VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Anh Nguyen

SMART CONTRACT – A PROMISING USE CASE FOR THE ADOPTION OF BLOCKCHAIN

Information Technology
2019

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Satoshi Nakamoto, who brought blockchain to existence and has inspired me for this thesis idea.

Throughout the writing of this thesis, I have received great support and assistance. I would like to express my great gratitude to Prof. Petri Helo, from the University of Vaasa, whose expertise and experience was invaluable in various areas, for having guided me extremely closely through the entire thesis work. I would like to acknowledge Dr. Ghodrat Moghadampour for having supervised me all along the process.

Furthermore, I would like to thank my beloved girlfriend, Dan Wang, who has taken great care of me and kept me up all the time, and thank my brother, Hieu Nguyen, for always being very supportive by my side in every aspect.

Finally, I would like to thank my parents for having saved me when I was 3, given me a life and selflessly let me take the opportunity to study in the best possible environment.

This thesis could have not been done without any of you, thank you all!

Anh Nguyen,

Vaasa, Finland

02.10.2019

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Anh Nguyen
Title	Smart contract – A promising use case for the adoption of blockchain
Year	2019
Language	English
Pages	101
Name of Supervisor	Ghodrat Moghadampour

Blockchain has been around for over a decade, however, the technology does not seem to have been adopted after a long time being in existence. The goal of this thesis was to get to study what are the barriers of the blockchain adoption. However, the thesis also indicated one use case that seems to stand out from the rest, smart contract, by implementing one specific example for demonstration purpose.

The thesis went through an in-depth concept of blockchain technology, studied the most well-known use cases being, including cryptocurrency, initial coin offering and smart contract and figured out their limitations particularly and blockchain's generally. Finally, one example of Ethereum smart contract was implemented to indicate how this particular use case could fit what is needed nowadays.

As a result, the studies showed that blockchain is promising and can be applied through developing decentralized applications by smart contracts but the use cases must be chosen very carefully due to the problems of privacy and data protection. Otherwise, the main obstacle for the adoption is the lack of serious regulations, resulting the lack of trust, therefore, as long as blockchain-related businesses are not clearly regulated, the mass adoption is probably still far away since no one gets fully protected in the game, neither the companies nor the customers.

Keywords Blockchain, Smart contracts, Ethereum

CONTENTS

LIST OF ABBREVIATIONS

P2P	**Peer to Peer**
PoW	**Proof of Work**
PoS	**Proof of Stake**
ICO	**Initial Coin Offering**
EoA	**Externally Owned Account**
CA	**Contract Account**
EVM	**Ethereum Virtual Machine**
MPT	**Merkle Patricia Trie**
PoA	**Proof of Authority**

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE SNIPPETS

1 INTRODUCTION

1.1 Background

Back in 2007-2008, the world was experiencing a global economic crisis which caused enormous financial loss and damaged people's trust in their organizations/institutions /1/. Not so long after, an anonymous called Satoshi Nakamoto unveiled a white paper, "Bitcoin: A Peer-to-Peer Electronic Cash System", explaining how such a payment system can solve the issue of trust with a trustless solution. But Bitcoin itself is not the solution, the solution lies in the technology behind it, then was named blockchain. Needless to say, there is indeed a pain point of trust to be cured in the world nowadays, however, over 10 years has passed, and despite a great number of news and countless advertisements, many attempts, yet blockchain is still struggling to be adopted. The above story has triggered my curiosity to go find the answer to the question of what the barriers of the blockchain adoption have been for such a long time. Meanwhile, there is the emergence of Smart contract, a use case that seems promising for the future of blockchain. Why not try to find a way out of it?

1.2 Objectives

Blockchain has recently received much attention by organizations, regulators and investors. However, due to the lack of a certain level of knowledge about the new technology, it seems to have not been approached properly and the invested money and time has unfortunately been wasted. And this is the issue that this thesis is attempting to solve.

Particularly, this thesis provides an academic concept of blockchain, studies some of its use cases and their limitations, resulting the limitations of the technology itself. Among the use cases, smart contract stands out to be the brightest one for the mass adoption. The thesis will attempt to balance between limitations and advantages of blockchain and eventually propose a method how to approach blockchain properly and apply the technology to businesses.

1.3 Outline

Chapter 2: Explains the concept of blockchain by its characteristics.

Chapter 3: Studies the most well-known use cases of blockchain and result their issues and limitations.

Chapter 4: Implements an example of smart contract with the purpose of indicating how to approach the blockchain technology, maximize the advantages and minimize the limitations.

Chapter 5: Tests and deploys the developed smart contracts.

Chapter 6: Concludes the thesis and discusses the future work.

2 WHAT IS BLOCKCHAIN?

This section will go through a detailed concept of blockchain technology and the idea behind it.

2.1 Concept of blockchain

The word “Bitcoin” seems to be much more well-known, but in fact, blockchain is the technology behind that makes the world crazy about it. So, what is blockchain? Blockchain is not an application or a digital currency. Blockchain is a technology that provides a reliable data storage solution. Simply put, a blockchain network maintains a growing chain of blocks containing all the transactional records that have ever happened within the network, known as a ledger, as shown in the following figure.

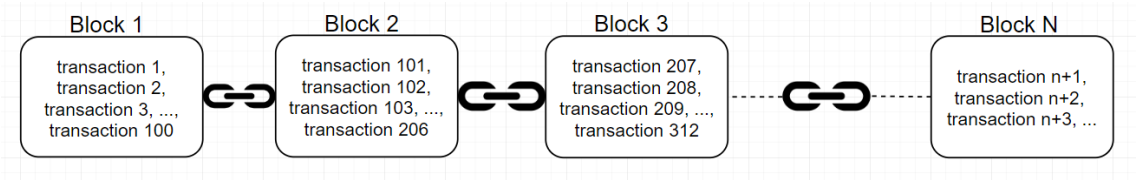

Figure 1. Block-chain

Because Satoshi Nakamoto wanted to remove the use of centralization for good, instead of storing this ledger in a central server, multiple copies are fairly distributed and stored in many computers in the network, in which no one is holding a master copy because there is nothing called “master copy” in blockchain, ensuring everyone’s right is equal, therefore, making it **decentralized** and **distributed**, the first two features of blockchain.

Similarly to traditional transactions, a blockchain transaction consists of such information as from whom and to whom the transaction is sent, what is transferred (could be either digital money or some sort of data), when it was sent and some other technical properties. In order to make sure a transaction does not contain any invalid information, for example, trying to send more money than the account is actually holding or double spending the same coin, the transaction must be validated

in a way that it is first broadcasted to the entire network and be verified by the network participants, here if everything is valid, the desired action will be proceeded accordingly and included in a block, otherwise any invalid actions will be detected immediately in the first place by any of the network participants and the transaction will be reverted as a result. And again, because Satoshi Nakamoto wanted to provide a trustless solution in a digital world full of fraud, all the transactions that are ever written into a blockchain network are entirely public, meaning they can be seen by anyone in the world and those pieces of data are permanent, therefore, making it **immutable** and **transparent**, the last two features of blockchain.

Eventually, to ensure blockchain technology is capable of the four mentioned features, any blockchain network must be built from three factors: cryptography, peer-to-peer network and consensus algorithm /2/. The following sections will go into detail of each one of them.

2.2 Peer-to-peer network

The peer-to-peer (P2P) network model is opposite to the traditional client-server model in which one or a group of servers is/are operated by an individual entity such as a company or organization, here all the requests will be sent through and handled by this server and all the user data will be stored in its data storage. So for example as long as any attackers can successfully attack that central server, all the data being is in danger. P2P model removes the need of a central server by having multiple servers all over the network with each being one peer or also commonly referred to as node. Every node is a computer having enough such computing resources as disk space or processing power will store an exact copy of the entire network's data in its own data storage, therefore, as long as there is at least one node remaining in the network, all the data can potentially be restored. P2P network enables all the data exchanges to be directly transferred between nodes without being controlled by any external party.

Blockchain technology utilizes P2P network to gain decentralization. Since P2P avoids the use of a central authority, all such decisions as which transactions to

accept or reject are fairly made based on some defined common rules within the network and no single party has the ability to decide over it. With blockchain there is no need to trust anyone as the system will take care of it. In the later sections will be presented how trust is maintained in blockchain.

2.3 Cryptography

Cryptography is truly what makes blockchain extremely secure and immutable. The types of cryptographic techniques that are nowadays commonly used in blockchain implementations will be described below.

2.3.1 Hashing

Hashing plays a key role in helping blockchain technology to gain immutability. Hashing is a mathematical process of turning an input of any length into a cryptographic fixed-length string as the output. A hash, or hash value, is the output generated by a hash function. The hash value will be the same as always if the same input is given, and any change, even the smallest as switching a letter from uppercase to lowercase will produce a completely different output. Table 1 shows several examples of hashes using SHA-256 (Secure Hashing Algorithm 256) – one of the most well-known hash functions nowadays and being used in Bitcoin implementation:

Table 1. Examples of hash values using SHA-256

Input	**Output**
Short Blockchain	c75aca9407a85d30a9220ac86d173a7198f03f167ab23dd71958ac3996a2e817
Short blockchain	26a3bf97d7a962030fc16fea3009470e5b05c55bf98110f9ccfef895ddf2c180
Shooooorter blockchain	9d72585da199554da611e44a638e33d109646e94221f999d92399fa98ae17e96

A hash function can only be considered useful once it is capable of acquiring all the following points, speed, pre-image resistance, second pre-image resistance and collision resistance /3/. Or briefly, an ideal hash function is one able to quickly generate hash values which are nearly non-reversible and “practically” completely unique for any given input. Strongly securing these features would make hashing extremely suitable for blockchain to protect the data and is integrity. How it is particularly implemented will be discussed in later section.

2.3.2 Digital signatures

In the real world, people prove their ownership over something by giving out their own handwritten signature. Similarly, signatures are used to bind a person or entity to digital data in blockchain, but instead, they are digital signatures.

Digital signatures employ asymmetric cryptography, also known as public key cryptography, which is a cryptographic system using pairs of keys to secure data on transfer. Every account is associated with one unique pair. One is public key, which may be shared to the public as an account address that everyone else can send transactions/data to. The other one is private key which is kept secret to only the owner of the key since owning a private key means owning whatever is associated with it such as the account, the data, the cryptocurrency, etc. The pairs of keys are generated through a key generation algorithm /4/ where the private key is always produced first, and the public key will be derived from it after that accordingly and both of them appear in the form of random numbers and letters. However, it is infeasible to calculate a private key if given a public. An example of a pair of private key and public key is respectively shown below:

L1AZF28gu8AckNKsyndWfsK2PNwpJ1wiSeNSAWksqevCkFs93w3j

17Vfv8K7WR4vffPH6RMCNZKbpmCmLkmYff

Now, a digital signature is signed and verified in the following way:

1. **Signing**: Before a transaction can be sent out, it must be signed by the sender proving that he/she owns this transaction and is willing to send

it. This step is done by combining the transactional data (usually hash beforehand) and his/her private key through a signing algorithm /4/, resulting a digital signature.

hash(transaction data) + private key = digital signature

2. **Verifying**: In order to prevent data tampering, the transaction must be verified to make sure that the transaction was really signed by the sender by combining the public key of the sender and the signature through a signature verifying algorithm /4/, resulting a hash.

   public key + digital signature = hash

   The resulted hash will then be compared to the hash generated by the same hash function from the transaction sent by the sender, if both hash values match, the data was not tampered and the transaction can be proceeded further, otherwise, something was modified in the middle, and the transaction is therefore rejected.

Since digital signatures are absolutely unique per transaction by the timestamp, one cannot fake a digital signature without knowing the private key of the owner.

2.4 Data structure

This section will explain how hashing is used in blockchain to maintain the immutability of such a huge ledger through its data structure. First, an example of a block in Bitcoin network is indicated in the following figure.

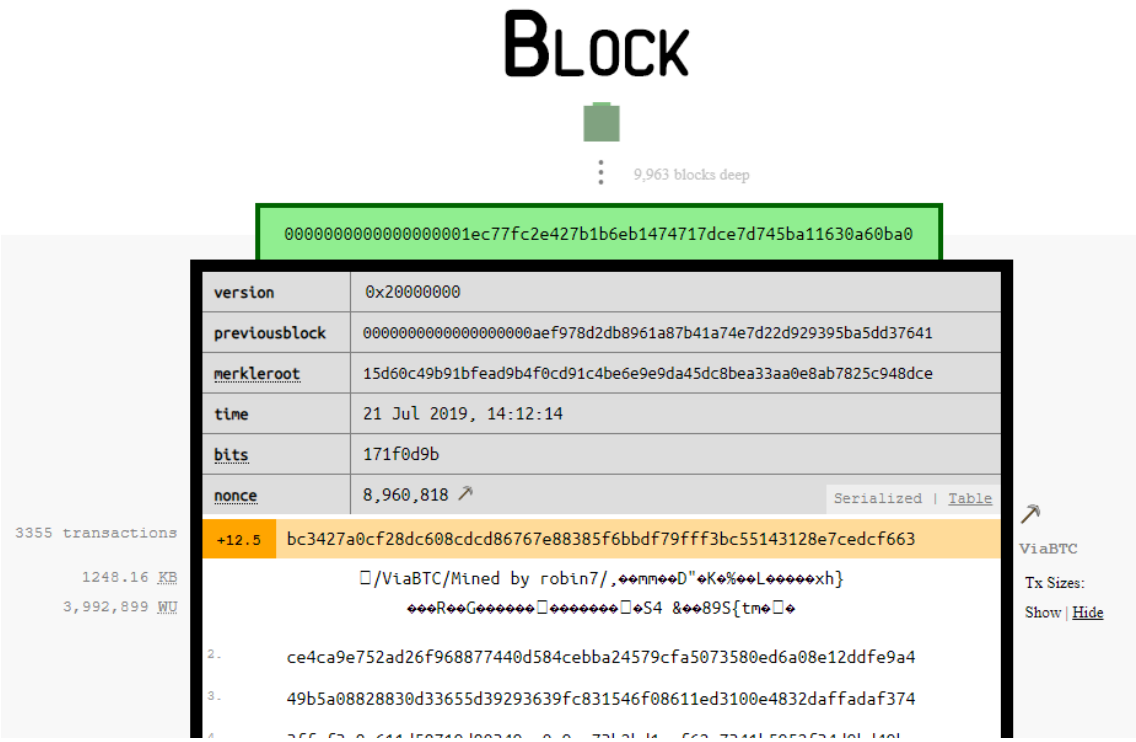

Figure 2. A Bitcoin block /5/

As previously mentioned, blocks in a blockchain network consist of transactions. A block is formed by two factors, a block header, which contains a bunch of metadata on top of a block and provides a unique overview of the entire block, the other factor is apparently, a long list of the collected transactions. Referring to Figure 2, a block header, specifically, contains the following:

- **Previous block**: The hash of the previous block. This is the factor that 'chains' the entire system altogether.

  NOTE: The very first block of any blockchain would have no previous block to be linked to, they are called "genesis block" and usually hardcoded in the implementation itself.
- **Merkle root**: A hash aggregated from all the transactions of this block. More on Merle tree later.
- **Time**: The Unix time of the moment that the block is mined, to be exact, the moment that the block's block header is hashed.

- **Bits**: The difficulty target for this block defines how difficult it is to find the block hash (only for Proof of Work consensus algorithm)
- **Nonce**: A random number of each block to be discovered by the miners. The first miner to find out will write the block into the chain and get the block reward (only for Proof of Work consensus algorithm)

In order to understand this structure better, the focus needs to be on the Merkle tree. Imagine a blockchain system processes hundreds of thousands of transactions a day, meaning it could be up to hundreds of millions of transactions per year, how much computing power and disk space would be needed to satisfy such systems. Here is one of the main reasons why Merkle tree comes into play. A Merkle tree is a data structure that allows computers to verify individual records in a system securely and efficiently without having to review the entire content of a large database. The following figure shows an example of a Merkle tree:

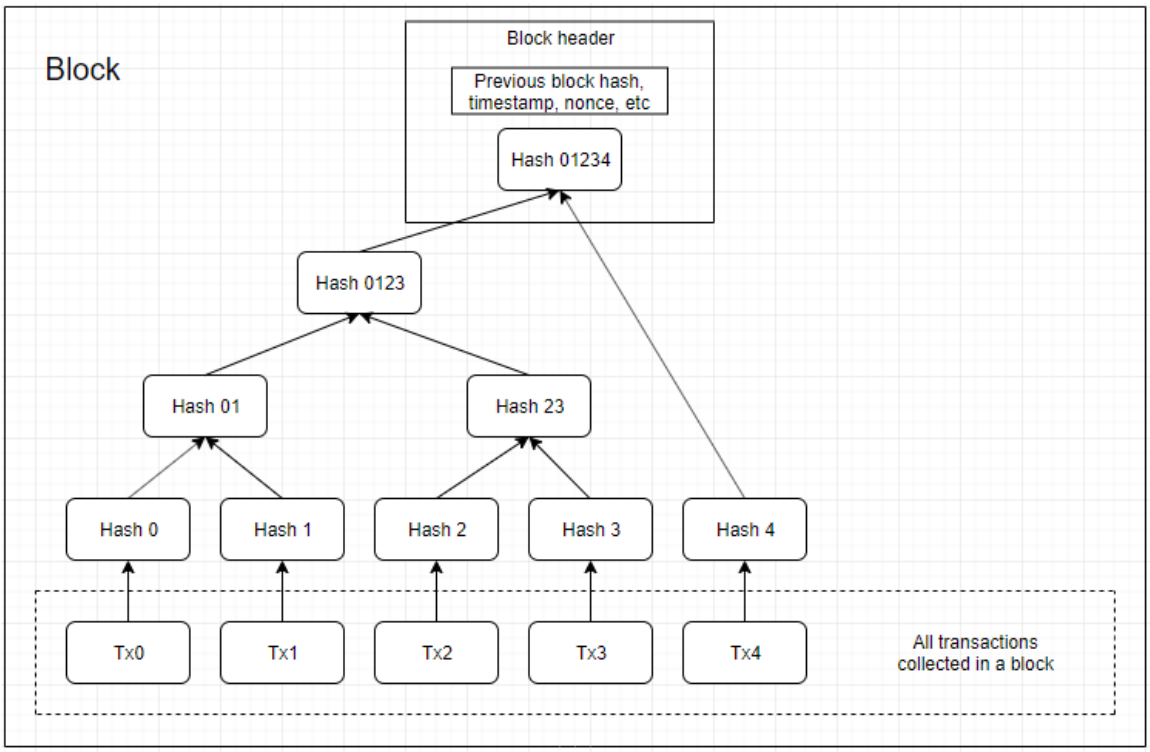

Figure 3. A Merkle tree in blockchain /6/

Each leaf node is the hash of a transactional data block, so accordingly**,** the leaf nodes in the example above are *Hash 0, Hash 1, Hash 2, Hash 3, Hash 4*, representing respectfully for Transaction 0 (*Tx0*), *Tx1, Tx2, Tx3, Tx4*. Now, a Merkle tree is created by hashing pairs of nodes until there is only node left in the highest position (they are constructed from bottom up). So, referring to Figure 3**,** there is *Hash 0 + Hash 1 = Hash 01, Hash 2 + Hash 3 = Hash 23* and so on, these are labelled non-leaf nodes where each of them is the hash of the value of its child nodes. The last non-leaf node on top of a Merkle tree is called the Root Hash or the Merkle Root, and as mentioned earlier, this Merkle Root is included directly in the block header /6/. Now suppose there is a transaction that claims to have been added into block *#123456*, what needs to be figured out is all the transactional data of block *#123456* and its Merkle Root from the block header, then a Merkle tree verification will be run, and finally a simple comparison between the resulting Merkle Root and the Merkle Root given earlier will tell whether that transaction is valid or not without having to know the data in block *#123455* or *#123457* or any additional

information. Hence, verifying any transaction by Merkle tree will greatly reduce not only the computing process but also an amount of time-consuming /7/.

From the structure of block header and the Merkle tree demonstrated above, it can be clearly seen that a blockchain network is not simply a data storage, but more than that, it is incredibly secure, as every single piece of transactional data is chained together and that every little change would consequently make significant changes afterwards and can be detected by anyone in the network.

2.5 Consensus models

Since blockchain networks are built to be decentralized, they do not rely on any central server, but each of the nodes maintains an independent version of the database, thus every node in the network must work based on common rules defined in advance so that all the databases are absolutely synchronized. Such rules combine altogether to be one thing called consensus. Consensus protocols describe how the communication and transmission of data between nodes in the network should work and it ensures the network can operate by itself without being corrupted. The consensus is reached when enough devices have acquired the requirements that were set earlier. This thesis will cover the most two common consensus algorithms right now, Proof of Work and Proof of Stake, each of them has its own pros and cons, but they are all dedicated to attempting to balance the scalability and security.

2.5.1 Proof of Work

Proof of work (PoW) was first used in the Bitcoin implementation by Satoshi Nakamoto. New bitcoins come into curricular as originally block rewards for miners as they find new blocks to append into the chain. One of the fundamental ideas of the reward is to cover the electricity cost for the computational power having been spent on mining new blocks. In order to maintain immutability and integrity, there should ever be one and only one chain that would keep growing. However, there are rather thousands or even hundreds of thousands of miners in one blockchain network, and because all miners work independently, it often happens that simultaneously, two

or miners broadcast different versions of the newly found block, e.g different transactions included in it, therefore different Merkle root hash value and resulting different block hash. As a result, the chain will be split into different forks, and the network would need to work based on a consensus to resolve this conflict so that only one would be accepted and officially included into the chain, as the figure below indicates.

