Analysis of Ethereum Smart Contracts and Opcodes

Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, and Francesco Tiezzi

Abstract Much attention has been paid in recent years to the use of smart contracts. A smart contract is a transaction protocol that executes the terms of an agreement. Ethereum is a widely used platform for executing smart contracts, defined by using a Turing-complete language. Various studies have been performed in order to analyse smart contract data from different perspectives. In our study we gather a wide range of verified smart contracts written by using the Solidity language and we analyse their code. A similar study is carried out on Solidity compilers. The aim of our investigation is the identification of the smart contract functionalities, i.e. opcodes, that play a crucial role in practice, and single out those functionalities that are not practically relevant.

1 INTRODUCTION

In recent years, increasing attention has been drawn towards the use of smart contracts for various application areas, such as public registries, registry of deeds, or virtual organisations. Smart contracts are a digitalised version of traditional contracts which should enhance security and reduce the transaction costs that are related to contracting. One of the most prominent platform for smart contract definition and execution is Ethereum[1] [14]. This is a blockchain-based distributed computing platform that allows to create smart contracts by using a Turing-complete language.

Various studies have been carried out to analyse smart contracts data from different angles. [1] analyses smart contracts in order to detect zombie contracts, while [3]

Stefano Bistarelli
University of Perugia, e-mail: stefano.bistarelli@unipg.it

Gianmarco Mazzante · Matteo Micheletti · Leonardo Mostarda · Francesco Tiezzi
University of Camerino, e-mail: {name.surname}@unicam.it

[1] https://www.ethereum.org/

inspects the usage of contracts with respect to their application domain. Finally [8, 10] study the contracts from technical, economic, and legal perspectives.

In this paper we present a study that gathers ten of thousands of verified Ethereum smart contracts that has been written by using the Solidity language[2]. A contract is *verified* when a proof that it can be obtained by compiling a (Solidity usually) source code can be provided. Our study analyses the hexadecimal bytecode instructions of smart contracts, by referring to their equivalent human readable format called *opcode*. We have analysed the opcodes frequency distribution for the considered contracts and for various compilers, in different period of times. We have discussed in details why some opcodes are more frequent than others, while some others are not used at all.

Our study permits to gain a precise understanding on how the linguistic constructs supported by Ethereum have been used in practice by contract programmers in the last two years. The results of our analysis can enable some simple, yet effective, checks on contracts concerning anomalous usage of opcodes (e.g., presence of opcodes never used in the practice). In addition, our study permits to identify a set of core features laying the groundwork for defining, as a long term goal, new formalisms and domain specific languages (DSLs) supporting the development of applications based on smart contracts. On the one hand, formalisms pave the way for the use of formal techniques for verification. On the other hand, frequently used opcodes can be linked to a set of widely used programming patterns related to specific domains of application. Such information can be exploited to devise different DSLs to more conveniently define smart contracts for specific application contexts.

The rest of the article is organised as follows. Section 2 outlines the basic concepts of Ethereum; Section 3 overviews the experimental setup that has been used to gather smart contract data and discusses the result of our analysis; Section 4 reviews the related work; finally, Section 5 concludes the paper and outlines future work.

2 Ethereum Background

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 [12]. This allows transactions to have public witnesses thus making some attacks (such as modification) more difficult. In this paper we focus on Ethereum [14] which is a blockchain-based distributed computing platform that allows the definition of **smart contract**, i.e., scripting functionality.

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

[2] https://github.com/ethereum/solidity

Virtual Machine (EVM) code. The instructions of the hexadecimal bytecode representation are often mapped into a human readable form which is referred to as **opcode**. An exhaustive list of EVM bytecodes and opcodes can be found in the Ethereum Yellow Paper [14]. High-level programming languages are available to write smart contracts, whose code is compiled into EVM bytecode in order to be executed in the blockchain. Currently, the most prominent language to write Ethereum smart contracts is **Solidity**. Two different Solidity compilers are available: `solc`[3] and `solc-js`[4]. The former is written in C++, while the latter in Javascript. Our study only considers `solc`, which is the official and most maintained compiler for writing smart contracts.The `solc` compiler was released on the 21*st* of August 2015 version 0.1.2 and is currently at version 0.5.1 released on the 3*rd* of December 2018.

