Design Patterns for Gas Optimization in Ethereum

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

Giuseppe Destefanis
Computer Science
Brunel University
London, United Kingdom
giuseppe.destefanis@brunel.ac.uk

Giulio Barabino, Danilo Tigano
DITEN
University of Genova
Genova, Italy
giulio.barabino@unige.it,
danilo.tigano@unige.it

***Abstract*— Blockchain technology is an emerging technology that allows new forms of decentralized architectures, designed to generate trust among users, without the intervention of mediators or knowledge between the parties. Since 2015, thanks to the introduction of Smart Contracts by Ethereum, it is possible to run programs on the blockchain, greatly extending the potential of this technology. The programming of Smart Contract, through the Solidity language is different from the traditional one. First of all, any action that requires to modify the blockchain costs gas, which corresponds to a fraction of the currency used by that given blockchain, and therefore to real money. Gas optimization is a unique challenge in this context and has obvious implications. This document aims to provide a set of design patterns and tips to help gas saving in developing Smart Contracts on Ethereum. The provided patterns are presented divided into five main categories, based on their features.**

***Index Terms*—Blockchain, smart contracts, Ethereum, gas saving, gas optimization, design patterns.**

I. Introduction

Recently, we witnessed the advent of blockchain technology. Initially a means of cryptocurrency management, it later evolved into the concept of Smart Contracts (SCs), real programs running on the blockchain. The most important ecosystem for the development and distribution of SCs, is currently the Ethereum blockchain, through the Solidity programming language. It is enjoying an increasing popularity, and several applications in the real world have already been developed [18] [19] [20].

The development of SCs has proved being very different from traditional software development, and involves a series of new problems and challenges. The issue of SC security, for example, is crucial as many of them are critical and deal with large sums of money [21] [22]. Furthermore, programming languages and virtual machines for the development on blockchain are still limited, as well as programming resources and development environments.

Other issues derive from the gas mechanism, typical and unique of this context, through which the execution of SCs in platforms such as Ethereum is managed. According to a study by Zou et al. [1], more than 86% of those surveyed on the issues related to the development of SC, declared to pay close attention to gas consumption; the main reasons being that gas corresponds to real money, and that transactions can fail due to an insufficient amount of gas. Furthermore, if on the one hand it is not easy to make an accurate estimation of the gas necessary for the execution of a specific SC, on the other hand it is not a good idea to set a high gas limit, because if there is a bug all the gas will be wasted. Our work tries to cope with this problem, outlining a series of design patterns and advices for the development of SCs, with a view to saving gas.

The followings of this paper are structured as follows: in Section II we provide a brief background of the Ethereum blockchain and the gas mechanism through which the running of SCs are managed. We also provide a definition of design pattern. In section III we provide the patterns and tips that are the subject of this work. Section IV concludes the paper.

II. Background

A. Ethereum and Smart Contracts

Ethereum is a platform for developing decentralized and reliable applications. Like other blockchains, it is extended globally, it is composed of a peer-to-peer network, there is no central authority and there is no single point of failure. The software associated with the transactions, and running on the nodes of the blockchain, is called Smart Contract (SC). A SC is typically written in a specialized high-level language, called Solidity. Most SCs are currently used to manage cryptocurrencies or tokens. Since Ethereum is the first and most popular blockchain able to run SCs [2] [3], in this work we will refer mainly to it.

In Ethereum, the SCs are created by special transactions. They can send messages to other SCs and inherit from other ones. A SC has a state, stored permanently in the "storage" variables of the blockchain. Once installed on the blockchain, it can no longer be undone or deleted. A SC is provided with public functions, which can be called during its creation or sending a transaction to it. Calling a SC public function has a different behavior depending on whether the call comes from a generic address or from another contract. In the first case, an Ethereum transaction is generated, which has an additional fixed cost, it is managed by the miners and takes a not negligible time, at least 10-15 seconds, to be performed. In the second case, the function is performed instantly, without delay, and the cost is strictly related to its execution. A SC can also send or receive Ethers to or from an address or another SC. If one of the functions of a SC, returns a value without changing the

state, or performing significant computations, it will cost no gas and return the result in real time. The gas must be paid in the cryptocurrency associated with the blockchain, in the case of Ethereum in Ethers. The required gas is proportional to the amount of computational power required to perform the operation.

