATACOPY":1},{"ADD":150},{"SHA3"
0x0033fd5819c111e5be558c52bd42fdc7481a94f1#code v0.4.18 [{"ADD":9},{"SHA3":3},{"CALL":1},{"SWAP
```


Fig. 2. Portion of the JSON with the information retrieved.

Table 1
Contract count and opcodes occurrences per month.

Month	Verified contracts	Opcodes count on verified contracts
10/2016	53	630859
11/2016	83	809555
12/2016	72	1491497
1/2017	108	1818251
2/2017	126	1830664
3/2017	120	1958167
4/2017	198	3332301
5/2017	270	3903969
6/2017	359	5436532
7/2017	702	10495739
8/2017	947	13541032
9/2017	1108	16653251
10/2017	1473	22308628
11/2017	1977	32242058
12/2017	2002	32415653
1/2018	2716	44550116
2/2018	3749	65411651
3/2018	3804	71645646
4/2018	3926	75555729
5/2018	3941	80398664

the address of each smart contract, the compiler version used to compile the contract, and all contract opcodes with the related frequency. This is the number of times the opcode appears inside the contract.

3.2. Opcodes statistics

This section describes the quantitative analysis that has been carried out on the smart contract data (see Section 3.1 for detials about data collection).

3.2.1. Opcodes frequency of all verified contracts

Table 1 shows for each month the number of verified smart contracts and the number of opcodes these contracts includes. It is worth noticing that the number of contracts exponentially increases from 2016.

The x-axis of Fig. 3 shows the hexadecimal encoding of all opcodes (a comprehensive list of opcodes can be found at [2]) while the y-axis displays the opcode global frequency. This is obtained by considering all smart contracts and counting the number of times the opcode appears inside them. We emphasise that only 5 opcodes have a global frequency higher

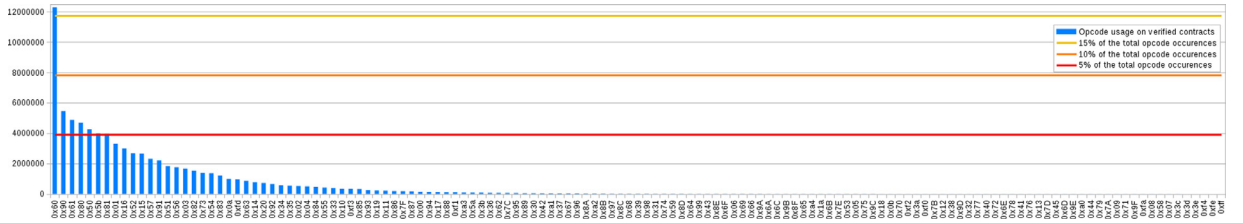

Fig. 3. Histogram of opcodes count on verified contracts.

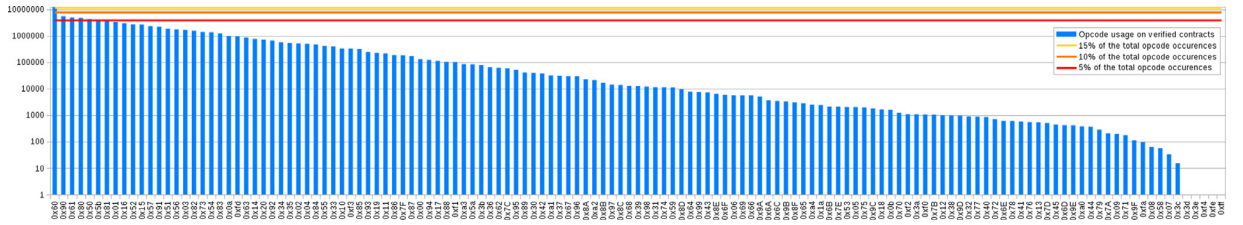

Fig. 4. Histogram of opcodes count on verified contracts (log scale).

Table 2
The ten most used opcodes of verified contracts.

	Most used opcodes on verified contracts
1	PUSH1
2	SWAP1
3	PUSH2
4	DUP1
5	POP
6	JUMPDEST
7	DUP2
8	ADD
9	AND
10	MSTORE

Table 3
First five most used push opcodes.

	Most used PUSH opcodes on verified contracts
1	PUSH1
3	PUSH2
18	PUSH20
23	PUSH4
41	PUSH32

than 5% of the sum of the global frequencies of each opcode. Fig. 4 shows the global frequencies of Fig. 3 with a logarithm scale.

3.2.2. Frequently used opcodes

This section discusses why some Ethereum opcodes appear more often than others.

Table 2 shows the ten most used opcodes. The majority of these opcodes are related to stack management operations, such as pop, swap and push since the EVM is a stack based machine. PUSH1 is used for adding 1 byte value onto the stack. This is the most frequent operation because it is a basic stack management operation and every contract starts with the sequence: `PUSH1 0x60 PUSH1 0x40 MSTORE`. This explains why MSTORE is amongst the most used opcodes. There are multiple variants of the `PUSH` opcode (the most used `PUSH` opcodes are shown in Table 3). The difference amongst them resides in the number of bytes they push onto the stack. While there are multiple `PUSH` opcodes there is only one `POP` opcode that works for every element of the stack regardless of their size in bytes. The number of `POP` opcodes is not necessarily equal to the number of every `PUSH` opcode in the contract. In fact the `POP` opcode is not the only one capable of removing elements from the top of the stack. For example opcodes such as `MUL` pushes the result of a multiplication between the two uppermost elements of the stack onto the stack while removing the factors of the multiplication from the stack.

