ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

SECURITY-RELATED EXPERIENCES WITH SMART CONTRACTS OVER THE ETHEREUM BLOCKCHAIN

Tesi in

Sicurezza delle Reti

Relatore

Gabriele D'Angelo

Presentata da

Francesco Maughelli

Sessione I

Anno Accademico 2016 - 2017

“se l’essenza della vita è racchiusa nel DNA, allora la società e la civiltà non sono altro che colossali sistemi di memoria”

- Batou

CONTENTS

SUMMARY

We live in a world ruled by states: higher expression of organized societies and civilizations, systems built by people, implicitly or explicitly establishing laws, conventions and regulations that society must adhere on in order to live. When systems such as these are constructed the inner self of people's majority is projected into the big picture, forming a vast array of features that can describe a civilization. G*estaltism* and *structuralism* fight over the assertion that "*the whole is other than the sum of the parts*" but in a way we can bear witness of this sharpness every time we look at an elegant engineered solution deployed into the real world.

However, man itself is a complex individual and during the process of defining a majority, some identities will be lost or something will be held back, confined to a minority that statistically cannot rise.

These grounds need a different approach, something that have to emerge from individual themselves aside from organizations and regulations that can deliver at the same time meaningful significance to people with an equal sharpness in its design. This kind of projects has always followed similar principles, whether being related to censorship, anonymity, persistency of information or truth. Services like *Tor, WikiLeaks* and *Bitcoin* are just examples. The decentralization process of already existing services is the reason of their success and the cause of their widespread usage: the ability to evade regulations if needed and express their potential *even* when ethic is at risk. These tools have been created often by unknown people that emerged from the sidelines to deliver a different mechanism than centralized services already provided with the clear intent to give voice and means to those that could not surface in the society cog.

But as with every *free-from-control* tool that exist, the responsibility, consequences and ethics of its usage rely solely in the hands of the final individual and his own judgment.

Cryptocurrencies blend into this world not differently from other distributed technologies: they can be deployed for a number of use cases as any fiat currency could. Their appealing side is of course their availability and the relatively simplicity in which a complex operation (like a fund transfer) can be achieved over the simple internet network, but this is just one side of the innovation.

1.0 INTRODUCTION

Starting from *bitcoin's* inception in 2009 the term "cryptocurrency" has been widely adopted to describe a different type of money in relation to classic "fiat" currencies (on printed papers). Taking advantage of a distributed environment and a security mechanism enforced by cryptography *bitcoin* started a new age of services for economics and transactions over the internet, edging with a new payment model for money transfers that set off many banks, companies and governments. The 2016 has been a great year for blockchain-based technologies like *Bitcoin* and *Ethereum*, the innovation introduced with distributed ledgers, led more and more start-ups, companies, researchers and common people (even non-tech ones) to experiment on it, testing, employing and starting to use it as an alternate way of carrying out their own business model. However, understanding the needs and complexities beyond a distributed ledger and the new platforms built on top of it is not an easy task. To get ahead of all this and in order to grasp this tech's *momentum* we first have to go back a few years into blockchain's background and analyze its history. This chapter will be a guide throughout all the available data on this topic and will give the reader means to understand the concepts and technicalities that will arise later in the work.

In the past few years blockchain technology has spread significantly, however we couldn't talk about blockchain while leaving out its "father": *bitcoin*. In a way, they each represent a side of a single coin, the first being the main backbone data structure behind bitcoin while the latter has been the main purpose of the existence of blockchain itself. In order to lay out the key concept behind this work we have to speak about *BC*[1] and introduce some insight about cryptocurrencies too, however since they are not the main topic of this document these details will be given out progressively as we peer deeper into the arguments. The amount of information given on these other topics will be limited to the scope of the actual paragraph's subject.

One of the major premise in the analysis of the blockchain phenomenon is that it has arisen completely online in the network and in an anonymous fashion that prevented the majority of fact checking and investigations from both governments and individuals. After the breakout of this technology however a significant amount of research and tests have been done exploring

[1] *Blockchain*

both its technical and structural aspects. This led to many new projects and forks that more or less did set a new kind of services developed by other parties interested in blockchain.

1.1 BITCOIN AND BLOCKCHAIN

A cryptocurrency can be defined as a digital asset that can interact as a medium used for an exchange, the term "crypto" is a prefix adopted to declare that transactions generated by this currency are cryptographically-secured (e.g. with *SHA-256*[2]). There are a number of digital currencies and cryptocurrencies in the network but Bitcoin is the first deployed payment system of its kind (Castillo, 2013) invented by the mysterious *Satoshi Nakamoto*. Bitcoin (or *BTC* for the currency itself) is the first decentralized virtual currency deployed (Calvery, 2013) and the largest of its kind in terms of total market value. The system is fully *peer-to-peer* with transactions taking place directly between users with no need of an intermediary (or a trusted administrator); in order to be a validated system the transactions are verified by network nodes and then recorded into a public *distributed ledger* called *blockchain*. This main decentralization feature, along with encryption, public availability and data persistency can be realized only through the blockchain data structure and this is the reason that made bitcoin very popular online and across the globe.

BC was first described in *Nakamoto's* paper as an elegant solution to achieve all those features and solve at the same time both the *infinite digital asset reproducibility* characteristic and the *double spending* problems involved in the development of electronic-money (Armstrong, 2016). The distributed ledger data architecture was initially overshadowed by the "bitcoin revolution" and the wave of news that the virtual currency brought in the web. Recently however it has become clear that the cryptocurrency is just a part of the innovation introduced and that Bitcoin in its former implementation is not suitable to be a silver bullet in payment systems (though the real question is "should it really be?"). As the spotlight moved away solely from bitcoin, blockchain risen from the shadows and became very popular.

A blockchain is essentially a decentralized digital ledger with duplicate copies that records transactions on thousands of computers around the world in a way that those transactions cannot be altered retrospectively. This enables and allows asset ownership and transfer to be recorded without external verification, in fact the authentication process comes from mass collaboration

[2] *Secure Hash Algorithm*, a family of cryptographic hash functions.

powered by collective self-interest (Don Tapscott, Here's Why Blockchains Will Change the World, 2016).

Under this guise, blockchain offers a way for people who do not know or trust each other to create a record of who owns what that will compel the assent of everyone concerned. A database that contains the payment history of every bitcoin in circulation, the blockchain provides proof of who owns what at any given time. In order to provide durability to this data, the distributed ledger is replicated on thousands of computers or "nodes" around the world and is publicly available. A BC database consists of transactions and blocks. Blocks hold batches of valid transactions that are timestamped, hashed and encoded into a *Merkle Tree*[3]. Each block includes the hash of the prior block in the blockchain, linking the two: linked blocks form a chain. This architecture maintains a growing list of blocks thus creating a digital ledger. Blocks are secured from revision and tampering, cryptography is used to allow each participant on the network for ledger manipulation in a secured way without any help from central authority.

As blockchain popularity increased between small BC-focused companies, a number of big-time firms (like IBM, Intel, Samsung, Microsoft and others) started to research on this technology and finding that its openness would grant a wide variety of freedom in its implementation and therefore in its usage. However the plain "old" structure of Bitcoin's blockchain was not viable to be developed with all the increasing concepts. In order to create tangible proof of business research companies started to develop their own blockchain implementations and protocols, taking Nakamoto's original one as basis and setting up new rules and new features.

With recent investments, many groups and firms have joined their forces to produce new blockchain services that can reduce costs in the banking and financial sector. This immediate and big advantage can be achieved out of this tech because of the natural similarities brought from its use with currencies and tokens. Many other ideas for usage beyond financial have been elaborated over the expectancy of BC, covering a wide variety of possibilities ranging from the use in public offices (for records, trades, loans and so forth) to supply chains, *IoT*[4], automation, messaging, data storage and so on. This growing customization created a schism about the fundamental question over which some parties still debate over:

[3] **Hash Tree** or **Merkle Tree** is a tree in which every non-leaf node is labelled with the hash of the labels or values (in case of leaves) of its child nodes.
[4] *Internet of Things*

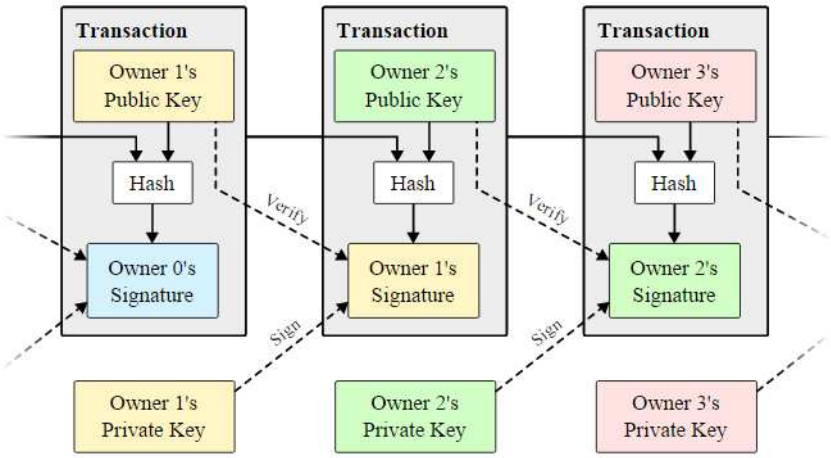
"is there any value in a blockchain without a cryptocurrency?"

To better understand this question we will point out that blockchain is both an economic and a computer science innovation. However the term "innovation" here comprehend a new combination of existing techniques, rather than something which has no precedent whatsoever. As a peer-to-peer technology we can compare BC to the *World Wide Web*, its invention is considered as an innovation, even though it did little more than combining hypertext with some existing *Internet* protocols. The point of having this question though is because some blockchain forks do stripe away its binding with a cryptocurrency, flushing away aspects that were initially conceived to strengthen its architecture. In light of other purposes we can say that blockchain without a token do serve a purpose which is just different from the original bitcoin BC one (Greenspan, Ending the bitcoin vs blockchain debate, 2015). The notion of shared public ledgers *per se* may not sound revolutionary or intriguing but the real innovation here are not the digital coins themselves, but the trust machine used to mint them, which promises much more besides simple financial transactions (The Economist, 2015).

1.2 THE RISE OF DISTRIBUTED LEDGERS

Nakamoto's paper states that "*commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust-based model*". What Nakamoto first described and then deployed is a complete payment system that overcomes this model, shifting the trust-based third-parties to peers on the internet, willing to cooperate with the goal to achieve the mutual benefits of this working payment system. Briefly, following the author definitions, we can define an electronic coin as a chain of digital signatures. The coins are transferred from an owner to the next by the digitally signing the hash of the previous transaction plus the public key of the next owner. Public keys are cryptographically generated addresses stored in the blockchain that are seldom tied to a real-world identity. A payee can verify the signatures to verify the chain of ownership but the problem is that the same payee can't verify that one of the owners did not double-spend the coin. The only way to confirm the absence of a transaction is to be aware of all transactions in fact, for our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. In order to accomplish this without the use of a third trusted party we need:

- Publicly announced transactions.
- A system for participants to agree on a single history of the transactions order.
- Proof (for the payee) that at the time of each transaction, the majority of nodes agreed it was the first received.



Picture 1 - Bitcoin's blockchain model

The solution for those needs has been laid out with the following features:

- A **timestamp server** that takes the hash of a block of items to be timestamped and then publishes the hash. The timestamp proves that the data must have existed at the right time and ordered to get into the hash. Each timestamp includes the previous one in its hash, forming a *chain*, with each additional timestamp reinforcing the ones before it.

- A solid and guaranteed mechanism that can eliminate the reproducibility problem of the digital medium called **Proof-of-Work (POW).** The concept of POW has been introduced by Dwork and Naor (Dwork C, 1992) and defines a mechanism for which the resources needed to solve a computational problem should not be easily acquired and may not be scaled at will. Formally we can consider the function:

$$\mathcal{F}\ (\ d, c, x\ ) \rightarrow \{\ True, False\ \}$$

where *d* is a positive number defined as *difficulty*, *c* and *x* are bit-strings where the first is the *challenge* and the second a *nonce*[5]. $\mathcal{F}$ is called a *PoW function* if it has the following properties:

[5] A *nonce* is an arbitrary number that may only be used once.

1. $\boldsymbol{F}$ *(d, c, x)* is fast to compute if *d*, *c* and *x* are given
2. For fixed parameters *d* and *c*, finding *x* so that $\boldsymbol{F}$ *(d, c, x) = True* using a unit-resource is distributed with *exp(1/d)*, i.e., computationally difficult but feasible.

The mining operation involves the process of scanning for a value (*x*) that when hashed (like with *SHA-256*), will make the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits wanted but can be verified by executing a single hash. The key feature here is asymmetry: the work must be moderately hard (but feasible) to resolve (hence the term "puzzle" in the slang of proof-of-work) but easy to check by other nodes in order to be validated. To align this feature with the timestamp network the POW is implemented by incrementing the *nonce* in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing all the work. As following blocks are chained after it, the work to change the block would include redoing all the blocks after it. Proof-of-work also solves the problem of determining representation in majority decision making, being able to surpass a one-IP-address-one-vote (that could be subverted by anyone able to allocate many IPs) in favor of a one-CPU-one-vote. The majority decision here is represented by the longest chain, which has the greatest proof-of-work effort invested in it. To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases (Nakamoto, 2008).

Blockchain workflow:

1. New digitally signed transactions (coming from users) are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding the proof-of-work for its own block, solving the puzzle.
4. When a node finds a POW, it broadcasts the block to all nodes.
5. Nodes accept the block only if all transactions in it are valid and not already spent.
6. Nodes express their acceptance of the block by start working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

The node's puzzle can only be solved by trial and error, therefore across the network, all nodes (called often "miners" for mined currencies) grind through trillions of possibilities looking for the answer. When a node finally comes up with a solution the other quickly check it (again,

solving is hard but checking is easy), and each node that confirms the solution updates the chain accordingly, nodes always consider the longest chain to be the correct one and will keep working on extending it. The hash of the header becomes the new block's identifying string, and that block is now a permanent part of the ledger. With this type of robust workflow, blockchain have been even described as a value-exchange protocol (Bheemaiah, 2015).

There are a number of factors in place to thwart attackers that can be summarized in:

1. *Chance*: It is virtually impossible predict which node (*miner*) will solve the puzzle, and so there can be no clue on who will get to update the blockchain at any given time.
2. *History*: Each new header contains a hash of the previous block's header, which in turn contains a hash of the header before that, and so on all the way back to the beginning. It is this concatenation that makes the blocks into a chain. Starting from all the data in the ledger it is trivial to reproduce the header for the latest block. Making a change anywhere even back in one of the earliest blocks will cause a chained reaction where all the subsequent block's headers will come out different. The ledger will no longer match the latest block's identifier, and will be rejected.
3. *Reward*: probably one of the most important key features of bitcoin's blockchain. Solving the POW puzzle (and forging a new block correctly) creates new bitcoins. As of now the winning miner earns 12 bitcoin, worth about $28.460 at current prices. The puzzle-solving step adds is an incentive which encourage nodes to stay honest.

With this kind of countermeasures in place even a skilled and resourceful attacker, able to assemble more CPU power than the rest of all the honest nodes, would have ultimately to choose between using that power to defraud people by stealing back his payments or using it to generate new coins. In the end, is all about considering a good profit out of the resources used: the puzzle-solving operation is very CPU-intensive, which drains computer power in form of electricity that has a non-negligible cost to the hardware's owner (not considering the hardware cost itself). The bitcoin reward profit must be matched with the hourly, daily or weekly cost of power consumption that could "waste" all the amount of bitcoins earned. This is a strong security policy.

1.3 WHAT THE BLOCKCHAIN IS

Studying its architectural side, we can observe that blockchain not only provides a way for secure transactions to take place, but also make it easy to recover corrupt data and in the same time minimizes loss possibility as every node inside the chain has a copy of data. Blockchain can thus be integrated into multiple areas: some of them are about payment systems related to digital (and physical) currency, like title tracking, payments, transactions, others range from permission distribution (like distributed sharing voting systems) to information anchoring, "truth proving", meta-token creation, identity demonstration, intellectual propriety handling, secure messaging, insurances and so on. Businesses learned that the potential of BC lies not so much in using it as a replacement technology, but rather in its ability to enable new business process improvement opportunities (Fredrik Milani, 2016). This concept has been absorbed by researchers and programmers and then re-engineered in different blockchain implementations that add other key-features. Successful use cases are (but not limited to):

Land Registry: one of the first tryout applications of blockchain outside the cryptocurrency scope has been the use in house and land registry area. Benefits in this sectors can be obtained on two sides: the first being the storage of land owning registry in a safe ledger, the second is about home-sales tracking, both encompasses the property over a crucial asset for citizens of a nation. On July 2016 Sweden and Scandinavian were conducting tests to put the country's land registry system on blockchain. The long shot of this planning is to put real estate transactions on blockchain once the buyer and seller agree on a deal and a contract is made, so that everybody (banks, government, brokers, buyers and sellers) will be able to track the progress of the deal (Chavez-Dreyfuss, 2016). This kind of application could potentially help all countries currently struggling with land title fraud since many databases are simply hacked and the contained properties' ownership faked.