A SC should be:

- **deterministic**: given the same input it must always provide the same output. To this purpose, it must not call non-deterministic functions, and it must not use non-deterministic data resources.
- **terminable**: by definition the SC must be able to finish within a certain time limit. To ensure this, there are several methods: use a timer, so if the execution lasts more than a given time limit it is externally interrupted; use incomplete Turing blockchains, which are not allowed to enter infinite cycles, like Bitcoin blockchain [14] [15] [16]; use the step meter method, in which a program is interrupted once a certain number of steps have been completed; or charge a cost to each operation and interrupt the execution if the prepaid commission has been reached. The last one is the case of Ethereum.
- **isolated**: each contract must be kept isolated to avoid corrupting the entire system in the event of a virus or bug.
- **immutable**: once deployed on the blockchain, a SC cannot be changed. However, it can be disabled forever. This property, together with the ability to publish the source code of the SC, guarantees the highest level of transparency and trust.

B. Ethereum Virtual Machine and Gas Mechanism

The Ethereum Virtual Machine (EVM) is the virtual machine on which all SCs work in Ethereum. It is based on 256 bits words, and is Turing Complete. It is simple but powerful. All SCs are sets of bytecode instructions, which are executed in sequence. However, the bytecode allows jumps, thus enabling a Turing complete behaviour.

In Solidity, when a SC is compiled, it is converted into a sequence of "operation codes", also known as opcodes. These are identified by abbreviations, for example ADD for addition, MUL for multiplication, etc. All the opcodes and their description are available in the so-called Ethereum yellow paper [2], the document which first described this system. Each opcode has a predetermined amount of gas assigned to it, which is a measure of the computational effort required to perform that particular operation. Bytecodes are similar to opcodes but are represented by hexadecimal numbers. The EVM executes bytecodes.

Table I [2] [17] shows the amount of fee (gas) due for the various operations in Ethereum; in the yellow paper, specifically in the appendix g, is available a complete table. For example, applying the blockhash operation requires 20 gas units, while the ADD operation requires 3 gas units. Is worth noting that a SC consists of numerous operations, some of them consume much more than a simple arithmetic operation.

The gas is therefore a fee for the execution of a computation, paid by who sends the transaction that triggers the computation.

TABLE I. GAS COSTS IN ETHEREUM

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
JUMP	8	Unconditional Jump
SSTORE	5,000/20,000	Storage operation
BALANCE	400	Get balance of an account
CALL	25,000	Create a new account
CREATE	32,000	Create a new account

By default, the minimum amount of gas for an operation that affects the state of the EVM, is 21000 gas. For example, this is the amount required to send Ethers from one account to another. To execute a function of a SC this amount will be 21000 gas plus the gas needed to perform each of the required opcodes. An exception to this behavior is when the called function is read-only and simple. Such a function is called a view function, and its execution is free and immediate, because it does not change the state of the EVM. So, calling a _view_ function within a Call in a local node does not cost gas, while calling the same function from a deployed SC within a Transaction costs gas.

Before performing an operation, the user sets a gas limit, which corresponds to the maximum amount of gas that he/she wants to pay. If the gas actually required by the operations overcomes the gas limit, that operation will be aborted, each change will be rolled-back, but all the gas will be spent and therefore lost. If the user sets a gas limit higher than the one required, the operation will be carried out, and only the used gas will be spent. It is almost impossible for a user to know in advance how much gas a transaction will exactly require. However, it is not wise to set a gas limit too large, because in the case of a bug or error in the SC, there is a risk of exceeding this limit and losing all the gas.