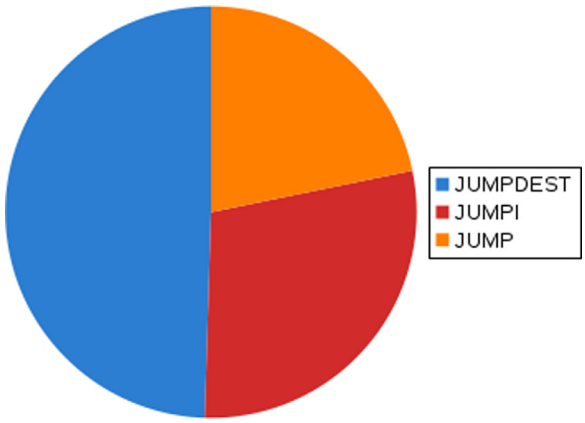

Fig. 5. Pie chart of JUMP,JUMPI,JUMPDEST opcodes occurences.

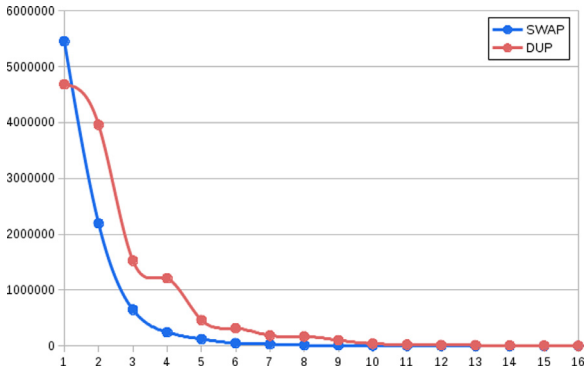

Fig. 6. Line chart comparison of SWAP and DUP occurences.

JUMPDEST, JUMP and JUMPI are used in order to translate control flow statements (i.e., if, loops and switch) from the smart contract source code. Fig. 5 shows a higher use of JUMPDEST with respect to the other jump operations. In fact, JUMPDEST is also used by the JUMP (unconditional jumps) and JUMPI (conditional jumps) opcodes in order to mark a valid destination. The frequency of JUMPDEST is also increased by their use for managing the internal code structure such as function linking. The ADD opcode is also present amongst the most used opcodes. ADD is not only used by programmers for performing algebraic operations, but it is also used for managing array positions. More precisely, the MSTORE opcode uses the ADD one in order to add an incremental value to the offset of an array. Opcodes PUSH20 and PUSH32 are also frequently used. The reason for their high usage becomes clear when we consider that accounts and contracts are identified by a 20-byte address while transactions are identified by a 32-byte address. The frequencies of SWAPs and DUPs opcodes decrease as the value targeted appears lower on the stack. Fig. 6 shows the frequency of these opcodes as the stack position increases from 1 to 16. We recall that DUPn duplicates the n-th item from the stack while SWAPn exchanges the 1st and nth stack items.

3.2.3. Less frequently used opcodes

Quite often an opcode is rarely used when it can be obtained by combining other opcodes. For instance this is the case of the RETURNDATASIZE opcode that was introduced by the Ethereum Improvement Proposal (EIP) number 211 [18]. This is used to obtain the output data size of the last external call. The behaviour of the RETURNDATASIZE is usually simulated with a sequence of other opcodes (see the EIP-211 proposal for details). In the same way, the RETURNDATACOPY opcode can be simulated by using other opcodes.

Table 4 shows a list of used opcodes that are rarely used since they introduce a peculiar variation of existing opcodes. For instance, the DELEGATECALL opcode is similar to the CALL one, except for the context used in the call (see [2] for details). The INVALID opcode was introduced with the EIP-141 proposal [19] and it is similar to the REVERT one. This was added in the EIP-140 proposal [20]. INVALID and REVERT terminate the code execution. INVALID drains all the remaining gas of the caller while REVERT does not. The INVALID opcode is not used because smart contracts never consume all the residual gas. Similarly, the SELFDESTRUCT opcode transfers the remaining Ether between sender and receiver accounts and destroys the contract. The sender account remains still available although interacting with it is a waste of Ether or gas. This behaviour is rarely used.

Table 4
Not used opcodes on verified contracts.

	Unused opcodes on verified contracts
131	RETURNDATASIZE
132	RETURNDATACOPY
133	DELEGATECALL
134	INVALID
135	SELFDESTRUCT

Table 5
Occurrences of environmental information opcodes on verified contracts.

	Environmental information opcodes on verified contracts
27	CALLVALUE
28	CALLDATALOAD
33	CALLER
50	EXTCODESIZE
51	CALLDATASIZE
56	ADDRESS
59	CALLDATACOPY
68	CODECOPY
70	BALANCE
100	GASPRICE
104	CODESIZE
106	ORIGIN
130	EXTCODECOPY
131	RETURNDATASIZE
132	RETURNDATACOPY

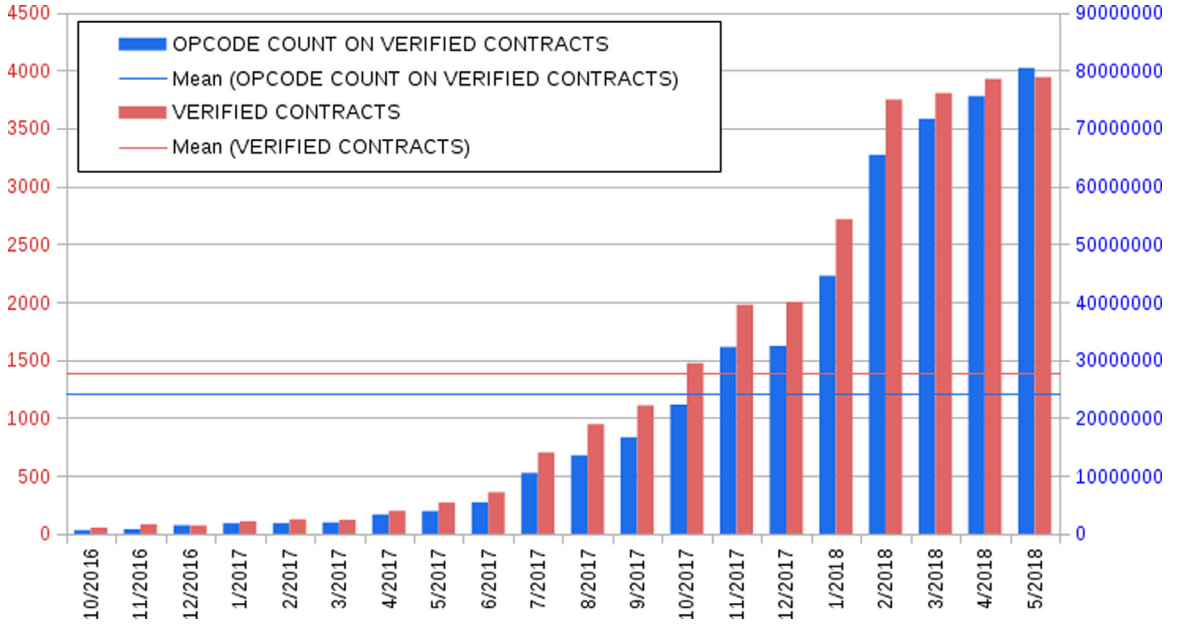

Fig. 7. Histogram of opcodes count per month and contracts count per month.