A smart contract is added to the blockchain as a transaction. **Explorers** can be used to read code and transactions of smart contracts. An explorer is a website that tracks all the information inside the blockchain and shows it in a human readable form. Explorers can perform various analysis on the blockchain and allow the *verification* of contracts. This is a three-step process where: i) the author publishes the compiled code of the contract in the blockchain, then ii) she loads the original source code and the version of the compiler into the explorer, and finally iii) the explorer marks the contract as verified when the compiled code can be indeed obtained from the source code. This process cannot be performed by only considering the blockchain which does not store any source code nor compiler information.

3 ANALYSIS OF SMART CONTRACTS AND OPCODES

This section overviews the experimental setup that has been used to gather various smart contract data and the result of its analysis.

3.1 Experimental setup

We have used Etherscan[5] in order to retrieve smart contracts information. Although several explores are available (e.g., Etherchain.org[6], Ethplorer[7] and Blockchair[8]) Etherscan is the only one that allows to obtain verified smart contracts. Our study considers the following smart contract information:

- the Ethereum unique address of the contract;

[3] https://github.com/ethereum/solidity
[4] https://github.com/ethereum/solc-js
[5] https://etherscan.io/ [13]
[6] https://www.etherchain.org/
[7] https://ethplorer.io/
[8] https://blockchair.com/ethereum

- the translation of the smart contract from its bytecode form into the opcode one;
- the Solidity compiler version that has been used to compile the smart contract;
- all dates where at least a smart contract was verified.

We have obtained the data of all contracts that have been verified between October 2016 and May 2018 (the date at which our data collection activity ended). Very few contracts were verified before October 2016 thus we have not considered these contracts.

We have implemented a Java program, available online[9], 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 an Etherscan API[10] that outputs the smart contract source in an opcode form. Our Java tool analyses the contract opcodes and store in a JSON format the address of the smart contract, the compiler version used to compile the contract and all contract opcodes with the related frequency (i.e., the number of times the opcode appears inside the contract).

3.2 Results

In this section we present the quantitative analysis that has been performed on the smart contract data we have described in Section 3.1.

3.2.1 Opcode frequency of all verified contracts

Table 1 reports the number of verified contracts per month and the total number of opcode these contracts used. Notice how the number of contracts exponentially incfrease from 2016 to today.

The histogram of Figure 1 instead, displays on the x-axis the hexadecimal value of all opcodes (the entire list of opcodes can be found at [14]) while on the Y-axis the global frequency of each opcode. This is obtained by summing up the number of times each opcode appears inside each contract. It is worth noticing that only 5 opcodes have a global frequency that is more then 5% of the sum of all global frequencies (see Figure 2 that represents the global frequencies of Figure 1 with a logarithm scale).

3.2.2 Frequently used opcodes

In the following we discuss why some of the opcodes are frequently used while others do not appear very often.

[9] https://github.com/GianmarcoMazzante/opcodeSurv
[10] http://etherscan.io/api?module=opcode&action=getopcode

MONTH	VERIFIED CONTRACTS	OPCODE 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

Table 1 Contracta count and opcode occurrences per month

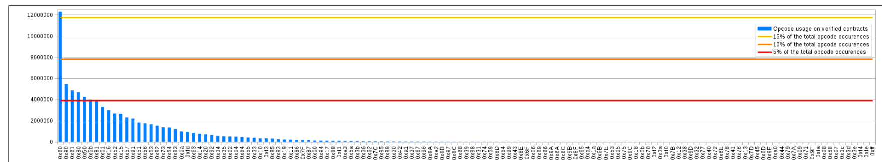

Fig. 1 Histogram of opcode count on verified contracts

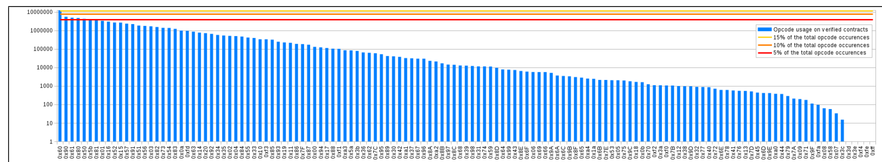

Fig. 2 Histogram of opcode count on verified contracts (log scale)

Table 2 summarises the ten most frequently used opcodes. Most of these opcodes are related to stack management operations, such as swap, push and pop, since the Ethereum Virtual Machine has a *stack architecture*. The PUSH1 operation adds 1-byte value into the stack. This is the most frequent operation since it is a basic stack management operation and every contract starts with the sequence: `PUSH1 0x60 PUSH1 0x40 MSTORE`. This also explains the presence of the memory storing opcode MSTORE amongst the most used opcodes. While there are various `PUSHs` (Table 3