The total fee actually paid, is equal to the total gas used multiplied by the "gas price". The gas price is not fixed but set by the user. Miners prefer transactions with higher fees; transactions whose fees are too low may never be included in the blockchain. On the contrary, setting a very high gas fee guarantees that the transaction will be executed quickly. At ethstats.net website [4] you can find the suggested gas price in real-time. At the time of writing this article the suggested gas price is 8 gwei, which corresponds to 0.000000008 Ether. Note that this number are constantly changing.

There is a gas limit also for every block. It corresponds to the maximum amount of gas that all the transactions included in the block can consume. This number also determines the maximum number of transactions to include in the block. Even the "block gas limit" is not fixed but is determined by the miners. The higher this limit is, the more the miners will earn in terms of transaction fees; however, the computational load to compute the block transactions will also increase.

Currently, the recommended gas prices related to the transaction execution frequency, based on https://ethgasstation.info/ website [5], are shown in Table II:

TABLE II. RECOMMENDED GAS PRICES IN ETHEREUM

Recommended Gas Price	Minimum transaction cost	Transaction Speed
2 gwei	0,006 US$	Safe Low <30 minutes
5 gwei	0,015 US$	Standard <5 minutes
10 gwei	0,030 US$	Fast <2 minutes

This table shows, for each gas price, the transaction cost corresponding to 21,000 gas – which is the minimum gas needed to execute a transaction – and the average time to have the transaction accepted. The costs refer to the present Ether value of about 125 US$.

Gas mechanism serves as an incentive system both for miners, to spend hardware and electricity costs to validate transactions, and against attackers, who would spend money to perform an attack.

Figure 1 shows the average Ethereum gas price from Ethereum inception, taken from Etherscan.io website [6], the leading block explorer for the Ethereum blockchain. It can be seen that the gas price is relatively stable, except in the first weeks, and for some occasional peaks.

The gas mechanism has the advantage of providing a good incentive for the miners and a disincentive for the attackers. However, it has the disadvantage that operations can become very expensive.

Furthermore, although the EVM is designed to be Turing-complete, in practice the limitations on the amount of gas limit the set of computable functions [23] [24].

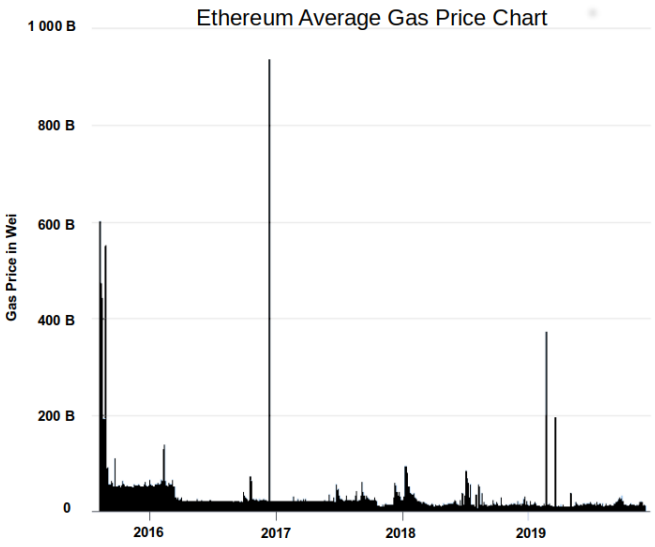

FIGURE 1. ETHEREUM AVERAGE GAS PRICE

C. Memory Usage in Ethereum

When developing a SC, different types of storage are available [2]:

- **Stack** (volatile stack access): it contains small local variables. It is almost free to use but can hold a limited amount of data. All operations are performed on the stack. It can be accessed with different instructions, such as PUSH, POP, COPY, SWAP, …
- **Memory** (volatile memory access): it contains temporary values, generated during the execution. It is erased at the beginning of every function call. It can be accessed with three instructions: MLOAD, MSTORE, MSTORE8.
- **Storage** (persistent memory): it contains all the SC state variables. Every contract has its own persistent storage. A contract can strictly read or write only its own storage. The opcodes to operate with it are: SLOAD and STORE.
- **Calldata**: special data location that contains the function arguments, only available for external function call parameters.
- **Event Log**: a special memory in the blockchain were data related to the Events raised by SCs are stored. These data cannot be accessed by SCs, but only by external applications.