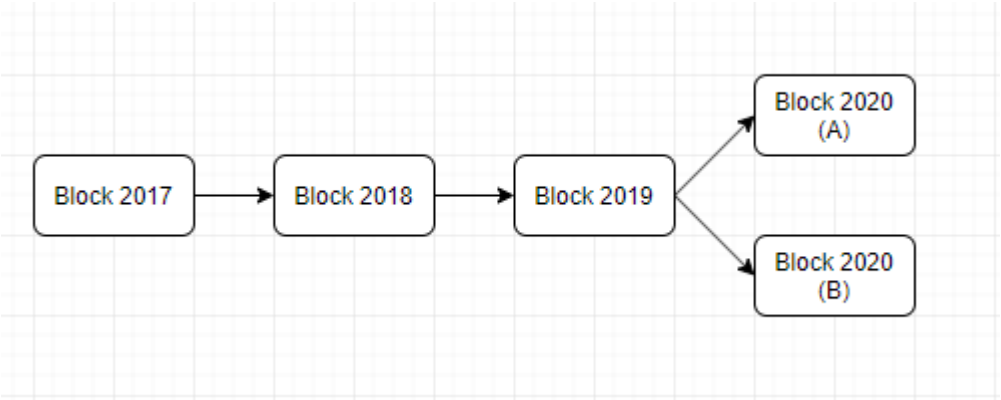

Figure 4. Block conflict

Since these branches are broadcasted just milliseconds from each other, miners can choose either branch to follow. All the miners would then keep on collecting and validating unconfirmed transactions, mining new blocks and adding them into the fork that they are on until one of the forks becomes longer than the other. The longer fork then would be known as “the longest chain” and officially be the main chain to be worked on by all miners as it has required the largest amount of effort, as known as computational energy to produce /7/. The other fork would disappear, many of the included transactions may have been included in the longest chain, and the others would just go back to the pool of unconfirmed transactions and wait for the coming blocks to come.

From the principle of the longest chain, if a person or organization attempts to attack the system, they would need to fool the rest of the network by obtaining at least 50% computing power of the entire network with the hope of being able to catch up with the honest chain and eventually be the longest chain of all. This attack is

known as 51% attack and being the biggest threat to PoW-model-based blockchain system. Such an attack can potentially lead to the following dangers:

- **Mining centralization**: Given earlier that mining is independently done by every individual, but when the market becomes more profitable, it would eventually be highly competitive. There start to come big mining groups obtaining a great amount of computing power and individual miners would just waste their time and energy mining for nothing since all the block rewards have been given to the mining giants who have mined out the blocks a long while ago. Therefore, most miners nowadays do not mine on their own but rather contribute their computing energy to other organizations by joining their mining pools and get paid more regularly by the pool administrators as little shares from the block rewards. For the past 1 year, 76.9% of total computing power is possessed by 7 mining pools, where 6 of them together with some other smaller pools are from China /8/. Imagine if those Chinese pools are not independent but cooperating altogether, a network disruption is very likely. Even though they would not have the ability to write whatever they wanted to the ledger or steal coins from other accounts as those operations would be considered as invalid and could be detected and rejected by other miners, they indeed could freely choose what transactions to include into the blocks and reverse their own transactions hence engaging in double-spending issue.
- **Double-spending issue**: This is an attack where one coin is spent twice. This can be done while the attacker is in manipulation of the network by creating two transactions sending the same amount of coins to two different addresses at the same time. One will be sent normally to an address that is supposed to receive the coins, for example a vendor selling some items to the attacker, and its transaction is publicly included in the next block of the main chain and the other one is sent to another address that is under control of the attacker and it is secretly included in an alternative fork, now the attacker would need to build the secret fork to be longer than the other public chain and reveal it afterwards, now the principle of the longest chain would

be applied and the fork including the dishonest transaction would be included into the blockchain, while the other one would become invalid and get reversed back to the attacker as it attempts to send the same coins at the same time and the other "fake" transaction is already confirmed, resulting the vendor would never receive his money while the items might have already been on the way to the attacker /9/. In fact, one Bitcoin transaction should not be considered a "safe" valid one unless it has reached 6 confirmations, meaning 6 new blocks have been included on top of the block which includes transaction, because the more confirmations of a block on the honest chain would mean the less possibility an attacker can catch up with his dishonest fork, and the chance is incredibly low with 6 confirmations /7/.

However, these attacks are extremely costly in practice since mining blocks is not as simple. In order to mine a block, the miners not only need to gather all the transactions and hash them altogether, but there is one factor that has made mining way harder, the nonce. A nonce is a very random number (Figure 2) that the miners need to find out during the mining process to altogether with other factors such as Merkle root hash, timestamp, previous block hash, etc. eventually generate the current block hash that falls under a so-called target hash, or difficulty target /7/. Finding the correct nonce consumes a lot of computational energy because the hashing action needs to be carried out over and over again until the one is discovered. The terms "computational energy" or "computing power" which have been repeatedly mentioned are in fact the hash rate. Hash rate represents the number of hashes produced per second. Therefore, the only way for a miner to increase his chance of solving a block before others is to increase his/her hash rate performance, which essentially requires more electricity and money.

As mining blocks gets faster if the hash rate of the entire network increases and vice versa, but Bitcoin was designed to update a new block approximately every 10 minutes because of latency issue, the target hash needs to be adjusted accordingly to maintain the block discovering speed. This variable number is updated once every 2016 blocks or approximately 2 weeks.

Going back to the 51% attack, the current Bitcoin hash rate is about 87,000,000 TH/s (trillion hashes per second) as of 04.09.2019 /10/, in order to manipulate the network, one must be able to produce 45,000,000,000 TH/s which is not impossible, but very unlikely in practice and this is why the more honest participants there are, the more secure the network becomes.

2.5.2 Proof of Stake

One of the principles of blockchain technology is decentralization where chances of creating new blocks are distributed fairly amongst the participants. However, mining centralization, which is happening right now with Bitcoin, seems to be going against it. Along with that, high energy consumption and 51% attack are clearly the three problems right now occurring with those digital currencies running on top of PoW algorithm, cryptographic developers try to solve all of them, hence resulting another consensus model, Proof of Stake (PoS). Instead of showing proof of computational work, the nodes are required to show the proof of stake. Therefore, with PoS, all nodes would not need to compete into a race of computational power to become the winner to write the next block into the chain, rather would the system choose the next block writer, or block validator, in advance based on the amount of tokens (or cryptocurrency) each node is owning/staking. With that given, the wealthier the node is, the more likely it will be chosen to be the one. As there would be no mathematical problems to be solved, there would not be any rewards either to be given and therefore there is no such thing as “mining”, but rather “forging” blocks in PoS /11/.

Now one of the questions being may be that the process then would most likely favour only the ones holding big stakes in the network, this seems to have been anticipated and therefore, some other methods are added to equalize the game, the most common one being “Coin Age Selection”. Specifically, nodes who want to participate in the forging process are likely to be asked to deposit a number of tokens into the system as their stake. The block validators are then chosen by calculating the coin age based on the combination of the number of days their tokens have been staked multiplying with the number of tokens being staked. Once the

block validator has forged a block, he takes all the transaction fees that are included in the transactions of the newly forged block as his own but otherwise, there is no block reward from the system, his coin age is to be reset to zero and cannot be chosen again after a while resulting a prevention of network domination coming from wealthy nodes /11/.

Therefore, PoS seems potentially to be able to solve the mining monopoly and costly energy consumption problems being on PoW, however, there comes problems of its own. Since the block validators do not need to commit a great amount computational power to show their proof of work, whenever there is a conflict in generating new blocks resulting an alternative fork, it is in their best benefit that they can choose to forge on whatever fork they see so that they can ensure the transaction fees to go to their pocket later on no matter which fork wins, and the other fork would just go away and they would lose nothing. This theoretical issue is called “nothing at stake” /12/. This scenario could potentially make double-spending attack more possible, as if everyone is forging on both forks, and the attacker intentionally chooses to forge only his “cheating” fork, then his fork would eventually end up being the longest chain and double-spending attack is simply successful. Seemingly, the arguments on whether PoS is eventually better than PoW, there would be solutions for existing issues or more issues are more to come yet remain open to be answered in the future.

3 NOWADAYS BLOCKCHAIN'S COMMON USE CASES

This section will discuss cryptocurrency, initial coin offering and smart contract as being the most common use cases of blockchain in recent years.

3.1 Cryptocurrency

Imagine if Dan wanted to send 5 EUR to Anh digitally, it would have been done in a way that the money would essentially need to go through a central bank or any third party between Dan and Anh. This raises questions such as whether the bank is reliable enough, whether the bank is corrupted, whether the transaction is processed properly or whether Dan's 5 EUR has been spent twice, also known as double spending problem in which the same digital currency is spent more than once. This is how cryptocurrency comes into play. The process for the earlier example now can be done exactly the same way as Dan gives 5 EUR to Anh physically by hand, but instead of sending euros or dollars or any fiat currencies, digital currencies are used. Cryptocurrency is the very first use case of blockchain ever when the technology is announced to the world. In short, cryptocurrency is a medium of exchange which is operated by blockchain under the hood to gain decentralization, tamper-proof and transparency. This means that this has removed the need of a central bank or any third party and the money is now not stored and controlled by no one else but directly in the wallet's owner. There have been many different cryptocurrencies that have come into existence and some of the most well-known ones will be presented in this section.

3.1.1 Bitcoin

Bitcoin was first introduced by the unknown Satoshi Nakamoto from a PDF white paper on October 31st 2008 /7/, then first released January 2009**.** Initialized with no value at all (1 BTC = 0.00 USD), Bitcoin in early days had no exchanges or markets, it was mainly back then cryptography fans sending bitcoins back and forth for fun. In March 2010, BitcoinMarket.com, which no longer exists nowadays, started operating as the first bitcoin exchange with the initial rate as 1 BTC = 0.03 USD. A decade has passed by, Bitcoin has ever peaked at $19,783.06 on December 17th,

2017, but then dropped significantly down to $6,300 at its 10th year anniversary due to the aftermath of an exchange breakdown and nowadays being evaluated at around $10,000 /13/.

There are many reasons that have driven the Bitcoin price incredibly high, 179.52 billion of US dollars of total market capitalization, from the aspect of a technology innovation. But, when considering the simplest scenario, the price increases when demand is higher than supply. There are two factors that affect the supply of Bitcoin. Firstly, new bitcoins come into existence as rewards to miners as a result of mining new blocks. And the Bitcoin implementation has made this block rewards at a fixed rate and this rate is designed so that the block reward value is reduced by a half once every 210,000 blocks are mined. As of the time writing this thesis, it stands at 12.5 BTC and is expected to reduce next time itself by 50% to 6.25 BTC in 2020.This can tell that if the demand of Bitcoin remains the same or increases, it can drive the price up and the supply is getting slower over time.

Furthermore, Bitcoin implementation is designed to also be limited to have a total of 21 million coins ever. It means that once this number is reached, there will be no longer reward as new block is mined out and the miners then will get compensated on transaction fees. When all 21 million bitcoins are in circulation, the price will depend on the demand and demand would depend on its practical use compared to other currencies.

Apparently, the world's leading cryptocurrency should have some value due to the underlying revolutionary technology – blockchain. However, all these have not explained what drives the demand all price up to over 10,000 USD / 1 BTC. One of the original reasons could be because of the belief that large institutional investors had in Bitcoin, but the key reason truly lies in the hype and news that social media has brought. The dramatic emergence of social media has been seen in recent years, and there has been a countless number of news and advertisements being posted on any means of social media on how potential Bitcoin is, Bitcoin quickly became the hot topic in any conversation all over the world. As a result, by instinct, humans

started to be curious and wanted to be part of the trend, and eventually hit the demand all the way up.

It is still early to say whether the Bitcoin hype is a bubble, however, when the hype is over, what is left is all about what Bitcoin can really do. Bitcoin, after all is not a store of value, but said by Satoshi Nakamoto, was originally meant to be "A Peer-to-Peer Electronic Cash System". In order to truly be an alternative for fiat currencies, Bitcoin must become an exchange currency and must be able strong enough to compete against other methods of payment such as Visa, Paypal, etc. Whether processing Bitcoin transactions consumes more electricity power than traditional central banks do or not remain a controversial topic as a lot of factors need to be taken into account, but one thing can be concluded right now is that central banking systems can handle transactions a lot faster than Bitcoin ever can do. Visa processes around 150,000 million transactions per day, meaning around 1,700 transactions per second (tps) /14/. So, what can Bitcoin really do?

Given that Bitcoin is limited to update a block once every 10 minutes, the size of one block is limited at 1MB and each transaction takes approximately 380B /7/, as a result, the number of Bitcoin transactions proccessed per second can be calculated as follows:

$$Number\ of\ Tx\ per\ block = \frac{Number\ of\ Tx\ per\ block\ in\ Bytes}{Tx\ size\ in\ Bytes} = \frac{1{,}048{,}576}{380}$$
$$= 2759.41\ Tx$$

$$Number\ of\ Tx\ per\ second = \frac{Number\ of\ Tx\ per\ block}{10\ minutes\ in\ seconds} = \frac{2769.41}{600}$$
$$= 4.60\ Tx$$

So, 4.6 tps on average and 7 at its best ever seen. Thus, even though Bitcoin came with several technological innovations, the Bitcoin adoption as a payment system/currency is now heavily bottlenecked by its scalability and unstable value, but it can be treated better in another way, as an asset, like gold where it can be bought, stored and sold for profits.

3.1.2 Ethereum

Ethereum was first proposed in a white paper in late 2013 and then officially released on July 30th 2015 by Vitalik Buterin, who is a cryptocurrency researcher and at that time was the co-founder of Bitcoin Magazine, and seven other co-founders /15/. Ethereum serves two purposes. First and foremost, Ethereum, same as Bitcoin and other cryptocurrencies, is a global decentralized financial system that uses ether, also known as ETH, token to be traded as its digital currency exchange. Secondly, Ethereum not only uses blockchain to decentralize money, but it also leverages the blockchain technology to become an open-source decentralized software platform that allows developers to build and execute applications on Ethereum network. Such applications are called decentralized applications (dApps) and they use a so-called smart contract to run on top of Ethereum. The subject about smart contracts will however be discussed further later in section 3.3.

Similarly to Bitcoin, there are accounts in Ethereum. However, Ethereum supports two types of accounts, externally owned accounts (EOAs) and contract accounts (CAs) /16/. EoAs, as already introduced, are to represent entities such as humans, organizations, and so on in Ethereum network. They can hold ether in its balance as a wallet. Each of them holds a unique pair of public key and private key and they can either process regular transactions to interact with other EOAs in the network or execute contract transactions by invoking functions within smart contracts (more on this later in section 3.3). On the other hand, CAs do not have any private key but only public key as their identifier. They can also have a balance in ether like EOAs, however, they can contain code written in smart contracts (more on this later along with smart contracts in section 3.3).

Ethereum nowadays can process roughly 15 transactions per second /17/, certainly faster than 4.6 from Bitcoin, but it currently is also using PoW consensus mechanism hence is facing such scaling problems as Bitcoin and other Proof-of-Work based applications are. However, while Satoshi Nakamoto, the inventor(s) of Bitcoin, has had no further contributions to the Bitcoin implementation since its first release in 2008 and most of the changes are proceeded by the community,

Ethereum's co-founders are strongly concerned about Ethereum's scalability and they certainly do have improvement plans to make Ethereum support as many users as it can. Table 2 briefly shows Ethereum's roadmap.

Table 2. Ethereum's roadmap /18/

Upgrade	**Date**	**Details**
Raiden Red Eyes	December 2018	Off-chain solution for faster and cheaper transactions.
Constantinople hard fork	February 28th, 2019	Lays the technical groundwork for significant scaling projects in the future.
Plasma	TBD	The introduction of "child" chains off the main Ethereum blockchain for faster and cheaper transactions. Similar to how the Lightning Network works on Bitcoin.
Casper	mid-2019	Ethereum's main scaling goal. Casper is the shift from Proof-of-Work to the more efficient Proof-of-Stake.
Sharding	2020-2021	Partition the existing blockchain into smaller pieces known as shards.
Serenity (aka Ethereum 2.0)	2019-2021	The culmination of Casper and Sharding will create "Ethereum 2.0."
Ethereum 3.0	2022-2025	Implementation of a 'super quadratic sharding' solution which could facilitate one billion transactions per day.

At the time of writing this thesis, the Constantinople hard fork is the latest major update concerning the transition from PoW protocol to PoS protocol (Caspter upgrade). After Casper upgrade, Ethereum development team will then move to implementing Sharding feature. Put simply, one of the existing scaling problems is that it requires a computer to have a full node to verify transactions, meaning those which store the full state of the network and every transaction that occurs. Every time a block is mined, the new block needs to be broadcasted and then verified by every single node in the Ethereum network in a way that one after another, meaning a linear manner. Currently, there are estimated to be 8,000 full nodes /19/ and transmitting the transactions through this large P2P network would take some time, and it is expected to get slower as the network grows over time. Therefore, sharding was decided to be the solution for this issue. With sharding, the Ethereum global state will be split into partitions, or so-called "shards" and each shard can be considered as one small blockchain network which will have its own state, set of addresses and transaction history. Each shard then would likely process transactions from and to addresses within itself. More complicatedly, if an address needs to communicate with another from another shard, there would be some sort of cross-shard protocol to be implemented. Shard blocks are created by so-called validators, who are assigned randomly, similarly to block validators in PoS (Sharding feature will be implemented after PoS, as shown in Table 2). Moreover, every block creation would be validated by a set of other validators, known as attesters in this case, one would be included into the main chain if it acquires at least 2/3 signatures of the attesters. Accordingly, there are a couple of different types of nodes in the system, and everyone is free to choose which type it wants to be /20/:

- **Super-full node**: Stores the full data of the chain, including all the shards' data.
- **Top-level node**: Maintains the main chain only, including the headers and signatures of shards but not their data.
- **Single-shard node**: Being similar to top-level node, but also downloads and verifies the data of some specific shards.

- **Light node**: This requires minimum disk space as most of the time it only downloads and verifies the block headers of the main chain. Otherwise, if needed occasionally, it would just also download the Merkle branch of some specific shards.

There are some questions about performance such as if one attacker attempts to take over the majority of the attesting to submit invalid block or so on. Currently, Sharding is not yet implemented, there is not much to talk about but only guessing, but Ethereum development team is working intensely on it so that it could be unveiled in time, then more detailed information will come and all the questions will get answered.

3.2 Initial coin offering

Initial coin offering (ICO) is considered one of the well-known use cases of blockchain that has made real impact. ICO is a fundraising method using virtual currencies. When a company is to launch a new digital currency or a decentralized application, they can look for some early support from outside investors by creating a new digital token via a number of different platforms to attract investors into their project. The company will eventually conduct a public ICO so that the interested investors then can buy the newly created tokens either with fiat or other pre-existing digital currencies like bitcoin or ether. This sounds similar to the initial public offering (IPO) method, but instead of buying shares from a public firm and actually owning a little part of it, in this case, the investors buy the tokens with the hope based on the promise that the digital tokens one day will perform exceptionally well and can be traded for a profit or used on a product that is eventually created, providing them a beneficial return on the investment.

The first token sale was conducted by Mastercoin in July 2013 where they raised $500,000 /25/. But the term ICO truly came into public's attention in July 2014 after Ethereum has successfully raised $18.4 million within a 42-day period, breaking all fundraising records back then. The funds raised by Ethereum were then used to develop the Ethereum decentralized platform with the smart contract functionality that exists today. The success of Ethereum's fundraising had later led to the ICO

boom in 2017 and first half of 2018. During this period, over 1,000 digital token sales had been conducted, raising a total amount of roughly $12 billion, the figure below shows ICO funding stats in 2018.

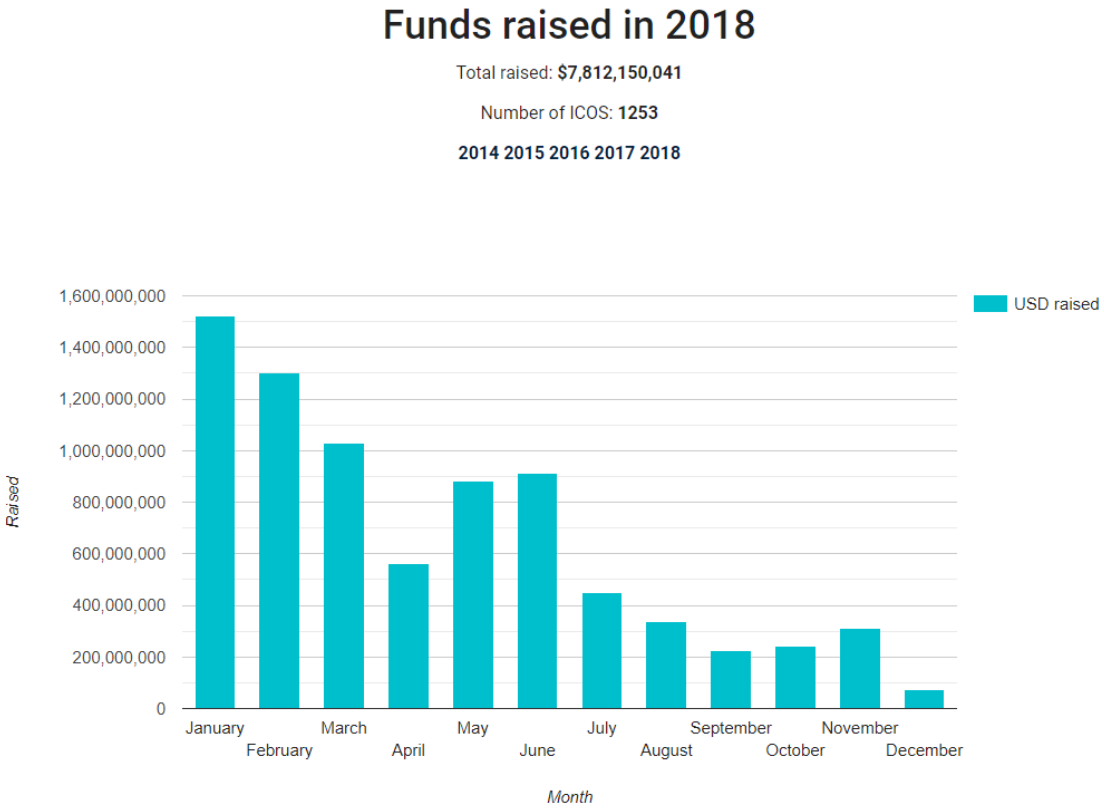

Figure 5. ICO fundraising in 2018 /22/

As it can be seen, the market trend seems to have started to go down dramatically from the second half of 2018, and thing has not gotten any better till August 2019, the time of writing this thesis, total 2019 ICO funding so far is only nearly $350 million /23/. When a new technology comes into existence with high hype and a large amount of money involved, often there comes frauds after. Most likely, when a company is to conduct an ICO fundraising, a white paper needs to be created, explaining the idea of the project. However, in the early stage, what the investors cared about was only what the project idea was and how it was going to be as described in the white paper, and they would just actually invest into the proposed ideas and hope that it will benefit them later. The only way for them to evaluate a token is whether it is later traded on any exchange, if not, the token is considered

as no value. On the other hand, as long as all the funds requirements are qualified within the required period of time, the money raised can no longer be refunded to the investors, but rather is then either ready to be used for a new scheme or to be completed. This has created a huge flaw that fraudsters could take advantage as all they need to do is to prepare a best white paper as they can and expect for a good fundraising result to eventually come. According to a report produced in July 2018 by SATIS group, an ICO advisory firm, approximately 85% of ICOs conducted in 2017 did not succeed, where 78% turned out to be scams and the rest 7% either had gone dead or failed, only 15% survived and got then listed on exchange /24/. This is one of the reasons that investment into ICOs have been dropping down unstoppably, a large number of investors have felt the risk of being frauded and no longer naively believe in the promises written on the white papers.