	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 2 The ten most used opcodes of verified contracts

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

Table 3 First five most used push opcodes

shows the First five most used PUSH opcodes) that differ from the amount of bytes they push into the stack, there is only one POP opcode that works equally on every element of the stack. The PUSH and POP behaviour does not ensure that the number of POP is the same as the number of all PUSH. In fact, the sum of all PUSH operations is 19788857 while the number of POP ones is 4247835 (less than one quarter of the previous number). This is consequence of the behaviour of various opcodes that automatically pop and push parameters into the stack. For instance the MUL opcode does not just insert the result of a multiplication on the top of the stack but it also removes the two factors of the operation, performing a double pop and a single push behind-the-scenes.

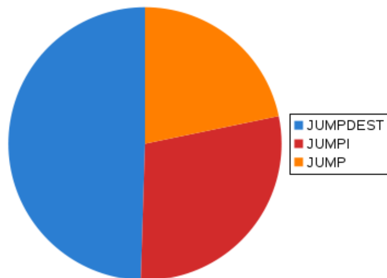

Fig. 3 Pie chart of JUMP,JUMPI,JUMPDEST opcodes occurences

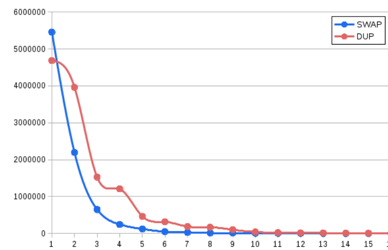

Fig. 4 Line chart comparison of SWAP and DUP occurences

Another opcode that is widely used is the JUMPDEST one which is used to specify the destination of the jump (JUMP) and the unconditional jump (JUMPI). These are used to translate i) loops, ii) if statements and iii) switches from the smart contract source code which justify the high frequency of JUMPDEST amongst the most used opcodes (see Figure 3 for the proportion of JUMPDEST with respect to the other jump operations). We can also find the ADD opcode in the list of most used opcodes.

This is not only used as an algebraic operation by the developers, but also as an internal command to manage array positions. In other words, `ADD` is used when adding an incremental value to the offset of an array from the `MSTORE` opcode. The opcodes `PUSH20` and `PUSH32` are also frequently used since *contracts* and *accounts* are uniquely identified by a 20-byte address while *transactions* are identified by a 32-byte address. Differently, the frequency of `SWAPs` and `DUPs` opcodes decreases as the number of bytes increase. Figure 4 shows the frequency of these opcodes as the number of bytes increase from 1 to 16.

3.2.3 Less frequently used opcodes

The behaviour of less frequently used opcodes can be often simulated by using other opcodes. For instance, the `RETURNDATASIZE` code has been introduced with the Ethereum Improvement Proposal (*EIP*) number 211[11] and can be used to get the size of the output data of the previous external call. The `RETURNDATASIZE` opcode can be simulated by using a sequence of various opcodes (see the *EIP-211* proposal for details). In the same way the `RETURNDATACOPY` can be simulated by using other opcodes.

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

Table 4 Not used 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

Table 5 Occurences of environmental information opcodes on verified contracts

There are various opcodes (see Table 4) which are rarely used since they introduce a peculiar variation of an existing opcodes. For instance the `DELEGATECALL` opcode is similar to the `CALL` one except for the context used in the call (see [14] for details). The `INVALID` opcode was introduced with the *EIP-141* proposal[12] and it is similar to

[11] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-211.md
[12] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-141.md

the `REVERT` opcode that was introduced in the *EIP-140* proposal[13]. Both the opcodes abort the code execution but `INVALID` drains all the remaining gas of the caller while `REVERT` does not. The `INVALID` behaviour is never used since smart contract never drain all the remaining gas. In the same way the `SELFDESTRUCT` opcode transfers all the ether between two accounts and destroy the contract[14]. This behaviour is never used.

There are also various environmental opcodes which are used to get financial information. For instance the `BALANCE` and `GASPRICE` opcodes are used to get the residual balance and the gas price, respectively. Table 5 shows that some of these opcodes are rarely used. For instance the `GASPRICE` opcode sets the gas price for transactions. This setting is rarely done since the default gas price is often used by smart contracts.

3.2.4 Opcodes and contracts count over the time

In this section we analyses the total count of opcodes of verified contracts. The X-axis of Figure 5 has a wide range of different months while the the Y-axis shows the following information:

- the number of contracts that have been verified
- the total count of opcodes that are contained inside verified contracts

The 10th of October 2016 corresponds to the release date of the Solidity version *0.4.2*. Figure 5 confirms that the usage of smart contracts raises in popularity.