Crowdfunding: being blockchain the structure behind a cryptocurrency one of the most successful project in the area is crowdfunding. The idea behind this concept is to provide a decentralized version of a funding application, this design is meant to function as a streamlined tool to commit pledges from people all around the world and use them to fuel special projects that will be more independent from countries policies and limitations. The money gained from pledges will be made available (and "unlocked") to the project owners only if and when the target amount is reached. This service has been used by different websites and organizations to

create not only projects related to technology or research but even medical, emergencies, cultural and so forth (Higgins, Bitcoin-Powered Crowdfunding App Lighthouse Has Launched, 2015).

Smart Contracts: arguably the most advanced feature integrated inside the blockchain technology. They were first defined in the early 1990s as a set of computer protocols and user interfaces intended for formalizing and securing relationships and agreements over computer networks, a SC thus encodes the terms of a traditional contract into a computer program that executes its clauses automatically (Szabo, 1994). Within blockchain technology, smart contracts can be self-executing and self-enforcing without the need for intermediaries. A particular clause could encapsulate, for example, complex terms and conditions which could be met only with a contingency on an external event (such as a required target amount of money for a crowdfunding operation). A blockchain-based SC is publicly visible to all users and can be extended with appropriate programming language instructions which both define and execute an agreement. This complex feature extends the blockchain domain to other important business areas that includes financial instruments like bonds, shares and derivatives, assurance policies, contracts and other instruments and transactions where nodes can monitor the events related to the rules dictated by the smart contract. In 2015, *UBS*[6] was already experimenting with "smart bonds" using bitcoin blockchain (Ross, 2015), but the group that has poured more resources and commitment in this direction is the *Ethereum Foundation*.
There are a number of potential benefits in using smart contracts that will be covered as we proceed but the most interesting feature is the possibility of embedding trust in a code that could overcome moral hazard problems and reduce costs of verification and enforcement. It is still debated however if the legal status of these contracts could raise serious consumer protection issues. Since blockchain-based smart contracts are still at an early stage, some believe that they are not reliable and with several unsolved problems (International Monetary Fund, 2016), however it would be wrong to neglect their wide capabilities just because their use is difficult and hard to comprehend at first.

Digital Organizations: in light of the feature offered by the previous examples, smart contracts can be custom-programmed and pushed onward resulting in the creation of a new level of organization scheme. A *decentralized autonomous organization* (*DAO*) is a complex set of rules

[6] **UBS AG:** a Swiss global financial services company based in Zurich and Basel.

and clauses defined by a number of smart contracts, creating what can be considered a full-working company or organization composed by a net of freelancers. This kind of system is run by people themselves but enforced by software rules, they work together on projects which are voted inside the organization' scope, the resources (money) available to the organization is then committed once a project is approved and people get paid on deliver or on completion of the project. All of this is achieved potentially online without the need to congregate physically or to form a brand new organization from scratch. All of the DAO's financial assets, transaction record and program rules are therefore kept on a blockchain that runs all the structure of the organization and, usually, supply even the necessary tools to handle projects and the interaction with people (Paul Vigna, 2015). It is fair to say however that even if this business model has a good number of successful cases, it is still a dangerous terrain to build something real on because there is no clear legal standing for this type of organizations and regulators are doubtful about the real advantages (Popper, 2016).

Finance: being blockchain the structure behind cryptocurrencies, one of its most common customization and use involves bank and financial areas. On September 2016 a number of major firms is Switzerland including: Swisscom, the Swiss stock exchange, Zurich Cantonal Bank and others, have formed a consortium to use blockchain technology for the facilitation of selling shares outside of a stock exchange. The *R3CEV* company is another consortium that allowed some of the biggest financial institution in the world to research on blockchain and integrate it in financial systems. The main driver for the use of BC in this area is that while payment requests can be fast over the web and internet, the actual financial assets being transferred still moves over old systems that connect all the institutions involved in the physical process of the transactions. It can take days for the funds to actually reach an account, therefore these systems both slow and really expensive too.

This kind of problems are not uniquely tided to banks or credit institutions; many companies and public bodies suffer from hard-to-maintain and incompatible databases, resulting in a high transaction costs because of the interoperability needed when interfacing to other systems. This is the problem that *Ethereum*[7], one of the most ambitious distributed-ledger project, wants to solve. The blockchain used in *Ethereum* can deal with more data than bitcoin's can and it comes with a programming language that allows users to write more sophisticated smart contracts able, for example, to create invoices that pay themselves when a shipment arrives or share certificates

[7] Link: "*https://www.ethereum.org/*"

which automatically send their owners dividends if profits reach a certain level. Strictly for finance world, blockchain would significantly lower the upkeep for the transaction systems and ease some of the procedures, lowering costs by making payment processing more efficient. On June 2015 *MasterCard*[8] company replied to a request for information about blockchain technology with a 4-page response stating that "*digital currency's risks outweigh the benefits*" (Spaven, 2015). On 21st October 2016 however *VISA*[9] announced new details about a forthcoming *B2B*[10] payment service developed with a blockchain startup to be launched in 2017 (Higgins, Visa to Launch Blockchain Payments Service Next Year, 2016). *MasterCard* probably changed opinion early before in the timeline and just after 10 days (on October 31st) they released an experimental API from *Mastercard Labs* that is connected to their internal blockchain work.

Private vs. Public / Token vs. Tokenless blockchains: from the moment when new implementations of blockchain technology risen there has been a wide degree of modifications and customizations. The difference between custom implementations is the use of public or private ledgers, bound with a token or tokenless scheme. The former structure of blockchain is, by definition, a public distributed ledger born with the specific purpose of being the backbone for bitcoin currency. Its public applications however are not restricted to a currency or token use of this structure: car leasing and sales automated by transaction that will lead to a programmable economy that will output on the *Internet of Things*, markets prediction, ride sharing, healthcare and supply chain management are just examples. *Everledger*[11] is a global distributed ledger built for the specific need of tracking the source, origin and trade of diamonds, in order to prevent fraud. A diamond blockchain can record each gem's unique combination of attributes, giving it a precise and distinct pattern which can then be put on the ledger in order to verify its tracking and status on a supply chain or in a chain of custody (Levy, 2016).

Other versions of blockchain that follow a token-free scheme have a different purpose from the original bitcoin's; by removing the medium contended by anonymous miners we lose some features like transparency and decentralized security based on proof-of-work. However if we consider a private blockchain maintained within a single organization there is no need of these

[8] *Mastercard Incorporated*
[9] *Visa Inc.*
[10] *Business-To-Business*
[11] Link: *"https://www.everledger.io/"*

features because we have *perfect trust*, in this scenario BC is useful for keeping decentralized databases in sync or can be used for creating consensus for specific types of transactions *between* organization that have only a *limited* degree of trust. Instead of using bitcoin or any currency as token, we can have a token-free model in which each row can represent multiple assets, this system would be built on top of a closed list of authorized miners, who identify themselves by signing the blocks that they create. This is a radical different approach from the traditional blockchain, but it serves well on highly regulated financial systems if you can accept the restriction that miners must be pre-approved (Greenspan, Ending the bitcoin vs blockchain debate, 2015). This type of BC, with the ability to restrict the participation and consensus process falls under the *permissioned* class of distributed ledgers. These ledgers are still subject to open debates and controversy because they would serve as a mere distributed version of the *multiversion concurrency control* (MVCC), which is usually implemented by traditional corporate-level databases. Therefore this process will reintroduce some security issues and pitfalls that cannot be longer mitigated from a public, token-mined environment (Don Tapscott, The Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World, 2016). After this description we can summarize the different designs of distributed ledgers into the following categories (Buterin, On Public and Private Blockchains, 2015):

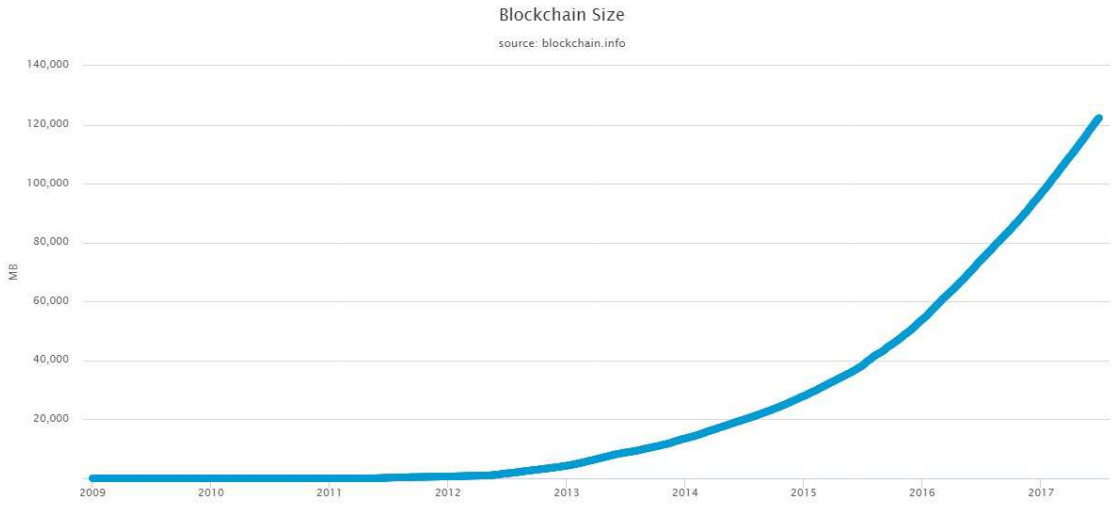
- **Fully public blockchains:** the more traditional approach is represented by decentralized ledgers open to all Internet users. Anyone can read, submit transactions and participate in the *consensus process* needed for determining which blocks will be added onto the chain. The security in this model is provided by a combination of economic incentives and cryptographic verification, using mechanisms such as proof-of-work or proof-of-stake. The general principle here is that the degree to which someone can have an influence in the consensus process is proportional to the real quantity of economic resources that they can bring to bear.

- **Fully private blockchains:** the opposite approach is one in which permissions are kept centralized and assigned by a trusted entity that replaces the proof needed while mining. Such a system does not need an embedded token or currency since his central entity can assign manually computers to verify transactions. Read permissions may be public or restricted to some extent based on the business model implied. Applications include database management, auditing, etc.

- **Hybrid or consortium blockchains:** another approach is to make up a mixed set of rules from the previous two, consensus validation process is controlled by a pre-selected individual or organization. The right to read the associated blockchain may be public or restricted to the participants. These systems are considered partially decentralized due to their nature of being shared between different companies/entities that may hold one single node of the computation that together form the blockchain. Business rules are applied in nodes to conform them to the BC procedures, the different degrees of trust at work here can be subject to both token and token-free models.

Disadvantages of Blockchain: after a complete readout on its main features and applications it is fair to point out even the drawbacks and difficulties that are compelled with the use of distributed ledgers. There is, of course, a tradeoff for using BC technology; the more influencing aspects will be summarized in the following list:

- **Space**: blockchain requires increasingly more storage space as the number of transactions climb up, this space is occupied in each and every simple node (or miner node) that is contributing to the consensus process of the ledger because every transaction is stored by everyone. This factor is mitigated by optimization techniques to prune the unneeded data but still remain a central issue while using a blockchain.



Picture 2- Blockchain total size for Bitcoin network

- Time: transaction completion takes more time compared to other technologies, this is because the transaction verification process is longer and is dependent on the miners for verification. After this process, a transaction is broadcasted to all nodes as new block. Although custom ledgers have mitigated this problem, it remains a huge drawback for bitcoin's BC that today can handle only 7 transactions per second due to its protocol restricting the block size to 1 megabyte and taking an average 10 minutes for a new block to be mined.

- Costs: from user prospective, the fees for transactions may vary from service to service and from miner to miner since every one of them decide the charge rate for the transaction's verification. On the other side the hardware cost for the mining process is non-negligible, tied with the hourly/daily/weekly power consumption required for the CPU calculus to be carried out.

- Security: the whole structure of cryptocurrency is not immune to the threat of hacking. During bitcoin's brief history the company has been attacked more than 40 times with a few thefts that exceeded $1 million in value, other projects (like Ethereum) have been attacked and drained too. The standard blockchain network is an implicit solution for the notorious *Byzantine General Problem* (Leslie Lamport, 1982), but relies on the fact that the majority of its miner nodes remain *honest* (> 50%). However, a number of research and studies pointed out that this is not enough: a sufficient large mining pool that employs a *Selfish Mining* strategy (Ittay Eyal, 2014) could subvert the network's protocol into one where blocks generated outside the pool would be ignored. Bitcoin protocol as it is now will never be safe against this type of attack if the mining pool manages to get more than 1/3 of the total mining power of the network. However, there are other consideration to take into account for double-spending attack to be deployed like effective resulting probabilities of success by hashrate, earned value vs costs, number of confirmations and others (Rosenfeld, 2014) .

- Objectiveness: when it comes to reality, the blockchain phenomenon has received a huge hype into the believing that it can be the final answer to a plethora of problems, this is misleading because BC is no silver bullet. Punctual and meticulous analysis must be done when striping BC from its former application (bitcoin) in order to understand

both benefits and drawbacks of this technology, this is because there is a complex interplay of many critical technology components that work together to make bitcoin secure, many of which can't be applied outside the scope of the cryptocurrency. An important notion to keep into consideration when parting from token models is that bitcoin isn't secure because of blockchain (primarily), instead the security is provided because the effort and cost of subverting the whole structure is greater than the value of what's being protected.

1.3 THE BIG PLAYERS / BLOCKCHAIN TODAY

Blockchain technology has come a long way from its introduction with Bitcoin in 2009, as new business possibilities emerges, new brands and developers are trying to get an edge on the market by providing a fully-personalized blockchain that can deliver more tailed-services than the original one. This section will introduce the state-of-the-art projects on distributed ledgers provided by organizations that are currently researching or providing an extensive blockchain-based product (a platform) on top of which third-parties can develop a variety of services tailed to their specific business needs. To better understand the size of the projects based on cryptocurrencies, we will list them by their total value of market capitalization as of July 2017:

-Bitcoin: although *Bitcoin* (BTC) is not designed to change its former implementation (therefore having limited applications other than the original) it has the largest capitalization between cryptocurrencies, being over $40.700.000.000. This large share is due to its age, moreover that the system has been adopted around the world by a number of official organizations and institutions and even because it is the main cryptocurrency used to exchange for minor digital currencies. A single BTC today has a value of $2.470 but as stated, the future development of this system is stuck on a debate over the better way to upgrade the backbone rules of the system.

-Ethereum: With a market total value of about $26.192.000.000 and a token price of $280, this project comprises both a cryptocurrency payment system (currency is called "ETH") and a distributed environment built on top of a custom-blockchain conceived for the deployment of *smart contracts* that can generate *Dapps* (decentralized applications). We will cover Ethereum in the next chapter.

-Ripple: A financial-target solution with a total market value of $9.888.000.000 built upon a distributed, open source, internet protocol, consensus ledger refined into the Ripple Transaction Protocol (RTXP). Once a company joins the network, Ripple is designed to enable payments (with both fiat and digital currencies) across different ledgers and networks persistently and globally. One of its main feature is the great degree of interoperability, giving banks and other financial parties located on different networks the ability to make transactions with one another directly even in the events of cross-border payments. Ripple therefore allows customers to have lower total costs when executing payments, and banks to offer new type of products and services without the need to worry about the underling provider or financial infrastructure used. (Liu, 2013).

-Others: There are a number of other smaller projects in the blockchain area (besides minor digital currencies as well), we will reference here just some of the most notorious.

The *Hyperledger* open-source project is a distributed ledger platform born in the end of 2015, by a collaborative effort created to advance cross-industry blockchain technologies, hosted by the *Linux Foundation*. The project aims at improving different aspects of the BC technology by combining new open protocols and standards with the goal to develop an enterprise-level modular framework that can be deployed in different type of businesses or industry-specific applications (The Linux Foundation, 2015). Hyperledger itself is the sum of different blockchain projects, each with an individual identity, features, purpose and objectives as stated by the project community.

Multichain is an off-the-shelf platform based on a fork of *Bitcoin Core*, it is focused on bringing the powerful features of *Bitcoin's* blockchain technology into institutional financial sector with relatively ease while extending its capabilities. All the main features are packetized in a ready solution that can create and deploy private blockchains, either within or between organizations, providing all controls needed for suiting the needs of the organizations. Being private, Multichain addresses the mining problem with openness declined with the use of integrated management of user permissions that ensures visibility and allows transactions only among chosen participants with enough privileges (Greenspan, MultiChain Private Blockchain Whitepaper) .