There are also various opcodes that are used to get financial information. For instance the `BALANCE` opcode is used for retrieving the balance of a specific address while `GASPRICE` is used for setting the gas price of the current transaction. This is not used since the default gas price is often used. Table 5 shows that some of these opcodes are rarely used used.

3.2.4. Opcodes and contracts count over the years

We discuss here the analysis of the total count of opcodes of verified contracts. In this regards, Fig. 7 reports an histogram where the x-axis has a wide range of different months while the y-axis shows (i) the number of contracts that have been verified, and (ii) the total count of opcodes that are contained inside verified contracts. The first considered month is October

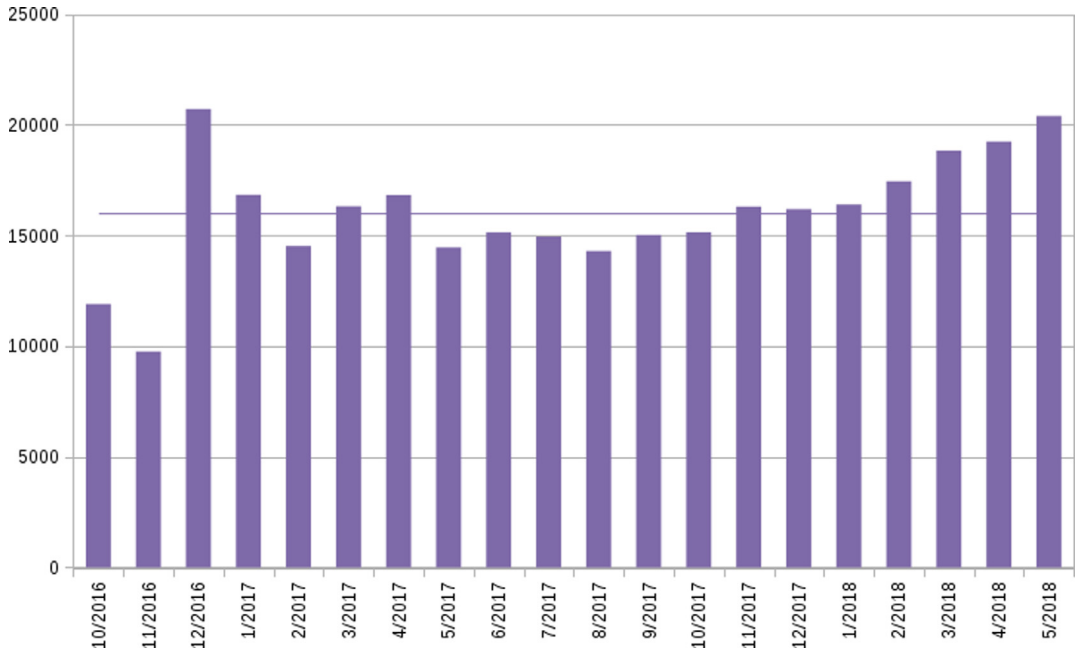

Fig. 8. Opcodes over contracts count per month.

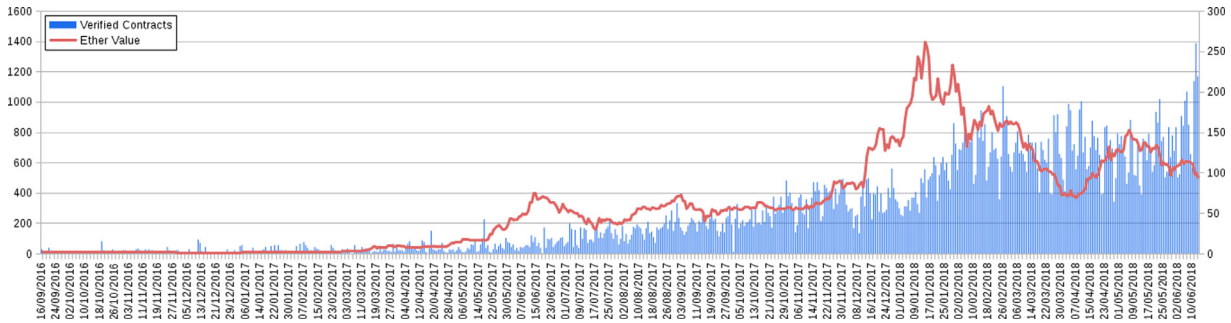

Fig. 9. Histogram of verified contracts per date and line chart of Ether value over time.

2016, as it corresponds to the release of the Solidity version 0.4.2. This histogram confirms that the usage of smart contracts significantly raised in popularity from 2016 to 2018.

For the same months, Fig. 8 instead shows the total count of opcodes that have been used in a month divided by the number of contracts verified in the same month. We kept indeed the same range of months of Fig. 7 in order to make the two charts comparable. Fig. 8 clearly shows that contracts are increasing in size.