Unlike the highly regulated IPOs, many of early token sales were conducted without a regulatory party such as lawyers therefore leaving the legal state of ICO very much undefined back then and causing most of the stolen money never be recovered /25/. The lack of regulation was one of the main factors behind major scams in ICOs. This has made many countries and regulatory organizations nowadays seriously look into the issues and take immediate actions. Some, such as China or South Korea, have decided to declare ICOs as illegal and ban funding via digital currencies /26/. Some others like the Securities and Exchange Commission (SEC) in the United States, or the European Securities and Markets Authority (ESMA) in Europe are trying to bring ICOs into under regulations and warn people of the great potential risk of investing into ICOs.

"As with any other type of potential investment, if a promoter guarantees returns, if an opportunity sounds too good to be true, or if you are pressured to act quickly, please exercise extreme caution and be aware of the risk that your investment may be lost." said Jay Clayton - SEC chairman /27/

Certainly, there are good and bad ICOs existing at the same time. The world has seen real successful projects with the funds raised from ICOs where one of the most well-known cases is Ethereum, and there has been also a countless number of

scams. Therefore, ICOs appear to be useful and very much promising, but identifying whether it is a good ICO requires a lot of more things taken into account such as who are involved in the project, whose idea is being proposed, how much reputation the company has gained on existing businesses or how active their community or social network are or is the ICO under any regulations, than just simply looking into a white paper, however, a clear and well dedicated white paper does indeed raise some trust on the success rate of the project.

3.3 Blockchain-based smart contract

This section will discuss smart contract, the concept of smart contract may be widely used, but this thesis only concerns the blockchain-based smart contract in general as a use case of blockchain and Ethereum smart contract in particular as a common platform for writing smart contracts.

3.3.1 Concept of smart contracts

In reality, a contract is a paper consisting of a set of “if”, “else” and “then” conditions and actions agreed between two parties. A smart contract is an ordinary contract whose terms and conditions are written as computer programs and automatically executed on computers. An Ethereum smart contract is a blockchain-based smart contract which is hosted on Ethereum blockchain network and executed by the Ethereum network participants. This is to remain decentralization and immutability, ensuring all the taken actions are honestly recorded, similarly to processing blockchain transactions. In fact, Bitcoin was an implementation of basic smart contract in early stage, but it is only limited to the digital currency use case, where the network nodes can transfer bitcoins and validate the transactions if certain pre-defined conditions are met. By contrast, Ethereum provides a platform that allows developers to write and deploy their own smart contracts that can do anything within the allowed resources.

Ethereum smart contracts are written and executed in terms of programming functions. Whenever a function within a smart contract is to be executed in Ethereum, it charges an amount of transaction fee in exchange for the computing power used

by the EVM to execute the contract. In Ethereum, this transaction fee is called gas (or gas fee), commonly seen as gwei or wei, and is paid in ether.

$$1\ ETH = 1{,}000{,}000{,}000\ gwei = 1{,}000{,}000{,}000{,}000{,}000{,}000\ wei$$

It is important to note that the gas fee to be paid for executing a smart contract's function is constant and does not get affected by the market value of ether, meaning that if today a contract execution costs 2 gwei, tomorrow it will still cost 2 gwei regardless 1 ETH = 1 USD or 1 ETH = 1,000 USD.

3.3.2 Ethereum virtual machine

Ethereum virtual machine (EVM) is a supercomputer being run in every Ethereum node, it serves as an isolated runtime environment to execute code written in Ethereum smart contracts. Since Ethereum is intended to be a platform for smart contracts, the EVM must be built as a so-called turing-complete machine so that it can process any given contracts. Briefly, turing-completeness is the ability of e.g a machine or programming language, etc. to solve computational problems /28/. One is considered turing-complete if it can solve any given problems, with enough resources provided such as time or computational power. However, the EVM is only "quasi" turing-complete and it was meant to be /29/. Each block in the Ethereum network has block gas limit which is the total amount of gas allowed to be spent to process all the transactions in a block. And this is the factor that limits the EVM from being truly turing-complete, a smart contract cannot be reliably predicted beforehand whether it will halt or run forever without really being executed. If a smart contract that contains an infinite loop can spend as much as gas it need, it would absolutely cause the network to slow down as the resources can never be freed up again. The gas limit is set at block level because it should not take time processing all transactions in a block as long as it takes Ethereum miners to mine out a new block, currently varying from 10 to 20 seconds, meaning by the time a new block is found out, all of its transactions must have been completed, otherwise it would cause network vulnerabilities that the miners cannot tell ahead of time. Therefore, the EVM can execute "complicated enough" smart contracts and that is why it is intentionally "quasi" turing-complete.

Smart contracts are usually written in high-level languages, the most well-known one is Solidity. The EVM does not understand these languages, thus contracts must be compiled in prior to be deployed to Ethereum network, resulting bytecode. The bytecode then would tell the EVM what to execute and especially how to manage the data. Basically, contract data is stored in three places, depending on the purposes, in the EVM /29/:

- **Stack (volatile)**: Used to store a limited number of small local variables and nearly free to use.
- **Memory (volatile)**: Used to store temporary data in random access memory (RAM). This data is always erased between function executions. Storing data in RAM is also cheap.
- **Storage (persistent)**: Permanent data is stored in storage that can be accessed across executions, making it the most expensive to use.

3.3.3 Transactions

Ethereum network is fundamentally a transaction-based state machine. It means that every transaction execution in Ethereum moves the current state to a new state. Transactions occurring between Ethereum accounts, including EoAs and CAs, can be created in three ways:

- **Transfer of funds transactions**: Transactions that transfer funds from an account to another account regardless of type of the account.
- **Contract creation transactions**: Transactions created on smart contract creations to transfer the compiled contract bytecode to the EVM resulting a CA that holds the compiled bytecode. Smart contract can only be deployed to Ethereum network by the contract owner, originally is an EoA.
- **Contract method call transactions**: Transactions are needed when invoking contract functions that change the state. However, if the function is to only read data, it does not require a transaction, therefore it takes no gas fee.

Accordingly, contracts themselves can send transactions between one another to either transfer funds or invoke methods. This type of transactions is called “internal

transactions", as shown in the following figure. However, all transaction executions must be originally initiated by EoAs, meaning that internal transactions can only happen after at least one external transaction executed by one human being.

All Ethereum transactions consist of the following components /29/:

- **Nonce**: Number of transactions sent by the sender.
- **Gas price**: The value (in wei) per unit of gas for the computation costs to execute this transaction.
- **Gas limit**: The maximum amount of gas to be used while executing this transaction.
- **To**: The address of the recipient, regardless type of the account. This field is empty in contract creation transactions as the CA does not exist yet.
- **Value**:
    - If the transaction is transferring ether, amount of wei that will be transferred to the recipient's account.
    - If the transaction is a contract creation transaction, amount of wei that will be added to the soon-newly created CA.
    - If the transaction is invoking a payable method of a contract, amount of wei that will be transferred to that CA.
- **v, r, s**: Used to generate the digital signature of the sender.
- **Input data**: Used either as a simple data storage or passed as parameter if the transaction is invoking a method within a contract.
- **Init (only for contract creation transactions)**: Contains the deployment script, or known as constructor code, that will be run only once in a contract's lifetime when deploying the smart contract. The EVM will execute the constructor function of the contract and initialize some pre-set variables (if set) and eventually return the contract code to the EVM.

It is important to note that the transaction's gas price is multiplied by the transaction's gas limit to determine the maximum gas cost for that transaction. This gas cost will then be paid in advance before the execution. Every operation during transaction execution consumes a certain quantity of gas, the gas cost will be deducted

step by step accordingly during the process. If the gas runs out before the execution is finished, the transaction will roll back, all the made changes to the state will be undone, but the all the gas will still be given to the miner due to the done computation. If after the transaction has completed, there is still gas left, it will be refunded to the transaction's sender. Another situation is that there's still gas to be used and code to be executed, but the block gas limit is already reached, then the transaction will fail, the used gas will still be taken by the miner and the remaining gas will go back to the sender.

3.3.4 States

While the Bitcoin system maintains only the transactions and the flow of bitcoins through accounts. At the most fundamental level, Bitcoin does not hold the account balances, instead, it tracks assignment of value. Every bitcoin comes to existence by the reward to miners as mining blocks and since then, the system just simply tracks the ownership of the coin across transactions throughout its lifetime. Therefore, the Bitcoin system can be seen as a book of history describing what happened when, as shown in Figure 6.

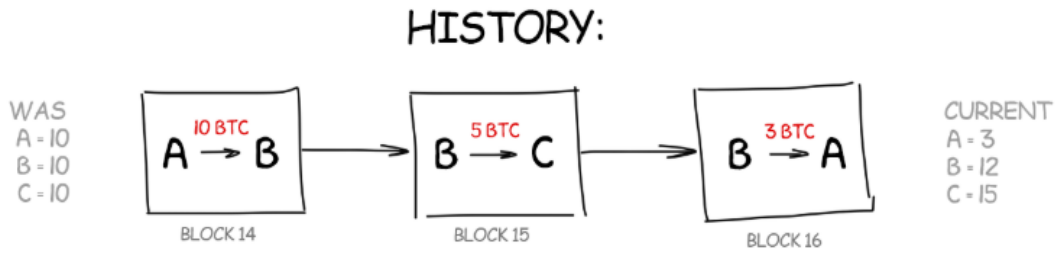

Figure 6. History /30/

This is not the case in Ethereum. Due to the existence of smart contracts, there is a lot more going on under the hood of Ethereum and the system certainly is more complicated. As mentioned above, Ethereum is a transactional machine of states that uses transactions to update the state of the ledger. So instead of describing what

happened when, the Ethereum book records what were stored when, as shown in Figure 7.

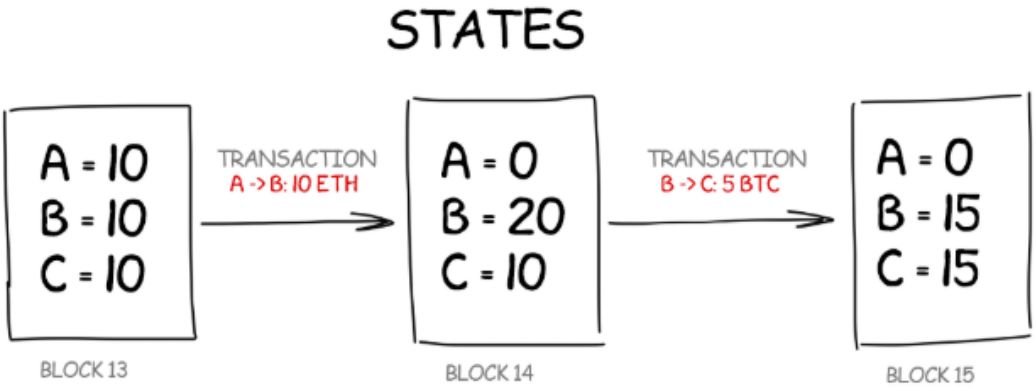

Figure 7. States /30/

There are basically two types of data in Ethereum, constant data and variable data. For example, transactions would belong to constant data type as once a transaction has been fully confirmed, it is immutable. On the other hand, such data as account balance would be mutable data as it changes every now and then. Therefore, they should be stored separately. Ethereum also utilizes Merkle tree structure to maintain the data, similarly to what is used in Bitcoin system, but it has been modified a bit to fit the Ethereum characteristics. Hence now it is modified Merkle Patricia Trie (MPT) as defined by Ethereum. An MPT is a key-value mapping trie from which the key is a path telling how to get to its corresponding value through the tree. Every block header in an Ethereum block contains not only one but three MPT roots for three different kinds of objects /29/:

- **transactionRoot**: This is similar to the Merkle root in Bitcoin. It is the hash of the root node of the transactions trie.
- **receiptsRoot**: After a transaction is executed, Ethereum generates a transaction receipt that contains the information of the transaction. This is the hash of the root node of the receipts trie.
- **stateRoot**: This is the hash of the root node of the state trie.

A state trie is only generated once all transactions in a block have been completed to represent the world state of the entire Ethereum network at the time the block is created. There is only one world state at a specific point of time. The state trie contains a key-value pair for all existing account in the Ethereum blockchain where the key is the account address, and the value is the encoded value of the state of the corresponding account using the Recursive-Length Prefix encoding method /29/. An account state contains the following /29/:

- **Account nonce**: The number of transactions sent from this account.
- **Balance**: The balance of the account.
- **Code hash**: The hash of the code of the CA. This field is empty if the account is an EoA.
- **Storage root**: The hash of the root node of the account storage trie that stores all the storage data of the account. This field is empty if the account is an EoA.

In conclusion, a full world state update can be described shortly as follows:

1. By default, the Ethereum environment is static, nothing happens and the world state remains stateless.
2. Some transactions are triggered by EoAs to either send ether, invoke a contract's function or deploy a new contract.
3. Miners collect the transactions in the order of time, take the gas cost, start to execute them one after another and deduct the gas cost during the process.
    a. If the transaction is a transfer of funds, simply verify the transaction as usual and transfer the ether included in the value field if valid and update the new balances of the involved accounts afterwards.
    b. If the transaction is deploying a new contract, the EVM runs the deployment script to create
    c. If the transaction is invoking a contract's function, the EVM executes the function and then accordingly either interact with its own internal storage by reading or writing data or trigger another contract until the execution halts.

NOTE: If there's some gas left after the transaction execution, that gas is refunded to the sender. If the gas runs out in the middle the execution, the transaction is cancelled, the state reverts to its previous confirmed version and all the gas cost is taken by the miner who eventually gets the new block confirmed.

4. After all transactions have been fully processed, transactions and receipts tries are generated based on the transactions included in the block, state trie is re-calculated based on the new states of each account (for accounts whose states did not change during the last block execution, the memory address can easily be referenced from the old trie hence there is no need to store the same data twice in the storage) and the new roots will be added into the block header representing the new world state of the Ethereum environment and it will remain till a new block is generated.

4 CAR SERVICE BOOKS USING SMART CONTRACT

This section demonstrates an example of smart contract by implementing, testing and deploying a contract of car service books.

4.1 Story and idea

Nowadays, except for modern cars, when most cars are taken to maintenance services, all the maintenance history is stored physically in car service books. This makes managing car maintenance extremely difficult as for example in some circumstances, the service books are somehow lost or maybe some parts of them are damaged. Furthermore, when it comes to selling used cars, the value of the cars with a missing or incomplete service book would drop no less than 10% as the buyers would not have enough trust in how the car was really taken care of. Therefore, having an honest complete service book is not only important for the health of the car, but also the value of the car.

Apparently, many garages today maintain their services in the on-board systems, but these systems are rather independent and not accessible by car owners. Seemingly, a smart contract storing all service history globally, where all garages, car owners and everyone can access, would be able to solve the pain being. Moreover, this implementation is dedicated to demonstrating how it is done.

NOTE: This particular idea was chosen because of the special characteristics of blockchain. In order for blockchain to gain immutability – full security, there must be a trade-off, the entire ledger must be public – zero privacy. Therefore, if this methodology is applied to businesses, the right idea can be found. Specifically, in this case, car service books' data can be stored publicly on the ledger as these are no private data, or to be exact, the car owners would not mind if one day his car service book is exposed to anyone. On the other hand, there is clearly a need for a system that can prevent the data from getting lost because of the reasons mentioned above. As a result, Ethereum smart contract seems to fit the need perfectly.

4.2 Design

The following figure sketches the most general picture of the entire network.

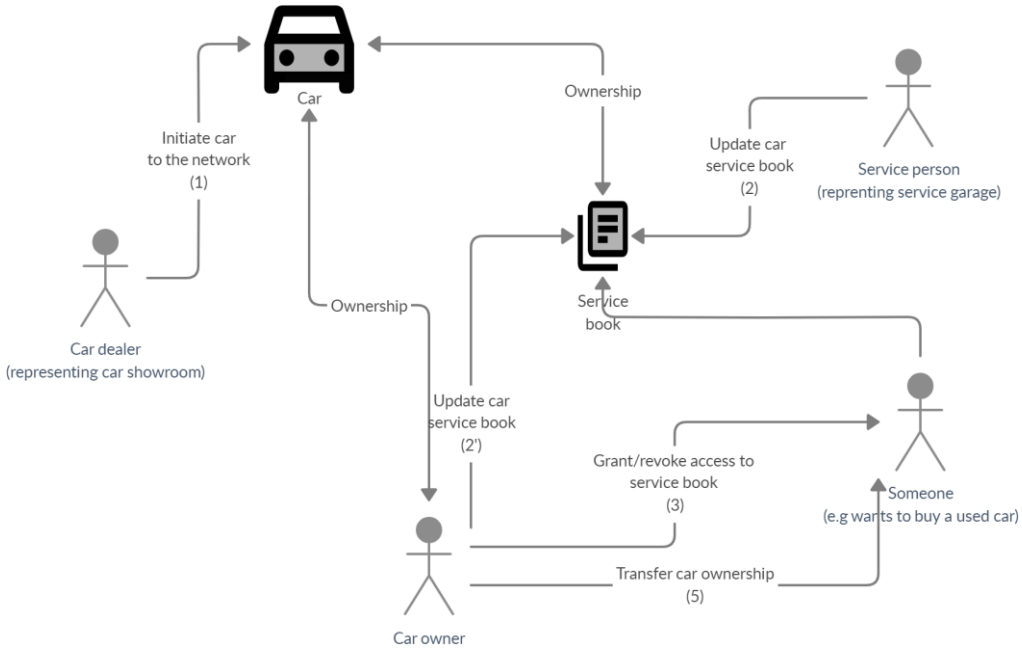

Figure 8. Network activities

Briefly, the entire network starts off empty, meaning no cars, no service persons, no service books and so on. Then car dealers and car service persons are the first objects to be added into the network by the contract owner (more on this later). When a person purchases a new car from a car dealer, s/he becomes the owner of the car and the car also gets initialized in the network. When the car needs to be maintained at a garage, the responsible service person will log what has been maintained into the contract and the logs can then be accessed by the car owner and any other service persons. The car owner can also grant or revoke access of his own car's service book to someone e.g who wants to buy his/her car so that it can be checked in advance. Because all the actions in the network are validated accordingly by the network participants based on the principles of blockchain, the service books remain honest and therefore it can potentially prevent fake service books.

4.2.1 Application diagrams

The application consists of five independent parties who will be able to perform different functions of the contract:

- **Contract owner**: When one deploys the contract to the Ethereum network, he/she will automatically become the owner of the contract on the contract creation. Figure 9 shows the use case diagram of contract owner.

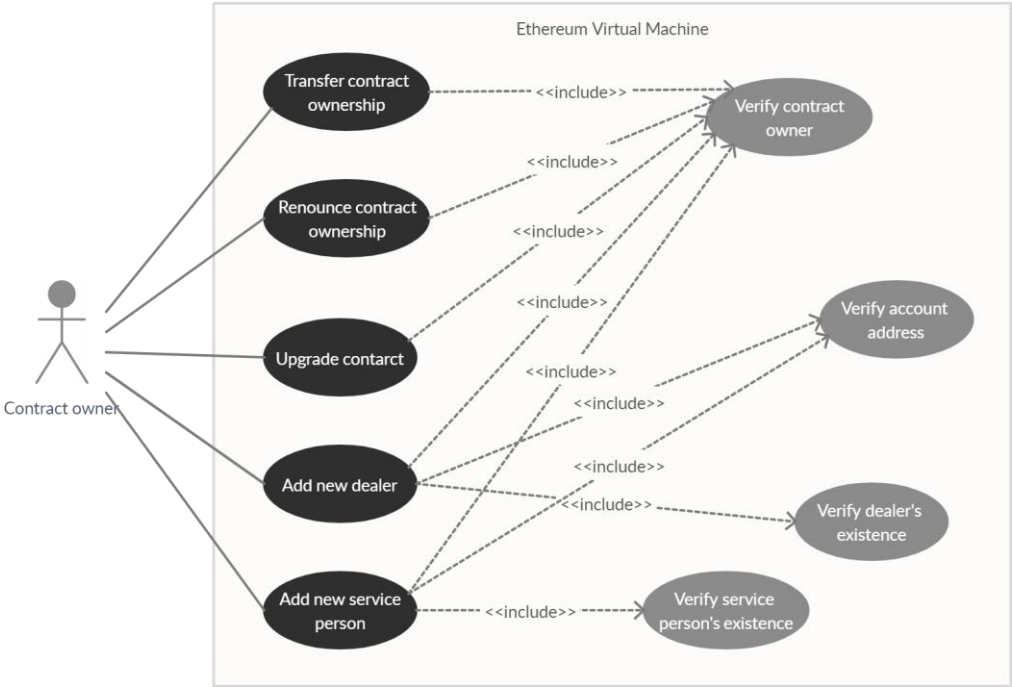

Figure 9. Use case diagram of Contract owner

- **Car dealers**: Car dealers are the ones making deals and selling car from showrooms/stores. Authorized dealers are added into the contract by the contract owner. Figure 10 shows the use case diagram of dealers.