2.0 ETHEREUM

Ethereum is an open source project created and maintained by the *Ethereum Foundation*[12], which develops a public distributed computing platform built on top of a customized blockchain network. The objective of Ethereum is creating and promoting development of both decentralized protocols and tools in order to build a new set of *decentralized applications*, with a different set of tradeoff that will be useful in solving a large class of problems in business domain (Foundation Team, 2014). To make an example of other on-chain implementation we can take Bitcoin: it offers a rudimentary scripting system that is neither expressive nor user-friendly. Many people in industry and research have tried to design and implement different smart-contract-like applications attempting to retrofit Bitcoin's scripting language (Marcin Andrychowicz S. D., 2013). However the effective expressiveness of this scripting language is very poor and the retrofitting process is both time consuming and costly, leading to more and more laborious work demanding a high effort for it to be efficient. Such need for custom implementations is the one that drove the Ethereum Foundation into the creation of a smart contracts ad-hoc platform, which has become the first and more viable to program at the moment than previous attempts and work-arounds. The key-features of this protocol are focused on rapid development times, security for small applications and the boosting of interaction capabilities between the different applications. All of this is accomplished by building *Đapps*[13] on top of a blockchain's abstract foundational layer, integrated with a built-in *Turing-complete*[14] (Ethereum Community, 2016) programming language capable of defining smart contracts. Ethereum is hence a complete platform: it provides a decentralized virtual machine called *EVM* (Ethereum Virtual Machine) that can execute coded computation on a "global-computer" realizing *peer-to-peer* contracts and services while using a token called *ether* (Buterin, Ethereum White Paper, 2014).

Being a background platform capable of providing an increasing number of ways to develop services, many small, medium or enterprise-level projects have adopted the Ethereum platform. The aim of this project is having a decentralized token-based "operating system" upon which all third-parties can develop their business solutions on. With this feature the Ethereum platform is natively inclined to support all sort of brand new tokenized-projects that can be implemented

[12] A non-profit organization, *https://www.ethereum.org/foundation*

[13] This writing style identifies *Ethereum Distributed Applications* specifically

[14] In computability theory, an instruction set or programming language is said to be **Turing complete** if it can be used to simulate any single-taped Turing machine.

with its programming language. Applications on this side range from anything related to digital currencies to contracting, savings wallets, wills and every specific-regulated need, all of this can be mapped with Ethereum smart contracts using the ETH currency to pay for services offered by the platform. We can then find all applications in which there is still a token component but the business model involves a non-monetary side, like identity and reputation systems, decentralized file storage or decentralized autonomous organizations. On a third category we can put all application related to decentralized governance, online voting, management and so on, which do not have a financial component at all. Beyond these categories, Ethereum has a longer list of applications, many of which have been proposed and funded, others are currently being scoped and tested more accurately. Domains that involve insurances, decentralized data feed, multisignature transaction contracts, cloud computing, peer-to-peer gambling, prediction markets and decentralized marketplaces are just examples (Buterin, Ethereum White Paper, 2014).

2.1 THE ETHEREUM PROJECT

The Ethereum environment and platform have been designed to be adaptable and flexible, unlike Bitcoin, Ethereum founders wanted to create a fully trustless smart contract platform. As a programmable blockchain, Ethereum provide users with means to create their own operations at any wanted level of complexity instead of relying just on currency transaction scripting. The core concept behind this programmable-feature is the *Ethereum Virtual Machine*, a runtime environment for the execution of smart contracts. It is a completely isolated environment, thus the running code inside it has no access to external resources like file system, network or other processes. The deploy process is carried out on an Ethereum client and follows a high level language compilation with a specific EVM compiler. Smart contracts are then deployed on the blockchain and reside on the network stored in a special binary-format called *EVM bytecode*. The EVM can execute code of arbitrary algorithmic complexity thus falling under the *Turing Complete* classification, its main programming language called *Solidity* is modelled on *JavaScript*.

The Ethereum environment has a *peer-to-peer* network protocol and blockchain structure way different than the Bitcoin's original, its database (about 20 GB in size for an Ethereum full node as of now) is constantly maintained and updated by the nodes throughout the network. Nodes that run the Ethereum client execute the same instructions set on a local EVM instance, this process is used to maintain a decentralized consensus across the blockchain granting interesting

features like high level of fault tolerance, no downtime and of course censorship-resistant data storing. This structure and protocol together create an environment that, as advertised, favors application that *"automate direct interaction between peers or facilitate coordinated group action across a network"* (Ethereum Community, 2016). As a both common and cheap infrastructure, users can take advantage of other "background" features like: user authentication verified by cryptographic signatures, easy-deployed payment logic, a certain degree of resistance in denial of service attacks (we will resume this point later), great interoperability between contracts, no server infrastructure (or single point of failure).

The roadmap for any average Ethereum-based project to become live starts with a concept of service that can be implemented on the blockchain: the designers describe that concept and lay out what can be defined as a "white paper" that states their goal and gives some use cases. After the presentation comes, a date and time period are chosen for the crowdfunding phase of the project, this process has been defined in jargon as *"Initial Coin Offer"* (or ICO, as opposed to the classic "initial public offering"). The ICO serves as a mean to raise funds for the new cryptocurrency venture, therefore bypassing the rigorous and regulated capital-raising processes required by venture capitalists or banks. For the duration of an ICO, a pre-mined fixed amount of the new currency's token is sold to early backers in exchange for other cryptos[15] (Investopedia, s.d.). Once the duration is expired or the target amount of tokens have been sold, the projects goes to development status and with good guidance and timing, it goes from first test version to a final product or service.

The important thing to notice here is that all of these processes (with exception of the presentation part) can be achieved solely with the Ethereum platform, using smart contracts to write code that operate the ICO phase, hold funds and later, the service itself. If the contract is well-coded, it can even refund money back to backers if the target is not reached within the initial offer time-window. To understand the scope of this platform's ecosystem we will summarize here a brief overview of the main Ethereum-based projects that are currently being deployed or funded to an active status (as of July 2017), describing their concept or proposal:

[15] Short for "Cryptocurrencies"

-**Aragon**: a distributed application designed for running DAOs (Decentralized Autonomous Organizations), anything needed to manage a digital company like cap table, governance, fundraising, payroll, accounting, bylaws and other necessities are packed together in an easy and manageable environment. Aragon is currently in alpha.

-**Augur**: a decentralized prediction market with the ability to forecast the outcome of an event based on the "*wisdom of the crowd*" principle. Following this method, information is collected from the crowd and averaged into the most realistic possibility and therefore the most probable outcome. Correct predictions are awarded by the network while incorrect reports are penalized, all of this is to create an incentive to truthful reporting and it is enforced with the usage of a tradable *Reputation* token. Augur is currently in beta.

-**Bancor**: a protocol that enables anyone to create a new type of crypto called "smart token" that can hold and trade other cryptocurrencies. It eases the market of other tokens by removing the need of second parties in token trades (exchangers). Bancor is deployed and live.

-**Brave & Bat**: Brave is a new blockchain-enabled browser that creates an environment resistant to both ads and trackers while introducing a new blockchain-based digital advertising model. Giving a new focus on the user attention and through the *Basic Attention Token* (BAT), the project has created a decentralized ad exchange, part of a new advertising strategy that aims to solve *malvertising* problems on the internet. The philosophy here is that user can receive rewards for their "attention" if they choose to see the ads on the website. Brave is currently available while Bat is in beta.

-**Status**: an open source messaging platform and browser that is designed to enable mobile devices in the use of Ethereum decentralized applications, turning devices into a light client node of the network that can peer in and interact. Status is currently in alpha.

-**PeerName**: an Ethereum-based DNS (Domain Name System) that servers as both a provider for Ethereum name system (ENS) and for other decentralized domain names that come from different DNS zones than the one usually provided by ICANN[16]. PeerName is a deployed and live service.

-**Sonm**: project that aims to provide a universal cost-effective super-computer designed for general-purpose computation. In this concept, miner on the network can make use of their idle

[16] *Internet Corporation for Assigned Names and Numbers*

computer power to become part of the Sonm network and earn its token or spend it in exchange for computation. Sonm concept has been funded in June.

-**Slock.it**: a decentralized smart physical lock that can listen to the blockchain. This *IoT*-related usage backs the fact that one can lock any asset (e.g apartment, car, bycicle) behind the *Slock* and anyone can rent the asset for a fee in Ether. This project showcased the potential of Ethereum connected to a real-world device in the beginning and today is enforced by smart contract and deployed by several businesses, for example *AirBnB*[17].

-**Swarm**: a *peer-to-peer* storage platform and content distribution service implemented in a serverless paradigm. From user prospective, swarm operates like WWW[18] but without a specific server with the integration of blockchain-based domain name resolution. Anyone with free space can rent it for a token reward or upload its data to the network, indexing headers will be maintained in the blockchain. Swarm is currently in alpha.

-**Truffle**: a development framework to ease smart contract writing; it enables support for special deployments, library linking, testing on public or private networks and other related tools. Truffle is currently in beta.

2.2 THE PLATFORM

Ethereum now is in its second, and stable, release called *Homestead*. The pre-release had launched on May 2015 (*Olympic* testnet), followed by a first release codenamed *Frontier* on August 2015 and then by Homestead in March 2016. The other two planned releases are *Metropolis* (precise date is still to be announced, should be before the end of 2017) and *Serenity*. One very important notion about the evolution of Ethereum is that at a certain point the protocol will shift from the use of *Proof-Of-Work* as a validation mechanism for miners in favor of *Proof-Of-Stake*. There will be substantial protocol changes due to this evolution but overall it will be a major feature providing new functionalities for top programmers while maintaining its legacy, however in order to resolve backward-incompatible changes usually a network fork is required.
Ethereum is composed by different basic key-components that we can break down as follows:

1 - Ethereum blockchain network and protocol

[17] https://www.airbnb.it/
[18] *World Wide Web*

2 - Nodes that run an up-to-date Ethereum client

3 - Gas

4 - Web-3 interface

5 - Ethereum Virtual Machine

6 - Smart Contracts

The decentralized structure (1) keeps record of transactions between accounts and their balance of ETH, no one controls or owns Ethereum and the project is open-source. Ethereum's basic unit is therefore the *account* and there can be two type of accounts:

-*Externally Owned Accounts* (EOAs) which are controlled by private keys and represent identities of external agents (e.g. humans, mining nodes or automated agents).

-*Contract Accounts* which are controlled by an internal contract code that can be triggered into activation only externally by an EOA. They can perform operations only when instructed to do so, this is due to the requisite that nodes (2) must be able to *agree* on the outcome of a computation, leading to a strictly deterministic execution.

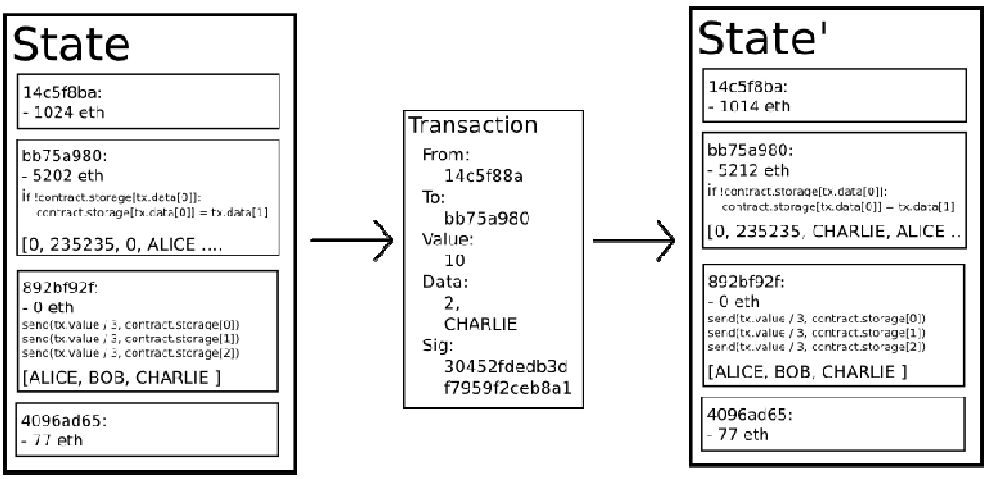
Both account entities are defined *state objects* because they implicitly incorporate attributes that define a state. Specifically an Ethereum account contains a 20-byte static address plus other four fields:

- A *Nonce* used as counter to ensure transaction uniqueness during processing
- The account's actual *ether balance*
- The *contract code* (if we are dealing with a Contract Account)
- The account's internal *storage* (empty by default)

From this prospective we can observe that the state of all accounts contribute to the state of the Ethereum network overall. Transaction sent from one account to another have an intrinsic cost called *Gas* (3) that must be paid by the transaction issuer. Gas is expressed units, each unit of gas as a price in ETH and its purpose is twofold: from the user side it discourages the submission of spam-like transactions or useless computational tasks (like DDoS[19] attacks or infinite loops). From the miner side it fixes a *transaction fee* that he can request as payment in order to mine (validate) a user transaction into a new block of the ledger. When a transaction is sent to a smart

[19] *Distributed Denial of Service*

contract activating some code, the computation is executed by one (or potentially every) node and the gas here is used to pay for each step of the "program" including computational power or memory storage therefore setting a *hard limit* to how much time, effort or resources are allocated for a single program execution. Miners obtain a reward from the system even when *their* block is successfully added into the chain, this represents the joint economic incentive for people to invest on mining hardware and electricity (this however will change with the future protocol migration in *Proof-of-Stake*). Usually a computational step costs 1 gas unit but there are operations that cost a higher amount of gas either because they perform more operations or because they need to increase the amount of data to be stored in the state. Plus, a fee of 5 gas is applied for every byte in the raw transaction data. A possible attacker is requested to pay proportionately for all the resources he wants to consume (computation, bandwidth and storage). If a code execution runs out of gas at any point an exception is raised inside the program, the state is reverted to pre-execution and all of the gas is lost.



Picture 3 - Ethereum state transition example

The state transition function of *Picture 3, APPLY(S, TX) -> S'* can be defined as follows:

1- Check if transaction is well-formed, the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.

2- Calculate the transaction fee as STARTGAS * GASPRICE (where STARTGAS represents the maximum number of computational steps allowed to be executed, and GASPRICE the fee payed *per* computational step) and determine the sending address from the signature.

Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.

3- Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4- Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5- If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6- Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

(Buterin, Ethereum White Paper, 2014)

From a "back end" prospective, Ethereum is seen as a *Web 3.0* technology, enabling a different version of internet where services like DNS and digital identity are decentralized and everyone can blend in this structure with economic interactions (Buterin, TNABC 2015 - Bitcoin 2.0 - Ideas and Applications, 2015). Specifically we can use an object provided by *web3.js* library (4) which is the Ethereum compatible *JavaScript API* that implements the *Generic JSON RPC*[20] specification. In order *to* make use of Đapps with an Ethereum node, the communication is handled through *RPC calls* to an exposed web3 interface, its API has an *eth* object that we can use for specific Ethereum interactions along with other commands (Triantafyllidis, 2016) (Nicola Atzei, 2016).

Down to the Đapps bytecode, inside the node's client we have the EVM (5) which has a simple stack-based architecture with a stack item size (word) of 256-bit (chosen to facilitate the *Keccak-256* hash scheme and elliptic-curve computations). The stack has a maximum size of 1024 elements and we can address its memory with a simple word byte array. The machine comes also with an independent storage model; this is similar in concept to the memory but with a word-addressable word array fashion. As opposed to memory, which is volatile, storage is persistent and is then integrated as part of the system state if computation ends successfully. More than that, the EVM does not follow the standard *Von Neumann* architecture; the program

[20] JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. See RFC 4627 for JSON spec.

code is stored separately in a virtual ROM by which we can interact only through a specialized instruction (Buterin, Ethereum White Paper, 2014).

The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately, we will see that some of this behaviours can lead to hazardous situation and security issues (Wood, Ethereum: a secure decentralised generalised transaction ledger, 2014).

Finally, Smart Contracts (6) provide functionalities of the Web 3.0 tech while built on top the Ethereum blockchain network, this gives them an edge over Bitcoin scripting or other form of "smartness" in digital currencies thanks to Turing-completeness, value-awareness, blockchain-awareness and state. To better grasp the concept of a programmable blockchain we can use a definition provided by Gavin Wood, one of the project creators that describes Ethereum as *"a collection of non-localized singleton programmable data structures"* (Wood, What is ethereum? | Ethereum Frontier Guide, s.d.).

2.3 OUR PROJECT'S GOAL

Since Ethereum has been aired as a streamline tool to launch secured blockchain-based applications without the need of a different ledger, protocol or currency, the present work aims to evaluate the system and its platform for the deployment of specific use-cases examples smart contracts in relation to speed, costs and security. The approach to this work has not been easy: the technology's *momentum* in the last months has grown exponentially (along with its market value) and this has attracted many attentions from the outside world, some looking for information and knowledge, others seeking to defraud people or attack the blockchain itself causing quite a lot of confusion. More than this, the steps to understand the basics of the programming language are tied with a prior understanding of the structure and its components for everything to work together, along with its dependencies and constraints. Even if Ethereum Homestead is in the first stated production release (Ethereum Community, 2016), there are still a number of components that are difficult to integrate and use, more than often some workarounds are needed to secure a correct deployment and testing. We will point out that for testing and deployment a private test network (with its own miner node) have been set up to avoid real-chain use difficulties. Firstly because, as stated, the smart contracts deployment,

transactions and calls do have a gas cost, paid with real *ether*, secondly because to get ahead of security evaluations and procedures we need to see the *side effects* of our computation in an observable environment from which empirically evaluate Ethereum, that can be obtained only locally.
The structure of this work can be break down with the following roadmap:

- Setting up a local Ethereum private network and test node
- Search for non-trivial problems to adapt into Smart Contract code
- Develop specific use-cases for chosen problems
- Deploy Đapps on the testnet and evaluate their execution
- Security audit for both platform and applications

The security audit will make some considerations about the structure of the blockchain and will investigate the correct conditions upon which a Smart Contract can safely deliver its intended execution without unexpected result. However, as we will see execution correctness by itself cannot guarantee the safeness of smart contracts. A number of security issues in Ethereum SC have been unveiled while developing custom code outside the scope of simpler examples (Kevin Delmolino, 2016) and by performing static analysis of all the contracts that reside on the Ethereum Blockchain (Loi Luu D.-H. C., 2016). Some of these vulnerabilities have been patched after a major attack drained more than $60 *M* from the contract of the DAO in June 2016 (Siegel, 2016).
The assessment part covered in Chapter 1 has been a general study and introduction of the blockchain phenomenon, while Chapter 2 a more accurate presentation and analysis of the Ethereum platform. Chapter 3 will cover all the staging of a local Ethereum environment, the development phase with a technical showcase of the functioning Đapps and some of the coding guidelines that have been used and why. In Chapter 4 there will be a deep examination of the result given out by our coded Đapps: we will highlight the current tech limitations of blockchain, the security issues behind its language and smart contract and a cost/consumption evaluation of Ethereum blockchain use at its state-of-the-art. Following chapter 4 there will be a summary of the whole experience with our conclusions based on both the gathered result data and our understanding of this innovative technology.