Fig. 9 shows the amount of contract deployed on the Ethereum blockchain over time along with the value of the Ether cryptocurrency. The number of smart contracts that were deployed have the trend of the Ether value that is as the number of smart contracts increases so does the Ether value. The value of Ether seems to follow the trend of Bitcoin value.

3.2.5. Analysis on different solidity compiler versions

An analysis focusing on the evolution of contracts usage in relation to the versions of the Solidity compiler (from 0.1.1 to 0.4.25) is depicted in Fig. 10. It shows two different series, concerning the total number of contracts and that of opcodes, thus providing a comparative view. The compiler version v0.4.19 results to be the most used.We can observe how the compiler version usage is strictly related to the Ether value trend and the Ethereum platform popularity trend. This is clearly depicted in Fig. 9.

Fig. 11 considers the Solidity compiler version v0.4.19 and shows the number of occurrences of each opcode. It shows that the most used are the stack management opcodes, along with memory and storage management opcodes.

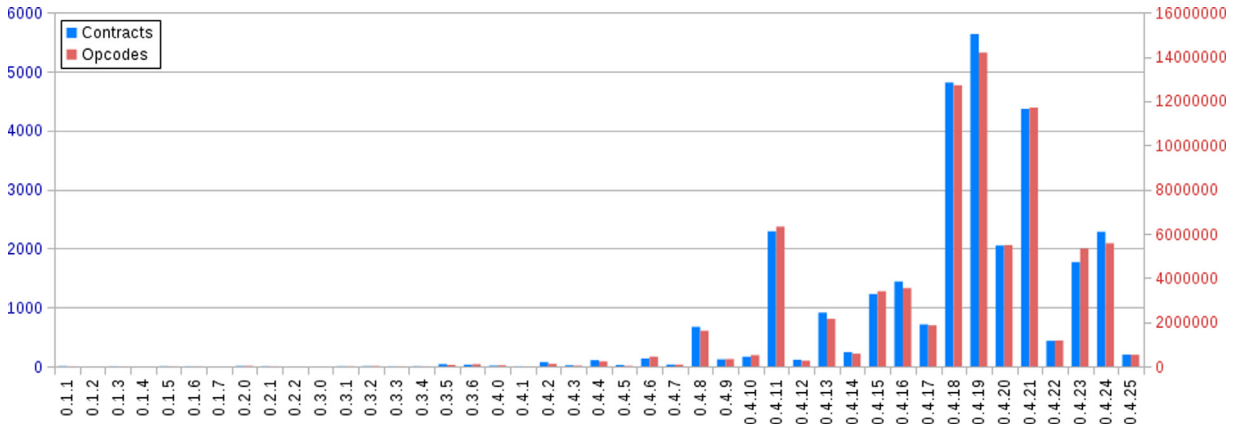

Fig. 10. Double histogram of opcodes count and contract deployments on different versions of the Solidity compiler.

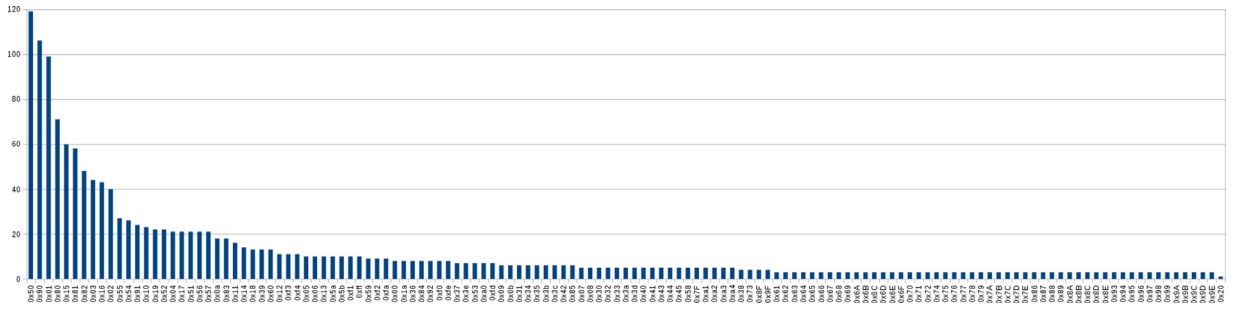

Fig. 11. Histogram of opcode occurrences on Solidity v0.4.19 source code.

3.3. Source code analysis

We report in this section our study of the Solidity source code of verified contracts. This analysis resulted to be quite complex, as it required to parse the smart contracts' source code and also the related compiled bytecode, for a large set of contracts.

3.4. Control structures analysis

We have checked the source code of smart contracts in order to keep track of the usage of all control structures, namely *if, if-else, for, while* and *do-while*. The *switch* and *goto* control structures are not available in Solidity. We have also considered the *break* and *continue* statements that are used inside loops. We have chosen to parse the Solidity source code in order to avoid the difficult task of reverse engineering the bytecode for identifying high-level control structures at opcodes level. In fact, as Figs. 12–14 show, the compiler translations of the *if, while* and *do-while* constructs use the same type of opcodes. Therefore, obtaining the statistics of control structures from the analysis of contract opcodes would be difficult. Notably, the translations reported in these figures have been generated by analysing the source code of the Solidity compiler (we have considered SolC 0.4.19, which is the most used compiler) and by analysing the bytecode produced by the compiler used on of several ad-hoc contracts.

Fig. 15 shows on the number of total occurrences of all Solidity high-level instructions. More precisely, for each instruction I we have calculated the total occurrences $O(I)$ by using the following equation:

$$O(I) = \sum_{k=1}^{N} f(C_k, I) \tag{1}$$

where N is the total amount of verified contracts, C_k the kth smart contract and f the occurrences-per-contract function that returns the number of times a given instruction appears in a given contract. The second column of Table 6 shows the same information in a numeric form.