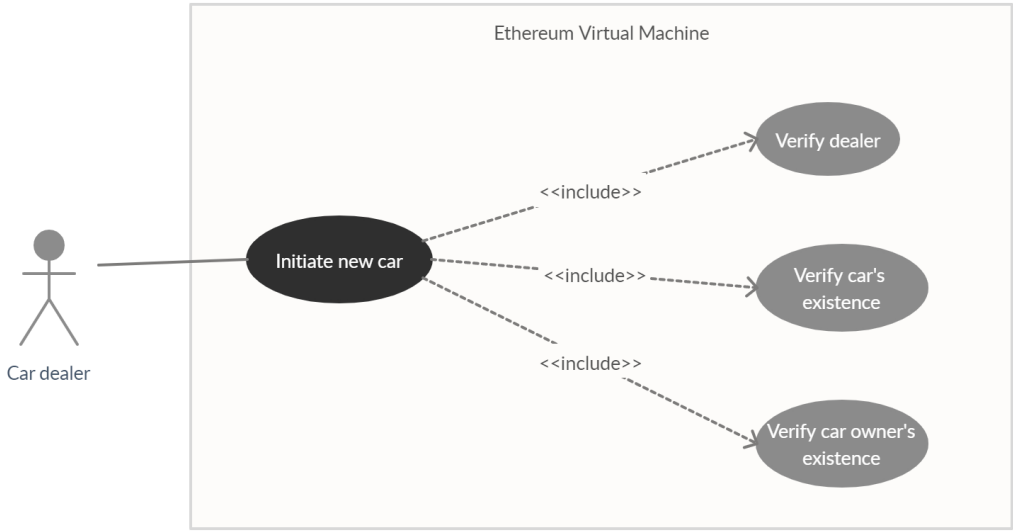

Figure 10. Use case diagram of Dealers

- **Service persons**: Service persons are the ones maintaining cars at the maintenance stores. Authorized service persons are added into the contract by the contract owner. Figure 11 shows the use case diagram of service persons.

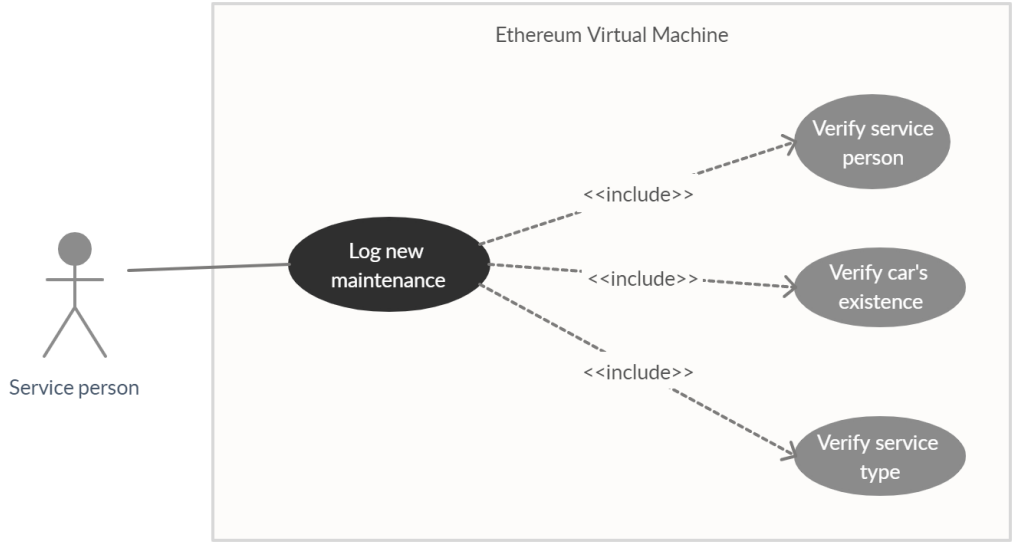

Figure 11. Use case diagram of Service persons

- **Car owners**: Car owners own cars. Everyone can register him/herself to be a car owner. Figure 12 shows the use case diagram of car owners.

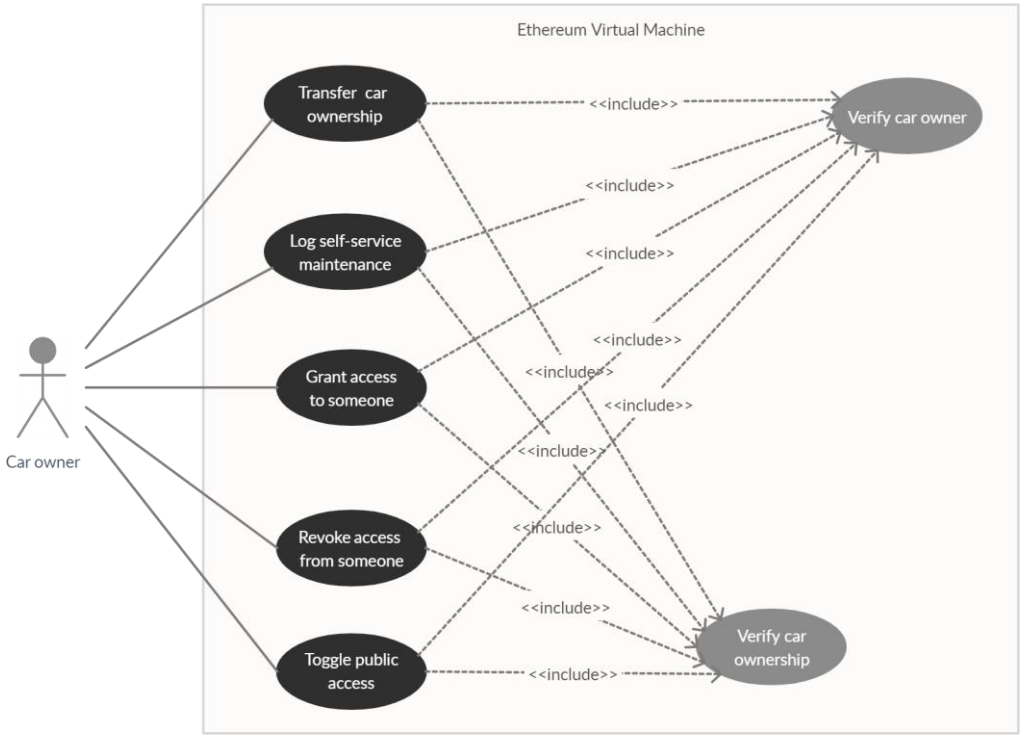

Figure 12. Use case diagram of Car owners

- **Everyone**: Everyone can perform certain functions without any authorization. Figure 13 shows the use case diagram of everyone.

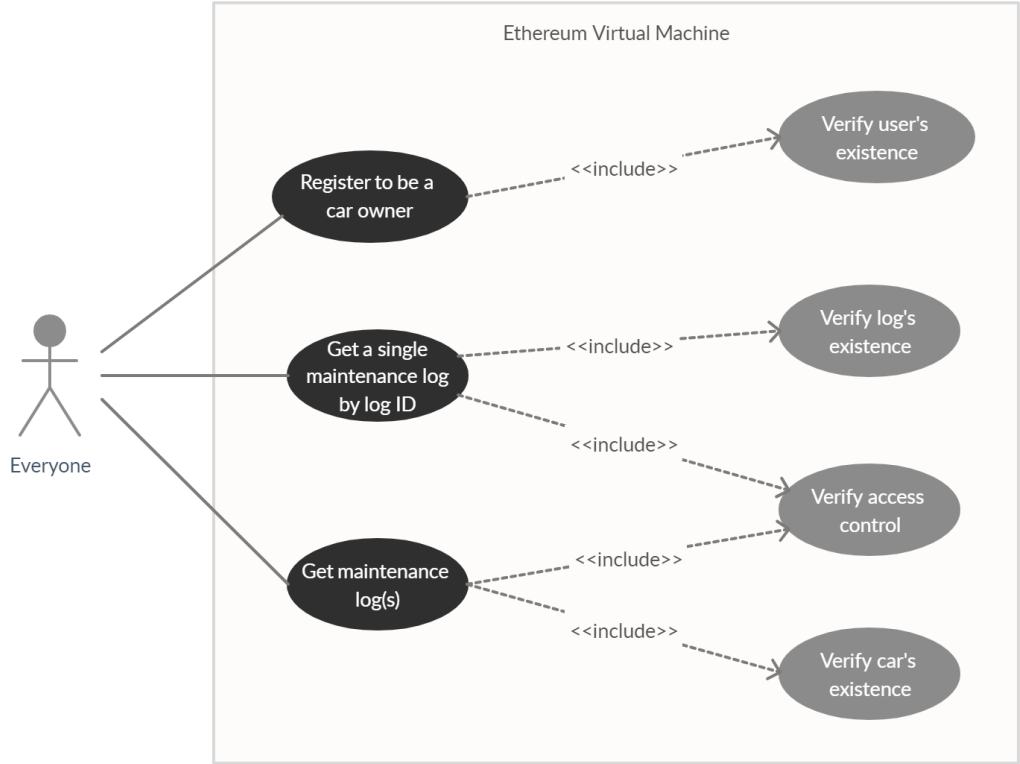

Figure 13. Use case diagram of Everyone

4.2.2 Contract structure

The implementation is split into 4 contracts and 1 library:

- **Contract Ownable**: Ownable contract is written by OpenZeppelin to manage contract ownership.
- **Library Utilizations**: Utilizations library contains some reusable technical functions. These functions are written as a library so that it would not consume as much gas fee as when written as a contract, this is one way to save gas when performing smart contracts.
- **Contract DataStorage**: DataStorage defines the data structure of the project and is where the data lives. DataStorage contains:
    - **structs**: A struct is a custom datatype that can store different properties inside of it. Code snippet 1 shows an example of struct:

```
struct Dealer {
    address id;
    bool exists;
}
```

Code Snippet 1. Example of `struct`

- **mappings:** A mapping in Solidity stores data in unique key-value pairs where key types and value types are defined in advance. Code snippet 2 shows an example of mapping:

```
mapping (address => Dealer) internal dealers;
```

Code Snippet 2. Example of `mapping`

It is important to note that using the internal visibility only prevent other contracts those are not deriving from it (external contracts) from technically accessing (reading or modifying) the data without using `this`. Otherwise, everything in smart contracts is visible to the whole world. The same to all other visibilities, including external, public and even private. They are just for technical uses.

- **Contract CarMaintenance**: CarMaintenance contains all the business logic functions that people can see and execute based on which party they belong to. Otherwise access control is managed by modifiers and logging is managed by events.
  - **modifiers**: Modifiers are functions but are usually used to check some requirements prior to executing the main function. In this project, modifiers are used as authorization and access control layer. Code Snippet 3 shows an example of modifier and how it's used:

```
modifier onlyDealer() {
        require(dealers[msg.sender].exists, "Only dealers
are allowed to invoke this function.");
        _;
}
function initiateCar(address _carOwnerId) public
    onlyDealer()
    carOwnerExists(_carOwnerId)
    returns(address)
{ // ...Function body here... }
```

Code Snippet 3. Example of `modifier`

So, the `initiateCar()` function can only be executed by an authorized dealer, otherwise the error message "Only dealers are allowed to invoke this function." will be thrown. The symbol `_;` indicates where the next part can be inserted, it could be the body of the function or in this case it is another modifier, `carOwnerExists()`. Therefore, _ can be used as many times as needed (1 is minimum) and placed anywhere inside of a modifier, depending on the requirements of the function.

- events: value of variables can be logged at certain points of time by triggering events, as shown in the following snippet.

```
event NewLogAdded(address _newLogId);
emit NewLogAdded(newLogId);
```

Code Snippet 4. Example of defining and triggering an event

- **Contract Proxy**: Proxy contract contains delegatecall which is a low-level function used to invoke functions from another contract, library. More on this will be covered later.

4.3 Implementation

This section will go into details how exactly the example of smart contract is implemented with Solidity.

Solidity is a contract-oriented language inspired by Javascript, Python and C++, so during the implementation, many of the features of Solidity will be seen very similar to those three languages.

4.3.1 Contract Ownable

Writing a good contract is extremely important as deploying a contract is very costly and once the contract is deployed, it is locked in the network forever. If by chance, some bugs are noticed on the deployed contract, the only way to fix it is to debug and re-deploy another contract and ignore the earlier contract. Therefore, auditing contracts before deploying is considered one of the most important steps when developing smart contracts where many factors are taken into account such as gas fee management, timing performance, etc. Ownable contract /31/, written by OpenZepplin, is one the contracts that was written and audited very carefully and strictly.

OpenZepplin wrote it with the purpose of managing the ownership of its own, but the contract itself does nothing else, it most of time is inherited by other contracts, so the ownership control is managed the same way there in the extended contracts. Specifically, when one EoA deploys a contract that inherits the Ownable contract, that account would automatically become the initial owner of the contract by using `constructor()`, as shown in Code Snippet 5

```
contract Ownable {
    address private _owner;

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }
    ...
}
```

Code Snippet 5. Ownable's constructor

The contract starts off with contract definition `contract Ownable`, followed by state variables, `address private _owner` in this case. `address` data type has the size of an Ethereum address, meaning it holds a 20-byte value.

After that, `constructor()` is defined. For other object-oriented languages, a constructor is called once an instance of that class is initialized. Similarly, a Solidity constructor is called once the contract is initialized/deployed in Ethereum network, this would be taken care of by the EVM. `msg.sender` indicates the account address

of the one who proceeds the action, it could be one who deploys a contract, or one who invokes a function. From now on, the term *sender* will be referred to as the account who invokes the function. In this case, the constructor assigns the address of the account who deploys the contract to the state variable `_owner`. But it is not clear how ownership is managed in the contract. As mentioned earlier, there are so-called modifiers in Solidity, allowing to verify some conditions in prior to executing functions. The following snippet demonstrates how access control could be defined with contract owner:

```
modifier onlyOwner() {
    require(isOwner(), "Ownable: caller is not the owner");
    _;
}
function isOwner() public view returns (bool) {
    return msg.sender == _owner;
}
```

Code Snippet 6. Modifier with contract owner

`require` keyword says that the statement inside it is required to be true to be able to advance, otherwise the exception message will be thrown. So, before any function is executed, if modifier `onlyOwner()` is specified, it requires the `msg.sender` to be equal to `_owner` since operators can be used to compare Solidity addresses, otherwise throw the error message `Ownable: caller is not the owner,` as shown in Code Snippet 7:

```
function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
}
```

Code Snippet 7. Example of function with modifier

Now some certain functions can be limited to be invoked by only the contract owner, this can be seen as a way to control user access in Solidity smart contract and will be used a lot in this implementation. For example, Code Snippet 8 indicates that OpenZeppelin has given the contract owner the ability to transfer the contract ownership to another account, or leave the contract without an owner if one day the current owner gives up on this role.

```
function renounceOwnership() public onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}

function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
}

function _transferOwnership(address newOwner) internal {
     require(newOwner != address(0), "Ownable: new owner is the zero
     address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
event OwnershipTransferred(address indexed previousOwner, address
indexed newOwner);
```

Code Snippet 8. Contract owner's ability in Ownable contract

Another thing to be noticed here is the `event OwnershipTransferred()`. Solidity allows logging through triggering events. Events would log the value(s) of the required variable(s) at the moment they are triggered and return them in the transaction receipt's logs. Events are triggered by the keyword `emit`. Therefore, if `transferOwnership()` is called properly, the event `OwnershipTransferred()` will be fired at the line `emit OwnershipTransferred(_owner, newOwner)` taking the value of _owner at that moment to be the value of `previousOwner` and the value of `newOwner` to be the value of `newOwner` in the event and get included in the receipt whose transaction invoked the function afterwards, as shown in the following figure.

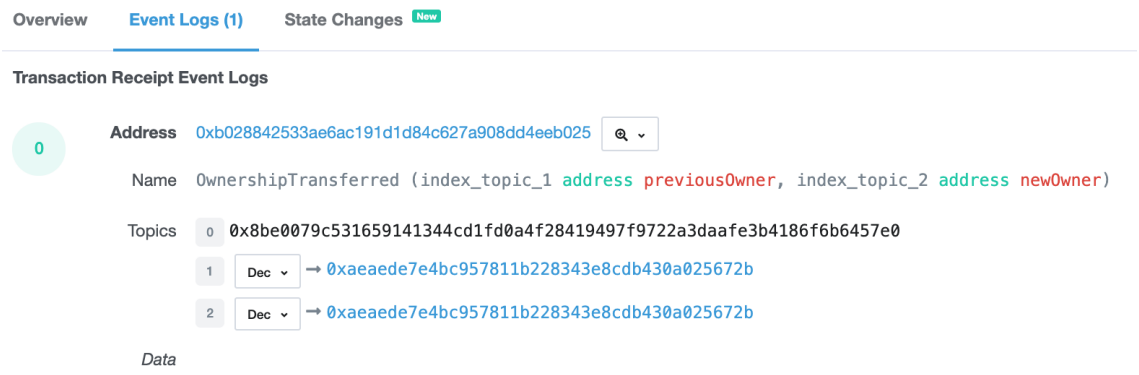

Figure 14. Event logs in Transaction receipt

4.3.2 Library Utilizations

Solidity libraries are similar to contracts, but they do not have any state variables as their purpose is not to store data but to contain reusable functions that can be called from other contracts to perform some simple operations based on input and return results. Libraries are usually supposed to be deployed only once, but since then, if there are no flaws being, they would just stay there on the network and be linked to other contracts as many times as needed.

Utilizations library consists of two main functions, `getUniqueId()` /32/ and `deleteByValue()` in which `getUniqueId()` was originally written by Nik Kalyani and shared on github. `getUniqueId()` is used to generate random Ethereum-looking-like addresses. However, in this project, that function has been modified a bit to fit in some certain requirements. Specifically, Code Snippet 9 shows that `getUniqueId()` now requires one parameter called `_blockAhead`. Solidity does not provide such functions to generate any random values, so the job is to manually generate addresses as randomly as possible.

```
function getUniqueId(uint256 _blockAhead) public view returns (address)
{
    bytes20 b = bytes20(keccak256(abi.encodePacked(msg.sender,
    block.number + _blockAhead)));
    uint addr = 0;
    for (uint index = b.length-1; index+1 > 0; index--) {
        addr += uint(uint8(b[index])) * ( 16 ** ((b.length –
             index - 1) * 2));
    }
    return address(addr);
}
```

Code Snippet 9. `getUniqueId()` in Utilizations library /32/

The idea is to rely on the most variable factors in Ethereum, the address of the sender (`msg.sender`) and the current block number (`block.number`). Additionally, `_blockAhead` is used as for example, one sender invokes a function that uses `getUniqueId()` more than once for different purposes, relying on only the sender's address and the block number would not be enough as they would result the same address. Therefore, `_blockAhead` can gives more flexibility. These three factors are afterwards hashed altogether and after some mathematical operations, it returns a "random" address in `return address(addr);`.

As Solidity does not provide function to delete element in arrays, the purpose of the second function, `deleteByValue()`, is to get the job done. As shown in Code Snippet 10, the function takes an array `_arr` and a value in the array `_value` as parameters.

```
function deleteByValue(address[] storage _arr, address _value)
    private {
    uint index = findIndex(_arr, _value);
    deleteByIndex(_arr, index);
}
```

Code Snippet 10. `deleteByValue()` in Utilizations library

The function then is operated in a way that it first needs to find the index of the given value in the array by calling another function, `findIndex`(). This is done by just a simple while loop through the array, as indicated in Code Snippet 11.

```
function findIndex(address[] memory _arr, address _value) private
    pure
    returns (uint)
{
    uint i = 0;
    while (_arr[i] != _value) {
        i++;
    }

    return i;
}
```

Code Snippet 11. `findIndex()` in Utilizations library

`findIndex()` was marked as `pure` (Solidity state modifier) because the function does not read or write the storage state at any point but only deals with some local variables.

And then the returned index will be used to remove the element from the array by `deleteByIndex()`. Since there should not be any gap in the array as for better performance, the deletion can be done by first switching the index of the last element and the element that is to be deleted and then just simply delete the last element from the array and reduce the length of the array as it is now one size shorter, referring to the snippet below.

```
function deleteByIndex(address[] storage _arr, uint _index) internal {
    _arr[_index] = _arr[_arr.length - 1];
    delete _arr[_arr.length - 1];
    _arr.length--;
}
```

Code Snippet 12. `deleteByIndex()` in Utilizations library

Throughout those three previously shown functions, one thing to be noticed is the `memory` and `storage` keywords. In Section 3.3.4, it was mentioned that EVM can store data in three areas, stack, memory and storage. The keywords `memory` and `storage` therefore indicate where the data should be stored. In the `deleteByValue()` and `deleteByIndex()` functions, `storage` data location was used for the `address[]` because an array located in storage was to be modified, defining the data location of the parameter as `storage` would simply reference to the data that is already allocated in storage, meaning no new data is created and it would be possible to modify

the state array. If it was defined as `memory` data, it would create a copy of the data in the memory and any modifications with the `memory` data would not reflect on the `storage` data. Another reason is that `memory` arrays require fixed size, meaning doing `_arr.length--` would not be possible. On the other hand, in the `findIndex()` function, as there was no intention to do anything with the state data but a simple while loop, `memory` data location could be applied there.

Using libraries is considered as an economical way of developing Solidity smart contracts not only because of its reusability, but also linking library to a contract can most of the time save some gas comparing to contract inheritance. Solidity also supports contract inheritance but it works in a way of copying code from the base contract(s) to the derived contract, so after the contract is compiled, the bytecode of the derived contract would literally be a sum of the bytecode of the base contract(s) and the bytecode of the derived contract itself. Meanwhile, if linking library is used, the final compiled bytecode would just leave a little placeholder where the library function is called for the address of the required library to be filled in after the contract is deployed, therefore reducing the size the bytecode and the gas to be consumed. As demonstrated in the following figure, deploying CarMaintenance2 contract with the inheritance of Utilizations library costs 230,885 gas.

```
Deploying 'CarMaintenance2'
---------------------------
> transaction hash:    0xc8f5abb532ec66e20a4bb27a3c502aedb7c68fd70318921abc8d4c0a37b503c3
> Blocks: 0            Seconds: 0
> contract address:    0xe044397A4E3660aC90bf0568396153e3D338205f
> block number:        10
> block timestamp:     1568190400
> account:             0x04ff275A3a8A62592493FBE515C47Fc8a69A222C
> balance:             99.5372279
> gas used:            230885
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.0046177 ETH

> Saving artifacts
-------------------------------------
> Total cost:           0.0046177 ETH
```

Figure 15. Contract with inheritance

But in Figure 16, deploying the CarMaintenance2 contract with the linking to Utilizations library costs only 217,211 gas.

```
Linking
-------
* Contract: CarMaintenance2 <--> Library: Utilizations (at address: 0x972504a4245e449D2F2C4155Da5bC346cD9191cA)

Replacing 'CarMaintenance2'
---------------------------
> transaction hash:    0x27104aa6cbce3d79da10865a2c1173fecebc35a448bddfbcc6cff55a2c20f660
> Blocks: 0            Seconds: 0
> contract address:    0x71aE40Cee9bCeD7461620c06Bd263651F56Dca0D
> block number:        13
> block timestamp:     1568190586
> account:             0x04ff275A3a8A62592493FBE515C47Fc8a69A222C
> balance:             99.5280563
> gas used:            217211
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.00434422 ETH
```

Figure 16. Contract with linking library

However, this trick is not always true as sometimes it depends on other factors such as the size of the library, or how many times the library functions are used. So, in some circumstances, deploying contract with linking library does not save but even consumes more gas than deploying contract with contract inheritance. But still, as previously mentioned, one clear advantage is that library does not need to be deployed over and over again which already skyrockets the gas saving as well as the use of reusability.

4.3.3 Contract DataStorage

All the contract data is organized and stored in DataStorage contract with `struct` and `mapping`. Every involved entity in this project has its own `struct`, including `Dealer`, `ServicePerson`, `Car`, `CarOwner`, `Log` and `AccessSession`.

Specifically, `Dealer`, `ServicePerson` and `CarOwner` are the three entities that are able to invoke functions externally, id of each individual is their own account address, otherwise no private user data is stored on the network since they should not be exposed to public, as shown in Code Snippet 13.

```
struct Dealer {
    address id;
    bool exists;
}

struct ServicePerson {
    address id;
    bool exists;
}

struct CarOwner {
    address id;
    bool exists;
}
```

Code Snippet 13. `struct` of `Dealer`, `ServicePerson` and `CarOwner`

Each log should also have an ID for management and lookup, they are randomly generated during the log creation process in form of an Ethereum address. Code Snippet 14 shows the `struct` of `Log`.

```
struct Log {
    address id;
    address servicePersonId;
    string carId;
    uint mileage;
    ServiceTypes serviceType;
    string description;
    string nextServiceDue;
    uint createdAt;
    bool exists;
}
```

Code Snippet 14. `struct` of `Log`

`SeriveTypes` in the `struct` of `Log` is defined as an `enum`, as seen in Code Snippet 15.

```
enum ServiceTypes {
    SelfService,
    MinorService,
    MajorService,
    Inspection
}
```

Code Snippet 15. `enum` of `ServiceTypes`

`Car` is the entity that connects others. The relationships of `Car` with other entities are visualized in the figure below.

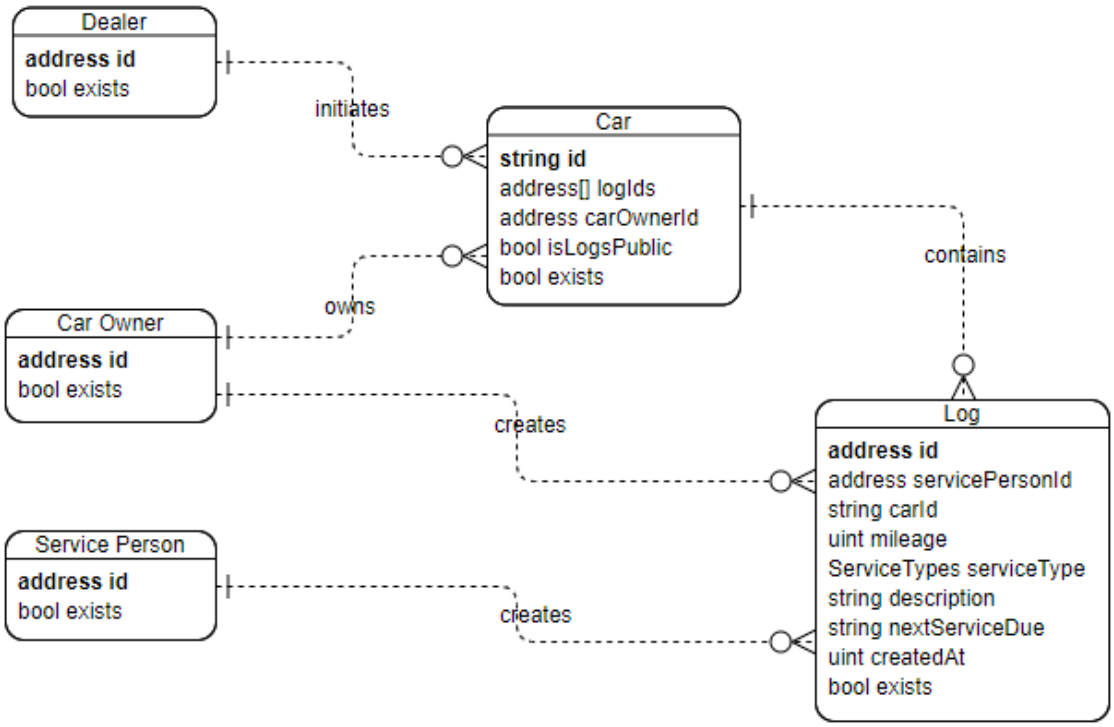

Figure 17. Entity relationship diagram

Id of each car is its VIN (vehicle identification number). `isLogPublic` is a boolean field of `Car` that is false by default, if it is set to true, the entire service book of the car can be seen by anyone without authorization. `struct` of `Car` is shown in Code Snippet 16.

```
struct Car {
    string id;
    address[] logIds;
    address carOwnerId;
    bool isLogPublic;
    bool exists;
}
```

Code Snippet 16. `struct` of `Car`

Read permission of service books is by default possible to only car owners and service persons, other accounts would need access granted by the car owners. `AccessSession` struct tells the time limit of each access to the logs, as Code Snippet 17 indicates.

```
struct AccessSession {
    uint validFrom;
    uint durationInMinutes;
    bool exists;
}
```

Code Snippet 17. `struct` of `AccessSession`

`AccessSession` is used as the value in key-value pairs stored in `bearerAccesses` mapping to maintain a list of accounts and sessions of those who have access to the maintenance logs, as shown in Code Snippet 18.

```
mapping (string => mapping(address => AccessSession)) internal
bearerAccesses;
//       carId  => mapping(bearerId => AccessSession))
```

Code Snippet 18. `mapping` of `bearerAccesses`

`logBearerList` mapping stores list of bearers grouped by car IDs, as shown in Code Snippet 19.

```
mapping (string => address[]) internal logBearerList;
//       carID  => bearerId[]
```

Code Snippet 19. `mapping` of `logBearerList`

The lists of bearers are stored as arrays because they will be needed when executing loops.

In Code Snippet 20, other mappings simply store lists of objects with defined structs as the value and the key is the id of each object.

```
mapping (address => Dealer) internal dealers;
mapping (address => ServicePerson) internal servicePersons;
mapping (string => Car) internal cars;
mapping (address => CarOwner) internal carOwners;
mapping (address => Log) internal logs;
```

Code Snippet 20. `mapping` of Entities

4.3.4 Contract CarMaintenance

CarMaintenance contract consists of three parts, logic functions, modifiers and events. All these together will help to operate the contract.

- **Modifiers**

Modifiers are used to manage access when external accounts attempt to invoke functions. Modifiers shown in Code Snippet 21 control access by user role, these modifiers are required at the first place in prior to invoking any functions that requires authorization. The check is done by checking if `exists` property in each object is `true` since when adding new object, this is set to true, otherwise, the object does not exist in the mapping.

```
modifier onlyDealer() {
    require(dealers[msg.sender].exists, "Caller is not a dealer.");
    _;
}

modifier onlyServicePerson() {
     require(servicePersons[msg.sender].exists, "Caller is not a
     service person.");
    _;
}

modifier onlyCarOwner() {
    require(carOwners[msg.sender].exists, "Caller is not a car owner.");
    _;
}
```

Code Snippet 21. modifiers checking user role

`modifier carBelongsToOwner()` requires the sender to be the car owner of the car in order to advance on the function. This is done by ensuring the value of `carOwnerId` property `Car` instance is equal to the sender's address, as shown in the following code snippet.

```
modifier carBelongsToOwner(string memory _carId) {
    require(cars[_carId].carOwnerId == msg.sender, "This car does not
belong to the person.");
    _;
}
```

Code Snippet 22. modifier authorizing car ownership

`modifier accessible()` makes sure the sender is allowed to view the requested log(s) of a car. This is done by calling `isAccessible()` function which would check first if the logs are set to public, else if the sender is a service person, else if the sender is a car owner and owns the car, else finally if the sender has valid access session listed in `bearerAccesses` mapping, as shown in Code Snippet 23.

```
modifier accessible(string memory _carId) {
    require(isAccessible(_carId, msg.sender), "Access denied.");
    _;
}

function isAccessible(string memory _carId, address _bearerId) private
view returns(bool)
{
    if (cars[_carId].isLogsPublic) { // everyone can access if all
maintenance logs of the car is public
        return true;
    } else if (servicePersons[_bearerId].exists) { // all service persons
have access to all maintenance logs
        return true;
    } else if (carOwners[_bearerId].exists && cars[_carId].carOwnerId
== _bearerId) { // car owners have access to their own car(s)' logs
        return true;
    }

    AccessSession memory session = bearerAccesses[_carId][_bearerId];

    return (session.exists && now <= (session.validFrom
+ (session.durationInMinutes * 1 minutes))); // accessible with time
limit decided earlier by the car owner
}
```

Code Snippet 23. modifier authorizing accessibility

`now` (alias for `block.timestamp`) keyword indicates the current block timestamp (when it was mined) instead of the current world timestamp. This may be some milliseconds to a couple of seconds behind the current world timestamp /33/. The time limit is calculated in minutes. It was checked in a way that it compares the current block timestamp to the timestamp of the session's end time (`session.validFrom + (session.durationInMinutes * 1 minutes)`). The `minutes` is a Solidity suffix used to convert from minutes to seconds /33/.

- **Events**

Events are mainly used for development or testing purposes. Code Snippet 24 illustrates all events defined and used in CarMaintenance contract.

```
event NewLogAdded(address _newLogId);
event NewCarInitiated(string _newCarId);
event newCarOwnerAddedDuringCarInitiation(address _newCarOwner);
event CarOwnershipTransferred(address indexed _previousOwner, address
indexed _newOwner);
event AccessGranted(string indexed _carId, address indexed _bearerId);
event AccessRevoked(string indexed _carId, address indexed _bearerId);
event AfterToggle(bool _isPublic);
event isServicePerson(bool  _result);
```

Code Snippet 24. `events` in CarMaintenance

- **Functions for Everyone**

Everyone fairly has the ability to register to be a car owner in the network or query individual log by a specific log ID or query the entire service book of a car by the car ID. Even though there is no user role control for these functions, but in order to get the log(s), the sender must have valid access first by going through modifier accessible(), as shown in the code snippet below.

```
function addCarOwnerMyself() public
{
    carOwners[msg.sender] = CarOwner(msg.sender, true);
}

function getLogsByCarId(string memory _carId) public view
    carExists(_carId)
    accessible(_carId)
    returns(address[] memory, ServiceTypes[] memory, uint[] memory)
{
    address[] memory logIds = cars[_carId].logIds; // array of logIds
    ServiceTypes[] memory serviceTypes = new ServiceTypes[](logIds.length);
// array of service types
    uint[] memory createdAts = new uint[](logIds.length); // array of
created times

    for (uint i = 0; i < logIds.length; i++) {
        Log memory r = logs[logIds[i]];
        serviceTypes[i] = r.serviceType;
        createdAts[i] = r.createdAt;
    }

    return (logIds, serviceTypes, createdAts);
}

function getLogByLogId(address _logId) public view
    logExists(_logId)
    accessible(logs[_logId].carId)
    returns(Log memory)
{
    Log memory l = logs[_logId];

    return l;
}
```

Code Snippet 25. Functions for everyone

Both of the get log functions above were marked as `view`, another Solidity state modifier, because these functions do not intend to modify the state of the contract but only read it, they then do not create any transactions on the function calls and the return will be fetched immediately.

In `getLogsByCarId()` function, instead of returning array of `Log` (`Log[]`), it returns separate arrays because Solidity at the moment does not support the option to return array of struct objects. Therefore, instead of looking like Code Snippet 26, the result will look more like Code Snippet 27.

```
[
    (Log_1.id, Log_1.servicePersonId, Log_1.description, etc),
    (Log_2.id, Log_2.servicePersonId, Log_2.description, etc),
    (Log_3.id, Log_3.servicePersonId, Log_3.description, etc),
];
```

Code Snippet 26. Array of struct objects

```
[Log_1.id, Log_2.id, Log_3.id],
[Log_1.servicePersonId, Log_2.servicePersonId, Log_3.servicePersonId],
[Log_1.description, Log_2.description, Log_3.description]
```

Code Snippet 27. Arrays of struct properties

- **Functions for Contract Owner**

The contract owner is able to add new dealers and service persons into the network by adding each of them into the corresponding mappings defined in `DataStorage` contract. The sender must be the contract owner to be able to invoke the functions, and the account addresses must be valid (invalid address is 0x0000000000000000000000000000000000000000), therefore `onlyOwner()` and `validAddress()` modifiers are used, as shown in the following snippet.

```
function addDealer(address _dealerId) public
    onlyOwner()
    validAddress(_dealerId)
{
    dealers[_dealerId] = Dealer(_dealerId, true);
}

function addServicePerson(address _servicePersonId) public
    onlyOwner()
    validAddress(_servicePersonId)
{
        servicePersons[_servicePersonId] = ServicePerson(
_servicePersonId, true);
}
```

Code Snippet 28. Functions adding new dealers and service persons

NOTE: This is not going to happen in reality as only the contract owner has the ability to add new dealers and service persons, which makes it very much centralized, but blockchain is meant to be decentralized, meaning no central server has control over anything. However, this project does not intend to solve that problem

in its scope, here it is assumed that all added dealers and service persons have successfully gone through a decentralized authorization step to be included in the network.

- **Functions for Dealers**

Dealers are able to initiate new cars in the network after they have been sold. The cars are registered by their VIN (vehicle identification number) as these are global unique ID for cars. During the initiation process, if the car owner is not yet registered in the network, it will create one `event newCarOwnerAddedDuringCarInitiation` will be triggered afterwards, given the account address of the car buyer, referring to Code Snippet 29.

```
function initiateCar(string memory _carId, address _carOwnerId) public
    onlyDealer()
    carNotExists(_carId)
{
    if (!carOwners[_carOwnerId].exists) {
        carOwners[_carOwnerId] = CarOwner(_carOwnerId, true);
         emit newCarOwnerAddedDuringCarInitiation(_carOwnerId);
    }

    address[] memory logIds;
    cars[_carId] = Car(_carId, logIds, _carOwnerId, false, true);
}
```

Code Snippet 29. Car dealer initiating new car

`carNotExists()` must be required in advance since if a registered car is initiated again with another car owner ID, it would override the former car owner ID and he would lose the entire control on his own car.

- **Functions for Service persons**

Service persons can upload new maintenance logs, telling which car was maintained, which service type it was, what is the current mileage of the car, what has been maintained, and when will be the next service due by either date or mileage. New log IDs are generated by using `getUniqueId()` function from Utilizations library, as shown in Code Snippet 30.

```
function logMaintenanceByServicePerson(
    string memory _carId, uint _mileage, ServiceTypes _type, string
memory _description, string memory _nextServiceDue)
    public
    onlyServicePerson()
    carExists(_carId)
    servicePersonExists(msg.sender)
    validServiceTypeByServicePerson(_type)
    returns(address)
{
    address newLogId = Utilizations.getUniqueId(0);
    logs[newLogId] = Log(newLogId, msg.sender, _carId, _mileage,
_type, _description, _nextServiceDue, now, true);

    cars[_carId].logIds.push(newLogId);
    emit NewLogAdded(newLogId);

    return (newLogId);
}
```

Code Snippet 30. Service persons logging new maintenance log

`validServiceTypeByServicePerson()` modifier makes sure the service person applies the proper service type (minor, major or inspection), there is self-service type but it can only be used by car owners.

- **Functions for Car owners**

Car owners have the ability to transfer their own car's ownership to another one when selling it to another owner by assigning the address of the new car owner (`_newOwnerId`) to `carOwnerId` property of the `Car` object. As a result, all the access right to the car's service book would belong to the new owner afterwards, as shown in Code Snippet 31.

```
function transferCarOwnership(string memory _carId, address
_newOwnerId) public
    onlyCarOwner()
    carBelongsToOwner(_carId)
    carOwnerExists(_newOwnerId)
{
     cars[_carId].carOwnerId = _newOwnerId;
     emit CarOwnershipTransferred(msg.sender, _newOwnerId);
}
```

Code Snippet 31. Car owner transferring his car's ownership

Before buying a used car, the buyer would like to check the car's service book, but currently he/she does not have access to it, the current car owner can grant the access to him/her with a time limit, as shown in the following code snippet.

```
function grantAccess(string memory _carId, address _bearerId,
uint _durationInMinutes) public
    onlyCarOwner()
    carBelongsToOwner(_carId)
{
    bearerAccesses[_carId][_bearerId] = AccessSession(now,
_durationInMinutes, true);
    logBearerList[_carId].push(_bearerId);

    emit AccessGranted(_carId, _bearerId);
}
```

Code Snippet 32. Car owner granting access to someone

Accordingly, the access session will be valid from the current block timestamp (`now`) until as long as the `_durationInMinutes` indicates in minutes.

Code snippet 33 shows that the permission can also be revoked immediately if the car owner invokes `revokeAccess()` function.

```
function revokeAccess(string memory _carId, address _bearerId) public
    onlyCarOwner()
    carBelongsToOwner(_carId)
{
    delete bearerAccesses[_carId][_bearerId];
    Utilizations.deleteByValue(logBearerList[_carId], _bearerId);

    emit AccessRemoved(_carId, _bearerId);
}
```

Code Snippet 33. Car owner revoking access from someone

Here `deleteByValue()`, the function defined in Utilizations library, was used to remove an element from `logBearerList` array. However, this function only prevents the bearer from querying the service book from the moment the access is revoked, but if the bearer has already successfully queried the log(s) when his access was permitted, he would still be able to see what was queried earlier.

Otherwise, the car owner can always toggle the service book's public access so that if it is switched to `true`, anyone could query it without any authorization, else only the car owner or service persons or permitted person(s), as seen in Code Snippet 34.

```
function togglePublicAccess(string memory _carId)
    onlyCarOwner()
    carBelongsToOwner(_carId)
    public
{
    Car storage car = cars[_carId];
    car.isLogsPublic = !car.isLogsPublic;
    emit AfterToggle(car.isLogsPublic);
}
```

Code Snippet 34. Car owner toggling a service book's public access

The `storage` data location must be used as it intends to modify the state data, otherwise if it was `memory` keyword, it would create a copy in the memory and state data would not get affected.

Finally, many car owners can maintain his/her car(s) him/herself at some certain level such as changing the tires, fixing the light bulbs, etc. All those changes can also be logged as self-service into the network so that they can keep track when it

was done the last time, and new log IDs will also be generated here for lookup and management, as shown in the snippet below.

```
function logMaintenanceByCarOwner(string memory _carId, string
memory _description) public
    onlyCarOwner()
    carBelongsToOwner(_carId)
    returns(address)
{
    address newLogId = Utilizations.getUniqueId(0);
    logs[newLogId] = Log(newLogId, msg.sender, _carId,
0, ServiceTypes.SelfService, _description, "", now, true);
    cars[_carId].logIds.push(newLogId);
    emit NewLogAdded(newLogId);

    return (newLogId);
}
```

Code Snippet 35. Car owner logging new self-service log

It can be seen that all the functions for car owners must go through `onlyCarOwner()` and `carBelongsToOwner()` modifiers to prevent unauthorized actions.

4.3.5 Contract Proxy

As all contract data is stored in the contract storage and given that once a smart contract is deployed to Ethereum network, the contract remains there unchanged forever. There comes a critical issue with smart contract development, that no matter how careful the contract auditing phase was proceeded, there is always possibility that the contract would have some flaws in the long run. The only way to correct the contract is to re-deploy a new version of it and start to use that new one, but new contract would mean new and empty storage and all the data being in the previous version is simply ignored. In short, if smart contracts are not designed in a way that they could be upgraded, those smart contracts are non-upgradeable and simply there comes a disaster. Here is how Proxy contract comes to play.

In this project, Proxy contract makes our smart contracts mostly upgradeable. Specifically, Proxy contract's mechanism is described as follows:

- Proxy contract uses delegatecall. Solidity delegatecalls are low level functions that help to invoke function from another contract or library at runtime, given the address of the destination contract/library.
- Proxy contract uses fallback function mechanism. The fallback function is triggered when invoking a function that does not exist in the contract and returns whatever the invoked function returns.