3.0 DEVELOPMENT

This development chapter will focus the attention on a growing group of projects we have selected among the developed ones to make out as a technical showcase of Ethereum capacities in both what can be achieved with the environment and how it is coded with Solidity, plus providing an overview on the main security issues and difficulties encountered. The highlights provided in the next paragraphs will be the input for the next one in which the results, methods and limitations will be further analyzed. There are a number of different base implementations of the Ethereum protocol upon which clients do rely on when executing its environment. The main implementation projects available as of (Ethereum Community, 2016), ordered by usage and diffusion are:

- *go-ethereum*, developed in *Go* language, it is the official Ethereum implementation and is focused on the use with *Mist* client and Đapps development, it also has a security audit for smart contracts.

- *Parity,* developed in *Rust* language by the *Ethcore*[21] it is both an Ethereum client and a Đapps-enabled browser.

- *cpp-ethereum,* developed in *C++,* best suited for miner nodes (currently the only one that supports GPU-mining), *IoT* and also smart contracts development.

- *pyethereum,* developed in *Python,* it implements the Ethereum cryptoeconomic state machine that aims at providing an easily hackable and extendable codebase.

- *ethereumj,* a pure-*Java* implementation provided as a library that can be embedded in any Java or *Scala* project to provide full support for Ethereum protocol and sub-services. It also supports CPU mining and the project is sponsored by *<ether.camp>*[22].

- *ruby-ethereum,* a *Ruby*-based implementation of the Ethereum Virtual Machine developed by Jan Xie[23].

Every one of these implementations follows the paradigm described in the *Ethereum whitepaper* (Buterin, Ethereum White Paper, 2014) and the protocol specified in the *Ethereum*

[21] A blockchain development startup started by one of Ethereum's original founder *Gavin Wood*
[22] *http://www.ether.camp/*
[23] *https://github.com/janx/*

yellowpaper (Wood, Ethereum: a secure decentralised generalised transaction ledger, 2014). All of them share the Ethereum Virtual Machine code, which is surprisingly simple: when the EVM is running, its full computational state can be defined by the tuple (*block_state, transaction, message, code, memory, stack, pc, gas*), where *block_state* is the global state containing all accounts and includes balances and storage. At the beginning of every execution round, the current instruction is found by taking the pc^{th} (*program counter*) byte of code and each instruction has its own definition in terms of how it affects the tuple. There are of course different ways to optimize the EVM execution via just-in-time compilation but a basic implementation of Ethereum can be developed in a few hundred lines of code (Triantafyllidis, 2016).

Each low-level operation executed by the EVM has a Gas cost in units of gas defined by a specific formula defined as: *full_memory_gas_cost = 3 * W + floor(W*W / 512),* the design choices for this formula are explained in the *yellowpaper* and a complete cost is listed in an online public spreadsheet[24] (Foundation, s.d.). The total *fee* of transactions or executions must then be calculated by multiplying the gas unit cost with the gas price cost and when a user submits a new transaction, he has to specify a fee that intends to send over. Many users use the default gas price from their wallet client when they make a transaction, this is generally the right way to proceed. However, it sometimes make sense to pay more if you want to assign a higher priority to the transaction: a higher fee might result in a faster mining operation while a lower fee is preferred for non-critical transaction or in order to save some money, especially if time is not required by the process. There are dedicated web services[25] that give a quick overview of the gas situation across the Ethereum blockchain and help to keep track of the related statistics.

As we mentioned earlier all entities in Ethereum environment are associated with an univocal addressable account, referred to by its 160-bit or 40 hexadecimal character long public key (e.g. *0xB465E96404611e85A79b3c4c5Af9C18bfD7b144c*).

This design works perfectly for the execution machine, but it is not very user-friendly, in that a human will have a difficult time in remembering the addresses of all interested parties. A useful service[26] has surfaced to counter this problem and provide an associated *name.eth* that allows users to register names that resolves into addresses using an auction process. However, the concept of unique address stands: when a new account is created on the blockchain the registrar

[24] *https://goo.gl/5mfkJC*
[25] Like *http://ethgasstation.info/*
[26] ENS – Ethereum Name Service

contract compiles the address after its creation and hard-code it in the ledger, this information cannot be changed ever. This feature provide uniqueness even for Smart Contracts registration: while anyone can deploy the same contract multiple times and interact with several of its versions, the value of a contract is defined by its usage across the network.

Since smart contracts operate as state machines, they could have certain stages in which they behave differently or in which different functions can be called. During development this can lead to frequent mistakes and errors made while encoding such states, one of which could be money leaking in contracts corner cases. Some fallback or defensive computation should always be kept in mind when designing smart contracts. Usually the contract's functions are responsible to transition a contract through its stages but is also common that some stages are automatically reached at a certain point in time.

3.1 THE SOLIDITY LANGUAGE

Smart Contracts in Ethereum are written with one of the specialized contract specification languages, there are three of them: *Solidity*, which resembles *JavaScript*, *Serpent* more close to *Python* and *LLL* that resembles *LISP*. Solidity however is the official language of the Ethereum Project and is suggested as the main language in the guidelines. It is an *Object Oriented* language where the internal definition of *contract* is very close to classes, a contract can have different features that we will quickly summarize (Ethereum Community, 2016):

- **Types**: Solidity supports a number of different data types but they have to be known at compile-time since the language is statically typed. The language supports Booleans, integers (signed or unsigned of 8 up to 256 bits) and fixed-size byte arrays. Strings can be used in the form of dynamically-sized byte array but are not a value type and there is no support for floating point variables as of yet. Another very interesting data type is the Ethereum *address*, it holds the 20 byte representation of an Ethereum account address and also have internal predefined members to check the balance or transfer Ether via a contract, as well as to call functions from other contracts. Solidity also supports *structs*, *enumerations* and *mappings* which are in essence key-value stores that map keys of any data type to values of any data type as well.

- **State Variables**: classic *variables* and values that will be permanently stored in the

contract internal storage. Variables can be of different Types and are subject to scope and visibility like in any other language.

- **Functions**: they define the executable units of code within the contract and are distinguished in two types of functions: constant and transactional. Constant functions have the sole purpose to return a value and cannot update the state of the contract (or of the blockchain), in a way we could define them as without any *side-effect* with exception of the returned value. They can be called directly and do not consume gas since they do not modify the blockchain. Transactional functions are used instead to obtain computation that will modify the state of the contract and, when called, an amount of gas has to be supplied to cover the transactional costs. There are four levels of visibility for Solidity functions:

  • *External*: functions part of the contract specification (interface), therefore they can be called by other contracts, but are not accessible by the contract itself. External calls are carried out via message call and they are susceptible to errors that could raise exceptions.

  • *Public*: functions that can be called by the contract itself internally or by any external contract or entity via message.

  • *Internal*: functions that can only be accessed by the contract itself and its derivative (inherited) contracts.

  • *Private*: Private functions are visible only to the contract itself and cannot be called by any external entity or derivative contract.

- **Function Modifiers**: they are constructs used to change the behavior of a specific function. They are mainly used to check if a given condition is satisfied before a function can be executed. Modifiers are inheritable properties of contracts, each function can belong to multiple modifiers and they can be overridden by derived contracts.

- **Events**: Events are the way for Solidity to provide information in the "outside world" of a smart contracts. They make use of EVM transaction logs, a special data structure in the Blockchain that can be used to make JavaScript callbacks interact with it in a user-side interface of a Đapp. Functions can emit these events populated with return

values, and event messages will be broadcast and stored on the blockchain. Event messages are not accessible from within contracts not even by the contracts that created them.

All the operations performed by Solidity on its variables and data have access to two types of memories in which manipulate or store data:

- *Memory*: an "infinitely" expandable and non-persistent linear byte array that is initialized to a new instance every time the contract receive a message call. Every new word (256-bit) of requested memory has a gas price that must be paid, its cost scales quadratically the larger it grows.

- *Storage*: a key-value store that maps 256-bit words to 256-bit words. Unlike memory, which reset after computation ends, storage is persistent in the long term but it cannot be enumerated. Storage operations like read or modify are more costly than their *memory* counterpart is, and a contract has only access to its own storage space.

Furthermore, contracts can inherit from other contracts and they can call code that resides in other SC on the blockchain. However every time a contract makes a message call the triggered code is executed in *his* environment using his memory space, moreover the caller has to pay for all the gas costs that will arise from the execution of the called contract. The code can also access the *value, sender* and *data* of the incoming message (the sender account), as well as block header data from its executing node. The code can also return a single value or a static-sized byte array of data as an output (Ethereum Community, 2016).
The Solidity structure similarities with a typed-language like JavaScript gives the false impression to a user that design and implementation can be similar, on the contrary Solidity implements its features differently thus causing code writing errors. This uncomfortable process can lead to a misalignment between the semantics of the language and the intuition of a programmer. The Ethereum programming language also lacks the appropriate constructs to deal with the fact that its code will be stored on a public blockchain, therefore the computational steps could be unpredictably reordered or delayed. Finally, while some bad habits and programming issues have been listed in the official documentation (Ethereum Community, 2016), the platform has a shortfall over a complete and exhaustive security overview, a developer has often to look up for details or answers online in research papers (Nicola Atzei,

2016) or discussion rooms (among the others *Gitter*, *Slack* and *Reddit*). A more precise and formal documentation on Solidity security would be needed.

3.2 SETTING THE ENVIRONMENT

In order to use an Ethereum environment we first need to download and install one of its clients. The client of choice for this work has been the main platform implementation, written in *Go* language and called *Geth*, which is the most maintained. Our version of the Geth client is *v1.6.5-stable* for *Windows* while the Go environment used is *go1.8.3* . In order to initialize a new private blockchain we need a special *Genesis Block* which is different from Ethereum's first block, that will be statically created and put on the chain. The properties and values of this block must be written into a *.json* file that will set the initial parameters of our blockchain network. After some initial testing we created our test network with this configuration:

```
{
  "config": {
    "chainId": 21,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "difficulty": "200000000",
  "gasLimit": "2100000",
  "alloc": {
    "0f6b7d05ece4916e6193129942091ce9a07c3009": { "balance": "400000" },
    "7Eb94c165f4Cb5986b97c05530bbd7667d94ADe0": { "balance": "250000" }
  }
}
```

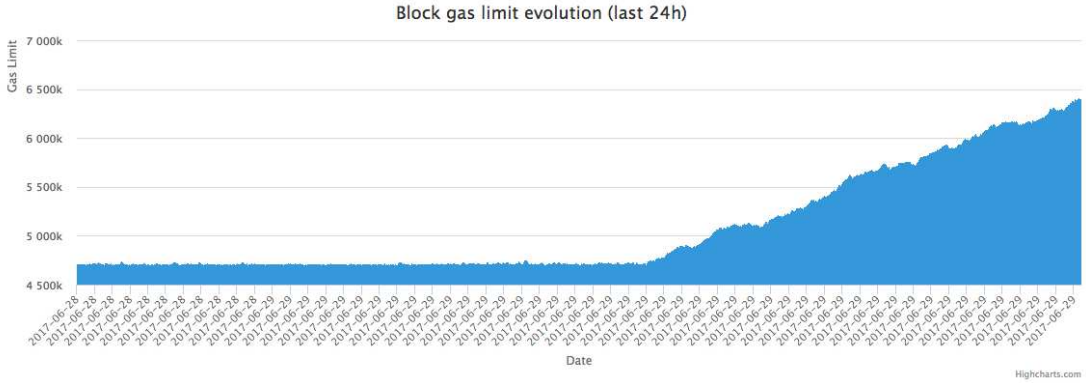
With the following parameters:

- *chainId*: this value is used to separate the private nodes network from the rest of the Ethereum's network. Connection between nodes are valid only if peers have both identical protocol version and network ID, therefore settings a value different than *1* (used for Ethereum *MainNetwork*) will guarantee the singularity of the network.

- *difficulty*: a scalar value that is applied during the calculation of this block, it also defines the mining difficulty target which will be calculated after the first block and is obtained from the previous block's difficulty level and the timestamp. The value impacts directly on the block generation frequency and on our test net is kept low and constant to favor a linear block

generation rate. In the real network this value is dynamically adjusted so that the block generation is set on an average of 12 seconds.

- *gasLimit:* a scalar value that defines the hard-cap of Gas expenditure per single block for all the nodes in the network. In order to be able to study the results from local smart contracts execution we keep this value high so we can "push" our application with more performance. However, we will point out that until 29 of June the gas limit for the *Main Network* was about 4.7 Millions, after major delays and network issues caused by a huge quantity of transactions the limit has been adjusted to ~6.3 Millions, therefore increasing the total transaction capacity of the network (Higgins, Miners Boost Ethereum's Transaction Capacity with Gas limit increase, 2017).



Picture 4 - Block gas limit increase on 29th of June

- *alloc*: it is used to define one or more pre-filled wallet accounts. This is an Ethereum specific functionality that is usually deployed to handle the "Ethereum pre-sale" phase period. We will use it here in order to get two accounts with some basic funds out of the system.

We could specify other properties and attributes in the *genesis* file but they are out of the scope of this work and this setup is more than enough to run our tests Đapps.

The next step is to initialize the network with a command that will take in our genesis file and a local path to store the future blockchain that will be created. Once the client has completed the creation of the genesis block and of the basic backend structure it is ready to be executed with the local command to start the node client:

```
geth.exe --datadir path\to\blockchain\folder --networkid 21 --cache 1024 --nodiscover
```

Where *cache* option specifies a custom quantity of memory allocated for the internal caching operation in order to increase efficiency (the default would be 128) and the *nodiscover* disables

the automatic *peer* discovery and addition feature, we want to make sure that we do not connect to the public blockchain by mistake.
Once the network and node are up we can use a terminal and a Geth instance to attach to the client and use all the needed commands, we can attach as many consoles as we want while other processes work through the node.
Once attached we are able to make the following steps (plus other operations):

- Definition of a *coinbase* account needed for mining operations (we can either define one of the two already-created accounts or create a new one via console).

- Use *miner.start()/stop()* to begin the mining process. While CPU is drained, the coinbase account will be rewarded with ETH every time a new block is minted (every few seconds of computation).

- Get basic information on the node or on the accounts within the blockchain, we can query the structure to ask for *balances* or prompt transactions and calls from one account to the other (assuming we have all the keys and password associated with the specific sender account). Transactions follow a precise definition.

We can start other nodes as well on the network but they require individual manual configuration in order to discover each other since they are not using Ethereum default discovery protocol, another solution for setting up a large set of private nodes could be a *bootstrapper* node. As the number of nodes (and eventually miner nodes) raises however there are some technical difficulties implied in the management of the network: too small difficulty in the genesis block could lead miners working on their own chain without the physical time to pair with each other therefore generating stale chains that will eventually breaking the network's functionality.