Fig. 16, instead, shows the number of contracts using specific high-level instructions. Specifically, for each instruction I we have calculated the number of contracts $C(I)$ that contains at least one occurrence of I. This is defined by using the

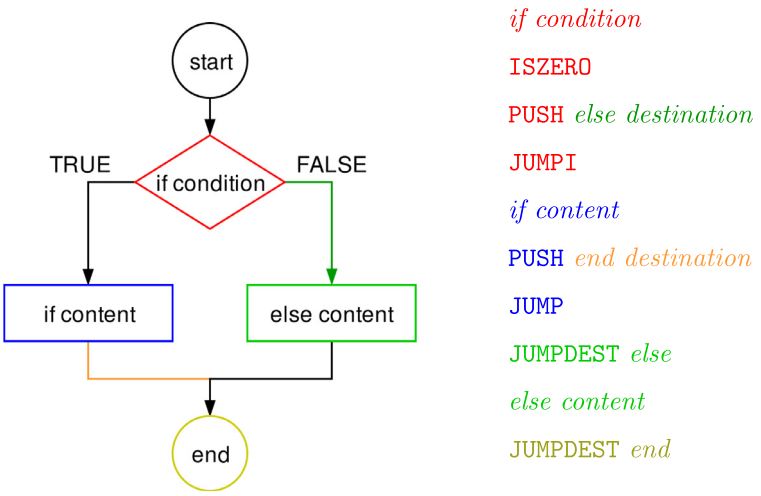

Fig. 12. Flow chart of *if-else* and corresponding translation in opcodes.

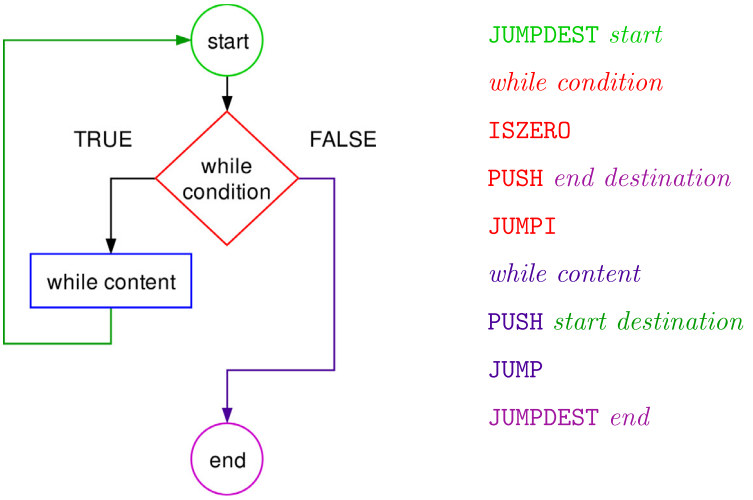

Fig. 13. Flow chart of *while* and corresponding translation in opcodes.

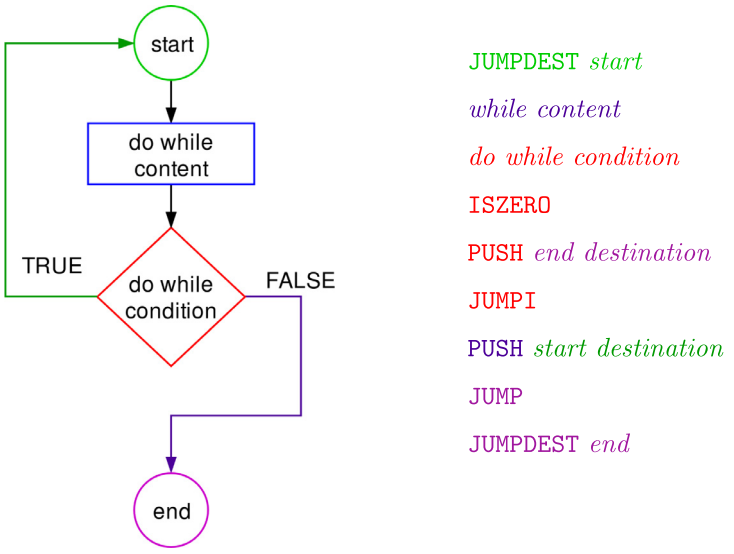

Fig. 14. Flow chart of *do-while* and corresponding translation in opcodes.

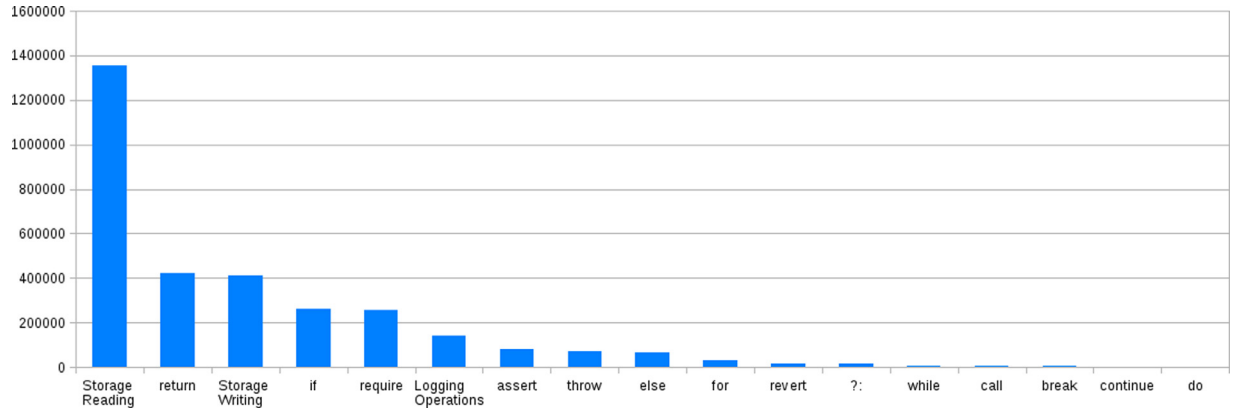

Fig. 15. Histogram of occurrences of high-level instructions on verified contracts.