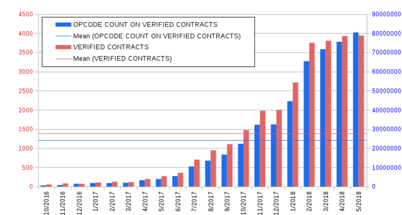

Fig. 5 Histogram of opcode count per month and contract count per month

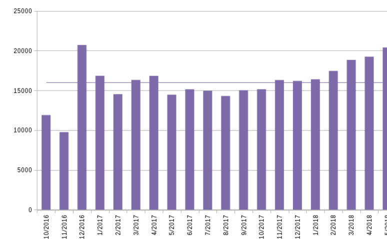

Fig. 6 Opcodes over contract count per month

The X-axis of Figure 6 has a wide range of different months while the Y-axis 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 the same range of chart 5 to make the two charts comparable. Figure 6 clearly shows that contracts are increasing in size.

[13] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-140.md

[14] The address will remain but any interaction with it will only waste gas or ether

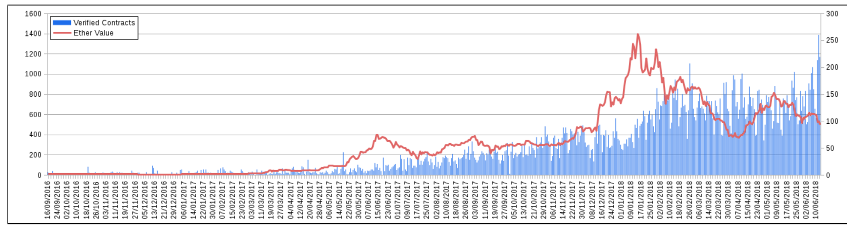

Fig. 7 Histogram of verified contracts per date and line chart of Ether value over time

The trend of contract deployment over the time can be better understood by considering Figure 7. This contains the number of verified contracts for each day together with a line chart representing the value of the *ether* cryptocurrency. We can easily see that as ether increased (it happened almost in parallel with the bitcoin) an increasing number of users were writing Ethereum smart contracts.

3.2.5 Different versions of Solidity compilers

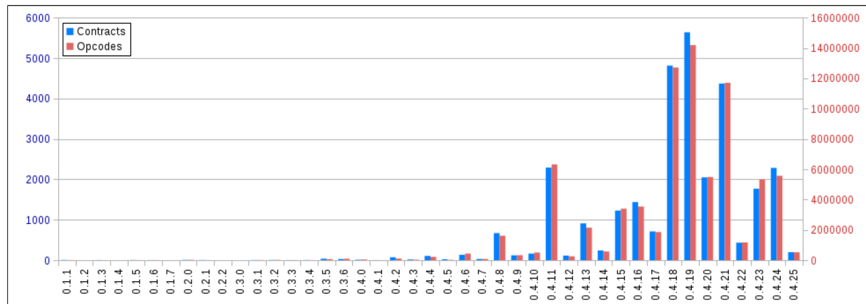

Fig. 8 Double histogram of opcode count and contract deployment on different version of Solidity

Figure 8 shows different versions of Solidity compilers (from 0.1.1 to 0.4.25), having on two different series the total number of contract and opcode calls respectively, in order to have a comparative view. This shows that the compiler version v0.4.19 is the most used. It also shows that the Solidity version usage follows the Ethereum trend both in terms of platform popularity and value of the currency (depicted on Figure 7).

Figure 9 considers the Solidity compiler version v0.4.19 and shows the number of occurrences of each opcode in the source code of the Go implementation. It shows that the most used are the stack management opcodes, among with memory and storage management opcodes.

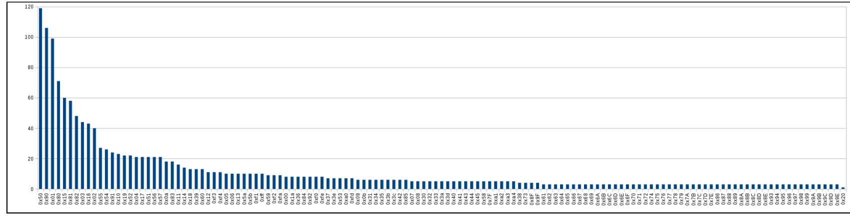

Fig. 9 Histogram of opcode occurrences on Solidity v0.4.19 source code

4 Related Work

In the literature there is a limited amount of works on studies of Ethereum smart contracts and theirs analysis and statistics, with respect to other well-known blockchains like Bitcoin [5, 6, 7].