Now it is important to have a deeper dive into the delegatecall. When delegating a call, the code will be loaded from the contract that is called upon (called contract) but executed in the context of the contract that delegates the call (calling contract). This means that storage, state variables, msg.sender etc will refer to the calling contract. In our case, calling contract which is Proxy contract would handle all the data storage, therefore called **Storage contract**, and called contract which is CarMaintenance contract has nothing to do with the storage but only logic functions that are called by the delegatecall, therefore called **Logic contract**. As shown in Code Snippet 36, OpenZeppelin has wrapped a delegatecall inside a fallback function, making contract upgradeability feasible. The implementation of delegatecall and fallback function for Proxy contract in this project is taken from Openzeppelin.

```
function () payable external { //fallback function
    address impl = _implementation;
    require(impl != address(0));
    assembly {
        let ptr := mload(0x40)
        calldatacopy(ptr, 0, calldatasize)
        let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
        let size := returndatasize
        returndatacopy(ptr, 0, size)

        switch result
        case 0 { revert(ptr, size) }
        default { return(ptr, size) }
    }
}
```

Code Snippet 36. Delegatecall wrapped inside a fallback funcion

With Proxy, transaction flow would go a bit differently. Function calls will be made from Proxy instead of CarMaintenance. The functions do not physically exist in Proxy contract, so fallback function will be triggered, delegating the calls to the

address (stored in storage of Proxy) of the newest version of CarMaintenance, as briefly illustrated in Figure 18.

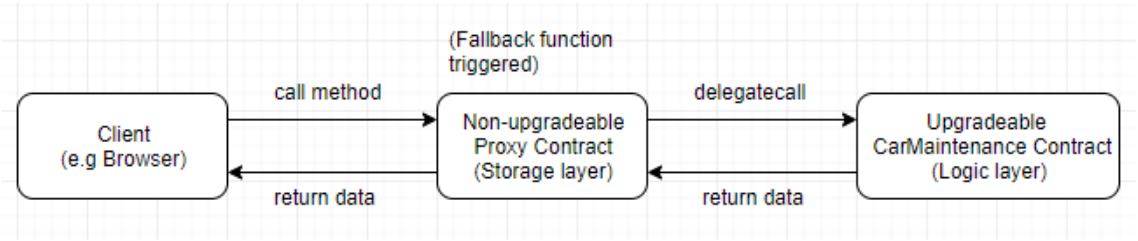

Figure 18. Transaction flow with Proxy contract

If by chance, some bugs appear in CarMaintenance contract, a e.g CarMaintenance_Ver2 will be deployed with the bugs fixed. Now Proxy contract needs to know about the address of the newly deployed contract so that the function calls can be delegated to there. This is done by calling `upgradeTo()` in Proxy contract, as shown in Code Snippet 37. `upgradeTo()` is called when the first version of logic contract has been deployed.

```
function upgradeTo(address _newImplementation)
    external onlyOwner
{
    require(_implementation != _newImplementation);
    _setImplementation(_newImplementation);
}
```

Code Snippet 37. `upgradeTo()` in Proxy contract

However, as all data is stored in Proxy contract, only logic functions could be upgraded, all new versions of logic contract must have the same data structure as the first one so that Proxy contract will not break. Usually, new versions of logic contract will just simply inherit its previous version to maintain the data structure and modify what is needed with the logics. This is why this Proxy implementation does not provide full ability to upgrade a smart contract.

5 TESTING AND DEPLOYMENT RESULTS

This project is developed with the support of Truffle Suite. Truffle is a development environment, testing framework and asset pipeline for Ethereum that helps to compile, link, deploy, etc smart contracts in a blink of an eye.

5.1 Testing

In order to be able to run smart contract tests, there needs a local Ethereum network. Truffle also provides such thing, called Ganache. Ganache is available both as a desktop application and command-line tool (ganache-cli) in Windows, Mac and Linux. This project is uses ganache-cli. As shown in the figure below, after the installation, `ganache-cli start` starts off the local network with 10 initial accounts that can be used during the testing.

```
Ganache CLI v6.5.1 (ganache-core: 2.6.1-beta.0)

Available Accounts
==================
(0) 0xb49548Afc101271B5A9609eADad8Ef2c07dEAf22 (100 ETH)
(1) 0x4A02eb0fC3fE4C4594D88C6499cD7E7654585a04 (100 ETH)
(2) 0x09563703a12c78a35d39fc90599ABc4cabddF6Fa (100 ETH)
(3) 0xB65455471e39A7F03a4EdB863dc07aC414CCd3AA (100 ETH)
(4) 0x1c5A9744109628D8b26Ee98b58C3ad992920327B (100 ETH)
(5) 0x827255aa3c79a84fd6e2D251087F81d2bC9b6D66 (100 ETH)
(6) 0x2c4F6af4985a58146D0CbA0f79B49BD7FEBD9cac (100 ETH)
(7) 0xFE0c8B4792BA029cA219100d2176dc64EC729d47 (100 ETH)
(8) 0x8F09b18d324cAa69D327c964B44ac8915F1E6e91 (100 ETH)
(9) 0xeb12b84aa975a2C54c10150426c98ecD77317417 (100 ETH)

Private Keys
==================
(0) 0x837c111374c055a4b1c6aa268264dfce1ee18a1c73462f29b1b11ff402bdfa16
(1) 0x60a4b50f51c641294be7a667f53d18c545080188ff86684deede00a1ada7144b
(2) 0xba23ad145e7005bcafcddea8aa5f75cbcf3c642c48fe7309dde8e6a05675f33f
(3) 0xe144936367ccf517b8f286c11803844730241e1d16e897a1dfdee879547287d0
(4) 0x260fb60b061263577acb3dc378be60a354a5559c10065cbfa0844b869058d3d1
(5) 0x8aa21aa614d7be68b77b0633bea7b2a159fa8117d0193e28d0fb2c12f1a777a9
(6) 0x4c86bcd79b1a72770eeb65c8874428424dd5d69426a224010312676fb9c30278
(7) 0x03217f59e325f73bf2f81963e7142ff1263023afdf98dc48c34fe0517c442b30
(8) 0x37b6f823192a0aa5adabfee391ca2e9e8ee726bee96bca2295bebc5c642e761b
(9) 0x1cfff50c1ffb970ae20477faefe8e7aa19b2aeda4a4b2692119357fc6a308f30

HD Wallet
==================
Mnemonic:      enact document awful sea control injury skirt useful genius gospel mind curtain
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
==================
20000000000

Gas Limit
==================
6721975

Listening on 127.0.0.1:8545
```

Figure 19. ganache-cli start

Smart contract tests were written in JavaScript in this project, otherwise it can be done in Solidity or JavaScript or Python. Even though Truffle provides a default assertion library, it does not seem enough, therefore another tool used for smart contract testing here is truffle-assertions. This package provides some additional options that makes writing Truffle tests much more convenient.

Before any test can be ran, the contracts must be first deployed to Ganache, the script in Code Snippet 38 shows how to deploy smart contracts with Truffle. `artifacts.require()` helps to interact with the required contracts and `link()` links a deployed with a un-deployed contract.

```
let CarMaintenance = artifacts.require("CarMaintenance")
let Utils = artifacts.require("Utilizations")
let ProxyContract = artifacts.require("Proxy")

module.exports = function(deployer) {
    deployer.deploy(ProxyContract)
    deployer.deploy(Utils)

    deployer.link(Utils, CarMaintenance)
    deployer.deploy(CarMaintenance)
};
```

Code Snippet 38. Contract deployment script

Now the contracts have been deployed, but as previously introduced, the contract functions will not be directly invoked through the main contract address but Proxy contract so that the logic contract could be upgraded if needed. Code Snippet 39 contains `connect()` which makes it possible to invoke functions of CarMaintenance contract through Proxy contract and from now on, `connect()` must be called in prior to developing any tests.

```
let impl_proxy
async function connect() {
    let CarMaintenance = artifacts.require("CarMaintenance")
    let ProxyContract = artifacts.require("Proxy")

    let impl_v1 = await CarMaintenance.new()
    impl_proxy = await ProxyContract.new()

    await impl_proxy.upgradeTo(impl_v1.address)

    return await CarMaintenance.at(impl_proxy.address)
}
```

Code Snippet 39. `connect()` in CommonScripts.js

`impl_proxy.upgradeTo(impl_v1.address)` calls `upgradeTo()` in Proxy contract to link the implementation of CarMaintenance to Proxy and afterwards `CarMaintenance.at(impl_proxy.address)` enables the all the function calls to CarMaintenance contract under the context of Proxy contract.

Now it is all set and the main testing phase can get started. The testing will be proceeded by per party, including Contract owner, Dealer, Service person and Car owner and final test is for contract upgradeability.

5.1.1 Test contract owner

Involved accounts for this test are as shown in the following snippet.

```
let owner = accounts[0]
let randomAccount = accounts[1]
let dealer1 = accounts[2]
let dealer2 = accounts[3]
let servicePerson1 = accounts[4]
let servicePerson2 = accounts[5]
```

Code Snippet 40. Involved accounts to Contract owner testing

When deploying contracts to Ganache network, the first account (`accounts[0]`) in the list of initial accounts is always chosen by default to be the one to deploy the contracts. Therefore, it is the contract owner.

Firstly, the ability of the contract owner adding new dealers will be tested. For this, only the contract owner is allowed to invoke `addDealer()` function, otherwise the transaction would revert. Here `fails()` from `truffle-assertsions` is used to assert that non-contract owner accounts should get the transactions reverted and `passes()` if everything is valid, as shown in Code Snippet 41.

```
it('Contract owner is able to add new dealers.', async function () {
    let proxy = await utils.connect()

    // ---------Testing starts here----------
    await truffleAssert.fails(
        proxy.addDealer(dealer1, { from: randomAccount }),
        truffleAssert.ErrorType.REVERT,
        null,
        "Only owner is allowed to add dealers."
    )

    await truffleAssert.passes(
        proxy.addDealer(dealer1, { from: owner }),
        "dealer1 should now be added into the storage."
    )

    await truffleAssert.passes(
        proxy.addDealer(dealer2, { from: owner }),
        "dealer2 should now be added into the storage."
})
```

Code Snippet 41. Test cases of contract owner adding new dealers

Secondly, the same thing is tested with adding new service persons, referring to Code Snippet 42.

```
it('Contract owner is able to add new service persons.', async function () {
    let proxy = await utils.connect()

    // ---------Testing starts here----------
    await truffleAssert.fails(
        proxy.addServicePerson(servicePerson1, {from: randomAccount}),
        truffleAssert.ErrorType.REVERT,
        null,
        "Only owner is allowed to add pharmacies."
    )

    await truffleAssert.passes(
        proxy.addServicePerson(servicePerson1, { from: owner }),
        "servicePerson1 should now be added into the storage."
    )

    await truffleAssert.passes(
        proxy.addServicePerson(servicePerson2, { from: owner }),
        "servicePerson2 should now be added into the storage."
    )
})
```

Code Snippet 42. Test case of contract owner adding new service persons

The test is then run with command `truffle test .\test\TestContractOwner.js` as the test is located in TestContractOwner.js file, and the results are shown in the following figure.

```
Contract: Contract owner
  √ Contract owner is able to add new dealers. (1218ms)
  √ Contract owner is able to add new service persons. (1150ms)


2 passing (2s)
```

Figure 20. Test results for Contract owner

5.1.2 Test dealer

Involved accounts for this test are as shown in the following snippet.

```
let owner = accounts[0]
let randomAccount = accounts[1]
let dealer1 = accounts[2]
let dealer2 = accounts[3]
let carOwner1 = accounts[4]
let carOwner2 = accounts[5]
```

Code Snippet 43. Involved accounts for Dealer testing

With dealers, the ability to initiate new cars will be tested in the first place. The test cases are as follows and illustrated in Code Snippet 44.

- Initiating a new car with a non-dealer account (`owner`).
- Initiating a new car with a dealer account (`dealer1`).
- Re-initiating an initiated car (`VN12345`).
- Initiating another new car for a non-existing car owner (`carOwner2`) with another dealer account.

```
await proxy.addDealer(dealer1, { from: owner }) // Adding dealer1
await proxy.addDealer(dealer2, { from: owner }) // Adding dealer2
await proxy.addCarOwnerMyself({ from: carOwner1}) // Adding carOwner1

await truffleAssert.fails(
    proxy.initiateCar("VN12345", carOwner1, { from: owner }),
    truffleAssert.ErrorType.REVERT,
    null,
    "Only dealers can initiate new cars."
)
await proxy.initiateCar("VN12345", carOwner1, { from: dealer1 })
await truffleAssert.fails(
    proxy.initiateCar("VN12345", carOwner2, { from: dealer2 }),
    truffleAssert.ErrorType.REVERT,
    null,
    "car VN12345 already exists."
)

let initiateCar = await proxy.initiateCar("FI54321", carOwner2, { from:
dealer2 })
truffleAssert.eventEmitted(initiateCar, 'newCarOwnerAddedDuringCarInitiation');
```

Code Snippet 44. Test case of dealer initiating new cars

`eventEmitted()` is another function provided by `trufflet-assertions` which asserts that `newCarOwnerAddedDuringCarInitiation` event is triggered. The test results are as shown in Figure 21.

```
Contract: Dealer
  √ Dealers are able to initiate new cars. (2129ms)


1 passing (2s)
```

Figure 21. Testing results for Dealer

5.1.3 Test service person

First of all is to test if Service person accounts are able to log new maintenance logs. The test cases and expected results are described in the following table.

Table 3. Test service person logging new maintenance logs

Test case	**Expected result**
An authorized service person logging a maintenance log of a non-existing car.	Should fail.
An authorized service person logging a self-service type log of an existing car.	Should fail.
An authorized service person logging a minor maintenance log of an existing car.	Should pass and trigger `NewLogAdded` event.

Then, the ability of a service person to query a log that was not created by him/her will be tested, asserting that all service persons are able to query all service books in the world. The test results for these two tests are as in Figure 22.

```
Contract: Service person
  √ Service persons are able to log maintenance logs. (2198ms)
  √ Service persons are able to query any service books. (1636ms)


2 passing (4s)
```

Figure 22. Test results for Service person

5.1.4 Test car owner

Here it is important to first test the ability of Car owner to grant/revoke access of service books to/from someone. The test cases and expected results are described in the following table.

Table 4. Test car owner granting or revoking access of service books

Test case	**Expected result**
A random account querying the logs of a car by car ID.	Should fail (Access denied).
A car owner querying the logs of a car which does not belong to him/her.	Should fail.
A car owner granting access of his/her car's service book to the random account above.	Should pass.
The random account querying the logs of that car again.	Should pass.
The car owner revoking the access of that car's service book from the random account.	Should pass.
The random account querying the logs again.	Should fail (Access denied).

Then, the test will continue with the ability of Car owner to toggle public access of a car's service book. The test cases and expected results are described in the following table.

Table 5. Test Car owner toggling public access of his/her car's service book

Test case	**Expected result**
A random account querying the logs of a car.	Should fail (Access denied).
The car owner of that car toggling public access of its service book.	Should pass (public access = true).
The random account trying to query the logs of the car again.	Should pass.
The car owner of that car togging public access of its service book again.	Should pass (public access = false).
The random account trying to query the logs of the car again.	Should fail (Access denied).

Next up is to test the ability of Car owner to transfer car ownership to someone else. The test cases are as follows:

Table 6. Test Car owner transferring car ownership to someone else

Test case	**Expected result**
carOwner2 querying the logs of carOwner1's car.	Should fail (Access denied).
carOwner1 transferring his/her car ownership to carOwner2.	Should pass.

carOwner2 querying the logs of the car again.	Should pass.
carOwner1 querying the logs of the car.	Should fail (Access denied).

Finally, Car owner role will be tested with the ability to log self-service maintenance logs where a car owner will try to log a new maintenance log by calling `logMaintenanceByCarOwner()`. As a result, the log should be fetched by calling `getLogByLogId()` and it should trigger `LogReturned` event.

The test results for all the cases above are shown in the figure below.

```
Contract: Car owner
  √ Car owners are able to grant access of service book to someone. (2069ms)
  √ Car owners are able to revoke access of service book from someone. (2003ms)
  √ Car owners are able to toggle public access. (1860ms)
  √ Car owners are able to transfer car ownership to someone. (2000ms)
  √ Car owners are able to log self-service logs. (1265ms)


5 passing (9s)
```

Figure 23. Test results for Car owner

5.1.5 Test contract upgradeability

With contract upgradeability testing, one mistake in a function in CarMaintenance contract will purposefully be made, where the function returns false in all cases, as seen in Code Snippet 45.

```
function isServicePersonExists(address _servicePersonId) public
view validAddress(_servicePersonId) returns(bool) {
    return false;
}
```

Code Snippet 45. Intentional bug in CarMaintenance contract

The scenarios will be as follows:

- Adding servicePerson1 into the contract storage by calling `addServicePerson()` from contract owner account.

  ⇨ Should pass.

- Invoking `isServicePersonExists()` function through Proxy contract on behalf of CarMaintenance.

  ⇨ Should return false.

- Creating CarMaintenance2 inheriting CarMaintenance and having the bug fixed here, as shown in Code Snippet 46.

```
contract CarMaintenance2 is CarMaintenance {
    function isServicePersonExists(address _servicePersonId)
    public view validAddress(_servicePersonId) returns(bool) {
        return servicePersons[_servicePersonId].exists;
    }
}
```

Code Snippet 46. The bug is fixed in CarMaintenance2

- Deploy CarMaintenance2 and upgrade the implementation address of Proxy contract to be the address of CarMaintenance2, as shown in the following snippet.

```
let CarMaintenance2 = artifacts.require("CarMaintenance2")
module.exports = function(deployer) {
    deployer.link(Utils, CarMaintenance2)
    deployer.deploy(CarMaintenance2)
};

async function upgrade() {
    let CarMaintenance2 = artifacts.require("CarMaintenance2")
    let impl_v2 = await CarMaintenance2.new()

    await impl_proxy.upgradeTo(impl_v2.address)

    return await CarMaintenance2.at(impl_proxy.address)
}
```

Code Snippet 47. Deploying CarMaintenance2 and upgrading Proxy

- Invoking `isServicePersonExists()` function through Proxy contract on behalf of CarMaintenance2.

  ⇨ Should return true.

The test is illustrated in Code Snippet 48.

```
it('CarMaintenance contract can be upgraded if needed.', async
function () {
    let proxy = await utils.connect()

    // ---------Testing starts here----------
    await proxy.addServicePerson(servicePerson1, { from: owner })

    let isServicePersonExists =
await proxy.isServicePersonExists(servicePerson1, { from: randomAccount })
    assert.isFalse(isServicePersonExists, "isServicePersonExists()
corrupted, function should return false.")

    proxy = await  utils.upgrade()

    isServicePersonExists = await proxy.isServicePersonExists(servicePerson1,
{ from: randomAccount })
    assert.isTrue(isServicePersonExists, "isServicePersonExists() is
now fixed, function should return true.")
})
```

Code Snippet 48. Contract upgradeability test

The test is proceeded in a way that first `servicePerson1` is added to the network, under the implementation of CarMaintenance. And due to the bug, `isServicePersonExists()` then should return `false`, meaning the service person does not exist. Then the logic contract is upgraded to CarMaintenance2 by calling `utils.upgrade()`. Now `isServicePersonExists()` should `true` indicating the bug has been fixed and the contract data was not lost.

And finally, the figure below shows the test result of this test.

```
Contract: Upgradeability
  √ CarMaintenance contract can be upgraded if needed. (1184ms)


1 passing (1s)
```

Figure 24. Test result for Contract upgradeability

5.1.6 Overall testing results

Overall, if all tests are run together at once, the results are shown in Figure 25.

```
Contract: Car owner
  √ Car owners are able to grant access of service book to someone. (1614ms)
  √ Car owners are able to revoke access of service book from someone. (1654ms)
  √ Car owners are able to toggle public access. (1636ms)
  √ Car owners are able to transfer car ownership to someone. (2032ms)
  √ Car owners are able to log self-service logs. (1067ms)

Contract: Contract owner
  √ Contract owner is able to add new dealers. (712ms)
  √ Contract owner is able to add new service persons. (690ms)

Contract: Dealer
  √ Dealers are able to initiate new cars. (1238ms)

Contract: Service person
  √ Service persons are able to log maintenance logs. (1282ms)
  √ Service persons are able to query any service books. (1250ms)

Contract: Upgradeability
  √ CarMaintenance contract can be upgraded if needed. (847ms)


11 passing (16s)
```

Figure 25. Overall test results

5.2 Deployment

This section will show how the contracts can be deployed and tested from the interface through user interactions.

5.2.1 Requirements

- **Truffle**

As mentioned earlier, this project uses Truffle Suite to manage all the stuffs like compiling, testing and deploying, etc.

- **truffle-plugin-verify**

After contracts have been deployed, they are not ready to be use as long as they are not verified. Contract verification can be done manually by uploading the source code of the contract(s) to the network to be compared against the bytecode compiled earlier during the deployment. This is a bit time-consuming when dealing with multiple contracts and libraries, but `truffle-plugin-verify` makes it a lot easier by

running only a single command from a command-line tool and the package will deal with the rest.

- **Kovan network**

There are 1 production (mainnet) and 4 test (testnet) Ethereum environments. Deploying contracts and making transactions on the mainnet would require some real ethers mined or bought by fiat currencies, this project will only be deployed to one of the testnets, called Kovan.

Kovan network utilizes Proof of Authority (PoA) consensus algorithm instead of PoW. There is no such mining activity here, but there are so-called "sealers" (a set of authorities) who will be responsible for creating new blocks and maintaining the world state. Ethers therefore are not mined but requested through a faucet service with time limits to prevent spam attack which has happened earlier on other networks. Kovan network updates a new block every 4 seconds, each block contains over 300 transactions, meaning that around 80 transactions are handled per second. Otherwise, all contract executions are processed exactly the same way as PoW-based networks do /34/.

- **Infura**

Since I am not running a full Ethereum node locally on my computer, and many developers are in the same situation as maintaining a full node consumes a great great amount of resoures, there needs to use a third party hosted node to be able to interact with the network, Infura in this case seems like the most reliable option. Infura is a collection of full Ethereum nodes that enables developers to connect with Ethereum network without requiring them to run a full node locally.

- **truffle-hdwallet-provider**

For security reasons, Infura does not manage private keys, therefore, cannot sign transactions. However, it can be done by `HDWalletProvider` provided by `truffle-hdwallet-provider`, a Truffle's separate package.

5.2.2 Configurations

When executing Truffle deployment command, it will look into the configurations defined in `truffle-config.js` to gather information of the deployment details. The code snippet below shows the definition of Kovan network and other configurations, where `42` is the ID of Kovan network and `mnemonic` is the 12 seed words generated from the private key of the account that is going to deploy the contracts from. The mnemonic together with the Infura project ID will be provided to `HDWalletProvider` to form up a provider that will be able to sign transactions. This mnemonic phrase should never be included directly in the project if it was for mainnet interactions but rather be stored locally in the computer. Otherwise, Etherscan's API key from the configurations is needed for `truffle-plugin-verify` to verify the contracts.

```
const HDWalletProvider = require('truffle-hdwallet-provider');
const mnemonic = "12_SECRET_SEED_WORDS";

module.exports = {
  networks: {
    kovan: {
      provider: () => new HDWalletProvider(
mnemonic, `https://kovan.infura.io/v3/INFURA_PROJECT_ID`),
      network_id: 42,
      timeoutBlocks: 100,
      skipDryRun: false,
      gasPrice: 20000000000,
      gas: 8000000
    },
  },
  compilers: {
    solc: {
      version: "0.5.11",
    }
  },
  plugins: [
    'truffle-plugin-verify'
  ],
  api_keys: {
    etherscan: 'Etherscan_API_key'
  }
}
```

Code Snippet 49. Configurations in `truffle-config.js`

5.2.3 Deploy and verify contract

The contracts are deployed by executing `truffle migrate --network kovan`, where Kovan is the desired network in our case, other network could be specified if needed. This command will tell to into the configurations in `truffle-config.js` and the deployment script in `1_initial_migration.js` (same as for testing in Code Snippet 38) will tell the deployment steps. To summarize, it will:

1. Deploy Proxy contract.
2. Deploy Utilizations library.
3. Link Utilizations with CarMaintenance.
4. Deploy CarMaintenance contract.

Figure 26 shows the deployment results, there it tells the total deployment cost, the addresses of the contracts and library, the hash of the contract creation transactions and so on.

```
1_initial_migration.js
======================

   Replacing 'Proxy'
   -----------------
   > transaction hash:    0xfa8706dbb688d767b049bf489de50dc3907b569bbc83fef6c869f91772740488
   > Blocks: 1            Seconds: 4
   > contract address:    0x54F401E9cE9813182652a3071Ac1f422580baeaf
   > block number:        13550956
   > block timestamp:     1568710904
   > account:             0xaEAEDe7E4bC957811b228343e8CDB430a025672B
   > balance:             2.367224487
   > gas used:            601523
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.01203046 ETH


   Replacing 'Utilizations'
   ------------------------
   > transaction hash:    0xd88c8215be46ddf2bc4c0ffcb25112598abc489792b8e966b5c6d047548881e0
   > Blocks: 0            Seconds: 0
   > contract address:    0x01255Cba89B3DdD6ca08208e37c216Ec052C2e99
   > block number:        13550958
   > block timestamp:     1568710912
   > account:             0xaEAEDe7E4bC957811b228343e8CDB430a025672B
   > balance:             2.360279447
   > gas used:            347252
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00694504 ETH


   Linking
   -------
   * Contract: CarMaintenance <--> Library: Utilizations (at address: 0x01255Cba89B3DdD6ca08208e37c216Ec052C2e99)

   Replacing 'CarMaintenance'
   --------------------------
   > transaction hash:    0x6e5d567ace969f53add8bdf7a6422194e6b7d0117653d56dc0712b8b69b40dbd
   > Blocks: 0            Seconds: 4
   > contract address:    0x42C599CC3965981cD797510DC7Af0DaE179020D1
   > block number:        13550960
   > block timestamp:     1568710920
   > account:             0xaEAEDe7E4bC957811b228343e8CDB430a025672B
   > balance:             2.266409927
   > gas used:            4693476
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.09386952 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:          0.11284502 ETH


Summary
=======
> Total deployments:   3
> Final cost:          0.11284502 ETH
```

Figure 26. Deployment results

Otherwise, smart contracts can also be deployed graphically from web browser such as via Remix. Remix is a web browser-based IDE to develop smart contracts. It also allows to deploy contracts directly, this way would require MetaMask installed in the browser as a connection to an Ethereum node. MetaMask is an extension for

Chrome or Firefox that connects to an Ethereum network using Infura on the browser so that the users do not need to run a full node locally. MetaMask can help to flexibly connect to the mainnet or any of the testnets. MetaMask does not store private keys on any remote server but in the browser's data store, therefore it is secure. Otherwise, the source code of MetaMask is public and auditable by anyone through https://github.com/MetaMask/metamask-extension.

The interface of MetaMask is illustrated in the following figure, telling that the current account (Contract Owner) is currently connected to Kovan network, the balance is 3.6775 ETH and the list of history transactions (this list does not maintain the full list of transactions of the account but only those directly signed on this browser particularly).

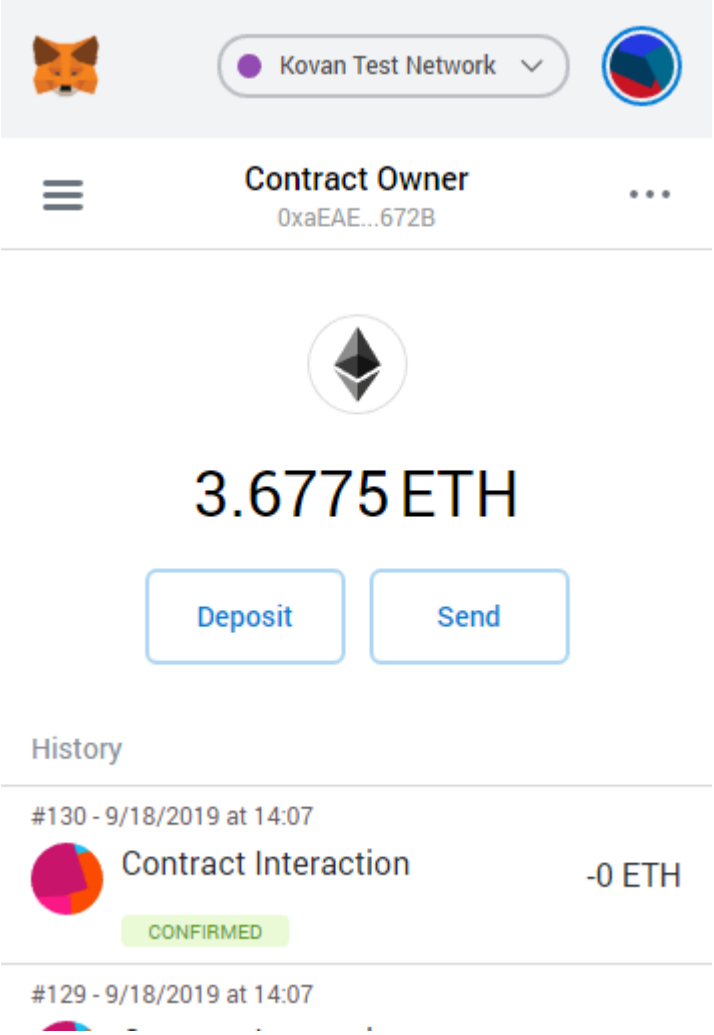

Figure 27. MetaMask interface

To deploy the contracts, all the related code must be presented and successfully compiled in Remix beforehand, then the deployment can be proceeded from the "Deploy & run transactions" tab with "Injected Web3" environment selected, as shown in Figure 28.

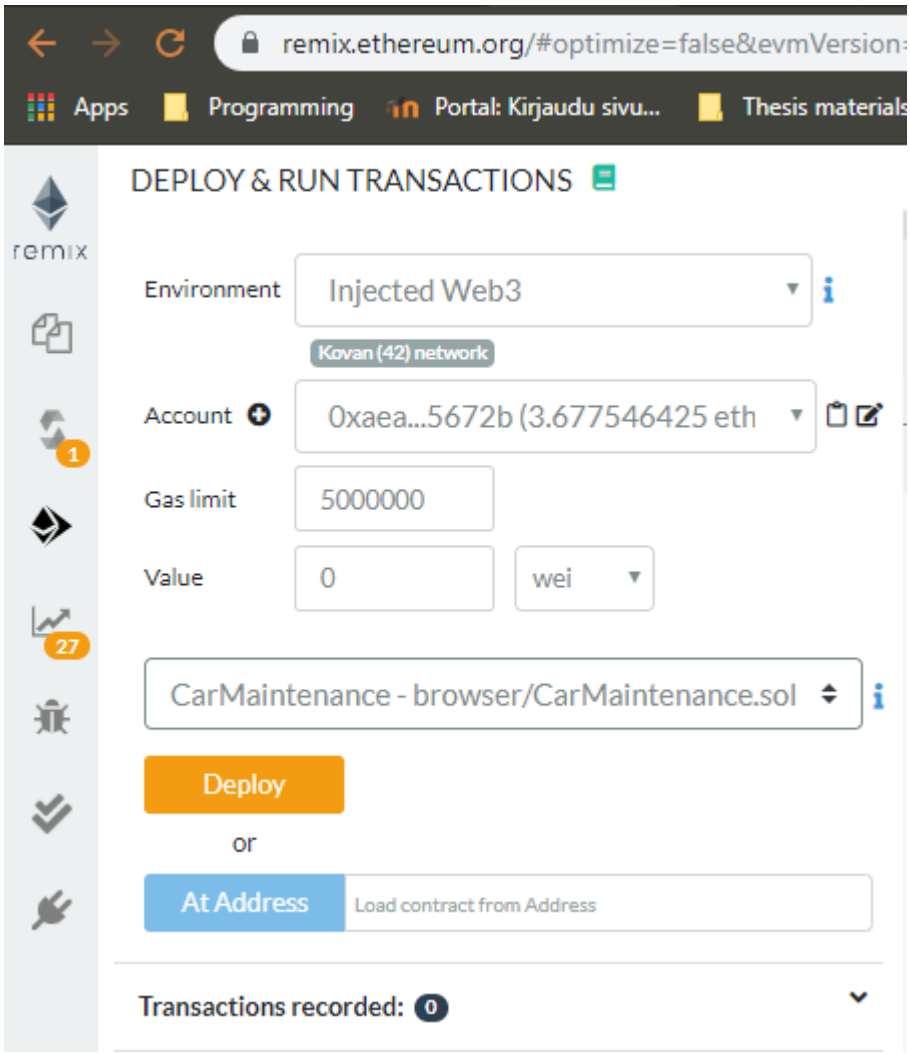

Figure 28. Deploying contracts with Remix

Now that the contracts have been deployed, but it is still one step away from being interacted, verification. To verify the newly deployed contracts, run the command `truffle run verify CarMaintenance --network kovan` and `truffle run verify Proxy --network kovan` or `truffle run verify CarMaintenance Proxy --network kovan`. After a few seconds, the verification results will return, as shown in the following figures.

```
Verifying CarMaintenance
Pass - Verified: https://kovan.etherscan.io/address/0x8de405c30d167B83daC72Dfa93A67e8B6eC45AcC#contracts

Successfully verified 1 contract(s).
```

Figure 29. Contract verification result for CarMaintenance

```
Verifying Proxy
Pass - Verified: https://kovan.etherscan.io/address/0xd3b960C4D893B183ba9d609EAbDc9a22974Dd8F3#contracts

Successfully verified 1 contract(s).
```

Figure 30. Contract verification result for Proxy

5.2.4 Interacting with the deployed smart contracts

There are several methods to interact with smart contracts, Remix and Etherscan will be used in this project.

- **Interacting with smart contracts via Remix**

First, the contract must be loaded by inserting the contract address, then all the functions will appear accordingly, referring to Figure 31.

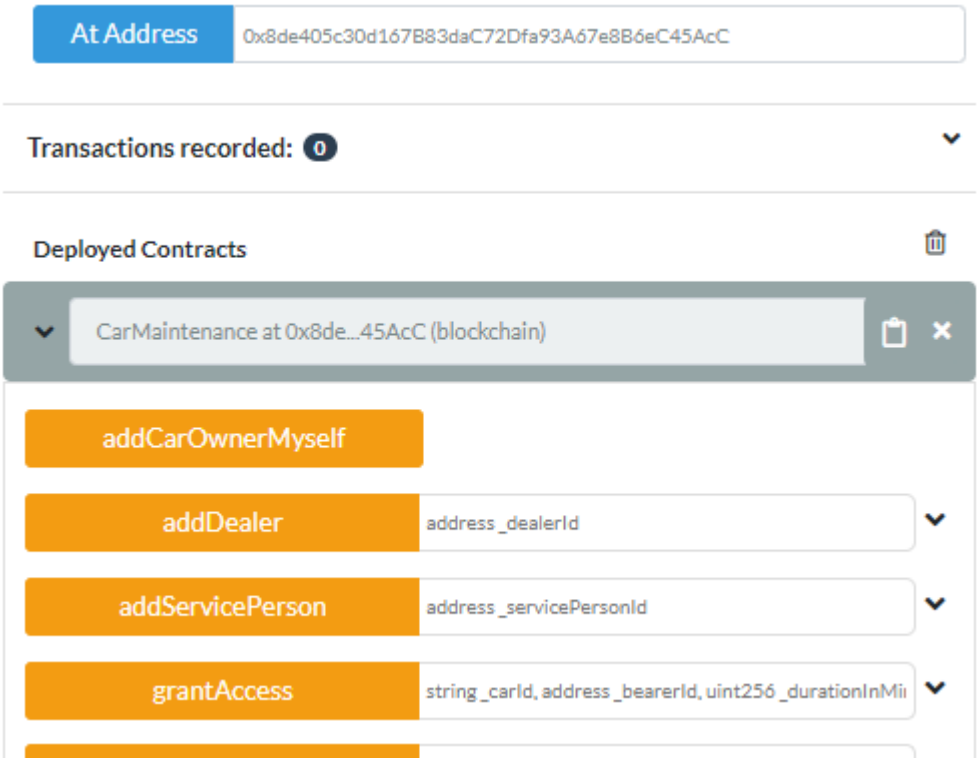

Figure 31. Loading a deployed contract

Some test accounts have also been prepared with MetaMask so that the contracts can be tested fully, as shown in the following figure.

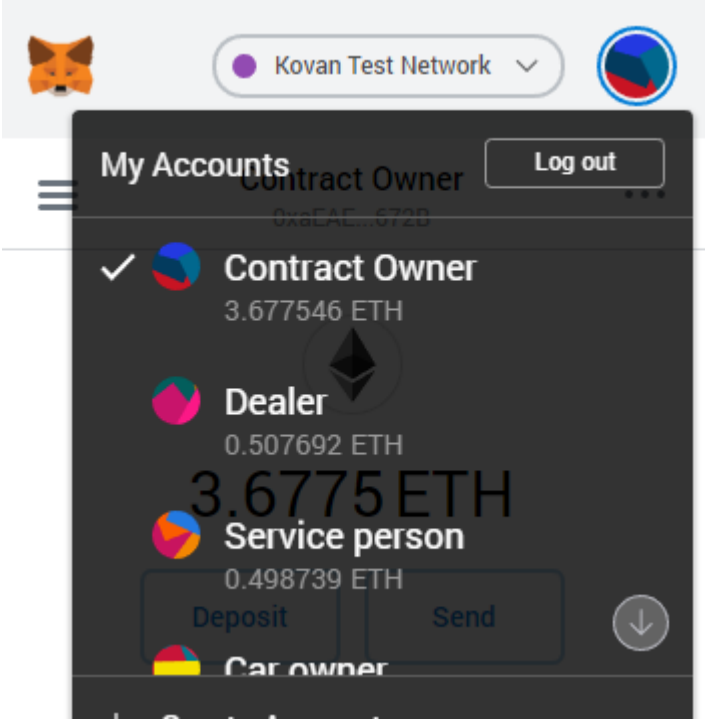

Figure 32. Imported accounts in MetaMask

Every function invocation will be asked for confirmation from MetaMask, as seen in the figure below.

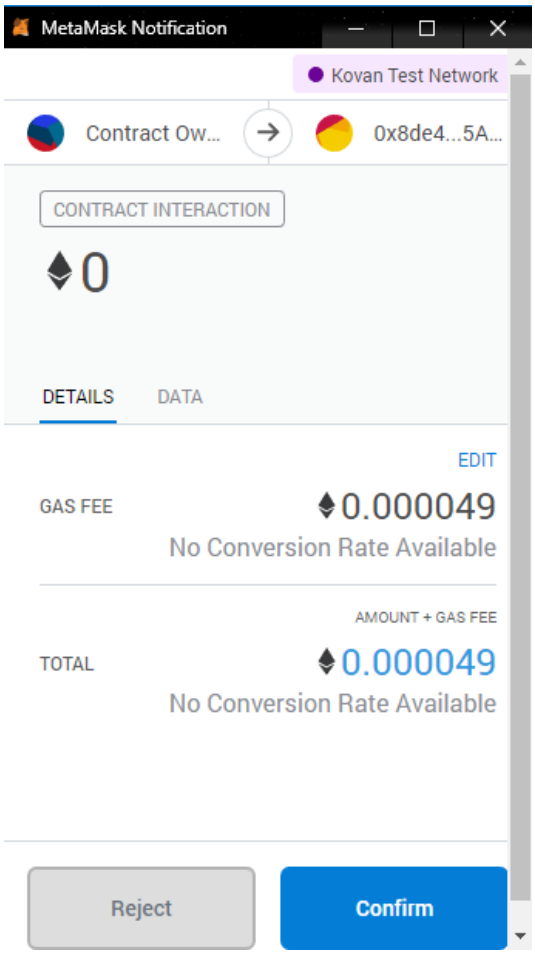

Figure 33. MetaMask transaction confirmation

Now, let us try to initiate a car, log some maintenance records, and grant access of the service book to someone. The figure below shows that a random account was able to query a service book and any of its log after the access had been granted by the car owner.

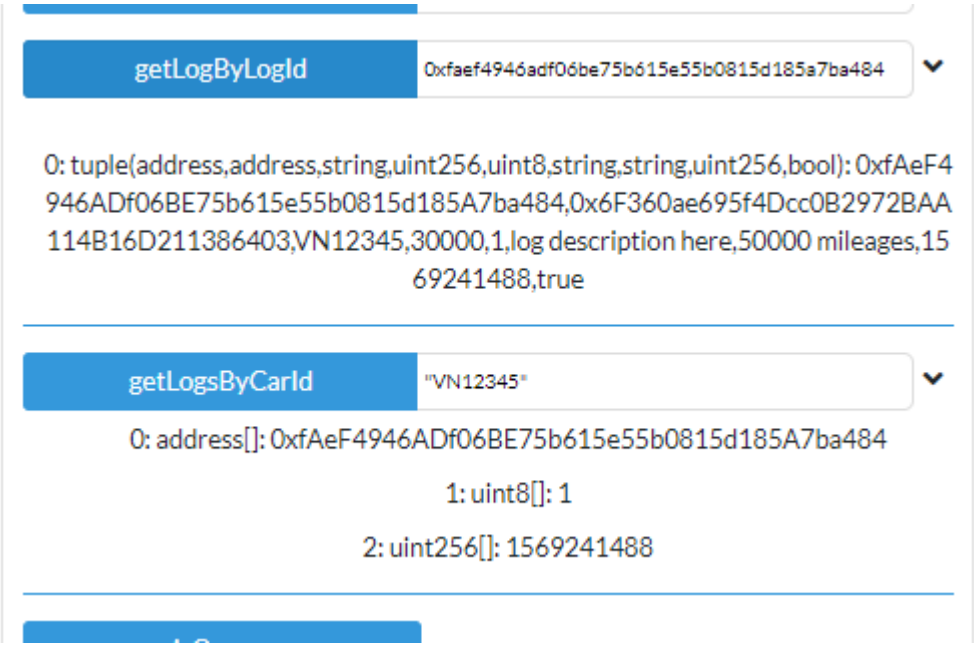

Figure 34. Contract interaction results with Remix

When calling functions, any potentially invalid function invocations such as non-authorized actions or insufficient funds will be warned by Remix in advance, if the sender insists, MetaMask will warn once more in the transaction confirmation dialog, then if the sender still insists, the transaction will be proceeded but will possibly fail, but the gas will still be taken and the transaction will still be recorded. Besides interacting with the contract, all the actions are also logged in the Remix terminal, as shown in the following figure.

```
0   listen on network   Search with transaction hash or address

transact to CarMaintenance.initiateCar pending ...

https://kovan.etherscan.io/tx/0xe31b953422a40f8b397e47d64587b11d36efd564e42bc5c58d36ac396c5f947c

[block:13682998 txIndex:0]  from:0x9dc...7e86a to:0x8de...45acc value:0 wei

transact to CarMaintenance.logMaintenanceByServicePerson pending ...

https://kovan.etherscan.io/tx/0x434c4c8186d2c59e82db6a3548ef2ea38c826254996f043d025c9346cf76c271

[block:13683057 txIndex:1]  from:0x6f3...86403 to:0x8de...45acc value:0 wei

transact to CarMaintenance.grantAccess pending ...

transact to CarMaintenance.grantAccess errored: The execution failed due to an exception.

transact to CarMaintenance.grantAccess pending ...

transact to CarMaintenance.grantAccess errored: The execution failed due to an exception.

transact to CarMaintenance.grantAccess pending ...

https://kovan.etherscan.io/tx/0x824f53129c998694eeadbbe18a48a81b0f1d518270c857c0a2115f2136435fc3

[block:13683086 txIndex:0]  from:0xc95...0a425 to:0x8de...45acc value:0 wei

call to CarMaintenance.getLogsByCarId

CALL  [block: txIndex:]  from:0x9dc...7e86a to:0x8de...45AcC value:0 wei

call to CarMaintenance.getLogByLogId

CALL  [block: txIndex:]  from:0x9dc...7e86a to:0x8de...45AcC value:0 wei
```

Figure 35. Remix terminal

Even though Kovan network was stated to be able to handle 80 transactions per second, but due to the busy traffic, real testing shows that all function calls took roughly 6 – 8s to be completed. Read functions return the values nearly instantly.

- **Interacting with smart contracts via Etherscan**

Etherescan is a leading Ethereum BlockExplorer. A BlockExplorer is a tool that users can see all the cryptocurrency transactions ever sent and all the blocks ever

mined, in short, it is a blockchain search engine. Etherscan also provides an independent interface for dApps, as shown below.