All operations used to manipulate memories cost gas. The most expensive ones are those affecting the Storage. This is due to the fact that the data are permanently stored in a database replicated across tens of thousands of nodes.

D. Design Patterns

A Design Pattern is a typical solution to a recurring design problem.

In 1979 the architect Cristopher Alexander noted that in building construction there are recurring design problems, and thus introduced the concept of pattern in his book entitled "The Timeless Way of Building" [7]. In 1987 Cunningham and Beck used Alexander's ideas to develop a pattern language for designing software architecture, that they presented at the OOPSLA Conference (Object-Oriented Programming Systems, Languages and Applications) [8]. However, the patterns were formally consecrated as an important innovation for OO (Object-Oriented) design only in 1994 with the publication of the book entitled "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Helm, Johnson and Vlissides, who began to create a catalogue of design patterns since 1990 [9].

A pattern is described by:

- a *Name*: it must be simple, focused on the issue and easy to remember. It allows you to unambiguously refer to the pattern;
- the *Problem*: it describes the issue that the pattern resolves. It may include a list of conditions that must be met for the solution to be valid.

- the proposed *Solution*: it clarifies what to do in order to overcome the *Problem*. It does not describe a specific concrete implementation, but an abstract and precise enough solution, that can be immediately applied, and reused multiple times.

Our pattern collection tries to cope with the issue of gas spending in Ethereum; it includes 24 patterns, presented in five categories, based on their features.

III. Gas Saving Design Patterns

In this chapter we present a collection of patterns for the design and development of Smart Contracts, with the aim of saving gas. So far, only a few efforts have been made to collect and classify patterns related to this issue, mainly published on blogs or discussion lists [10] [11] [12][13].

As already discussed, during the development and execution of a SC there are costs, calculated in gas/gwei; we can divide these costs in fixed costs related to the creation of the SC or to the sending of the transaction; costs related to the permanent storage of the SC state, in storage; costs relating to the storage of temporary variables, necessary for the execution of functions, in memory; costs related to the execution of the operations.

We collected 24 design patterns, which we assigned to 5 categories. The categories resulted from analyzing the collected patterns, and observing in which context they are used – for instance, limiting the storage (which is ludicrously costly), or limiting the computations made by function execution. Some patterns might belong to more than one category. We assign it to the most appropriate one.

Applying patterns to an application can better align it with the unique properties provided by the blockchain, overcoming its limitations. The presented patterns are based on multiple sources, such as the Solidity documentation, the study of blogs and discussion forums on Ethereum on the Web, and the examination of existing SCs.

A. External Transactions

This category includes patterns related to the creation of contracts and the sending of transactions from external addresses, including JavaScript applications, using Web3.js standard library.

Name:	**Proxy**
Problem:	SCs are immutable. If a SC must be changed due to a bug or a needed extension, you must deploy a new contract, and also update all SCs making direct calls to the old SC, thus deploying also new versions of these. This can be very expensive.
Solution:	Use Proxy delegate pattern. Proxy patterns are a set of SCs working together to facilitate upgrading of SCs, despite their intrinsic immutability. A Proxy holds the addresses of referred SCs, in its state variables, which can be changed. In this way, only the references to the new SC must be updated.