3.3 TECH SHOWCASE

Our focus now is to develop some examples and show that even if complex solutions can be achieved with relative speed, evaluating the code correctness and safeness against bugs and malicious attacks is way much harder. To test the result output of our Đapps *correctness* against their design, we will deploy the code to our local blockchain test net, the code snippets in the document sometimes do omit unnecessary or repeated code:

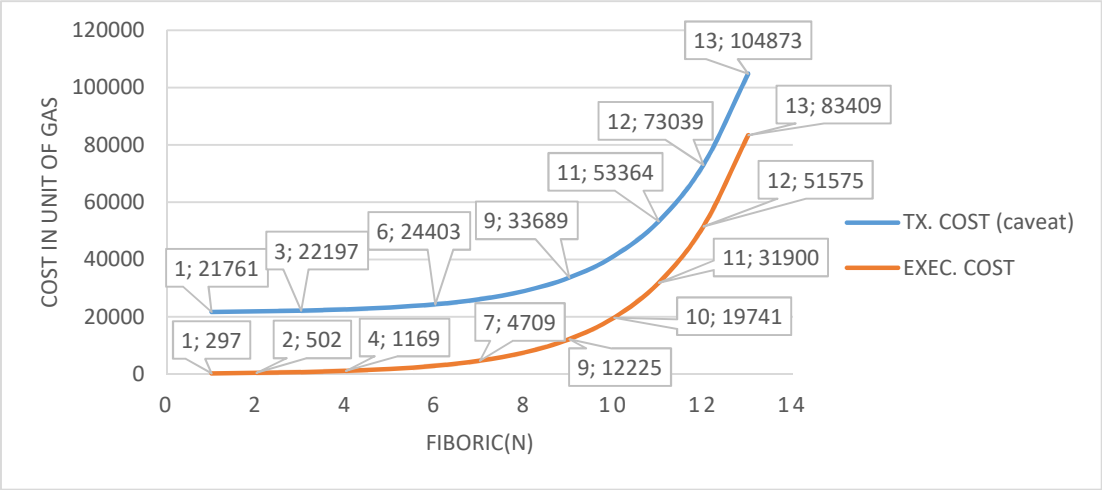
3.3.1 FIBONACCI

As a base case for Solidity programming, we coded a Fibonacci Smart Contract to observe the bare computational power that the platform can achieve with the execution of a heavy computation:

```
01 contract Fibonacci
02 {
03     function fiboRic(uint number) constant returns(uint result)
04     {
05            if (number == 0) return 0;
06            else if (number == 1) return 1;
07            else return Fibonacci.fiboRic(number - 1) +
08                        Fibonacci.fiboRic(number - 2);
09     }
10 }
```

Code Snippet 1 - Recursive Fibonacci

This simple case that shows the *recursion* features of Solidity is probably one of the worse way to implement a Fibonacci sequence but it gives us the opportunity to analyze the function's chained call and its results. Here we have to think in terms of *transaction* and *execution costs;* the idea is that every operation performed by the *stack machine* (EVM) has a unique cost that must be eventually summed up with the transaction cost from the length of the transaction, both expressed in units of gas. When a user wants to invoke a smart contracts execution he must supply enough gas to cover all of that cost multiplied for the actual *gas price* value (expressed in *wei*). Starting from a value of *number = 1* we observed the results of our computation and depicted them in the following chart:



Picture 5 - Gas cost for Recursive Fibonacci

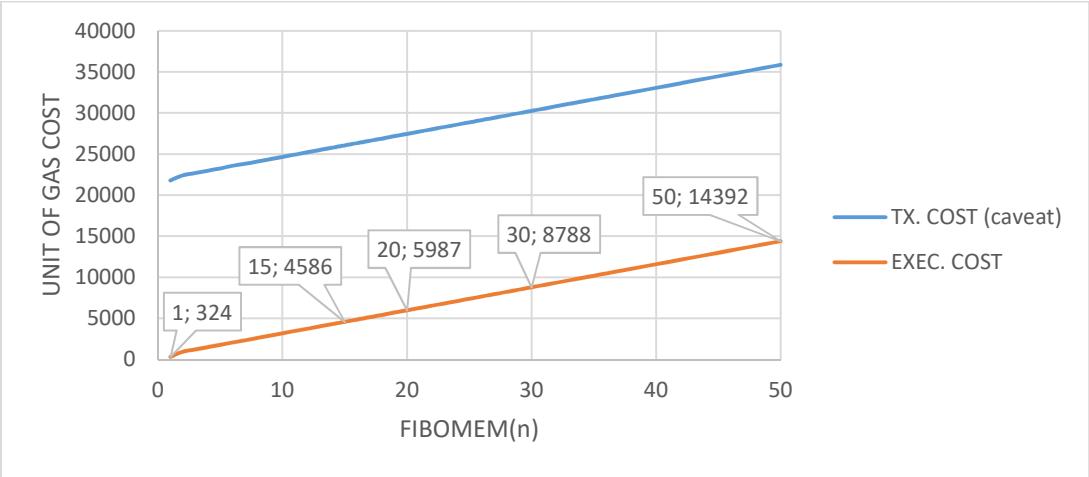
As we can see, the harder the computation becomes the higher our execution fees raises almost doubling at every new step. For a value of *number = 13* the execution times becomes very long and the lag is physically visible, while for 14 our computation is discarded, probably after an *out-of-gas* exception is raised. Because *fiboRic* function is *constant*, it is expected not to modify the chain state and we do not need an actual transaction to trigger its execution. We used a JSON-RPC *eth_call* which is a specialized function that executes a new message call without transacting on the blockchain, indeed it is expected that this execution would not consume any gas at all. However, to prevent an idle scenario a small fallback quantity of gas (defined *stipend*) is kept inside a contract that is used to trigger its *constant* activations, this gas is spent if no other gas is provided in the message call. We could of course manually provide more gas for the execution but we have to keep in mind that there is an upper limit for total gas expenditure in a single Block when it has to be validated and that total amount is the sum of *all* transactions currently candidate to be validated by that node. Again it is a tradeoff between how much we want to invest on this execution, averaged between other users' gas bets and total costs. This example illustrates the importance that *gas* measurement must have during the design phase of our smart contract, that said, on our private test-net we can have more resources than the *Main Net* would allow us to use.

A more intelligent solution for the Fibonacci problem is the following:

```
01 contract Fibonacci
02 {
03     function fiboMem(uint256 number) constant returns(uint256 result)
04     {
05         if (number < 2) return number;
06 
07         uint256[] memory fib = new uint256[](number+1);
08         fib[0] = 0;
09         fib[1] = 1;
10         for (uint256 i = 2; i <= number; i++)
11         {
12             fib[i] = fib[i-1] + fib[i-2];
13         }
14         return fib[number];
15     }
16 }
```

Code Snippet 2 - Memoized Fibonacci

This code provides a *memoized*[27] version of the problem that levers on the use of arrays and an iteration to store already computed results. Again this solution may seem harmless but we have to think at our operations cost and constraints: with increased performance we can easily compute a higher Fibonacci number and its cost will be relatively low compared to the previous solution growing at a slow linear rate:



Picture 6 - Gas cost for Memoized Fibonacci

However, for a sufficient high value of *number* this computation will inevitably lead to the *unsigned integer* overflow for Solidity language, this issue will not be detected anyhow by the program, easily breaking up its design and functioning if no checks are made.

[27] *An optimization technique that stores the results of expensive function calls and then uses them when the same input happens again during execution.*

3.3.2 RANDOM GENERATION

A more challenging and critical problem to develop inside the Ethereum network is the safe use and generation of random numbers in any fashion. From the first release of smart contracts a number of research and businesses have performed studies on the subject, since Ethereum involves tokens and therefore money transfers, a significant effort has been made by backers of online gambling and similar projects to find a reliable and safe solution. The main issue here is that the blockchain network is a *deterministic* environment: its nodes, EVMs and smart contracts all rely on a *consensus protocol* that favors a natural synchronization between peers and, since none of those components can access the external world it is very difficult to find sources of *randomness* capable of increasing the system's entropy. To be able to simulate non-deterministic choices, many smart contracts that need this feature generates pseudo-random numbers with their initialization seed chosen uniquely for all miners.
A first example of naïve random generation is the following:

```
01 contract Random
02 {
03     uint256 FACTOR = <integer max number>;
04
05     function randStatic() public constant returns (uint256)
06     {
07         uint256 lastBlockNumber = block.number - 1;
08         uint256 hashVal = uint256(block.blockhash(lastBlockNumber));
09
10         return uint256(uint256(hashVal) / FACTOR) + 1;
11     }
12 }
```

Code Snippet 3 – static Random generation

This contract, which gives access to a random number, uses the hash of the last validated block as seed, then divided for a *factor* that is equal to the max value of unsigned integers in order to produce a result that is between 0 and 100. This example is problematic because even if the content of a future last block cannot be predicted, for a time of at least ~12 seconds (average mine time on network) any call to this contract produces the *same* output value for that updated node in the network, providing a very poor result. A different situation could be obtained with the use of *block.timestamp* object that provides a time representation snapshot in seconds since its *Unix epoch*[28]. However even this single solution suffers from the time-window problem that can occur between nodes with same timestamps: because the Ethereum nodes tries to

[28] *Also know as POSIX time, starts from 00:00:00 UTC of January 1 1970, follows ISO 8601 data format.*

synchronize, their block timestamps are related to their system clocks at the moment of mining. In order to resolve slightly different timing issues, the protocol tolerates an amount of discrepancy between timestamps of a few seconds. This is beneficial to our random generation, but still not enough to guarantee a good measure of randomness.
We then defined another version of the random generator:

```
01 contract Random
02 {
03            function rand(uint seed) constant returns (uint randomNumber)
04            {
05                  return(uint(sha3(block.blockhash(block.number-10),
                          seed ))%100);
06            }
07
08            function timeRand(uint seed) constant returns (uint randomNumber)
09            {
10                  return(uint(sha3(block.timestamp, seed ))%100);
11            }
12
13            function multiBlockRand(uint seed, uint size) constant
                        returns (uint randomNumber)
14            {
15                  uint number = 0;
16                  for (uint i = 0; i < size; i++)
17                  {
18                        if (uint(sha3(block.blockhash(block.number-i-1),
                                seed ))%2==0)
19                                    number += 2**i;
20                  }
21                  return number;
22            }
23 }
```

Code Snippet 4 – complex Random generation

This second solution implements three different random functions: all of them have been updated with a *sha3* call that computes the *Ethereum-SHA-3* (*Keccak-256*) hash of the provided arguments. The first one at line *03* make use of both a blockhash and a user-provided seed to compute a hash that will generate a number in the 0 – 100 interval. The second one at line *08* is pretty similar but is provided with a timestamp instead of a blockhash. The last one at line *13* is a bit more complex: what we are doing here is using a seed with an iterative calculation of a (provided) number of previous blocks, the general idea is to thwart a possible attacker by providing a set of blockhashes instead of a single one during the computation. The operation carried out in the looping *for* produces a number between 0 and 2^n (defined by *size*) and can be seen as a computation that will halve the possibilities for an attacker to influence the random generation at every iteration.

There have been different considerations for the use of random generators inside Ethereum network (Loi Luu D.-H. C., 2016), one of the major concern we want to point out is that even with the use of *salt* and cryptographic functions *inside* smart contracts, we still need to consider the fact that all data sent to the blockchain is completely *public* by default. In order to make good use of a random engine, which takes data or a seed from a user, we need to secure the communication client-side or a skilled attacker might just listen to the message and use its content to make himself a random request and (possibly) obtain the same number.

Finally when it comes to random generation, we have to carefully inspect the role that miners will have in our model structure. A miner has *always* the final word over the block generation if he is validating our smart contract execution; a malicious miner can see the results of the random computation *before* anyone else and could therefore decide to *discard* the block if the obtained result is not favorable to him. The worst-case scenario is that a miner could try to *forge* his own block to purposefully bias the result of the number generation; it has been shown that if the costs to carry out such attacks balance the profit accordingly, there is no need for lots of resources (Cécile Pierrot, 2016). This kind of bad influence can lead to security issues and fraud if an organized party, trying to secure a gamble or winning a game, deploys this kind of attack. The player might raise its stakes knowing that the miner won't accept execution blocks that are not favorable to them.
Some alternative solutions have been proposed for this problem involving time-commitment protocols (Marcin Andrychowicz S. D., 2014), they are based on secrets communicated by participants and sent over in a hashed version, to guarantee for the safety of this protocol every user pays a fee on the secret deposit operation. Later on the (pseudo) random generation is achieved by the combination of all the provided secrets, if a malicious participant chose not to reveal his own then he loses his deposit fee. Again, the attacker has to consider his own tradeoff between costs and profit in order to carry out an attack. Examples of complex random generation are the *RANDAO Đapp* (a DAO working as RNG of Ethereum – *https://github.com/randao/randao*), while a game based on random generation is the *MAker DART (*a random number generating game*).*

3.3.3 RUBIXI / DYNAMIC PYRAMID

Being smart contracts both a technological and economic innovation the difficulties to design and code applications are subject to technicalities of both worlds. Therefore during the modeling phase not only we have to follow the correct specification of the Solidity language, but also be sure that there are no pitfalls in the logical process of transactions, activations and payments. *DynamicPyramid* is a smart contracts that implements its own version of a *Ponzi Scheme*[29] that is designed to make participants gain money from the high investments made by newcomer subscribers, attracted to the application by promised high-revenues in a small, mid, or long term scenario. A dynamic pyramid always follows a similar scheme deployed in different fashion: this example is of course trivial to identify but there could be systems that are initially used in an honest way to attract people (even paying them out) and then subverted into fraudulent execution. The owner of *DynamicPyramid* contract has also the ability to collect some of the fees sent by subscribers after their association. After a first deploy the developers updated the code of the contract and renamed it to *Rubixi*, the following is just a fragment of the complete code:

```
01 contract Rubixi
02 {
03      ...
04      address private creator;
05
06      function DynamicPyramid() { creator = msg.sender; }
07
08       modifier onlyowner { if (msg.sender == creator) _; }
09
10       function collectAllFees() onlyowner
11       {
12            if (collectedFees == 0) throw;
13
14            creator.send(collectedFees);
15            collectedFees = 0;
16      }
17       ...
18 }
```

Code Snippet 5 – Rubixi

The developers did update the code but forgot to rename the contract's *constructor* at line *06*. A constructor is executed only once during deployment and here it sets the owner's address of the contract; however, a constructor is required to have a function name *equal* to the contract's

[29] *A fraudolent investment operation named after the famous Italian swindler Carlo Ponzi*

name. By leaving this code as depicted anyone has been able to become a temporary owner of *Rubixi* by just calling the *DynamicPyramid()* function, this has led to a number of people trying to race and exploit the contract's maliciousness to drain funds from victims until the name of the contract became famous.

This type of problem has been classified as "immutable bugs" (Nicola Atzei, 2016) and more generally refers to the *immutability* feature of a blockchain itself. Once a smart contract bytecode has been deployed into the network it is impossible to update: there are means to create some sort of extendibility (obtained with libraries and reference to other contracts) but nothing can actually be changed without re-uploading a newer version of the contract. Moreover every deploy comes with a static address definition that *cannot be* reused, created once on deploy. A user of that specific contract needs to be informed of a newer version by other means or tools and has to update his private list of Đapps with the new coordinates in order to find the contract on the blockchain. This leads to a maintenance and patch issues that cannot be overcome by quick fixes: if a serious bug or problem is found the contract should have a safe and designed method to be disabled because there is nothing provided by the language to do so (Bill Marino, 2016). It is possible to *kill* the contract and prepare the new one with speed (if the service can afford the related downtime): a contract can be destroyed with the *selfdestruct(<recipient address>)* invocation, all his funds will then be transferred to the specified account. This functionality might seem useful but has to be encoded first and will disable permanently the contract's address, leaving up any party involved in the use of that contract with the risk of losing all the eth sent forever and without notice since a transaction to an orphan address cannot be distinguished from another one (Ethereum Community, 2016). In any case the contract code will remain on the chain for the time being as garbage.

3.3.4 PAYMENTS AND TRANSACTIONS

At the heart of the Ethereum environment and smart code usage related to finance are the transactions. Payments and fund transfer patterns are of paramount importance because they are susceptible to security attacks that can drain funds or disable some (or all) the contracts functionality. The following snippet contains the code for a savings Đapp that implements both a simple registration system and the savings implementation.
In this example, the contract *owner* has an administration role and can subscribe users to the system (a registered user becomes a *client*) granting them access to its functionalities. Once registered a client can:

- Deposit some funds
- Get his savings balance
- Withdraw an amount of his funds.

When the eth is sent to the contract via *depositFunds()* at line *31* the client-balance mapping is updated with the amount deposited. However all the eth sent to the contract is kept within its account balance; the information recorded on the mapping is simply the personal amount. A client can query its balance with the *getBalance()* function at line *38*, this property could be created *public* if we want a client to *always* visualize its balance without asking the contract. Finally at line *44* we have *withdrawFunds()* that is used to retrieve the correct amount of ether from the contract's balance once the availability is confirmed.

```
03 contract SavingsContract
04 {
05     mapping (address => uint) clientFunds;
06     mapping (address => bool) clientStatus;
07     address owner;
08
09     event UpdateStatus(string message);
10     event UserStatus(string message, address user, uint amount);
11
12     function SavingsContract()
13     {
14         owner = msg.sender;
15     }
16
17     function addNewClient(address client)
18     {
19         if(msg.sender != owner) throw;
20
21         clientFunds[client] = 0;
22         clientStatus[client] = true;
23     }
24
25     modifier ifClient()
26     {
27         if(clientStatus[msg.sender] != true) throw;
28         _;
29     }
30
31     function depositFunds() ifClient payable returns(bool success)
32     {
33         clientFunds[msg.sender] = msg.value;
34         UserStatus('User has deposited money', msg.sender,
                           msg.value);
35         return true;
36     }
37
38     function getBalance() ifClient returns(uint balance)
39     {
40         UpdateStatus('Someone called a getter');
41         return clientFunds[msg.sender];
42     }
43
44     function witdrawFunds(uint amount) ifClient
45     {
46         if(amount <= clientFunds[msg.sender])
47         {
48             clientFunds[msg.sender] -= amount;
49             msg.sender.transfer(amount);
50             UpdateStatus('User transferred money');
51         }
52         else
53         {
54             UpdateStatus('Requested amount too large');
55         }
56     }
57 }
```

Code Snippet 6 – Savings Wallet

The withdraw function is usually a very important section of any value transaction-based Đapp; in order to comply with safety standards the operations coded here have been designed as stated by the "Checks-Effects-Interactions pattern" described in the official documentation (Ethereum Community, 2016).