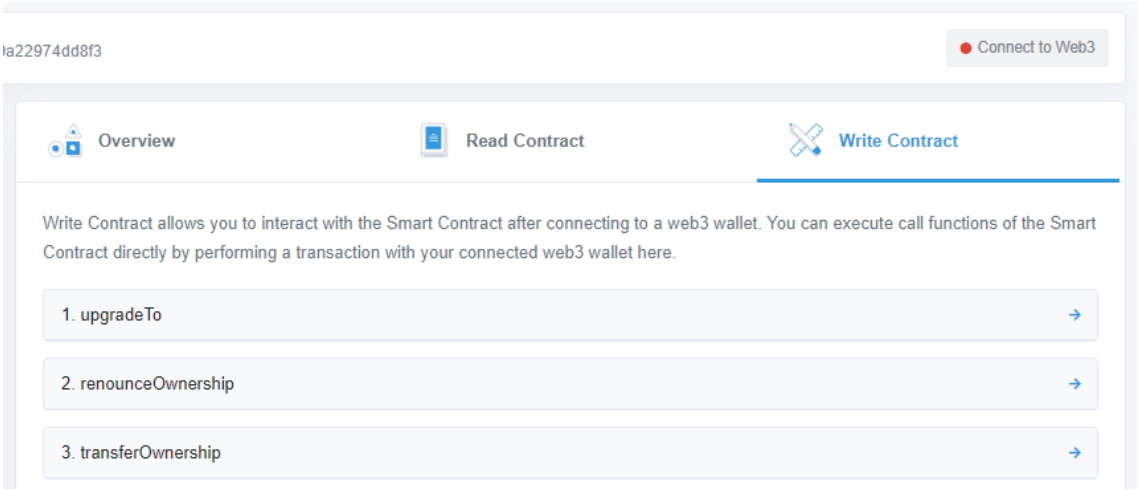

Figure 36. dApp interface on Etherscan

In the figure above, “Read Contract” tab shows all the view functions and “Write Contract” tab shows all the write functions. Here the contract upgradeability with Proxy contract can be verified. To do so, a CarMaintenance implementation must be linked to the Proxy contract first, as shown in the following figure.

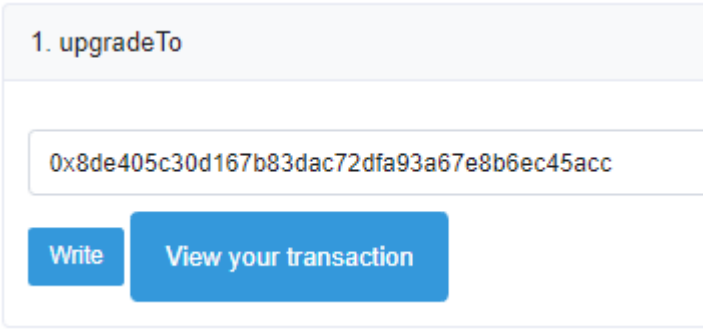

Figure 37. Link logic contract version to Proxy contract

Then all the named functions of CarMaintenance contract will be shown through the fallback function in Proxy contract and can be invoked here, referring to the figure below.

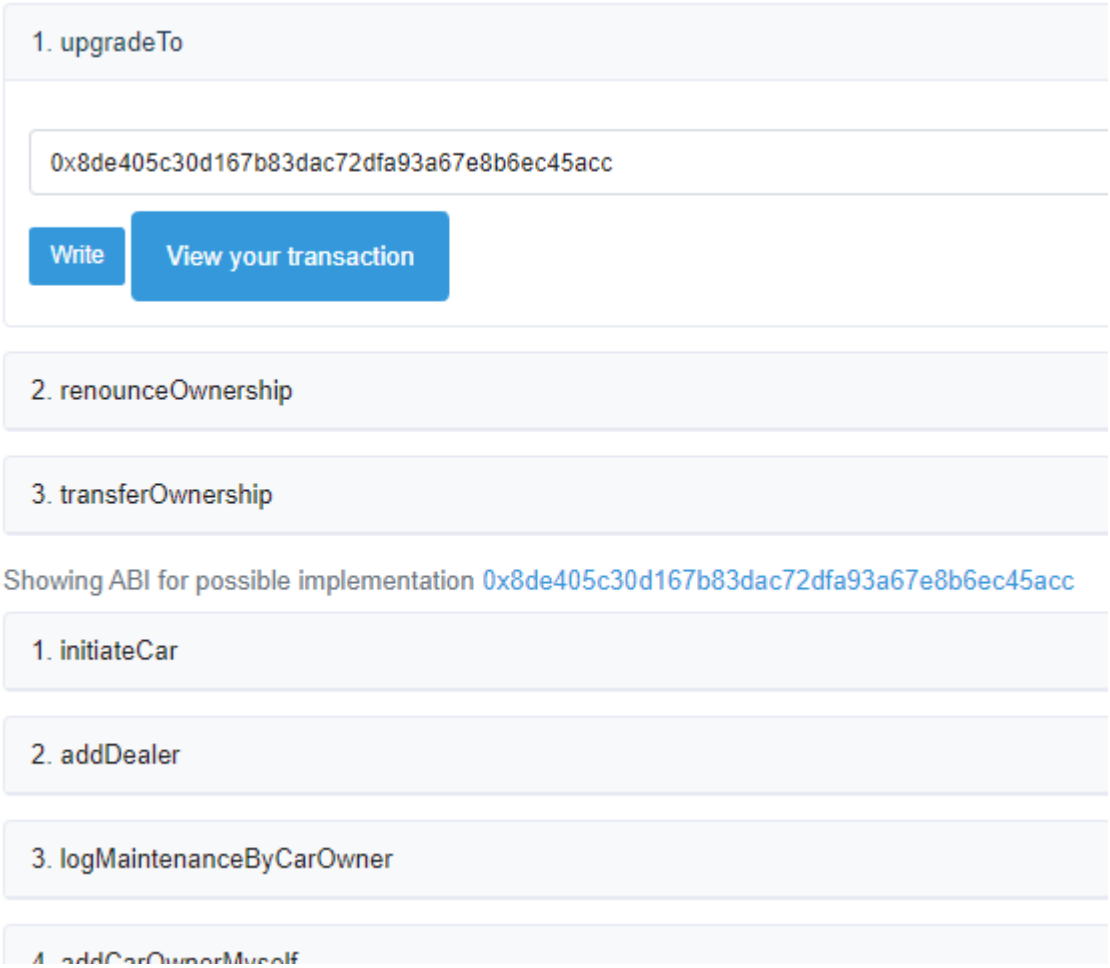

Figure 38. Logic functions can be used in Proxy contract

Then the contract owner will add a new service person into the contract storage, as shown below.

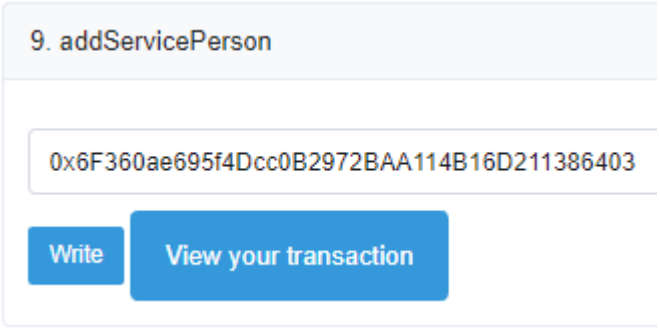

Figure 39. Add a new service person

Figure 40 shows that however, `isServicePerson` event returns `0` (`false`), indicating that there are flaws in the current contract as it's supposed to return `true`.

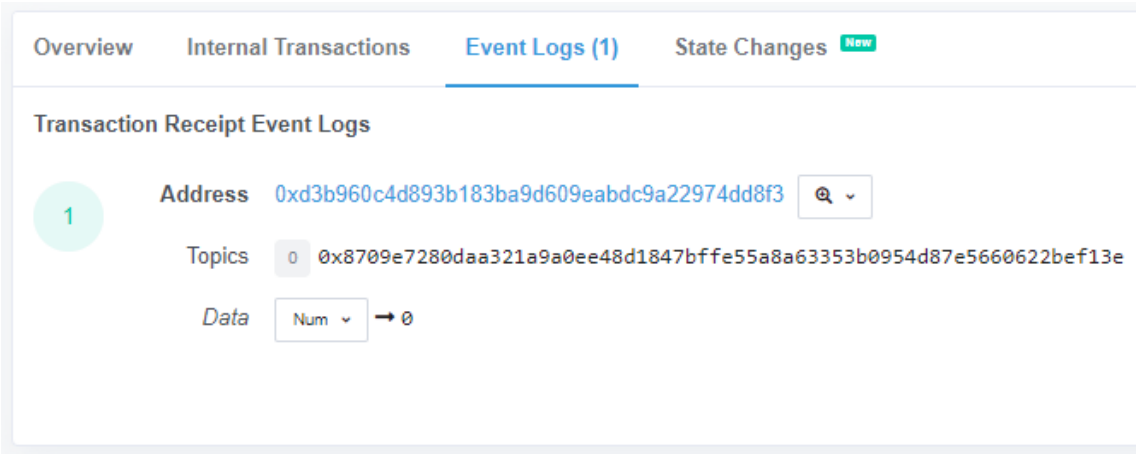

Figure 40. `isServicePerson` event returns `0` in event logs

So, another contract must be deployed to and upgraded. The following figure shows that the contract has been upgraded and ready to be used.

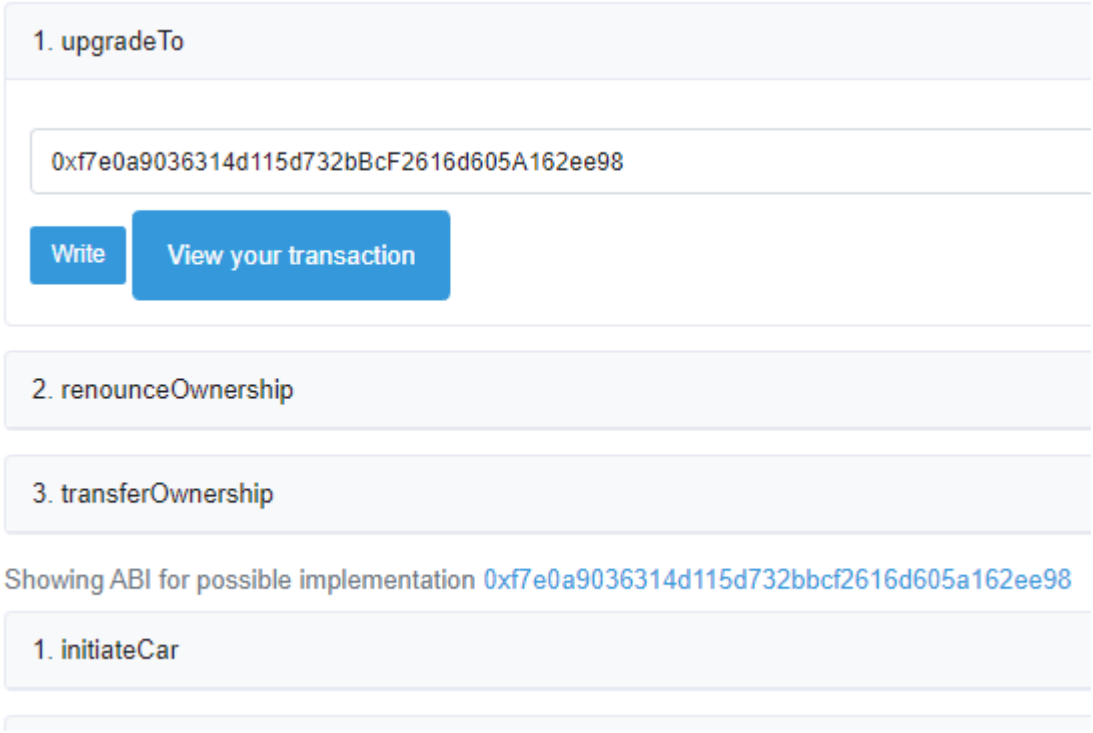

Figure 41. Link a newly deployed logic contract to the same Proxy contract

And now `isServicePerson` event returns `1` (`true`), meaning that the error is resolved, the service person really exists in the contract state and the data storage remains, as shown in Figure 42.

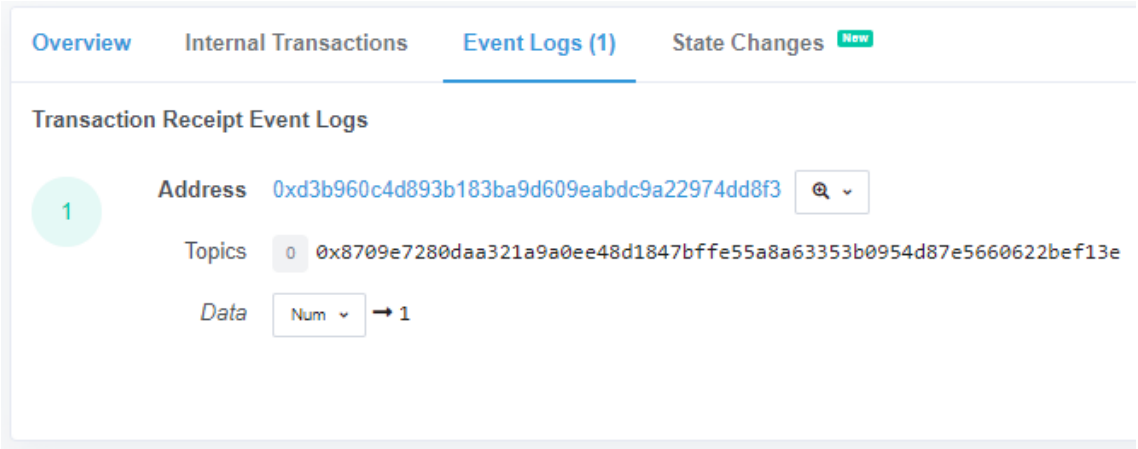

Figure 42. `isServicePerson` event returns `1` in event logs

6 CONCLUSIONS

To summarize, the implementation of car service book system using smart contract has quite a lot of things to be improved. One of those is to investigate how to migrate the contract data in case the storage structure needs to be upgraded, meaning that making the contract entirely upgradeable. This would make the contract to be more flexible and very much more scalable.

Moreover, as mentioned earlier in section 4.3.4, the implementation is somehow centralized, in which car dealers and service persons are added into the network by the single contract owner, this should be studied so that a decentralized method will be integrated. Digital identity could be a good option since then the entire eco-system can be totally decentralized, thus nobody has control over the network and the identity is linked directly to the entity (person or organization).

On the other hand, one of the most important factors when developing smart contracts is transaction fees, as the data grows larger over time, more computational power may potentially be needed during the contract executions, the contract must be studied and written in a way that it costs as little fees as possible. Also, the current Ethereum transaction speed is around 15 tps on the mainnet and 80 tps on Kovan testnet, and each function call takes approximately 6 – 8s to write the data into the storage on the testnet, this number may be higher on the mainnet due to busier traffic which does not seem like a good rate as compared to under 5s from the client-server model, but we can expect better performance from the upcoming Ethereum 2.0 with the sharding implementation and other features.

The idea to bring blockchain technology to the world as a decentralized, transparent and immutable solution is extremely brilliant, especially in the digital world nowadays, a digital trust solution is more than needed. Blockchain has indeed the potential to solve some critical issues, but no blockchain solutions so far have seemed to be fully adopted because they remain such inherent problems as scalability or the lack of truth in a trustless system. Bitcoin was originally to be a digital currency,

but it has never been and never will be because of its transaction speed and instability, most of ICOs project failed as they turned out mostly frauds and cheats. Yet one stays alive strongly, Ethereum, as they are serious and dedicated to their goals.

But above all, the biggest reason that is holding back the adoption of blockchain is the issue of trust, even though it intends to solve the trust problem. With blockchain, the users may not need to trust anyone in the network, but at least they need to trust the implementation itself, meaning that instead of relying on any central servers or network participants, it now relies on the people who are making all the work. Therefore in my opinion, the technology is only trusted and truly adopted if its implementations are as good and honest as they say to be and regulatory impacts are soberly concerned for all cases of blockchain implementations, including Ethereum smart contract.

In conclusion, the implementation of smart contract has demonstrated that developing decentralized applications on top of smart contract can be considered one of the brightest directions to go for blockchain technology. But due to the fact that blockchain is a highly secure data storage in which it is strengthened by its immutability, however, there is nothing called “privacy” there as literally everything is public to the whole world, finding the right idea to optimize the use of smart contract is not an easy work. But if used correctly, it can be truly disruptive to the current traditional systems in various industries like supply chain management, healthcare and so on. Yet, I believe that with the strong dedication and determination from the Ethereum development team and the consideration in regulation aspect from governments, the blockchain generally and the use of Ethereum smart contract to build decentralized applications particularly are very much promising and worth noticing in the future.

REFERENCES

/1/ The Age of Trust — the problem blockchain solves that others cannot. Last accessed 07.07.2019. Available at: https://medium.com/swlh/the-age-of-trust-the-problem-blockchain-solves-that-others-cannot-6024ebf47cad

/2/ What is Blockchain Technology?. Last accessed 24.09.2019. Available at: https://www.coindesk.com/information/what-is-blockchain-technology

/3/ Cryptographic hash function. Last accessed 20.09.2019. Available at: https://en.wikipedia.org/wiki/Cryptographic_hash_function#Properties).

/4/ Digital signature. Last accessed 17.09.2019. Available at: https://en.wikipedia.org/wiki/Digital_signature

/5/ Bitcoin block 586,381. Last accessed 20.09.2019. Available at: https://learnmeabitcoin.com/browser/block/0000000000000000001ec77fc2e427b1b6eb1474717dce7d745ba11630a60ba0

/6/ What is hashing?. Last accessed 01.10.2019. Available at: https://blockgeeks.com/guides/what-is-hashing/

/7/ Bitcoin: A Peer-to-Peer Electronic Cash System. /Last accessed 03.10.2019/. Available at: https://bitcoin.org/bitcoin.pdf

/8/ Bitcoin hashrate distribution. Last accessed 25.09.2019. Available at: https://www.blockchain.com/en/pools

/9/ Deep Chain Reorganization Detected on Ethereum Classic (ETC). Last accessed 01.10.2019. Available at: https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de

/10/ Bitcoin hashrate – live chart. Last accessed 01.10.2019. Available at: https://www.blockchain.com/en/charts/hash-rate

/11/ Proof of Stake explained. Last accessed 30.09.2019. Available at: https://www.binance.vision/blockchain/proof-of-stake-explained

/12/ Understanding Proof of Stake through it's Flaws. Part 2 — 'Nothing's at Stake'. Last accessed 01.10.2019. Available at: https://medium.com/@abhisharm/understanding-proof-of-stake-through-its-flaws-part-2-nothing-s-at-stake-8d12d826956c

/13/ History of Bitcoin. Last accessed 25.07.2019. Available at: https://en.wikipedia.org/wiki/History_of_bitcoin

/14/ VISA: Security and reliability. Last accessed 01.10.2019. Available at: https://usa.visa.com/run-your-business/small-business-tools/retail.html

/15/ Wikipedia: Ethereum. Last accessed 30.09.2019. Available at: https://en.wikipedia.org/wiki/Ethereum

/16/ Ethereum Development Tutorial. Last accessed 31.10.2019. Available at: https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial

/17/ Ethereum average block time chart. Last accessed 01.10.2019. Available at: https://etherscan.io/chart/blocktime

/18/ The Future of Ethereum: A Scaling Roadmap to Casper, Plasma, and Sharding. Last accessed 05.08.2019. Available at: https://blockexplorer.com/news/ethereum-scaling-roadmap-casper-plasma-sharding/

/19/ Ethereum node tracker. Last accessed 01.10.2019. Available at: https://etherscan.io/nodetracker

/20/ Sharding FAQ. Last accessed 03.10.2019. Available at: https://github.com/ethereum/wiki/wiki/Sharding-FAQ

/21/ Wikipedia: Initial coin offering. Last accessed 25.09.2019. Available at: https://en.wikipedia.org/wiki/Initial_coin_offering

/22/ Funds raised in 2018. Last accessed 25.09.2019. Available at: https://www.icodata.io/stats/2018

/23/ Funds raised in 2019. Last accessed 25.09.2019. Available at: https://www.icodata.io/stats/2019

/24/ CRYPTOASSET MARKET COVERAGE INITIATION: NETWORK CREATION. Last accessed 25.09.2019. Available at: https://research.bloomberg.com/pub/res/d28giW28tf6G7T_Wr77aU0gDgFQ

/25/ Token Sales: ICOs, IEOs, ITOs, STOs. Last accessed 06.08.2019. Available at: https://blockchainhub.net/ico-initial-coin-offerings/

/26/ Tokenization: The world of ICOs. Last accessed 06.08.2019. Available at: https://www.cnbc.com/2018/07/13/initial-coin-offering-ico-what-are-they-how-do-they-work.html

/27/ Statement on Cryptocurrencies and Initial Coin Offerings. Last accessed 06.08.2019. Available at: https://www.sec.gov/news/public-statement/statement-clayton-2017-12-11

/28/ Wikipedia: Turing completeness. Last accessed 01.11.2019. Available at: https://en.wikipedia.org/wiki/Turing_completeness

/29/ ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. Last accessed 08.08.2019. Available at: https://ethereum.github.io/yellowpaper/paper.pdf

/30/ How Ethereum and Smart Contracts Work. Last accessed 13.08.2019. Available at: https://vas3k.com/blog/ethereum/

/31/ OpenZeppelin Ownable contract. Last accessed 01.10.2019. Available at: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/ownership/Ownable.sol

/32/ UniqueId. Last accessed 01.10.2019. Available at: https://gist.github.com/techbubble/8f9db0f3ddd83ae0b1786ccf7805c461

/33/ Units and Globally Available Variables. Last accessed 01.10.2019. Available at: https://solidity.readthedocs.io/en/v0.5.11/units-and-global-variables.html#time-units

/34/ Kovan - Stable Ethereum Public Testnet. Last accessed 01.10.2019. Available at: https://github.com/kovan-testnet/proposal