Table 6
Table of occurrences and usage of high-level instructions on verified contracts.

Instruction	Occurrences	Contracts using it	Contracts using it (in %)
Storage Reading	1356355	29379	98,2378118103391
return	419916	27830	93,0582491807664
Storage Writing	412913	28718	96,0275529993981
if	258824	24554	82,1039256336521
require	256079	22275	74,4833812612854
Logging Operations	140581	25867	86,4943489600749
assert	79060	17131	57,2828195011035
else	66123	16628	55,6008827660001
throw	68452	7575	25,3293653447469
for	31239	7225	24,1590316324483
revert	14447	5739	19,1901290710894
?:	12911	4000	13,375242426269
while	5633	2056	6,87487460710225
call	3660	3307	11,0579816759179
break	3109	1916	6,40674112218284
continue	669	381	1,27399184110212
do	93	75	0,250785795492543

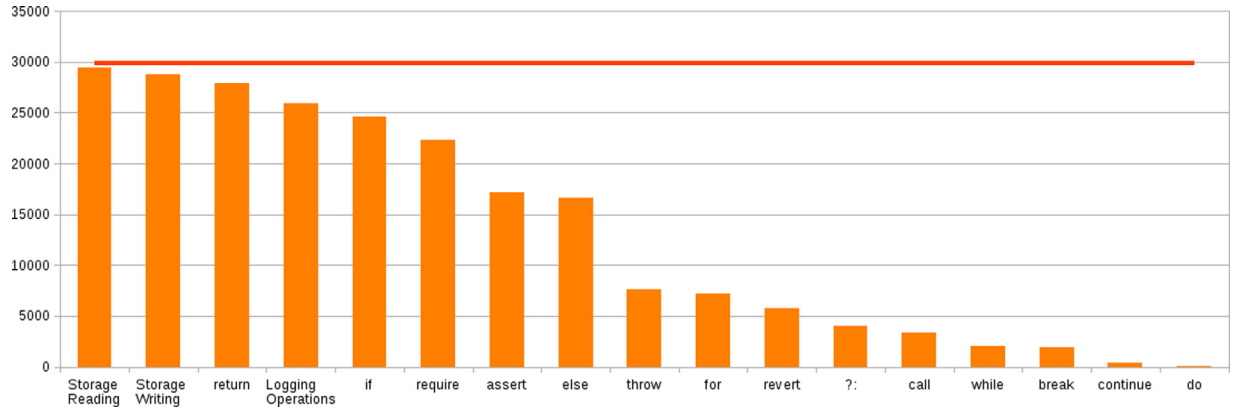

Fig. 16. Histogram of verified contracts using specific high-level instructions.

following equation:

$$C(I) = \sum_{k=1}^{N} IN(C_k, I) \tag{2}$$

where N and C_k, again, are the total amount of verified contracts and the k-th smart contract, respectively, while IN is the presence-per-contract function that returns 1 if a given contract contains a given instruction, 0 otherwise. The third and forth columns of Table 6 show the same information in a numeric form.

Our results show that only 0.25% of the contracts use the *do-while* loop, whereas 6.8% use the *while*. The *for* control structure is the most used, as it occurs in 24% of contracts. In contrast, *if, if-else* and the ternary operator ?: are used by most of the contracts. The *continue* and *break* instructions are rarely used and smart contracts rarely call other smart contracts. This is confirmed by the occurrences of the *call* instructions in the contract source code, that is 11%. This instruction is indeed used by a running smart contract in order to call smart contracts that are stored in the blockchain. We can conclude that the majority of the smart contracts do not use control loops but have a rather simple linear structure. Indeed, they are mostly used for reading and writing data on the blockchain. Application domains include logging, certification and tracking. This is also confirmed by the analysis presented in the next section.

3.5. Reading and writing data on the blockchain

Analysing reading and writing operations on the source code is not an easy task. Indeed, not only assignments to variables can produce writings on the blockchain but also other variable manipulations, such as the increment (resp. decrement) operator `++` (resp. `--`). Assignments can also produce more than one writing on the blockchain at once. For instance, the structure creation `myStruct = Struct(x,y,z)` produces three assignments. This complexity has lead us to analyse writes and reads on the blockchain at the bytecode level. More precisely, we have analysed the SLOAD and SSTORE opcodes that are used for reading and writing data on the blockchain. We use the terminology 'storage reading' and 'storage writing' in order to denote SLOAD and SSTORE opcodes in Figs. 15 and 16, and in Table 6. We have also considered all logging opcodes, i.e., LOG1, LOG2, LOG3 and LOG4. These provide a cheaper way to store data in the blockchain and their statistics are easier to gather at bytecode level. In Fig. 15 the 'logging operation' is the sum of the total number of occurrences of the instructions LOG1,LOG2,LOG3 and LOG4. More precisely, the occurrences-per-contract function $f(C_k, \textit{logging operation})$ can be defined as follows (see Eq. 1 for details about the definition of f):

$$f(C_k, LOG1) + f(C_k, LOG2) + f(C_k, LOG3) + f(C_k, LOG4)$$

Figs. 16 shows the total number of contracts where at least a logging instruction appears. More precisely, the presence-per-contract function $IN(C_k, \textit{logging operation})$ can be defined as follows (see Eq. (2) for details about the definition of IN) :

$$IN(C_k, LOG1) \vee IN(C_k, LOG2) \vee IN(C_k, LOG3) \vee IN(C_k, LOG4)$$

As clearly shown in these figures and in Table 6, reading and writing operations on the blockchain are the most frequent constructs: reading is performed by 98.2% of the contracts while the writing by 93%. Logging is instead performed by 86.4% of the contracts.