The *checks* part is done first and is related to verification of some pre-conditions that must be met before proceeding with the execution: namely the *ifClient modifier* at line *44* that checks if the caller of the function is actually registered and the *if* at line *46* (a preemptive check to see if the amount requested for withdrawal is lower or equal than the available) is true.

The *effects* section is accessed after the various checks have been done and it comprises the changing of contract's state variables. These reflect the internal state of the smart contract and should always be consistent during any intermediate change with no intervention or interruption from external sources. In our code this part is carried out by line *48* which updates the new value of a client's balance.

Lastly we can instantiate and use *interactions* thanks to the fact that we modified all our parameters in a safely fashion: the call *msg.sender.transfer(amount)* at line *49* executes a transaction that will transfer the ether from the contract's address to the client's. If, for any reason the transactions fails to deliver the eth, an exception will be thrown back. In Solidity however, a thrown exception cannot be caught: what happens is that the execution stops, the gas fee is lost and all the previously produced side effects (including the ether transfers) are reverted. The previous design pattern used in our code is a meant to avoid security issues like *reentrancy* and *call to the unknown* (Ethereum Community, 2016) that could arise during execution and have been problematic since the inception of smart contracts. Both this security problems have been examined in depth by a number of authors like (Nicola Atzei, 2016) and can be described as follows:

- *Reentrancy:* it involves the apparent atomicity and sequentiality that transactions may seem to possess. In reality, what happens is that an attacker could re-enter a caller function thanks to the behaviour of the *fallback* [30] function. The immediate consequences of such an attack is an unexpected invocation loop that will terminate either only reaching the EVM's stack limit or after consuming all the gas, preventing further execution. Moreover, if a *transaction* is generated inside the one-time attacked

[30] A special function with no name and no arguments that can be arbitrarily programmed. The fallback function is either executed when a function invocation doesn't match any signature or also when the contract is passed an empty signature.

function the malicious user could trick the contract into multiple execution of the same lines thus generating multiple transactions that will quickly drain all funds from the contract's account. Reentrancy must be checked especially on complex implementations such where contracts interact with other contracts or external resources, this situation poses a higher security threat because of the way exceptions are handled: when a function is directly called (like in our code snippet) if an exception is thrown it is capable of reverting every side effect produced until its occurrence. In case of any external *call()*, *delegatecall()* or *transaction.send()* the exception is propagated along the chain inside called contracts, reverting every side effect *until* a subsequent call is found. From that point the code resumes execution but depending on how the *call* has been done, its return Boolean may or may not be propagated back to the caller (Loi Luu D.-H. C., 2016).

- *Call to the unknown:* the problem involved here is related to the fact that when a *call* is performed (on the contract itself or another one) its signature is matched against the definition of all the functions in the contract's interface. If no match is found for the signature or if we are executing a *transfer* operation then the *fallback function* of the targeted contract is executed instead thus leading into the execution of unexpected code. The fallback function could or could not have been implemented by developers: in order to engineer this limit into an attack a party can develop a smart contract which relies on malicious code put inside the fallback function. Then, after the upload, if the party manages the victim smart contract to make use of the malicious one they are able to execute foreign code inside the environment of the targeted Đapp with obvious consequences. This vulnerability has been spotted even in few other cases such as type cast or state operations (Nicola Atzei, 2016).

The infamous DAO attack (Siegel, 2016) has been carried out exploiting these two security vulnerabilities. After the painful situation was resolved a number of corrections have been made to the language, with introduction of new security patterns, however not every smart contract follows a disciplined approach and sometimes not all solutions can be coded with that pattern, resulting in a vast number of contracts being at large still vulnerable to similar or other security issues (Loi Luu D.-H. C., 2016).

3.3.5 ENERGY TRADE

Finally, in order to provide an example of the language and platform full-fledged capabilities for making a decentralized reality using a token-based project we lay out the code of an energy production and sale market system called *EnerTrade*. The scope of this project is to give means to a private energy producer to sell is own energy power into a market that will automatically award him with a fee based upon an updated power price rate that can change dynamically. The unit used to measure energy power output is kWh (kilowatt – hour).

The features provided by *EnerTrade* are:

- Energy selling for the producer, while being connected to an Ethereum node, he can issue transactions for each kWh produced and get a proportionate payout based on the last updated price rate and the energy amount.
- Any consumer user can also purchase some kWh at the updated price rate.
- The price rate can vary based on multiple factors that are externally computed, however the user buying and selling rate could influence this factor, therefore the contract has a function that returns the total amount of energy traded per user. This concept could be extended with a collector smart contract taking the results and aggregating them on the blockchain for subsequent external reading.
- The contract will expose an up-to-date price rate for any convenience.

The rewards and fees are coded into a custom token system called *EnerCoins*¸ that has its own definition in a separate dedicated smart contract. Since the energy price can change due to demand and offer in the external market its value is obtained from an external web service that provides a simple *.xml* which always has the updated price listed. This feature has been integrated into our *EnerTrade Đapp* thanks to the use of an external service called *Oraclize*, its API enables us to make “queries” out of the Ethereum environment and return simple results in a safe and verifiable fashion with no side-effects. Moreover, with the use of a custom token there is no need for *payable* Ethereum transactions since no actual ether is moved (aside from transaction fees), the business logic related with compensations works directly within the code and follows the rules defined in the coin’s smart contract.

```
01 import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";
02
03 contract EnerTrade is usingOraclize
04 {
05     uint public kWh_price;
06     mapping (address => uint) energyBalance;
07     mapping (address => uint) enerCoinBalance;
08     address owner;
09
10     event newOraclizeQuery(string description);
11     event newEnergyRating(string price);
12
13     function EnerTrade()
14     {
15         owner = msg.sender;
16         updatePriceRate();
17     }
18
19     function __callback(bytes32 myid, string result)
20     {
21         if (msg.sender != oraclize_cbAddress()) throw;
22         newEnergyRating(result);
23         kWh_price = parseInt(result, 2);
24     }
25
26     function updatePriceRate() payable
27     {
28         newOraclizeQuery("kWh price update ongoing, stand by..");
29         oraclize_query("URL",
               "xml(https://www.enertrade.com/rest/ratePrices).rate.kwh");
30     }
31
32     function sellEnergy(uint kwh) public
33     {
35         coinBalance[msg.sender] += (kwh * kWh_price);
36     }
37
38     function buyEnergy(uint coin)
39     {
40         if (coin <= enerCoinBalance[msg.sender])
41         {
43             coinBalance[msg.sender] -= coin;
44             energyBalance[msg.sender] += (coin / kWh_price);
45         }
46     }
47
48     function getEnergyBalance() constant returns (uint kwh)
49     {
50         return energyBalance[msg.sender];
51     }
52
53     function getCoinBalace() constant returns (uint coin)
54     {
55         return enerCoinBalance[msg.sender];
56     }
57
58     function updateCurrentRate() { updatePriceRate(); }
62 }
```

Code Snippet 7 - EnerTrade

All the features of the *EnerTrade Đapp* are coded into this snippet with the power selling being handled in line *32* in function *sellEnergy*, the buying at line *38* in *buyEnergy* and the Oraclize functions in *19* and *26*. We have to point out that since the Oraclize queries and updates are *asynchronous* the Đapp cannot ensure a precise updated price during an operation. This lack can be adjusted by using an "updater" feature of the Oraclize API that enables us to specify a time frequency for the *oraclized answer* to be sent to our contract, that way for example, if we specify a parameter of 60 seconds the price would be updated in a timed fashion. In doing so we have to consider the relative *gas* expenditure since it is our contract that has to provide the right amount of gas to Oraclize to cover the price for every update to be sent back.
The *EnerTrade Đapp* can be extended with some features that could increase its core service value in being a reliable service: a subscription system could be integrated in order to make clients register first to the platform and then give the ability for them to interact with selling and buying features. A registration could benefit even a data collector smart contracts that thanks to registrations could provide trading information to the EnerTrade provider party or to an external market that could in turn provide better price rates based on the given feedback.

This example illustrates the high capacity and seamless integration features that an Ethereum project can have while being distributed and easily deployed. The EnerTrade producer user could be a private owner of solar arrays connected with a simple Raspberry Pi that maintains a light Ethereum client capable of making queries to the blockchain environment. The user has the ability to choose between making manual transactions to EnerTrade basing his decisions on its own mind or could *code* a smart contract capable of evaluating the hourly solar production rate comparing it with actual price rates and make it sell energy on his behalf. With the support of off-chain software this behaviour could be tuned into automation (since smart contracts cannot auto-execute). This concept of distributed user-end capacity for doing businesses with a wide degree of freedom and customization is what inspired the work of the Ehereum foundation and represents the ethos of the project itself (Buterin, Ethereum White Paper, 2014).

3.4 BLOCKCHAIN DEPLOYMENT

Each coding phase has been followed by a deployment on a blockchain network to test the correctness of our designs and in order to be able to collect results wherever possible. Since setting up a complete functioning and private blockchain environment is not an easy task and not all test networks are suitable for every measurement we relied on two main test networks:

- Testnet21, a private instance (on a local physical computer) with a development Ethereum network being launched and mined locally with some accounts being created and used for testing purposes.

- Remix IDE testnet: Remix is a web-based Solidity tool for developers that has an easy and quick-to-deploy testnet where we write, compile and can instantiate our smart contracts and it comes equipped with 5 test accounts.

All things considered, we point out that both these networks relies on our local processor when it comes to mining operation or smart contract execution of any kind. We can relate with the local Testnet21 environment directly via the *geth* console which gives us all the necessary operations for doing basic interactions like: account creation, handling, eth transfer, transactions, smart contract deploy, execution, calls, all of which is obtained through the interaction with the Web3 API and its commands. While using the console we can even embed and execute *JavaScript* code with the use of *.js* files or inline.

In order to be deployed, a smart contract must first be compiled with either the *solc*[31] or the *Remix* compiler and if there is no error, it will output a bytecode, an *Application Binary Interface* (or *ABI* in short) and a Web3 deploy code. Now for the next step we must use one of these objects based on what tool we are using for deployment. For a low-level console Web3 deploy we can save the output code into a *.js* file, unlock a user account that will create the creation transaction and load the code using:

```
>_ personal.unlockAccount(0x07c48c6baa13aa4f974b219bcb731ace47f28f95, “password”, 60)
>_ loadScript(‘deployFile.js’)
```

[31] The official *Solidity compiler*

The script will issue a transaction with the request of a smart contract creation with the specified code and interface. After a moment our miner node will validate the transaction and output:

```
Contract mined! address: 0xd405d4f2dcde9ff66fb3aa4615d296b357e4feff
transactionHash: 0xa8eb4dcd2d4a4b6e5f34edb5f049779354dc336d897f6d4ed68cf9f4d59f9868
```

And it is done! We just need to store the address of our newly created Đapp in order to start using it; generally, when we want to interact with a smart contract we just need its address and a bit of documentation on its usage in order to query its functions. We can deploy the contract even from Ethereum's official client called *"Mist"* (in our tests we used *version 0.8.10*), it is a more straightforward process since the only requirement is the source code as the client will perform every step necessary to publish the code into the blockchain. Mist is both a Đapp-enabled browser that can interact directly with deployed blockchain services and a wallet Đapp that is used for handling user account operations and eth transfers.
If we want to add an already deployed smart contract to our list from another node (different than the deploy node) we need its address and *ABI*. The contract's interface system has been designed to be strongly typed, known at compilation time and static with no introspection provided. The assertion made here by developers is that all contracts will have the interface definitions of any called contracts available at compile-time (Catalano, 2017).

4.0 CONSIDERATIONS

This chapter will collect all the results from tests, considerations and research made on the Ethereum blockchain topic and the Solidity smart contract development and will discuss both the advantages and drawbacks of this innovation in order to better understand its implications. In this paragraph we will sum up all the questions we asked ourselves about Ethereum during the development of this project and give preliminary short answers that will introduce key insights into our analysis in the following sections. As with every new technology that comprises a vast environment in which different ideas and projects can flourish, there are a number of factors that must be kept in mind during an unbiased evaluation

First of all is to set aside all the excitement and hype that a tool such as this can generate in people's mind, fueled by often wrong media gossip. As we already stated, Ethereum, like other blockchain technologies is no *silver bullet* for any immediate usage, it surely enables a time shortening in development and deployment paradigms that more classic technologies do not have. Fast web development and ubiquitous services are features that we saw only in last years and in newer technologies (apart from enterprise tech of course), however one must not rush into believing that distributed and non-centralized structures are free of hindrance. A precise security evaluation must always be done in order to evaluate risks and benefits from the use of a new technology, especially in a system that involves digital money transfer over transactions. Some businesses have been so much lead astray by enthusiasm that have converted their local services into blockchain-based services without even considering benefits or issues of this architecture, out of the blue.

We questioned ourselves with the following matters:

1. *Can we use Ethereum to develop real applications that are useful both in a blockchain and non-blockchain environment?*

Our experience suggested that we are generally positive on the answer, however we also want to point out that application developed inside the Ethereum environment do rely on its architecture, this alone reflects what kind of projects are suited for this use and which does not. Decentralized app that can take advantage of a distributed database are natural candidate for development because of the availability and persistency level guaranteed by the system: a non-distributed counterpart could face failures that isolates part (or all) of its functionalities. Other than this, we can argue that Smart Contracts alone are not sufficient to build a complete service,

its limitations compels us with the need of a user-end interface (to integrate with the official *JavaScript* API) that translates the client's necessities into interaction with the blockchain. In a way this taxonomy is similar to the standard software division between *front-end* and *back-end*, however, here the connections are more loose because they rely on a time-inefficient structure which on average updates his status every ~12 seconds (if we need to store data) and this limit is hard-capped.

Moreover we have to think to the network stack related to our blockchain projects: starting from the final user we have a (probably proprietary) web-based front-end (1) which act as a friendly CLI[32] that translates functionalities and send them over to a light Ethereum client (2) or directly to a miner node (3). He in turn will execute some code, validate our transactions and pass back our results but the process could even go further if the service is reasonably complex and could rely to external libraries or off-chain features (4). Now if we watch through the layers of the communication stack we could say that objects 1-2-4 rely on an internet connection while 3 relies on a *second-tier* network built on internet too, therefore this model relies *heavily* on connection speed and will inherently suffers from any performance issue related to both networks. At the end of all this the "no downtime" and "censorship-resistant" advertised features we spoke of earlier may sound a little optimistic because rely on something that is not under complete control of the environment. Although this is true for all web services, Ethereum could also be bottlenecked by its own network like what happened in June after the launch of the *Status'* ICO (Valenzuela, 2017).

2. Do we have acceptable development and application performances, given the environment constraints?

There are clear difficulties that arise starting from setting up a complete working development environment to developing a correct smart contract code. This is due to the project relatively new coming out from preliminary test phase and will be probably balanced out as Ethereum will continue to evolve into the new *Metropolis* and following releases. However, as of right now development can be achieved following the basics from the docs (which too are not complete) and after that by practicing severe trial-and-error result evaluation on smart contracts. Some Solidity development frameworks are currently being developed and are in *beta* (like *Meteor* and *Truffle*) but their overhead and set up is still buggy although functioning.

[32] *Command Line Interface*

Application performance is very good for execution of code that involves only reading and simple interaction operations; it is obviously slowed when transaction or on-chain storage is required but this can vary from case to case. A part from its tools, the development process is an intuitive operation once one has mastered the main key concepts around blockchain, transactions and accounts, the language natural similarity with other object oriented languages greatly helps in this.

Performances could not be evaluated without considering debug and release too: the first is rather difficult because requires an accurate and tested working local environment in order to sort out some observable results, that is, only very simple Đapps can be tested "as they are", while other ones with incorporated business-logic must be thoroughly examined and checked for corner cases. Even if project speedup can increase thanks to the absence of heavy server or database infrastructure requirements and overhead, release is a difficult process too: as we pointed out, since every minor change or fix requires the contract to be republished again, propagation time an effort must be considered when making versioning plans.