Name:	**Data Contract**
Problem:	When a SC holding a significant amount of data must be updated, also all its data must be copied to the newly deployed SC, consuming a lot of gas.
Solution:	Keep the data in a separate SC, accessed by one or more SC, using the data and holding the processing logic. If this logic must be updated, the data remain in the Data Contract. This pattern usually is included also in the implementations of the Proxy pattern.

Name:	**Event Log**
Problem:	Often events maintain important information about the system, which must be later used by the external system interacting with the blockchain. Storing this information in the blockchain can be very expensive, if the number of events is high.
Solution:	If past events data are needed by the external system, but not by SCs, let the external system directly access the Event Log in the blockchain. Note that this Log is not accessible by SCs, and that if the event happened far in time, the time to retrieve it may be long.

B. Storage

This category includes patterns related to the usage of Storage for storing permanent data.

Name:	**Limit Storage**
Problem:	Storage is by far the most expensive kind of memory, so its usage should be minimized.
Solution:	Limit data stored in the blockchain, always use memory for non-permanent data. Also, limit changes in storage: when executing functions, save the intermediate results in memory or stack and update the storage only at the end of all computations.

Name:	**Packing Variables**
Problem:	In Ethereum, the minimum unit of memory is a slot of 256 bits. You pay for an integer number of slots even if they are not full.
Solution:	Pack the variables. When declaring storage variables, the packable ones, with the same data type, should be declared consecutively. In this way, the packing is done automatically by the Solidity compiler. (Note that this pattern does not work for Memory and Calldata memories, whose variables cannot be packed.)

Name:	**Packing Booleans**
Problem:	In Solidity, Boolean variables are stored as uint8 (unsigned integer of 8 bits). However, only 1 bit would be enough to store them. If you need up to 32 Booleans together, you can just follow the Packing Variables pattern. If you need more, you will use more slots than actually needed.
Solution:	Pack Booleans in a single uint256 variable. To this purpose, create functions that pack and unpack the Booleans into and from a single variable. The cost of running these functions is cheaper than the cost of extra Storage.

C. Saving Space

This category includes patterns related to saving space both in Memory and Storage.

Name:	**Uint* vs Uint256**
Problem:	The EVM run on 256 bits at a time, thus using an uint* (unsigned integers smaller than 256 bits), it will first be converted to uint256 and it costs extra gas.
Solution:	Use unsigned integers smaller or equal than 128 bits when packing more variables in one slot (see Variables Packing pattern). If not, it is better to use uint256 variables.

Name:	**Mapping vs Array**
Problem:	Solidity provides only two data types to represents list of data: arrays and maps. Mappings are cheaper, while arrays are packable and iterable.
Solution:	In order to save gas, it is recommended to use mappings to manage lists of data, unless there is a need to iterate or it is possible to pack data types. This is useful both for Storage and Memory. You can manage an ordered list with a mapping using an integer index as a key.

Name:	**Fixed Size**
Problem:	In Solidity, any fixed size variable is cheaper than variable size.
Solution:	Whenever it is possible to set an upper bound on the size of an array, use a fixed size array instead of a dynamic one.
Name:	**Default Value**
Problem:	It is good software engineering practice to initialize all variables when they are created. However, this costs gas in Ethereum.
Solution:	In Solidity, all variables are set to zeroes by default. So, do not explicitly initialize a variable with its default value if it is zero.

Name:	**Minimize on-chain data**
Problem:	The gas costs of Storage are very high, and much higher than the cost of Memory.
Solution:	Minimize on-chain data. The less data you put on-chain in Storage variables, the less your gas costs. Store on-chain only critical data for the SC and keep all possible data off-chain.

Name:	**Explicitly mark *external* function**
Problem:	The input parameters of *public* functions are copied to memory automatically, and this costs gas.
Solution:	The input parameters of *external* functions are read right from Calldata memory. Therefore, explicitly mark as *external* functions called only externally.

D. Operations

This category includes patterns related to the gas used for the operations performed within SC functions.