3.6. Throw, revert, assert and require operations

The instructions *throw, revert, assert*, and *require* play a basic role in error handling. More precisely, the *if-throw* pattern is used in order to throw an invalid opcode error, undoing all state changes, and using up all remaining gas. *Assert* and *require* are 'guard' functions: the former steals all gas, while the latter calls out for errors and steals a lower amount of a gas. *Revert* will still undo all state changes, but it will be handled differently, i.e., it will return a value and refund any remaining gas to the caller. Finally, we have analysed the *return* statement. The analysis of the use of all the aforementioned instructions has been performed at source code level. The use of these error handling mechanisms is quite widespread, that is 74% of the contracts use the *require* operation while 57% the *assert*. The *throw* operation is used by 25% of the contracts. This low percentage is a consequence of the Etheremum policy which has deprecated the use of the *throw* instruction.

4. Related work

The amount of related work that focuses on Ethereum smart contracts is scarce when compared to other famous blockchains such as Bitcoin [21–23].

Atzei et al. [24] presented a survey on possible attacks on Ethereum smart contracts. They define a taxonomy of common programming deadfalls that may lead to different vulnerabilities. The work provides helpful guidelines for programmers to avoid security issues due to blockchain peculiarities that programmers could underestimate or not be aware of. With a similar aim, Delmolino et al. provide a step by step guide to write "safe" smart contracts [25]. The authors asked to the students of the Cryptocurrency Lab of the University of Maryland to write some smart contracts, and guided them to discover all the issues they had included in their contracts. Some of the most common mistakes included: failing to use cryptography, semantic errors when translating a state machine into code, misaligned incentives, and Ethereum-specific mistakes such as those related to the interaction between different contracts.

Anderson et al. [4] provide a quantitative analysis on the Ethereum blockchain transactions from August 2015 to April 2016. The study they performed applies to smart contracts referenced before creation or to zombie contracts. They investigated the usage of unprotected commands - such as SUICIDE - in order to analyse the security of the contracts. In addition they analysed the code for finding patterns and similarities on contracts that were written by using tutorials and variants.

Bhargavan et al. [26] provide a novel formal approach for verifying the correctness of a smart contract. The authors implement a frameworks that compiles the contracts into F* and checks functional and safety correctness for detecting runtime errors. The EVM bytecode is also translate into F* in order to analyse low-level properties of the smart contract such as lower and upper bounds on the amount of gas required to run a transaction. Although the aforementioned approaches analyse smart contracts, they are profoundly different from our study. In fact, previous studies focus on understanding the security aspects of smart contracts while the goal of our study is to identify smart contract functionalities. More precisely, we want to find out opcodes that play a major role in practice, and identify the features that are not relevant in practice.

Other works cover financial aspects of blockchains and their impact on the current economy as well as introducing the blockchain technology in some existing application domains. In [7], Fenu et al. aim at finding the main factors that influence an ICO success likeliness. An ICO is a public offers of new cryptocurrencies in exchange of existing ones with the purpose to finance projects, mostly in the blockchain scenarios. First, they collect 1387 ICOs published on December 31, 2017 on icobench.com. From that ICOs they gather information to assess their quality and software development management. They also get data on the ICOs development teams.Second, they study, at the same dates, the financial data of 450 ICO tokens available on coinmarketcap.com, among which 355 tokens are on Ethereum blockchain. Finally, they define success criteria for the ICOs, based on the funds gathered and on the trend of the price of the related tokens.

Boceck and Stiller highlight various sets of functions, applications, and stakeholders which appear into smart contracts and put them into interrelated technical, economic, and legal perspectives [6].Examples of new applications areas are remittance, crowdfunding, or money transfer. An existing application is CargoChain, a Proof-of-Concept which shows how to reduce paperwork, such as purchase orders, invoices, bills of lading, customs documentation, and certificates of authenticity. Other popular non-financial areas with active blockchain projects are: fraud detection (Everledger, Blockverify, Verisart, Ascribe, Provenance, and Chronicled); global rights databases (Mediachain, Monegraph, and Ujo Music); identity management (Blockstack, UniquID, ShoCard, and SolidX); ridesharing (LaZooz and Arcade City); and document verification (Tierion and Factom).

Other works related to smart contract analysis are as follows. In [27], Kiffer et al. examine how contracts in Ethereum are created, and how users and contracts interact with one another. They find that contracts today are three times more likely to be created by other contracts than they are by users, and that over 60% of contracts have never been interacted with. Additionally they find that less than 10% of user-created contracts are unique and that there is substantial code re-use in Ethereum. In our study we have not considered the contract creation, but we have studied interactions among contracts by means of the *call* instruction. Our results shown that smart contracts rarely call other contracts. [28] presents EthIR, a framework for analysing Ethereum bytecode which aims at improving sound and automated reasoning about high-level properties of Ethereum smart contracts. EthIR decompiles EVM bytecode into a high-level representation in a rule-based form. More precisely, EthIR takes as input a low-level encoding provided in the CFGs generated by Oyente (a CFGs generator tool) and produces a rule-based representation (RBR) of the bytecode that enables the application of (existing) high-level analyses to deduct properties of EVM bytecode. The authors make use of EthIR to perform an automated resource analysis of existing contracts inside the blockchain. Resources are out of scope in our study, which only focuses on the high- and low-level linguistic constructs for writing smart contracts.

Bartoletti and Pompianu in [5] present an empirical analysis of Ethereum and Bitcoin smart contracts which is very similar to ours. They analyse the use of smart contracts with respect to their application domain in order to find design patterns in Ethereum smart contracts. They use a dataset of 811 verified smart contracts that were submitted to Etherscan.io between July 2015 and January 2017. The authors define a taxonomy of smart contracts based on their application domain to quantify their usage on each category and to study the correlation between patterns and domains. While this study categorises the smart contract transactions loaded in the blockchain on a certain time period, we only focus on verified smart contracts. We are interested in finding trends and patterns in their code. More precisely, our focus is not on transactions, but only on opcodes.

5. Concluding remarks

We present a novel study whose main goal is to understand how programmers use the linguistic instructions supported by Ethereum. We gathered and analysed verified Ethereum smart contracts that have been used in recent years. The contract source code and the related bytecode have been analysed in order to understand the control structures and core instructions that are used for smart contract definitions. Our analysis shows that most of the smart contracts do not use control loops but have a rather simple linear structure. More precisely, contracts are often used for reading and writing data on the blockchain. In addition, the identification of relevant instructions, at both high and low level, can contribute to lay the groundwork for defining new formalisms and domain specific languages supporting the development of smart contracts and related blockchain-based applications.

In the future, we plan to analyse and optimise the gas consumption of contracts. We also plan to benefit from our studies to provide formal methodologies to analyse smart contracts and develop domain-specific languages on top of Solidity.

Declaration of Competing Interest

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

Acknowledgments

The authors would like to acknowledge the financial support from the National Science Foundation of China (Grant No. 60934007, 61074060, 61104078).

References

[1] P. Shi, H. Wang, S. Yang, C. Chen, W. Yang, Blockchain-based trusted data sharing among trusted stakeholders in IoT, Software: Pract. Exp. (2019), doi:10.1002/spe.2739.
[2] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, 2018, (https://ethereum.github.io/yellowpaper/paper.pdf). [Online; accessed 08-December-2018].
[3] Solidity, 2018, (https://github.com/ethereum/solidity). [Online; accessed 09-December-2018].
[4] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, I. Weber, New kids on the block: an analysis of modern blockchains, 2016.
[5] M. Bartoletti, L. Pompianu, An empirical analysis of smart contracts: platforms, applications, and design patterns, Lect. Notes Comput. Sci. (2017), doi:10.1007/978-3-319-70278-0_31.
[6] T. Bocek, B. Stiller, Smart Contracts – Blockchains in the Wings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 169–184. 10.1007/978-3-662-49275-8_19
[7] G. Fenu, L. Marchesi, M. Marchesi, R. Tonelli, The ico phenomenon and its relationships with ethereum smart contract environment, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2018, pp. 26–32, doi:10.1109/IWBOSE.2018.8327568.
[8] M. Swan, Blockchain, O'Reilly Media, 2015.
[9] N. Satoshi, Bitcoin: a peer-to-peer electronic cash system., 2018, (https://bitcoin.org/bitcoin.pdf). [Online; accessed 09-December-2018].
[10] R.C. Merkle, A digital signature based on a conventional encryption function, in: C. Pomerance (Ed.), Advances in Cryptology — CRYPTO '87, Springer Berlin Heidelberg, Berlin, Heidelberg, 1988, pp. 369–378.
[11] B. Singhal, G. Dhameja, P.S. Panda, How Ethereum Works, Apress, Berkeley, CA, pp. 219–266. 10.1007/978-1-4842-3444-0_4
[12] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426, doi:10.1145/362686.362692.
[13] B. Vitalik, Ethereum: a next generation smart contract and decentralized application platform, 2018, (https://github.com/ethereum/wiki/wiki/White-Paper). [Online; accessed 08-December-2018].
[14] Installing the solidity compiler, (https://solidity.readthedocs.io/en/develop/installing-solidity.html). [Online; accessed 11-February-2020].
[15] M. Tan, The Ethereum block explorer, 2018, (https://etherscan.io). [Online; accessed 09-December-2018].
[16] Ethplorer - ethereum tokens explorer and data viewer. (https://ethplorer.io/a). [Online; accessed 11-February-2020].
[17] Blockchair - universal blockchain explorer and search engine, (https://blockchair.com/b). [Online; accessed 11-February-2020].
[18] Eip - 211, (https://github.com/ethereum/EIPs/blob/master/EIPS/eip-211.mdc). [Online; accessed 11-February-2020].
[19] Eip - 141, (https://github.com/ethereum/EIPs/blob/master/EIPS/eip-141.mdd). [Online; accessed 11-February-2020].
[20] Eip - 140, (https://github.com/ethereum/EIPs/blob/master/EIPS/eip-140.mde). [Online; accessed 11-February-2020].
[21] S. Bistarelli, I. Mercanti, F. Santini, An analysis of non-standard bitcoin transactions, in: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20–22, 2018, IEEE, 2018, pp. 93–96, doi:10.1109/CVCBT.2018.00016.
[22] S. Bistarelli, I. Mercanti, F. Santini, A suite of tools for the forensic analysis of bitcoin transactions: Preliminary report, in: G. Mencagli, D.B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R.R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J.D.G. Sánchez, S.L. Scott (Eds.), Euro-Par 2018: Parallel Processing Workshops - Euro-Par 2018 International Workshops, Turin, Italy, August 27–28, 2018, Revised Selected Papers, Lecture Notes in Computer Science, 11339, Springer, 2018, pp. 329–341, doi:10.1007/978-3-030-10549-5_26.
[23] S. Bistarelli, F. Santini, Go with the -bitcoin- flow, with visual analytics, in: Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29, - September 01, 2017, ACM, 2017, pp. 38:1–38:6, doi:10.1145/3098954.3098972.
[24] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: M. Maffei, M. Ryan (Eds.), Principles of Security and Trust, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 164–186.
[25] K. Delmolino, M. Arnett, A. Kosba, A. Miller, E. Shi, Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab, in: Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC„ 9604, 2016, pp. 79–94, doi:10.1007/978-3-662-53357-4_6.
[26] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, S. Zanella-Béguelin, Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, in: PLAS '16, ACM, New York, NY, USA, 2016, pp. 91–96, doi:10.1145/2993600.2993611.
[27] L. Kiffer, D. Levin, A. Mislove, Analyzing ethereum's contract topology, in: Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31, - November 02, 2018, ACM, 2018, pp. 494–499.
[28] E. Albert, P. Gordillo, B. Livshits, A. Rubio, I. Sergey, Ethir: A framework for high-level analysis of ethereum bytecode, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2018, pp. 513–520.