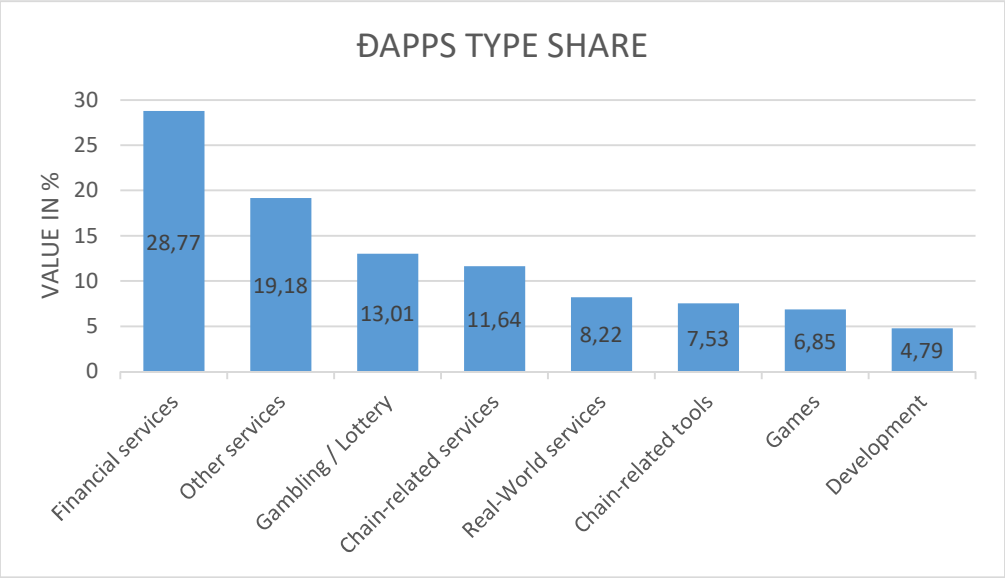
Time performance must also be acknowledged, with the computational power peak set at every 12 seconds the Ethereum viability for real-time or time-critical applications is practically out of the equation, therefore limiting its usage in industrial operations. Other distributed ledgers tailed for these use cases have or are currently been developed (like some with the *Hyperledger* project) but they are way different from Ethereum of course (The Linux Foundation, 2015).

3. Can we bridge the evaluation of our limited Đapps with real Ethereum applications?

An interesting question that follows all the work done until now. However to give an appropriate answer we will first analyze some numbers and statistics collected from the available information on the Ethereum Đapps ecosystem and their usage. As of July 2017 there are more than 550 confirmed Đapps on the Ethereum blockchain[33] a number that can be refined into approximately 230 applications marked as *live*, therefore running their service throughout a Đapp-enabled web-page. We precise that this number is *not* the effective number of uploaded smart contracts in the blockchain but rather a count of complete products that provide a supported service.

We classified the collected data and plotted the results in the following graphic:

[33] As read from "State of the Đapps" web service available at *https://dapps.ethercasts.com/*



Picture 7 - Distributed Apps shares by type in Ethereum

With the following examples of decentralized services listed inside this classification:

- *Financial services:* all applications that are built around economics (not including custom token unless directly tied with financial world) such as exchanges, prediction systems, markets, notary services, advertisement platforms, investment, remittance, insurance, virtual checks, microcredit and so on.

- *Other services:* the most colorful such as image creation and storage, avatars, blog generation, DNS resolving, contest creator, validator and voting platform, messaging, forum creation, Bitcoin bridging, Bitcoin full implementation, fitness motivational community apps, educational Ponzi schemes, short messages pegged to URLs and files, document and information dissemination, whitepaper-like companies record, data scraping, links and address validation and so on.

- *Gambling / Lottery:* every application that stakes a certain amount of money for a promised (and an improbable) payout.

- *Chain-related services:* cloud-storage, Ethereum naming, DAOs, token-based projects, enterprise blockchain implementations and so on.

- *Real-World services:* estate crowdfunding, ether time-store bank, lending platform, frequent flier program, e-commerce payments, real transportation of goods with

Ethereum-aware pegged devices that can track expeditions, gold storage, asset propriety, electric car refuel systems sharing and so on.

- *Chain-related tools:* wallets, Đapp-enabled browsers, chain explorers, chain stats and gas statistics providers.

- *Games:* all applications with a simple gaming purpose (with no money involved).

- *Development:* Đapp developing frameworks, external support libraries, random generation DAOs, code designers, Solidity test simulation environments.

As we can see, the majority of these services implements a blockchain enabled app either for storage purposes of some important information or the use of the *p2p* architecture as it is, with leading cases and applications tied in the financial and transactional area. However, the quality of these applications reside primarily in the goodness of the economic algorithms built behind the forecasting operation of stocks, titles and investments making it hard to evaluate as they are. The blockchain here is seen as a cheaper and simpler infrastructure than the classic enterprise solutions but still, the know-how remains in the hands of the service providers and not on the smart contract by itself.

4. *Is there any advantage in the use of Ethereum instead of a traditional approach?*

Ethereum has indeed materialized some very powerful concepts and has managed to build a platform around them using features that were previously just theorized like state machine replication systems (Rachid Guerraoui, 2009), or never deployed from both an algorithmic (modified GHOST[34] protocol implementation) and business (distributed economy) point of views (Yonatan Sompolinsky, Secure High-Rate Transaction Processing in Bitcoin, 2015). All of this will surely benefit the decentralization process of internet services and create a streamlined channel for private users to get to know better the web and to trust its architecture. The advantages we saw here in this work have been mainly related to the ease that Ethereum aims to get to starting from the design process to the deployment of a working economic Đapp; the steep learning curve is justified by the high complexity that the platform hides from final users.

[34] The "Greedy Haviest Observed Subtree" was first introduced in (Yonatan Sompolinsky, Accelerating Bitcoin's Transaction Processing Fast Money Grows on Trees, Not Chains, 213)

5. Which security does this platform offer to smart contracts?

A short answer to this question would be: “still not enough”. Even with the embedded cryptography functions (an expected feature that *any* type of modern transaction-based system should have) and a “List of Known Bugs” provided at the end of Solidity documentation (Ethereum Community, 2016) there are still a number of ways to circumnavigate checks and submit faulty or bugged contracts. In the first days of July a new compiler version of Solidity has been released that now has a major impact when encoding possibly dangerous variables or features inside a source code. However Ethereum is currently in development phase and its low maturity can be understood but have to be acknowledged by both Solidity developers and final users that often consider this an underestimated reality. Even the Solidity language itself is still under development as some features are currently not available (floating points are just an example), the absence of proper experienced-documentation and more comprehensive security-related guidelines therefore makes it harder to code applications efficiently. Stability issues have become less frequent but still present sometimes, while official guidelines warns against deploying anything that is production-ready to the network at its current stage, postponing everything with the following release.

4.1 TECHNOLOGY GAPS / LIMITS

Our work has documented a wide range of possible implementations and use cases of the Ethereum blockchain and its environment praising its advantages and features; now we will focus solely on its limitations based on our experience and its architecture evaluation. We will proceed by summarizing its limitations starting from technical ones and then proceed to different points of view related to the technology itself and the stakeholders tied to it.

1. *Consensus protocol:*

Currently *Proof-of-Work* delivers goods results on average, the problems related with PoS are mainly tied with its *enormous* energy consumption (the Bitcoin network alone burns about 14.43 *TWh*[35] on a yearly estimate, close to the total energy consumption of the whole *Turkmenistan*) (Digiconomist, s.d.), and the fact that this consensus protocol could be influenced with the use of a certain amount of resources (Ittay Eyal, 2014). Although Ethereum

[35] 1 *Terawatt-hour* equals 10^{12} *watt-hour*

PoW is slightly different, safer and more efficient this protocol will likely be abandoned in favor of *Proof-of-Stake.* The idea behind PoS is that instead of having to prove an amount of work spent on the block generation, the nodes will have to prove ownership of their currency balance. In a PoS system the blocks are mined by the nodes voting on which will be the next block in the chain while the voting rights are distributed according to the "stake" each node (*validator*) has in the network. For the consensus part the PoS validators can rely on both *chain-based* proof of stake *or BFT-style* proof of stake algorithms which have a different validator selection policy. The approach used by Ethereum in PoS is different than already deployed projects like *BlackCoin* (Vasin, 2014) and *PPCoin* (Sunny King, 2012), having an array of benefits that range from low-energy consumption, less need of new coins, "mining" centralization risk discouragement and penalties to make 51% attacks more expensive (Buterin, Proof of Stake FAQ, 2017). Ethereum leader and creator *Vitalik Buterin* is currently working on the *Casper* algorithm, which is the Ethereum implementation of Proof-of-Stake that will replace the current PoS. All information about Casper can be found online in the community website however, this falls outside the scope of the present work.

2. *Scalability requirements:*

As time progresses and the blockchain becomes streamlined and longer, the space needed to store all the distributed ledger information increases as well, generating a scalability requirement that cannot be easily resolved. This problem has been central point of discussion since Ethereum creation, a number of official and unofficial threads have been opened on the topic (like in *Ethereum Reddit*). A number of partial solutions have been proposed by developers and researchers, summarized in this article (Simon, 2017) but the only proposal which does not implies a radical change in the mining operation, block size or the use of sub-chains is the *sharding* technique. Sharding was first introduced in (Loi Luu V. N., 2016) and later suggested for implementation into the Bitcoin network: the paper describes techniques and operations to split the transaction processing state or the state itself into multiple partitions called "shards". The hindrance here is that the effort done could lead to some optimization but in the end, Ethereum developers should choose which problem to solve: the processing one, resulting in a very high transaction throughput capacity or the space one, partitioning state information to multiple nodes. The overall problem is very actual, with different hybrid proposals and solutions being actively discussed (Simon, 2017). For the time being however, the space requirement to store a full Ethereum node will not shrink at all.

3. *Unpredictable state:*

As we saw previously, the state of a smart contract is determined by the value of its address fields and balance. However, even a simple call to the contract could take several seconds in order to take place, this could result in finding an unexpected state of the contract. Generally, when a user sends a transaction over to the network to invoke a contract, he cannot be sure that the transaction will be run in the same state the contract was at the time of sending that transaction. This may happen because during the same time-window other transactions or contract activations could have changed its state. There is an intrinsic mutual exclusion and state propagation problem here that must kept in mind when designing the logic of a viable Đapp. Even if the user is fast enough to be the first to send a transaction it is not guaranteed that such transaction will be the first to be run thanks to the lack of total ordering in transaction pools. Miners that group transactions in blocks are not required to preserve any order and they could choose not to include some transactions at all, this can also easily happen if a user assigns substantial different *fees* to transactions.

4. *Smart Contracts issues:*

As we saw, there are a number of problematic factors that do not encourage smart contract development and neither contribute to their spreading. First of all **speed**: even 12 seconds are still a *lot* of time for end-users usually accustomed to buying stuff or services with a feedback provided in a few clicks; *Proof-of-Stake* promises a significant drop in this time, but it has to be proven. Transactions do have a **cost**, in order to complete some basic and not-academic operations they always require a fee; surely, the amount paid *per transaction* is not as expensive as other Ethereum's competitors are (like Bitcoin) but still it is to be noted. Moreover, the carried out computation is largely forced to be **public** with no direct means to maintain secrets or provide any privacy at all: this is a feature that Ethereum *wants* to provide in the future (Buterin, Privacy on the Blockchain, 2016). As of right now however a solution to this problem must always be custom-developed and is difficult to enforce and maintain until new features are made out of Solidity. Finally, an off-chain serious **integration** is very hard to obtain: while there are some very useful services (like the one we used, *Oraclize*) it is still dragon's *land* with few certainties and legion of workarounds.

5. *Other related risks:*

As a new technology that covers a wide area of applications and peers into economics there are other kind of risks that we can categorize under the following arguments:

- *Regulatory*, governments could realize the versatility of an economic platform such as this and choose to limit or forbid its usage. Since right now we have to rely mostly on exchanger sites in order to convert fiat to cryptocurrencies they could censor those sites and cut down its usage. Or they could reach a point of regulations that will make the use of the system not viable or convenient anymore.

- *Reputational*, a bad reputation is always a token of low consideration, although this concept is cross related to everything the cryptocurrencies tech can be directly influenced by a number of third-party rep image like exchangers (like what happened in 2014 with *Mt.Gox*), new fraudulent pyramidal scheme Đapps, a number of user's wallet hacks and other security-related violations.

- *Adoption*, related to reputation too but focused more on the user end. It is very hard to convey an Ethereum explanation to the general public, often the press and news sources appoint the word "cryptocurrency" to *everything* which is financial related on the web and dubs "blockchain" with no reference to specific products or systems. Another problem is that it is still difficult for a non-tech user to acquire Ether and secure it safely, the absence of a third-party (like a bank) entity that takes all the risks of managing (or losing) the user's money is not easily accepted concept and distributed responsibility is a great burden. All of this must be acknowledged and should be understood by the final user at the same time.

4.2 CODE EXECUTION SECURITY

We provided examples and proof that code execution inside the Ethereum environment is critical to its usability, moreover security must be adequately audited in order to create a correct and valid Đapp service. Apart from specific code errors and behaviours we reviewed that can lead to security breaches, we will summarize some highlights regarding what has been seen contributing to the disruption of proper services.

Denial-of-Service attacks in Ethereum can be mounted either against the platform or against a specific smart contract service, these attacks target vulnerabilities in the EVM specification level, combined with security flaws in the Ethereum client. The community has experienced an

example of this on September 2016 (Wilcke, 2016); the attackers flooded the network with a huge quantity of instruction execution requests in which the cost in gas was too low compared to the computational effort needed to carry them out. This resulted in a heavy slowdown of the network and of the synchronization process. After the end of the attack a number of fixes and low-level gas cost EVM corrections had to be made and rolled-out, the outcome of all this was another blockchain fork like in the past with *TheDAO* incident (Swende, 2016).

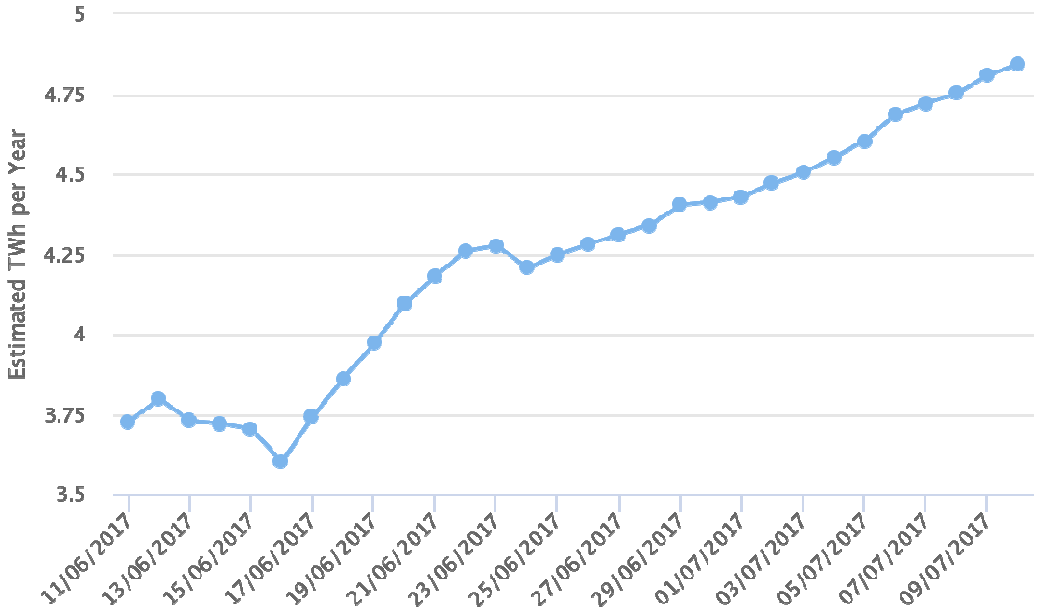
Even clients have been proven to be critical in the safe upkeep of the Ethereum protocol and must be subject to severe security audit in order to guarantee an adequate level of security as an attacker could use a flaw in the system as a vector to undermine its protocol (Karl Wüst, 2016). A single Denial-of-Service attack deployed against a contract relies on a malevolent fallback function being coded inside a smart contract by an attacker. An example is that if the function is implemented with just a "throw" exception, any situation in which an honest caller smart contract calls the attacker one (with an execution that prompts the fallback) would end up in being reverted every time, possibly disabling the service. A thorough example of this is given in the *"King of the Ether Throne"* game application in different papers (Nicola Atzei, 2016), while other research papers like (Kevin Delmolino, 2016) and (Loi Luu D.-H. C., 2016) have shown that even a simple smart contract as a "Rock, Paper, Scissors" game can contain several logic problems.Code security is therefore critical in Solidity development and it requires an "economic thinking" prospective different from other development processes: application designers should always consider costs, fees and defensive coding prior to any business logic. Contracts have to be written to ensure fairness (wherever possible) when multiple parties may attempt to access the service or result, but the key-factor to safety is keeping the economic incentive for performing an attack always greater than the payout of its eventual success.

From a different prospective however we have smart contracts that are bound to follow a rigid application logic greatly limiting one's capacity to write malware over the platform, even because a contract has access only to its own memory context (Triantafyllidis, 2016). The security of the platform itself is hence relayed on the security of the EVM and with the single client implementation. As we pointed, the effort must be focused on maintaining a healthy EVM implementation with no bugs or exploitable security issues, as long as this task is accomplished, the code security is reasonably safe. Moreover, once deployed on the blockchain only the contract's bytecode is stored, thus a user must always put a degree of *trust* in both the Đapp provider and in the executing node.

There will be always bugs, pitfall that may come into light since the project is continuously in motion, however, following the updated documentation and guidelines from community can ease the process while newer and safer Solidity design patterns are unveiled.