Name:	**Limit External Calls**
Problem:	Every call to an external SC is rather expensive, and even potentially unsafe.
Solution:	Limit external calls. In Solidity, differently from other programming languages, it is better to call a single, multi-purpose function with many parameters and get back the requested results, rather than making different calls for each data.

Name:	**Internal Function Calls**
Problem:	Calling *public* functions is more expensive than calling *internal* functions, because in the former case all the parameters are copied into Memory.
Solution:	Whenever possible, prefer *internal* function calls, where the parameters are passed as references.

Name:	**Fewer functions**
Problem:	Implementing a function in an Ethereum SC costs gas.
Solution:	In general, keep in mind that implementing a SC with many small functions is expensive. However, having too big functions complicates the testing and potentially compromises the security. So, try to have fewer functions, but not too few, balancing the function number with their complexity.

Name:	**Use Libraries**
Problem:	If a SC tends to perform all its tasks by its own code, it will grow and be very expensive.
Solution:	Use libraries. The bytecode of external libraries is not part of your SC, thus saving gas. However, calling them is costly and has security issues. Use libraries in a balanced way, for complex tasks.

Name:	**Short Circuit**
Problem:	Every single operation costs gas.
Solution:	When using the logical operators, order the expressions to reduce the probability of evaluating the second expression. Remember that in the logical disjunction (OR, ‖), if the first expression resolves to true, the second one will not be executed; or that in the logical disjunction (AND, &&), if the first expression is evaluated as false, the next one will not be evaluated.

Name:	**Short Constant Strings**
Problem:	Storing strings is costly.
Solution:	Keep constant strings short. Be sure that constant strings fit 32 bytes. For example, it is possible to clarify an error using a string; these messages, however, are included in the bytecode, so they must be kept short to avoid wasting memory.

Name:	**Limit Modifiers**
Problem:	The code of modifiers is inlined inside the modified function, thus adding up size and costing gas.
Solution:	Limit the modifiers. Internal functions are not inlined, but called as separate functions. They are slightly more expensive at run time, but save a lot of redundant bytecode in deployment, if used more than once.

Name:	**Avoid redundant operations**
Problem:	Every single operation costs gas.
Solution:	Avoid redundant operations. For instance, avoid double checks; the use of SafeMath library prevents underflow and overflow, so there is no need to check for them.

Name:	**Single Line Swap**
Problem:	Each assignment and defining variables costs gas.
Solution:	Solidity allows to swap the values of two variables in one instruction. So, instead of the classical swap using an auxiliary variable, use: (a, b) = (b, a)

Name:	**Write Values**
Problem:	Every single operation costs gas.
Solution:	Write values instead of computing them. If you already know the value of some data at compile time, write directly these values. Do not use Solidity functions to derive the value of the data during their initialization. Doing so, might lead to a less clear code, but it saves gas.

E. Miscellaneous

This category includes patterns that cannot be included in the previous ones.

Name:	**Freeing storage**
Problem:	Sometimes, Storage variables are not longer used. Is there a way to take advantage of this?
Solution:	To help keeping the size of the blockchain smaller, you get a gas refund every time you free the Storage. Therefore, it is convenient to delete the variables on the Storage, using the keyword *delete*, as soon as they are no longer necessary.

Name:	**Optimizer**
Problem:	Optimizing Solidity code to save gas in exhaustive way is difficult.
Solution:	Always turn on the Solidity Optimizer. It is an option of all Solidity compilers, which performs all the optimizations that can be made by the compiler. However, it does not substitute the usage of the presented patterns, most of which need information that is not available to the compiler.

IV. CONCLUSIONS

In this work we provided a brief introduction to Ethereum and its gas mechanism, through which the execution of the SCs is managed. The main problem that this mechanism involves is due to the difficulty in estimating execution costs in advance, with the risk of making the execution of the associated transaction fail. Furthermore, the gas corresponds to real money, whose usage must be spared. The need to have gas saving techniques is therefore clear.

Most of the general principles of sound programming and optimization also apply to Solidity, but there are some peculiarities in this language that make it more challenging to optimize the code. By applying the patterns and advices presented in this work, developers can solve or mitigate these problems, typical of blockchain development. To ease their usage, the patterns have been organized into categories based on their characteristics.

Future work will address empirical analysis of SCs actually used and whose source code is available, to assess if and to

which extent the presented patterns are actually applied by Ethereum programmers.

REFERENCES

[1] W. Zou et al., "Smart Contract Development: Challenges and Opportunities", IEEE Transactions on Software Engineering PP(99):1-1, September 2019.

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

[3] H. Rocha, S. Ducasse, "Preliminary Steps Towards Modeling Blockchain-oriented Software", Proc. 1th Workshop on Emerging Trends in Software Engineering for Blockchain, ICSE 2018, Gotheborg, Sweden, May 2018.

[4] Ethereum Network Status, available at: https://ethstats.net, last access on December 2019.

[5] ETH Gas Station, available at: https://ethgasstation.info/, last access on December 2019.

[6] Etherscan, available at: https://Etherscan.io/, last access on December 2019.

[7] C. Alexander, "The Timeless Way of Building" (TTWoB), Oxford University Press, 1979.

[8] K. Beck, W. Cunningham, "Using Pattern Languages for Object-Oriented Program", OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming. Retrieved 2006-05-26, September 1987.

[9] Gamma, Helm, Johnson, Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley. ISBN 978-0-201-63361-0, 1995.

[10] Eattheblocks, "How to optimize gas cost in a Solidity smart contract? 6 tips", available at: https://eattheblocks.com/how-to-optimize-gas-cost-in-a-solidity-smart-contract-6-tips/, last access on December 2019.

[11] M. Gupta, Mudit Gupta's Blog, available at: https://mudit.blog/solidity-gas-optimization-tips/, last access on December 2019.

[12] M. Gupta, "Solidity tips and tricks to save gas and reduce bytecode size", available at: https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6, last access on December 2019.

[13] W. Shahda, "Gas Optimization in Solidity Part I: Variables", available at: https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde, last access on December 2019.

[14] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system", 2019, available at: https://bitcoin.org/bitcoin.pdf, last access on Jenuary 2020.

[15] R. G. Brown, J. Carlyle, I. Grigg, M. Hearn, "Corda: An introduction", 2016, available at: https://docs.corda.net/releases/release-M7.0/_static/corda-introductory-whitepaper.pdf, last access on January 2020.

[16] A. Churyumov, "Byteball: a decentralized system for transfer of value", 2016, available at: https://byteball.org/Byteball.pdf, last access on Jenuary 2020.

[17] T. Chen, X. Li, X. Luo, X. Zhang, " Under-Optimized Smart Contracts Devour Your Money", IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017.

[18] Nomura Research Institute: "Survey on blockchain technologies and related services", 2016, available at: http://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf, last access January 2020.

[19] S. T. Aras, V. Kulkarni, "Blockchain and Its Applications – A Detailed Survey", International Journal of Computer Applications (0975 – 8887) Volume 180 – No.3, December 2017.

[20] P. Tasatanattakool, C. Techapanupreeda, "Blockchain: Challenges and applications", International Conference on Information Networking (ICOIN), 2018.

[21] N. Atzei, M. Bartoletti,T. Cimoli, "A survey of attacks on Ethereum smart contracts", Cryptology ePrint Archive, Report 2016/1007, 2016.

[22] V. Buterin, "Thinking about smart contract security", available at: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/, last accessed January 2020.

[23] M. Bartoletti, L. Pompianu, "A n empirical analysis of smart contracts: platforms, applications, and design patterns", Financial Cryptography and Data Security, 2017.

[24] P. Grau, "Lessons learned from making a chess game for Ethereum", 2016, available at: https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6, last accessed January 2020.