Some of these studies focus on security issues. Atzei, Bartoletti and Cimoli provide a survey on attacks to Ethereum smart contracts [2]. 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 [9]. 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. provide a quantitative analysis on the Ethereum blockchain transactions from August 2015 to April 2016 [1]. Their investigation focuses on smart contracts with a particular attention to zombie contracts and contracts referenced before creation. They performs a security analysis on that contracts to check the usage of unprotected commands (like SUICIDE). They also inspects the contracts code to look for similarities which could result from a contract being written by following tutorials or from testing and variants. In the aforementioned works, correctness of smart contracts is checked by inspecting source code for known pattern. A more formal approach is proposed by Bhargavan et al. [4], who provide a framework to verify Ethereum smart contracts by *(i)* compiling them into F*, to check functional correctness and safety towards runtime errors, and *(ii)* decompiling EVM bytecode into F* code to analyse low-level properties (e.g. bounds on the amount of gas required to run a transaction). Even if the works described above report analyses of smart contracts, these studies significantly differ from ours, because they focus on security aspects while the aim of our study is to identify the smart contract functionalities, i.e. opcodes, that play a crucial role in practice, and single out those functionalities that are not practically relevant.

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 [10], Fenu et al. aim at finding the main factors that influence an ICO success likeliness. 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 highlights various set of functions, applications, and stakeholders which appear into smart contracts and put them into interrelated technical, economic, and legal perspectives [8]. 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.

The work in the literature closest to ours is the one by Bartoletti and Pompianu in [3]. They perform an empirical analysis of Ethereum and Bitcoin smart contracts, inspecting their usage according to their application domain and then focusing on searching for design patterns in Ethereum contracts. Their analysis on Ethereum contracts starts from a dataset of 811 verified smart contracts 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. Our work differs from theirs on some important aspects. In fact, they study and categorise the smart contratcs transactions loaded in the blockchain on a certain time period. Instead, we only concentrate on verified smart contracts, because we are interested to find trends and patterns in their code. Our focus indeed is not on transactions, but on opcodes.

Also, In [11] Kiffer, Levin, and Mislove 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.

5 Conclusion and future work

In this paper we gathered and analysed the verified Ethereum smart contracts used in the last two years. In particular, we identified most and less used opcodes. As future work, we plan to better investigate the correlation between opcodes usage and the corresponding Solidity code to identify relevant patterns, and to extend our study to non-verified contracts.

We also plan to study and analyse the gas consumption of the contracts in order to try to optimize smart contract compiler on this direction. Finally, as longer term goal, we intend to exploit these studies to i) support formal analyses on smart contracts and ii) define DSLs as on top of Solidity for specific application domains.

References

1. L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber. New kids on the block: an analysis of modern blockchains, 2016.
2. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In M. Maffei and M. Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
3. M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. *Lecture Notes in Computer Science*, 03 2017.
4. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.
5. S. Bistarelli, I. Mercanti, and F. Santini. An analysis of non-standard bitcoin transactions. In *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*, pages 93–96. IEEE, 2018.
6. S. Bistarelli, I. Mercanti, and 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, and S. L. Scott, editors, *Euro-Par 2018: Parallel Processing Workshops - Euro-Par 2018 International Workshops, Turin, Italy, August 27-28, 2018, Revised Selected Papers*, volume 11339 of *Lecture Notes in Computer Science*, pages 329–341. Springer, 2018.
7. S. Bistarelli and 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*, pages 38:1–38:6. ACM, 2017.
8. T. Bocek and B. Stiller. *Smart Contracts – Blockchains in the Wings*, pages 169–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.
9. K. Delmolino, M. Arnett, A. Kosba, A. Miller, and 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,*, volume 9604, pages 79–94, 02 2016.
10. G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 26–32, March 2018.
11. L. Kiffer, D. Levin, and 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*, pages 494–499. ACM, 2018.
12. M. Swan. *Blockchain*. O'Reilly Media, 2015.
13. M. Tan. The Ethreum block explorer. https://etherscan.io, 2018. [Online; accessed 09-December-2018].
14. G. Wood. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://ethereum.github.io/yellowpaper/paper.pdf, 2018. [Online; accessed 08-December-2018].