4.3 POWER CONSUMPTION / COSTS

Lastly, as energy consumption and costs are another critical factor into the evaluation of the blockchain technology itself we will give some insights about the topic. In order to better understand this details we will provide data and graphics, while discussing costs implications:



Picture 8 - Ethereum Energy consumption index

Although this data source is still in beta and collected by an external observer[36] it relays a good esteem of the actual hash power being used for mining blocks in the Ethereum environment. Ethereum uses way less power than Bitcoin does (has less nodes, and a different hashing algorithm called *Ethash*) but with his roughly annual average of ~4.84 *TWh* consumption, it is close *Moldova* country, with an amount of energy spent *per transaction* equal to 50 KW/h.
As of July 2017, the estimated price for Bitcoin miners is about 5 $ cents per KWh, while Ethereum miners are assumed to pay about 12 $ cents. This is due to the fact that Bitcoin miners, after coming a long way down from CPU, GPU and FPGA mining [37], nowadays relies heavily

[36] *digiconomist.net*
[37] CPU: Central Processing Unit, GPU: Graphical Processing Unit, FPGA: Field-Programmable Gate Array

on ASICs[38] that can deliver easily *10,000 GH/s*[39] at *0,25 W/Gh*[40] and from the inception of crypto this hardware has been well-industrialized. Ethereum on the other hand is mined only over GPUs because *Ethash* is ASIC-resistant and has been design such as this for two reasons: Firstly, in order to diminish the feasibility of miner aggregation into big *pools* (like for Bitcoin) that could retain a major total hashpower, secondly because PoW is used has a bootstrap algorithm to mint the initial coins but the long term goal has always been PoS. This calculations do not take into consideration the revenue generated by the cryptocurrencies, that is, when block generations mints new tokens there is an effective payout that must be considered if we want to make a complete year evaluation based on the cost of energy against revenues. In this sense Ethereum generates just a smaller value of gross income than Bitcoin (~$2 vs $2.4 billion) meaning that Ethereum efficiency is way higher as the circulating supply and volume of eth overcome is elder brother (Coinmarketcap, 2017).

As for the maintenance costs of Smart Contract we gathered our own data on the following table:

ĐAPP NAME	DEPLOY COST (gas units)	USE COST (avg use)	MEAN TTC (N° of Blocks)	MEAN TTC (Seconds)	TX Fee (ether)	TX Fee (USD)
<simple transaction>	21.000	---	3,6	69	0,00042	$ 0,080
Fibonacci	326.954	$ 0,325	5,2	99	0,006539	$ 1,275
Random	304.210	$ 0,075	5,2	99	0,006084	$ 1,186
Rubixi	2.032.749	$ 0,585	5,2	99	0,040655	$ 7,928
SavingsContract	883.432	$ 0,943	5,2	99	0,017669	$ 3,445
EnerTrade	3.005.475	---	5,2	99	0,06011	$ 11,721

For every issued transaction, the gas costs has been fixed to 20 Gwei as this is the signaled[41] Gas price mid-range for a safe and relatively fast transaction validation (TTC is Time-To-Confirmation). We have separated the *deploy cost* (which is the sum of the transaction and payload costs) and used it as the reference for all the remaining table data apart from the *use cost* that is a direct estimate of an average use (given by a round-trip of a full functionality or function calls). The *TX Fee* fields are the effective cost of our contracts first deployment over the Ethereum network, as we can deduce when external libraries and tools are involved in the

[38] Application-specific integrated circuits
[39] *Gigahash per second*
[40] *Watts per Gigahash*
[41] Information taken from *http://ethgasstation.info*

project, the cost is greater because all the referenced or *imported* code in our file will be added dynamically to the lines right before the compilation in byte-code. This information gives a rough esteem of smart contracts upkeep.

5.0 CONCLUSIONS

In this work, we started by studying what the blockchain phenomenon is, trying to understand why this technology has received such a hype in the media and what can it really achieve. However, we wanted to look for a solid application or use case that could benefit from this tech and not just from its hype. As we peered deeper into the papers that described the main technical features of distributed ledgers and gained insights on their limits, we discovered the Ethereum blockchain project and our attention was then captured by the concepts conveyed with its vision. We covered a lot of ground in the Ethereum blockchain development and assessed wherever possible, all the features and innovations proposed by it focusing on Smart Contracts. The Ethereum team is making a great work into delivering a next-generation tool that has enabled an innovative tech such as blockchain to better integrate with everyday life and necessities. Whereas Bitcoin and the other *altcoins* provided a "dark" and cloudy way to financially-achieve just an end, Ethereum managed to create a decentralized mean to use policies and code into a more comprehensive system that has a great potential.

Some questions do remain: will a killer-Đapp be found ever? The rise of other Ethereum-like projects may have more success than Ethereum? Bitcoin could significantly update its structure and become more capable? These are all good points to think on, but as long as the Ethereum community carries on with development, keeps their goals clear and their mind open the maturity of the whole project will be just a matter of time. People must understand what this technology is really about and what are the correct use cases that can lever its features and not just use it for everything that comes by. We saw throughout this work that the importance of the platform in being a common ground (a *global computer*) where new applications and ideas can grow on, with the ability to interact with each other relying on a networked set of peers that can transfer even money value and currencies. This degree and freedom and flexibility has only be seen in the past with the invention of the *World Wide Web* and its *HTTP* protocol in 1980, and that was too an attempt to decentralize a set of services that were before only created specifically ad-hoc. Like for any distributed technology that has been invented and deployed (*Peer-to-Peer* alike), some time is required to reach its full functional state and potential. The paradigm shift of decentralized features in operations such as money transfers, public verifiable votes and online contracting needs to be digested by the whole internet community, however the simplicity expressed in Ethereum is unprecedented and other traditional approaches would be too complex and very difficult to understand for the general public. We can argue that

Ethereum today is the most advanced form of programmable distributed ledger actively deployed and maintained and this uniqueness is what will enable the technology to take the necessary steps to be considered a next-generation environment and platform.

There is however plenty of security work to do ahead as both with the current state of Đapps and with the future features that will likely to come in the platform; a significant effort could be invested into discovering and potentially fixing its vulnerable components in order to make the community and the project grow alike.

IMAGE INDEX

Bibliography

Armstrong, S. (2016). *Move over Bitcoin, the blockchain is only just getting started.* Wired. Retrieved from http://www.wired.co.uk/article/unlock-the-blockchain

Bheemaiah, K. (2015). *Block Chain 2.0: The Renaissance of Money.* Wired. Retrieved from https://www.wired.com/insights/2015/01/block-chain-2-0/

Bill Marino, A. J. (2016). *Setting Standards for Altering and Undoing Smart Contracts.* Cornell Tech (Jacobs Institute).

BitcoinWebHosting. (2016). Retrieved from http://historyofbitcoin.org/

Buterin, V. (2014). *Ethereum White Paper.* Retrieved from https://github.com/ethereum/wiki/wiki/White-Paper

Buterin, V. (2015). *On Public and Private Blockchains.* Ethereum Blog. Retrieved from https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

Buterin, V. (2015). TNABC 2015 - Bitcoin 2.0 - Ideas and Applications. (Bitcoinist.net, Interviewer)

Buterin, V. (2016). *Privacy on the Blockchain*. Retrieved from blog.ethereum.org: https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/

Buterin, V. (2017). *Proof of Stake FAQ*. Retrieved from github.com: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ

Cachin, C. (2016). *Architecture of the Hyperledger Blockchain Fabric.* Zurich: IBM Research. Retrieved from https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf

Calvery, J. S. (2013). Statement of Jennifer Shasky Calvery, Director Financial Crimes Enforcement Network United States Department of the Treasury. Financial Crimes Enforcement Network.

Castillo, J. B. (2013). *Bitcoin: A Primer for Policymakers.* (G. M. University, Ed.) Mercatus Center.

Catalano, R. (2017). *Ethereum Contract ABI.* Retrieved from https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI

Cécile Pierrot, B. W. (2016). *Malleability of the blockchain's entropy.* Sorbonne Universités, EPFL IC LACAL. IACR Cryptology ePrint Archive 2016.

Chavez-Dreyfuss, G. (2016). *Sweden tests blockchain technology for land registry.* New York: Reuters.

Coinmarketcap. (2017, July). *CryptoCurrency Market Capitalizations*. Retrieved from coinmarketcap.com: https://coinmarketcap.com/

Digiconomist. (n.d.). *Bitcoin Energy Consumption Index*. Retrieved from digiconomist.net: http://digiconomist.net/bitcoin-energy-consumption

Don Tapscott, A. T. (2016). *Here's Why Blockchains Will Change the World.* Fortune. Retrieved from http://fortune.com/2016/05/08/why-blockchains-will-change-the-world/

Don Tapscott, A. T. (2016). *The Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World.* Portfolio / Penguin.

Dwork C, N. M. (1992). *Pricing via processing or combating junk email.*

Ethereum Community. (2016). *Ethereum Homestead Docs*. Retrieved from Ethdocs.org: http://ethdocs.org/en/latest/introduction/index.html

Ethereum Community. (2016). *Solidity Official Documentation*. Retrieved from solidity.readthedocs.io: http://solidity.readthedocs.io/en/develop/index.html

Foundation Team. (2014). *Ethereum Foundation Mission and Vision Statement.*

Foundation, E. (n.d.). *Gas Costs for EVM operations 1.0*. Retrieved from docs.google.com: https://docs.google.com/spreadsheets/d/1m89CVujrQe5LAFJ8-YAUCcNK950dUzMQPMJBxRtGCqs/edit#gid=0

Fredrik Milani, L. G.-B. (2016). *Blockchain and Business Process Improvement.* BPTrends.

Greenspan, G. (2015). *Ending the bitcoin vs blockchain debate.* Retrieved from http://www.multichain.com/blog/2015/07/bitcoin-vs-blockchain-debate/

Greenspan, G. (n.d.). *MultiChain Private Blockchain Whitepaper.* Coin Sciences. Retrieved from http://www.multichain.com/download/MultiChain-White-Paper.pdf

Higgins, S. (2015). *Bitcoin-Powered Crowdfunding App Lighthouse Has Launched.* Coindesk. Retrieved from http://www.coindesk.com/bitcoin-powered-crowdfunding-app-lighthouse-launches-open-beta/

Higgins, S. (2016). *Visa to Launch Blockchain Payments Service Next Year.* CoinDesk. Retrieved from http://www.coindesk.com/visa-blockchain-payments-service/

Higgins, S. (2017). Miners Boost Ethereum's Transaction Capacity with Gas limit increase. Retrieved from http://www.coindesk.com/miners-ethereum-transactions-gas-limit/

Hyperledger. (2016, November 1). *Hyperledger Welcomes Iroha*. Retrieved from Hyperledger: https://www.hyperledger.org/blog/2016/11/01/hyperledger-welcomes-iroha

Hyperledger. (2016, November 2). *Meet Sawtooth Lake*. Retrieved from Hyperledger: https://www.hyperledger.org/blog/2016/11/02/meet-sawtooth-lake

Hyperledger. (2017, Jenuary 17). *Hyperledger Says Hello To Cello*. Retrieved from Hyperledger: https://www.hyperledger.org/blog/2017/01/17/hyperledger-says-hello-to-cello

International Monetary Fund. (2016). *Virtual Currencies and Beyond: Initial Considerations IMF Discussion Note.* International Monetary Fund. Retrieved from https://www.imf.org/external/pubs/ft/sdn/2016/sdn1603.pdf

Investopedia. (n.d.). *Definition of 'Initial Coin Offering (ICO)'*. Retrieved from investopedia.com: http://www.investopedia.com/terms/i/initial-coin-offering-ico.asp

Ittay Eyal, E. G. (2014). *Majority is not Enough: Bitcoin Mining is vulnerable.* Cornell University, Department of Computer Science.

Karl Wüst, A. G. (2016). *Ethereum Eclipse Attacks.* ETH Zurich. Retrieved from https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/121310/eth-49728-01.pdf?sequence=1&isAllowed=y

Kelly, J. (2016). *Exclusive: Blockchain platform developed by banks to be open-source.* London: Reuters. Retrieved from http://uk.reuters.com/article/us-banks-blockchain-r3-exclusive-idUKKCN12K17E

Kevin Delmolino, M. A. (2016). *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab.* University of Maryland, Cornell University, Department of Computer Science.

Leslie Lamport, R. S. (1982). *The Byzantine Generals Problem.* SRI International.

Levy, H. P. (2016). *The CIO's Guide to Blockchain.* Gartner. Retrieved from http://www.gartner.com/smarterwithgartner/the-cios-guide-to-blockchain/

Liu, A. (2013). *Beyond Bitcoin: A Guide to the Most Promising Cryptocurrencies.* Vice. Retrieved from http://motherboard.vice.com/blog/beyond-bitcoin-a-guide-to-the-most-promising-cryptocurrencies

Loi Luu, D.-H. C. (2016). *Making Smart Contracts Smarter.* ACM CCS.

Loi Luu, V. N. (2016). *A Secure Sharding Protocol For Open Blockchains.* National University of Singapore, Singapore. Retrieved from https://www.comp.nus.edu.sg/~loiluu/papers/elastico.pdf

Marcin Andrychowicz, S. D. (2013). *Secure Multiparty Computations on Bitcoin.* IEEE Symposium on Security and Privacy.

Marcin Andrychowicz, S. D. (2014). *Secure Multiparty Computations on Bitcoin.* University of Warsaw, Poland.

Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* Retrieved from https://bitcoin.org/bitcoin.pdf

Nicola Atzei, M. B. (2016). *A survey of attacks on Ethereum Smart Contracts.* Università degli Studi di Cagliari.

Paul Vigna, M. J. (2015). *The Age of Cryptocurrency: How Bitcoin and Digital Money Are Challenging the Global Economic Order.* St. Martin's Press.

Popper, N. (2016). *A Venture Fund With Plenty of Virtual Capital, but No Capitalist.* The New York Times.

Rachid Guerraoui, V. Q. (2009). *The next 700 BFT Protocols.* Retrieved from https://infoscience.epfl.ch/record/121590/files/TR-700-2009.pdf

Ripple. (2017). *Ripple Solutions Guide.* ripple.com. Retrieved from https://ripple.com/files/ripple_solutions_guide.pdf

Rosenfeld, M. (2014). *Analysis of hashrate-based souble-spending.*

Ross, R. (2015). *Smart Money: Blockchains Are the Future of the Internet.* Newsweek.

Sayer, P. (2016). *Bankers plan to give Corda blockchain code to Hyperledger project.* PCWorld. Retrieved from http://www.pcworld.com/article/3134014/bankers-plan-to-give-corda-blockchain-code-to-hyperledger-project.html

Siegel, D. (2016, 06 25). *Understanding The DAO Attack.* Retrieved from coindesk.com: http://www.coindesk.com/understanding-dao-hack-journalists/

Simon, A. (2017). *On sharding blockchains.* (The Ethereum Foundation) Retrieved from github.com: https://github.com/ethereum/wiki/wiki/Sharding-FAQ

Spaven, E. (2015). *MasterCard: Digital Currency's Risks Outweigh the Benefits.* CoinDesk. Retrieved from http://www.coindesk.com/mastercard-digital-currencys-risks-outweigh-the-benefits/

Stefan Thomas, E. S. (n.d.). A Protocol for Interledger Payments. 25. Retrieved from https://interledger.org/interledger.pdf

Sunny King, S. N. (2012). *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake.*

Swende, M. (2016, October 13). *Announcement of imminent hard fork for EIP150 gas cost changes.* Retrieved from blog.ethereum.org: https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/

Szabo, N. (1994). *Smart Contracts.* unpublished manuscript. Retrieved from http://www.erights.org/smart-contracts/

The Economist. (2015). The great chain of being sure about things. *The Economist.* Retrieved from http://www.economist.com/news/briefing/21677228-technology-behind-bitcoin-lets-people-who-do-not-know-or-trust-each-other-build-dependable

The Linux Foundation. (2015). *Linux Foundation Unites Industry Leaders to Advance Blockchain Technology.* San Francisco: The Linux Foundation. Retrieved from https://www.linuxfoundation.org/news-media/announcements/2015/12/linux-foundation-unites-industry-leaders-advance-blockchain

Triantafyllidis, N. P. (2016). *Developing an Ethereum Blockchain Application.* University of Amsterdam, System & Network Engineering, MSc.

Valenzuela, J. (2017, June 21). *ICO Mania Grinds Ethereum to a Halt, Scaling Issues Not Limited to Bitcoin.* Retrieved from dashforcenews.com: https://www.dashforcenews.com/ico-mania-grinds-ethereum-halt-scaling-issues-not-limited-bitcoin/

Vasin, P. (2014). *BlackCoin's Proof-of-Stake Protocol v2.* www.blackcoin.com. Retrieved from https://bravenewcoin.com/assets/Whitepapers/blackcoin-pos-protocol-v2-whitepaper.pdf

Wilcke, J. (2016, September 22). *The Ethereum network is currently undergoing a DoS attack.* Retrieved from blog.ethereum.org: https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/

Wood, G. (2014). *Ethereum: a secure decentralised generalised transaction ledger.*

Wood, G. (n.d.). *What is ethereum? | Ethereum Frontier Guide*. Retrieved from Ethereum.gitbooks.io: https://ethereum.gitbooks.io/frontier-guide/content/ethereum.html

Yonatan Sompolinsky, A. Z. (2015). *Secure High-Rate Transaction Processing in Bitcoin.* Retrieved from http://fc15.ifca.ai/preproceedings/paper_30.pdf

Yonatan Sompolinsky, A. Z. (213). *Accelerating Bitcoin's Transaction Processing Fast Money Grows on Trees, Not Chains.* Retrieved from http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf